# OPTIMIZATIONS BASED ON TEMPORAL COHERENCE FOR RENDER FARMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AHMET UMUT GÜLKÖK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2014

Approval of the thesis:

**OPTIMIZATIONS BASED ON TEMPORAL COHERENCE FOR RENDER FARMS**

submitted by **AHMET UMUT GÜLKÖK** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** ________________

Prof. Dr. Veysi İşler
Supervisor, **Computer Engineering Department, METU** ________________

**Examining Committee Members:**

Assoc. Prof. Dr. Murat Manguoğlu
Computer Engineering Department, METU ________________

Prof. Dr. Veysi İşler
Computer Engineering Department, METU ________________

Assoc. Prof. Dr. Alptekin Temizel
Graduate School of Informatics, METU ________________

Assist. Prof. Dr. Kayhan İmre
Computer Engineering Department, Hacettepe University ________________

Assist. Prof. Dr. Ahmet Oğuz Akyüz
Computer Engineering Department, METU ________________

**Date:** ________________

# ABSTRACT

OPTIMIZATIONS BASED ON TEMPORAL COHERENCE FOR RENDER FARMS

Gülkök, Ahmet Umut
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Veysi İşler

January 2014, 99 pages

Temporal coherence is very important for various computational motion picture applications. Animation rendering and video encoding are good examples of these applications. This study proposes an optimization technique using temporal coherence for distributed animation rendering environments so called “Render Farms”. The proposed approach consists of separate procedures for reducing the network communication cost and providing dynamic computational load balancing between render farm worker computers. The network communication cost between the render farm controller and the workers is reduced by compressing the output images with the H.264 video codec which provides significant compression as the coherence between the adjacent frames increases. In addition, computational load balancing is achieved by a cost prediction method based also on temporal coherence. However, these two methods work best with different task distribution schemes. Therefore, these two optimizations are combined with a new task distribution algorithm considering the tradeoffs.

Keywords: distributed rendering, render farm, load balancing, animation rendering, temporal coherence, temporal coherence, video encoding

# ÖZ

GÖRSELLEME ÇİFTLİKLERİ İÇİN ZAMANSAL TUTARLILIĞA DAYALI OPTİMİZASYONLAR

Gülkök, Ahmet Umut
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Veysi İşler

Ocak 2014 , 99 sayfa

Bu çalışma dağıtık görselleme ortamları ya da diğer bir deyişle görselleme çiftlikleri için zamansal tutarlılığı kullanan bir optimizasyon tekniği önermektedir. Zamansal tutarlılık görüntünün ardışık kareleri arasındaki benzerlikleri ifade eder ve hareketli görüntüler için en önemli özelliklerden biridir. Önerilen yaklaşım görselleme çiftliğindeki işçi bilgisayarlar ve kontrolcü bilgisayar arası iletişim yükünü ve işçilerin aralarındaki işlem yükü dengesizliklerini azaltmak için zamansal tutarlılıktan faydalanan iki ayrı metot kullanmaktadır. Ağ iletişimi yükünü azaltmak için çıktı görüntüler H.264 codec'i ile sıkıştırılmıştır. İşlemsel yük dengesini kurabilmek için ise zamansal tutarlılık kavramından faydalanan bir kalan süre tahmin etme yöntemi geliştirilmiştir. Ancak bu iki yöntemin en verimli çalıştığı iş dağıtım biçimleri farklıdır. Bu nedenle bu iki metodun birlikte olabildiğince verimli çalışabilmesini sağlayan yeni bir iş dağıtımı algoritması geliştirilmiştir.

Anahtar Kelimeler: dağıtık görselleme, görselleme çiftliği, yük dengeleme, animasyon görselleme, zamansal tutarlılık, kareler arası tutarlılık, video kodlama

*To my parents Ferah Gülkök and Yusuf Ziya Gülkök*

# ACKNOWLEDGMENTS

I would like to thank my supervisor Professor Veysi İşler for his faithful discussions with constant support and guidance.

I appreciate Cem Ulu from Zor Zanaat Animation Studio for his efforts on helping me with understanding the basics of the render farms, for providing me with the testing material and giving me ideas about the techniques used in this thesis.

The hardware resources of MODSIMMER (Modeling and Simulation Research and Development Center - Middle East Technical University) are used for all test runs of the software developed for thesis work.

Special thanks to Mr. Can Soysal (Coordinator of “TRT Çocuk") and Mr. Bekir Yıldızcı (Assistant Coordinator of “TRT Çocuk”) for starting the idea of an academical study on render farms.

I also thank my friends Emra Aşkaroğlu, Berkhan Deniz and Deniz Karatay for their friendship and patient discussions about the thesis.

Special thanks to “Kulis” for making life fun during this hard period of time.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| CPU | Central Processing Unit |
| DDS | Data Distribution Service |
| DR | Distributed Rendering |
| DRE | Distributed Rendering Environment |
| DLL | Dynamic-Link Library |
| GI | Global Illumination |
| HDFS | Hadoop Distributed File System |
| FPS | Frames Per Second |
| MODSIMMER | Modeling and Simulation Research and Development Center |
| NFS | Network File System |
| PPP | Processing Power Point |
| RAM | Random Access Memory |
| SLI | Scalable Link Interface |
| WAN | Wide Area Network |

# CHAPTER 1

# INTRODUCTION

## 1.1 Background

The process of realistic image synthesis in computer graphics consists of several phases such as preparing the 3D model, determining and setting materials and textures, positioning and adjusting the virtual light sources and finally rendering [9]. The term "rendering" in computer graphics is the process of generating a 2D image from a model of a 3D environment. This 3D environment is named as scene file or scene data structure. It is a collection of 3D geometry, texture data, material definitions, light positions and light characteristics [8]. Naturally, the more accurate and detailed this data is, the more realistic the final 2D image will be. However, the detail of the scene definition is not the only criteria for the photo-realism. The algorithms used for the rendering also affect the photo-realism significantly.

The ultimate goal of photo realistic image synthesis is to acquire an image which does not differ from the real world image of a scene [8]. The success in photo-realism of a rendering technique depends on the precision of the algorithms while simulating the key aspects of the illumination such as shadows, soft shadows, reflections, transparency, translucency, refraction, diffraction, indirect illumination, caustics etc. To achieve this, since the 1970s researchers proposed various photo-realistic rendering techniques such as "Ray Tracing", "Ray Casting", "Radiosity", "Photon Mapping" etc. Each of them has various advantages compared to the others. For example, one of the earliest techniques is the "Ray Casting" rendering technique that is proposed first by Arthur Appel in 1968 [4]. Actually the idea behind it is significantly old. The first ray casting idea is proposed by Rene Descartes in 1637 and he applied the laws of refraction and reflection to a spherical water droplet to demonstrate the formation of rainbows. The techniques "Ray Casting" is very successful for spot lights. Lately, in 1979 Turner Whitted came up with the technique called "Ray Tracing" which is an evolved version of the idea of Appel, but unlike the "Ray Casting" this idea solves the lighting equation completely the phenomena refraction and reflection at the same time [18]. The Radiosity technique, on the other hand, proposed first by Goral et. al. in 1984, is better for diffuse lights [15]. These techniques are generally named as "Global Illumination Techniques".

Although, the global illumination techniques provide great photo-realism, they bring a major issue which is extremely high computational cost. This cost is so high that with today's graphics hardware these techniques cannot be applied to the real-time applications such as computer games or virtual reality simulators. Instead global illumination techniques are the major tool for non-real-time photo-realistic computer graphics applications. In the past decade, the software industry has produced great products using the global illumination techniques in their renderers. These products are the most important tools for the areas such as architectural drawing, animation films, movie special effects etc. The huge demand in this industry pushed the artists to continually improve the details of the scenes they produce. Therefore, the extreme computational cost of global illumination has been continuing to be an issue for the non-real-time rendering applications too. For example, the company Weta Digital required 35000 CPU cores and 104 terabytes of RAM for rendering the famous animation movie "Avatar" [6].

As Gonzalez-Morcillo et. al. mention, distributing the processing is a natural way to reduce the rendering time. In other words, having a rendering that takes $t$ amount of time on a single processing node, would take $t/n$ amount of time in ideal cases with $n$ identical processing nodes working in parallel on different parts of the same rendering. This kind of parallel processing settings is called Distributed Rendering Environment (DRE) [8]. Actually the DREs not only for non-real-time rendering with GI. They are also used for some real-time applications without GI. An example of the real-time non-GI applications of DREs is the flight simulators with dome shaped screens. Because the huge size of the screen, the resolution of the image that needs to be rendered in each frame is so high, several GPUs need to be utilized in parallel for this processing task. Additionally, even a multi-GPU computer combining 2 GPUs with SLI (Scalable Link Interface) technology of NVIDIA can be considered as a DRE. On the other hand, when the non-real-time DREs are considered, it is observed that the variety in the scale of computation cluster is greater than the real-time DREs.

Today the DREs are so indispensable in the area of non-real-time rendering with GI, this kind of DREs has a special name which is "Render Farm". There are various render farm forms. The main choice for middle sized organizations which has plenty of workstations for their staff is to build a centralized cluster based render farm. In this kind of render farm there is a central resource queue like famous *"DrQueue"* and all computers are in a fast local area network. The centralized resource controller holds a job queue and gives these jobs to the worker computers whenever they are idle. Secondly, there are some render farm companies. In a render farm company, idle staff computers are not used as render farm workers. Instead, some dedicated servers are built for the rendering. Unlike the previous form, this solution may be decentralized because the cluster size is extremely high and the task of controlling the resources may need to be distributed as well. Finally, there is a render farm category based on volunteer computing. A famous example for this kind is *"Renderfarm.fi"*, alternatively called *"The Publicly Distributed Rendering Service"*. In this grid based render farm approach there is a public community with volunteers connected by the Internet. To be able to use other people's resources for rendering, each volunteer must serve some free CPU resource to the community whenever his/her computer is idle. This mutual benefit relation generates an ecosystem in which each person changes the role as producer or consumer from time to time [3].

## 1.2 Scope and Objectives

In the ideal cases, the render farms are expected to decrease the render time linearly as the number of processing nodes increases. In other words, the job which is rendered with one computer in $t$ amount of time is expected to be rendered with $n$ identical computers in $t/n$ amount of time. However, in practical cases where straight-forward techniques are used, this ideal case can almost never be observed due to some bottlenecks, tradeoffs and overheads. For example, while splitting the rendering job into sub tasks[1], the granularity of the tasks results in the most crucial tradeoff due to the task submission overhead. Moreover, the increase in the number of render farm workers means increase in the communication costs, which is another obstacle to achieve the linear speedup. This thesis work concentrates mainly on the optimizations that can be applied to a render farm controller to deal with these problems. These optimizations try to overcome the load imbalance between the worker computers and the high communication cost between the workers and the render farm controller. The optimizations about the rendering algorithms that can be applied to the renderers running on the worker computers are out of the scope of this work. The techniques that will be proposed in this study is based on temporal coherence that is the similarities between the adjacent frames of an animation scene. Therefore, none of the proposed techniques in this study is applicable to single-frame still image rendering, which is also out of the scope of this study.

One of the side works of this research is to propose a thorough distributed rendering environment architecture which is suitable for an environment such as campus network. The software developed to assess the proposed algorithms of this research is also designed with this constraint. For example, the system does not require a closed and safe local area network. Moreover, both the application and the algorithms are designed to deal with hardware diversity. In other words, both a super computer and an old computer can be workers of the same render farm. Another important design consideration about the software and the algorithms is being dynamic and agile. The target environment for this research does not consist of dedicated servers with 24/7 up-time. Instead, the render farm will probably contain laboratory workstations which have variable up-time and workload. Because of this fact, the system must deal with momentary changes in the distribution of processing power.

## 1.3 Document Organization

In the rest of this document, firstly the related work about the area of distributed

[1] Throughout this document the whole file submitted to the render farm by the user will be called *"job"* and the sub parts of this job submitted to the rendering nodes by the resource controller will be called *"task"*

rendering optimizations is presented in Chapter 2. Secondly the problem addressed in this research about the render farms is explained in detail in Chapter 3. In Chapter 4 the algorithms and formulas used in this work explained in detail with the architecture of the software used. Then, Chapter 5 includes and discusses the test results of the proposed approach.

# CHAPTER 2

# RELATED WORK

In this chapter, some recent studies about the distributed rendering optimizations will be presented. When the studies in field of distributed rendering optimizations is examined, it is observed that majority of the research is conducted on real-time distributed rendering environments. The reason for this case appears to be the fact that real-time distributed rendering environments are far more sensitive to load balancing than the non-real-time DREs. As a result, balancing the workloads of the renderers in a real-time DRE provides higher performance gains than load balancing in the non-real-time DREs. The cause of extra sensitivity of the real-time distributed rendering to load balance can be explained with the example of physical simulation environment such as a flight simulator. These simulators usually provide an extremely high resolution image on a dome shaped screen. In order to generate that high resolution image in real time, distributed rendering is required. Thus, the image on the screen should be partitioned and each partition should be rendered by a separate renderer. However, if the screen is partitioned equally, the load of renderers rendering the above partitions may probably be lower than the renderers rendering the below partitions since the cost of rendering the sky is lower than rendering the terrain. In this situation the renderers of top partitions will wait for the other renderers in an idle state until the rendering of the next frame is started. In this case the closest deadline is the completion of current frame. On the other hand, in the non-real-time DRE such as animation rendering, the closest deadline is current job that consists of multiple frames. As a result, small and short term load imbalances in non-real-time DRE can be tolerated in a long rendering job whereas a real-time DRE may become completely useless if there is load imbalance. This phenomenon is illustrated in Figures 2.1, 2.2, 2.3 and 2.4.

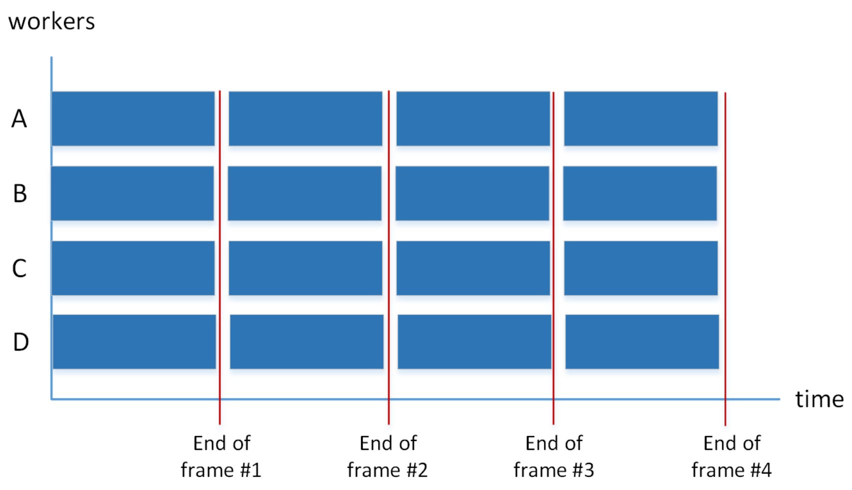


Figure 2.1: Real-Time DRE with Perfect Load Balance

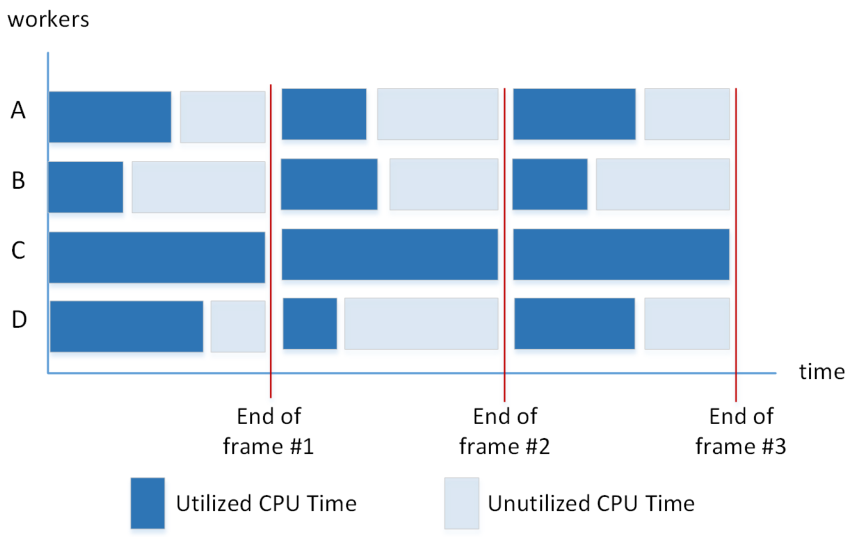


Figure 2.2: Real-Time DRE with Load Imbalance

In Figure 2.1, a real-time DRE with perfect load balance is shown. In this illustration, there are four rendering units with the same processing power. The load balance can be considered to be “perfect” since each renderer A, B, C and D is assigned to jobs with equal cost. Therefore, completion of the current frame, which is the closest deadline, is finished as soon as possible by using all renderers most efficiently. On the other hand, Figure 2.2 shows the utilization of the renderers in case of load imbalance. As a result of this load imbalance, the renderers A, B and D wait for renderer C during each frame since the load of C is higher than the others.

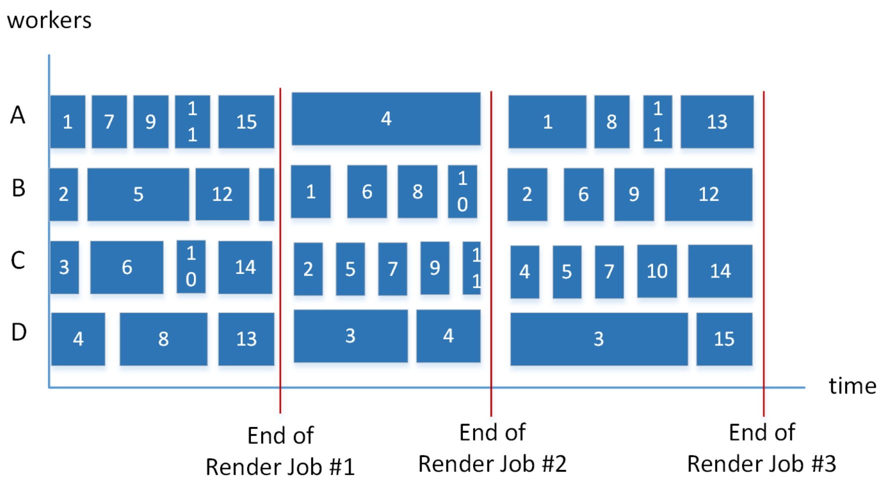


Figure 2.3: Non-Real-Time DRE with Perfect Load Balance

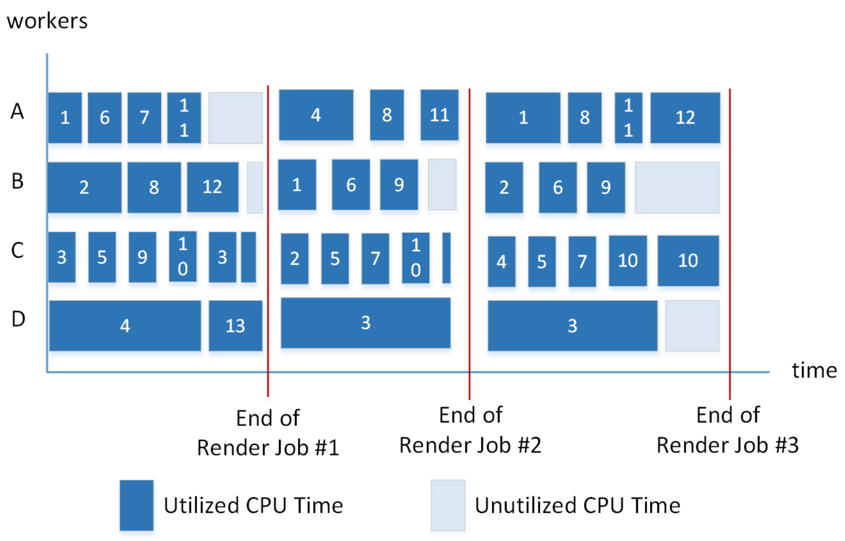


Figure 2.4: Non-Real-Time DRE with Load Imbalance

In Figure 2.3 the environment is non-real-time unlike Figure 2.1 and 2.2, where an animation having multiple frames is to be rendered. In this figure, the load balancing can be considered as perfect. Although, each frame has a different cost, each renderer finishes their assigned tasks at the same time at the end of the animation job. This is the only requirement for a non-real-time DRE since the closest deadline is the completion of whole animation instead of a single frame.

In Figure 2.4, the load imbalance is illustrated for the case that the task assignments are not planned according to the costs of tasks and the power of the renderers. For example, in Render Job #1 the costs of assigned tasks of renderers A and B are less than the costs of tasks assigned to C and D. Because of the shown unutilized CPU times, the deadline is delayed. In this figure, a renderer which finished all tasks for a job could have started to process the next job. In that case, the total completion time of the three jobs would be optimal but still the Render Job #1 would be delayed.

Although the main focus of this research is the optimizations in a non-real-time DRE like shown in Figure 2.4, some of the studies which have been conducted in the real-time DRE field will also be examined closely. These studies will be presented in the next section. Later in Section "Studies on Non-Real-Time DR Optimizations" the studies related directly to the field of this research will be presented.

## 2.1 Studies on Real-Time DR Optimizations

Most of the researchers working in the distributed rendering field draw a strict line between non-real-time and real-time DR optimizations and they conducted their research only on one side of this line. Whereas, in this research the other side of the line will also be considered although the main focus is non-real-time distributed rendering environments. The main reason for this case is that some ideas in the real-time side can be very inspiring for the non-real-time side of DR optimizations. Three important researches will be examined in this section. These papers are chosen especially since they address some problems which will be examined closely in Section 3. Namely these problems are, dividing the whole job so that the number of sub-tasks is equal to the number of renderers and acquiring sub-tasks with costs proportional to the power of the renderer.

According to Molnar et. al. [13] the rendering pipelines can be divided into three categories which are sort-first parallel rendering systems, sort-middle parallel rendering systems and sort-last parallel rendering systems. As mentioned in Section 1.1, because the computational cost of the global illumination techniques is extremely high, the real-time distributed rendering is mostly implemented with the "Rasterization" rendering technique which has significantly lower cost when compared to GI. Indeed, the three rendering pipe-line category stated above are all related to the Rasterization rendering technique. As can be understood from the names, the Rasterization problem contains some sorting phase, which is on the objects in the scene. This sorting process is necessary to determine which object is visible in the final image in case one object hides some other. The moment that this phase occurs is critical for DR because it is main dependency point of independent rendering nodes. As a result, the way of handling this problem affects the DR performance significantly.

In their studies Molnar et. al. [13] analyzed each of three approaches and tried to discover some tradeoffs. In addition, they propose a framework to decide which approach to use according to the needs. The study of Molnar et. al. [13] explains the three rendering pipeline category as follows:

**Sort-Last:** In this approach all rendering nodes renders the full frame with different subsets of the geometric objects in the scene. This approach requires a complex process in the last sorting phase. This last phase is a kind of blending and it is relatively hard to implement. According to Molnar et. al. [13], the advantage of this approach is that the renderers are independent until the pixel merging phase, which is the last phase, and it is less prone to load imbalance. Its disadvantage is that it creates very high traffic between the renderers especially when oversampling is used [13].

**Sort-Middle:** The sort-middle approach is also a complex approach. At the beginning of the process the primitives are converted into 2D screen coordinates according to their geometry and position. Then, the primitives are distributed to the renderers according to their coordinates on the screen. Therefore, instead of rendering a full frame image, each renderer can work on a sub-set of the frame which contains the primitives that are assigned to that renderer. The major disadvantage of this approach is that it is susceptible to load imbalance between renderers when primitives are distributed unevenly over the screen [13].

**Sort-First:** In this approach one frame is split into several responsibility areas before the Rasterization and each area is assigned to a renderer. Unlike the sort-last or the sort-middle approaches, the scene content is not divided at all and it is given to each renderer without any change. In other words, each renderer is aware of each primitive in the scene. The fact that all renderers are aware of each primitive in the scene does not cause a significant performance drop since the primitives outside the view frustum will be ignored. The biggest advantage about this approach is that it creates very little network traffic since the renderers do not need to switch information during the rendering. Unfortunately, this approach is highly susceptible to load imbalance [13].

Among these three approaches, the sort-first can be considered as the most important one. It is a better choice for distributed rendering since it generates lower communication traffic. To prevent bottlenecks caused by insufficient network bandwidth, this advantage of sort-first approach is crucial when there is high number of renderers. The sort-first technique is also important for non-real-time DR. The distributed rendering techniques used in non-real-time DR show great resemblance to the sort-first technique. The load balancing techniques in this area may provide a good start point for load balancing of non-real-time DR.

To solve the load balancing problem of the sort-first approach a dynamic method is proposed by Ji and He [12]. In their method the cost of a sub-task (i.e. tile in the screen) is calculated as the sum of the load of geometric transformations and the load of rasterization. They described it with the following formula:

$$L_{rendering} = N_{vert} \cdot t_v + A_{frag} \cdot t_f$$

In this formula $L_{rendering}$ is the predicted rendering load of a partition in frame. $N_{vert}$ and $A_{frag}$ is the number of vertices in the scene and the number of pixels in the final image respectively. The $t_v$ and $t_f$ values are the predefined unit times required for processing one vertex and one pixel in the geometric transformation and rasterization phases respectively.

The proposed approach in the study of Ji and He partitions the frame by creating a kd-tree. While creating the kd-tree, cost of each kd-tree node is calculated by the above formula. By an iterative method the kd-tree is refined until the aimed load balance precision is achieved. Lately, these partitions are assigned to the renderers as part of the sort-first approach [12].

Another important research in real-time DRE field that may be inspiring for the non-real-time DRE belongs to Abraham et. al. [1]. The preferred rendering technique in this paper is again sort-first. Abraham et. al. [1] stated that they propose a very simple to implement load balancing algorithm which provides good load balancing. Moreover, the cost of the algorithm is negligible when compared to the cost of rendering. The success of the algorithm comes from usage of temporal coherence which is one of the most important ideas in this field. Temporal coherence uses the fact that in movie contents, the objects in the scene manifest a continuous movement except for the scene and camera changes. This fact provides the predictability of the next frame by the previous frame. Of course there is the possibility of two completely unrelated consecutive frames but the probability of this is very low because of the high frame rates around 30 fps. In other words, there will be probably so many coherent frames until a scene or camera change. The approach of Abraham et. al. [1] uses this coherence to determine the load distribution of a frame. To be precise, the algorithm starts with a blindly created tiling. Then the system renders it and the algorithm collects the run times of each sub-task. Actually this run time data gives information about the load distribution of the last rendered frame whereas the problem is to find the load distribution of the next frame. Although the run times of last rendered frame can just be an approximation for the next frame, because of the temporal coherence, the research states that the algorithm provides a high degree of load balance. When compared to the approach of Ji and He [12] this approach gives slightly less load balancing precision with gain of performance. The performance is higher in this approach because it uses each frame as an iteration step and executes only one load balancing per frame whereas the approach of Ji and He [12] continues to iterate to refine the load balancing until a threshold is reached for each frame.

In their study Abraham et. al. [1] also proposed a smart method for task-renderer matching. The solution to this problem can also be very important in the non-real-time distributed rendering environments. The importance of this method is that it changes a disadvantage into an advantage. As seen in Figure 2.2 the imperfections in the load distribution causes unutilized CPU or GPU resources. In the load balancing method of Abraham et. al. [1] there will always be some amount of imperfections in load balancing, which is inevitable. Therefore, instead of fighting that fact, the renderer which finished its task earliest starts rendering the most costly tile of the next frame. At first glance this seems to be meaningless since it does not solve the delay in the closest deadline. However it actually solves the problem by just changing the sharp boundaries into smooth transitions at the start and finish of frames.

## 2.2 Studies on Non-Real-Time DR Optimizations

In this section, important studies on non-real-time distributed environments will be presented. As mentioned in Section 2.1, these studies also remain on one side of the boundary between non-real-time DRE and real-time DRE. On the non-real-time side of that boundary, the studies are examined under four categories which are Render Farm Frameworks, Communication Optimizations, Cost Prediction, Task Distribution Optimizations.

### 2.2.1 Render Farm Frameworks

One of the most comprehensive render farm frameworks belongs to Patoli et. al. [16]. They defined 11 constrains for render farms. These constrains are “Platform Independent”, “Grid Based”, “Registry Support”, “Freeware”, “Fully Automatic”, “Workflow Support”, “Job Monitoring”, “Animation Support”, “Open Source”, “Script Generator” and “Service Oriented”. The paper also contains a table which compares all well-known render farm solutions with the proposed framework of Patoli et. al. According to this table, the presented framework in the paper is claimed to satisfy all of these constraints, whereas the closest and one of the famous render farm solutions “DrQueue” can satisfy only six of these constraints. The main contribution of this study is that it shows that all these render farm features can be provided at the same time. In order to provide grid based distributed rendering, the framework uses Condor, which is a general purpose and advanced tool for distributed computing developed by the University of Wisconsin-Madison. In order to provide platform independence, the preferred 3D software in the study is open source Blender but by the “Script Generator” feature theoretically all 3D software can be integrated to this framework.

Gooding et. al. also proposed a render farm framework [11]. They propose a comprehensive design for a render farm that is built upon the TerraGrid network. TerraGrid is an extremely big grid computing architecture contributed mainly by universities. It is mainly used for very costly scientific calculations. Another contribution of this study is that they open the way of very new possibilities in this area by founding this massive render farm. Gooding et. al. claimed that their proposed render farm framework solves many software problems which can be encountered by any software developer while developing a render farm controller software [11]. First one is the problem about the communication of controller and the farm worker computers. To solve this problem, Gooding et. al. also used Condor. The second problem solved by this study is the security of the artifacts. To deal with the security issue, a submission process is designed having authentication and encryption. Since the framework in this study contains a comprehensive design even the procedures for job name creation or asset relocation are presented.

### 2.2.2 Communication Optimizations

One of the biggest problems about the render farms is the limitation of the network bandwidth. Because of this problem, most of the render farms are cluster based and built on a local area network. Considering the bandwidth of local area networks may even be insufficient, building a render farm across WAN requires some special techniques. In their paper Cao et. al. [20] propose some techniques to deal with the communication cost problem and build a render farm across WAN. Their approach is based on using multi-level controller nodes hierarchically. In other words, the workers in this farm are grouped in an efficient way to both overcome communication bottlenecks and increase scalability. In a naive communication schema for render farms, there are usually a controller and some number of workers all connected to the controller directly. Assuming the number of the workers is $n$, the communication load of the controller will be $n$ time higher than the render farm workers. To solve this issue, Cao et. al. [20] propose to group the clusters and pick some sub controller in each group. In this schema the main controller only communicates with the sub controllers directly. The sub controllers only communicate with their workers in its cluster. Unfortunately, this schema also has a drawback. Since the communication between the workers and the main controller is provided by the sub controllers, all workers go offline in case of sub-controller failure. Therefore, each cluster must be able to entitle another machine as sub-controller automatically in case of sub-controller failure.

The study of Chong et. al. [5] is one of the studies which propose a framework for distributed rendering. However, they provide a communication optimization algorithm also. The main contribution of this study is that it proposes a novel approach to solve the problem of high network traffic caused by the huge input files (scene definition files). The framework handles this problem by a special lossless 3D compression method. This novel 3D compression method mainly focuses on the scene definition files. They mention that most of the size in these files is occupied by 3D objects which generally are duplicated because of the temporal coherence. The technique of Chong et. al. [5] overcomes this duplication by hashing the geometric objects. Naturally the hash value of an object will be very small compared to the object itself. Therefore, by sending only one copy of the object and the hash values whenever necessary reduces the network traffic significantly. According to the test results presented in the paper of Chong et. al. [5] this technique can achieve compression ratios up to 97%.

### 2.2.3 Cost Prediction

One of the papers proposing a method for predicting the global illumination rendering cost of a scene is by Reinhard et. al. [17]. Their proposed approach is based on the knowledge that the cost of one traced ray in global illumination increases on each ray-object intersection. Therefore, Reinhard et. al. [17] claim that the cost of the scene converges to the average tree depth of the spatial subdivision. However, at the ray-object intersections some amount of light passes through for transparent objects. Therefore, the formula they use for cost prediction also takes the average amount of surface transparency into account.

Another important study about the cost prediction problem is by Gillibrand et. al. [7]. Unlike Reinhard et. al. [17] Gillibrand et. al. propose to calculate the cost by building spatial subdivision tree and using its average depth and surface transparency. The proposed approach of Gillibrand et. al. profiles some pixels in the scene by tracing a ray from each of these pixels like real ray tracing algorithm. However, to achieve a run time which is significantly less than rendering the scene, the algorithm carries out ray tracing for very small subset of all pixels. In the first phase of the algorithm, scene is rasterized like a preview of the geometric objects. In the result of this rasterization phase, they change the colors of the materials according to their material properties. For example, red for transparent materials, blue for glossy and specular materials and green for diffuse materials. After acquiring this snapshot of the scene their algorithm has knowledge about both material and complexity distribution of the scene. In the next stage the algorithm decides the positions of the sample lights. Having the complexity map, the algorithm puts fewer samples on the simple objects such as walls and puts more samples on the complex specular objects. After that process the algorithm profiles the rays. The main contribution of this study is that the results converge to the real cost quickly because the algorithm calculates more samples around the complex objects.

### 2.2.4 Task Distribution Optimizations

The study of Abramson et. al. [2] is very important since it proposes a solution to the problem of fairly distributing the shared resources to the users on a grid computing environment. This problem becomes even harder if the users are clients of a company providing rendering service because the billing model of a company like that may have the possibility of getting more resource by paying more. To solve this problem Abramson et. al. propose the use of Nimrod-G which is a resource broker framework. The main parameters for this application are the aimed deadline for the job, budget, and the choice of minimizing the cost or time. This framework simulates the stock market for a render farm. In other words, high demand results in cost increase. Each user of this render farm declares a budget which is the maximum possible payment planned for a rendering job. Then user selects the choice of time or cost minimization together with the planned due date for the job. If the user selects to minimize the cost, the system allocates the resources to this user whenever possible but retracts the resources when there is a customer who is selecting the time minimization choice and paying more for the resources. This scheduler tries to provide a fair resource allocation by giving service to each user proportional to their payments and by considering the final due dates declared by the users.

Another important study in this field belongs to González-Morcillo et. al. [9]. This study uses a multi-agent approach for task distribution optimizations. They call this *"MAgarRO"* (standing for "Multi-Agent Approach to Rendering Optimization"). This approach is also grid based and suitable to work with the free CPU cycles in a campus network. In the paper, "MAgarRO" is claimed to have the following features:

**Decentralized control:** In MAgarRO, all render farm workers are not controlled by a single controller. Instead, the system contains multiple clusters called agents which show autonomous behavior. In other words, all these agents have a local controller and make decisions by themselves. In addition there is also a main controller to which the agents report and communicate.

**Higher level of abstraction:** The MAgarRO is claimed to be suitable to work with all renderers by defining the input parameters.

**Use of expert knowledge:** This feature is one of the most important features of MAgarRO. According to González-Morcillo et. al. [9], the render farms are not always used in an optimized way because of the wrong render parameters. In other words, inexperienced users may increase the quality unnecessarily and that results in excessive rendering times with very small quality gains. To prevent this, the MAgarRO is claimed to optimize the render parameters by some fuzzy set rules called expert knowledge.

**Local optimization:** By the autonomous behavior of multi-agent approach each agent can build its own expert knowledge according to its hardware configuration.

The MAgarRO contains several phases before the real rendering phase. By applying these phases the system collects information about the tasks and distributes them to the agents according to the power of each agent. Meanwhile, the processing powers of the agents are also updated. To be precise, the procedure consists of the following phases. In the first phase an agent can be added to the system. When an agent is connected for the first time, a benchmarking job runs on it to assess its processing power. To achieve load balanced task distribution, MAgarRO tries to divide the frames into zones by detecting the parts which are more costly. To do that it generates a special image called "Importance Map". This approach also uses fast rasterization to generate the importance maps like the technique used in the study of Molnar et. al. [13]. After the generation of importance maps the resulting image is partitioned into zones that have roughly the same complexity. In the next stage the master of the render farm distributes the sub tasks of the rendering job to the agents. Then the agents profile the zones by just rendering the 5% resolution of each zone. By doing that, the system acquires more precise cost estimation about each zone. After the accurate cost estimation is acquired, a communication service called "Blackboard" is used so that the agents share information with each other. Namely, each agent registers the cost estimation of the assigned task together with some properties of the zone such as its resolution etc. After this process a phase called "Auctioning" is carried out. In Auctioning phase all idle agents of the system compete for the available unfinished tasks and the master distributes the zones to agents. As a final phase before rendering, the expert knowledge is applied to the zones. In other words, after the task submissions each agent uses its fuzzy rule sets to model the expert knowledge to determine the optimal rendering parameters. The main parameters in this procedure are the recursion level, the light samples and the interpolation band size. The agents try to find an optimal value for these parameters to prevent unnecessary quality for the unimportant areas on image. For example, for the simple objects such as walls there is no need for a high number of light samples whereas for the complex specular objects the light samples should be increased. After all these optimizations, the agents render the zones fully. However, at this point a problem emerges because agents used different render parameters for each zone. After connecting the zones, some difference in tone and detail may become visible at the connection line of each zone. To solve this problem, González-Morcillo et. al. [9] used an interpolation technique. The MAgarRO determines an interpolation band size according to the difference in the render parameters between two zones. Then, a linear interpolation is applied to provide a smooth transition between the zones.

Although the proposed approach of González-Morcillo et. al. [9] contains a good task distribution optimization, it does not take granularity of the tasks into account. On the other hand, modeling the expert knowledge seems to solve one of the biggest problems in this area. Unfortunately, the test results in the study is not suitable for distinguishing how much of the performance gain is because of the expert knowledge and how much of it is because of the task distribution optimizations. One of the most important aspect of the results of this study is that it shows the importance of the task granularity. In other words, the sub-task rendering time does not decrease linearly while the number of agents increases. This phenomenon is illustrated in Figure 3.3.

The main reason for this non-linearity is the per-task overhead. Namely, for execution of each task the renderer carries out some initialization before rendering. In addition, the communication overhead per pixel also increases. This problem will be explained in Chapter 3 more detailed.

The main problem of MAgarRO proposed by González-Morcillo et. al. [9] is that it analyzes the cost distribution in each frame and the tasks of a rendering job are parts of a frame. Although, this high granularity is inevitable for still-image rendering jobs, for multi-frame animation jobs this high granularity results in a high overhead. Therefore, for animations the cost distribution should be analyzed in frame level instead of analyzing in sub-frame level.

# CHAPTER 3

# THE PROBLEM STATEMENT

Because of the high demand in the animation industry in the past decade the detail and the complexity of the scenes created by the artists have been increasing continuously. Although, the hardware technologies improved significantly in the meanwhile, this improvement was never enough to make a single computer fast enough to render even a small animation by itself considering the tight due dates of the animation companies. Because of this high computation cost, for now the only practical solution in this area is to use large number of computers in parallel like workers in a farm.

## 3.1 Communication Cost Problem

In ideal cases, render farms are expected to decrease the time required to render linearly while the number of rendering nodes in it increases. In other words, the job which is rendered with one computer in $t$ amount of time is expected to be rendered with $n$ identical computers in $t/n$ amount of time. However, in practical cases with straight-forward approaches this linear speedup can almost never be observed due to some bottlenecks, tradeoffs and overheads. An example of the bottlenecks is the case of centralized approach for large render farms [8]. In this case, the render farm is so big that one computer is not enough to handle all the controlling tasks such as providing inputs, collecting outputs, accounting, aliveness checks etc. This results in low scalability. Another bottleneck example is about the networking. Network bottlenecks occur with the grid based render farms connected with some narrow band network connection. In this case, the wait for the download operations of the input and the upload operation of the output can decrease the render farm performance [5]. This performance drop can be very significant if the scene input file contains so many high resolution textures or the resolution of the produced output is very high. This is illustrated in Figure 3.1 and Figure 3.2.

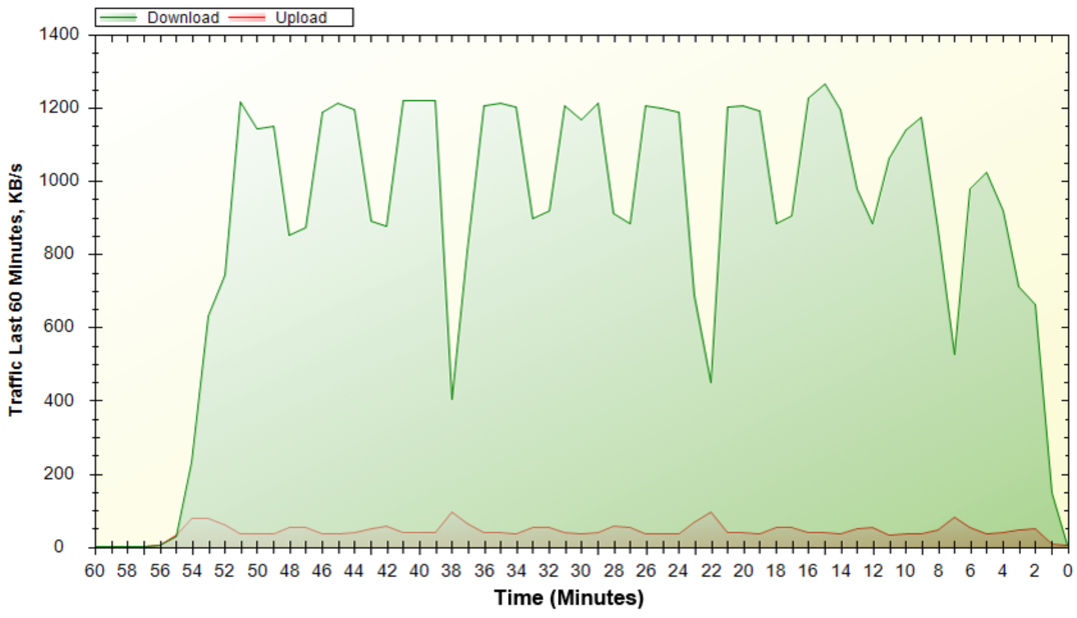


Figure 3.1: Network Bandwidth Usage History of Render Farm Controller

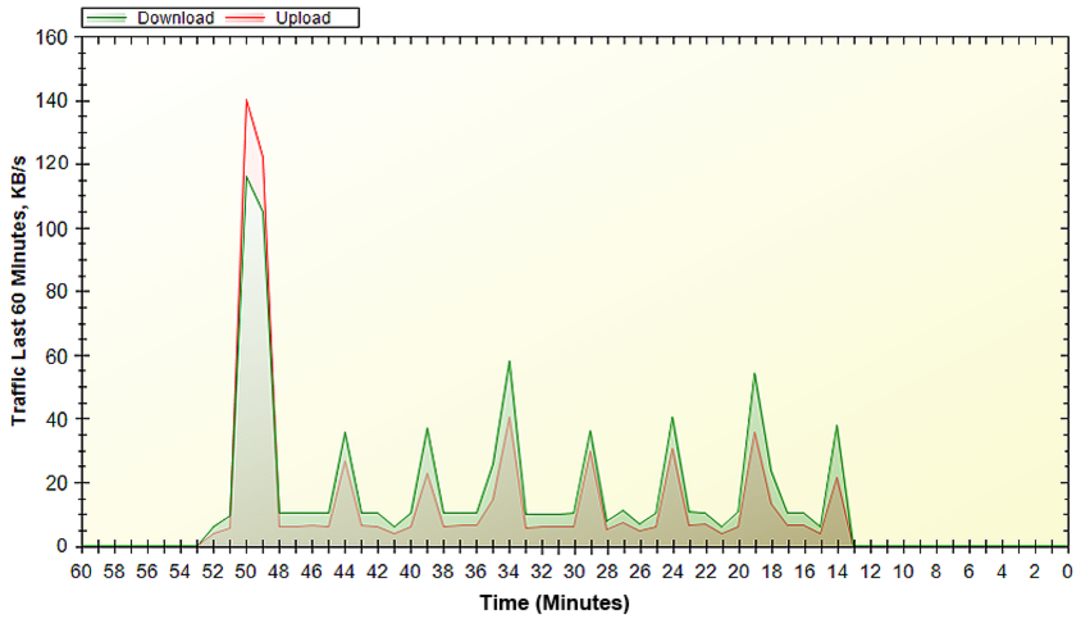


Figure 3.2: Network Bandwidth Usage History of a Render Farm Worker

Figure 3.1 and Figure 3.2 show the network load measurements for a render farm with 20 workers and one controller connected with a 802.11g Wi-Fi network. Figure 3.1 shows the network traffic history of the render farm controller and Figure 3.2 shows the network traffic history of one of the render farm workers. A rendering job having 1000 frames with JPEG format and 1920x1080 resolution is submitted to the render farm for this test. The average file size of each image file is 1.2 megabytes and the average rendering time is measured as 5 minutes. The periodic peaks which can be seen in Figure 3.2 correspond to the uploading of output frames to the render farm controller. While the network adapter of the render farm worker only deals with this one file in approximately five minutes, the network adapter of render farm controller has to deal with 20 times more data at the same time. Thus, the load on the network adapter of render farm controller is 20 times higher as can be seen in Figure 3.1. As the worker count increases or the output file resolution increases this unbalanced load turns into a significant bottleneck in the communication. Therefore, the network costs of the operations between the render farm workers and the controller should be optimized as much as possible. The study of Chong et. al. [5] can be a good example for this kind of optimizations. Although, decreasing the network costs is not the ultimate solution for the problem of decreased scalability due to network load imbalance, it may still improve the system performance.

Moreover, render farms do not always have a dedicated network. Even if the bandwidth of the network is enough for the operation of render farm, the same network may be in use by some other users. A render farm built in a campus network can be a good example for this situation. Campus networks usually have high bandwidths, which is enough for the render farm but during the operation of the render farm the personal users of the network may suffer from some latencies that decrease the usability of interactive applications such as web browsers. For this reason too, the network usage of the render farms should be minimized as much as possible.

## 3.2 Computational Load Imbalance Problem

Besides the above problems there is another issue about splitting a rendering job into sub-tasks and submitting these sub-tasks to rendering nodes. As mentioned in the previous chapter, the granularity of the tasks results in a crucial tradeoff due to the overheads occurring during the task submission. Knowing the costs of different parts of the job while splitting job into sub-tasks and matching sub-tasks with the processing nodes according to their capacity is a major optimization to deal with the load imbalance between the render farm worker computers.

To explain this problem, the stages of submitting a rendering job to a render farm should be explained. In all render farms, the process starts with the submission of the input file. This file can be placed in a shared network location if the whole system and the user are in a local area network or the file can be uploaded by a portal application [5]. After the server acquires the input file then comes the sub-task submission phase. In this phase, firstly the render farm controller splits the whole job into sub-tasks. After that, the render farm controller could submit each sub-task to a rendering node or the rendering nodes could check the system to detect if there is any unfinished task. If there is any unfinished task, the workers may pick a task by themselves. The first alternative which gives full control to the render farm controller is called “Data Driven Approach" [10]. This second approach is called “Demand Driven Approach" [10]. Both of these approaches have different advantages and disadvantages. To gain insight knowledge about the problem, these two approaches can be examined closely.

The demand driven approach has some advantages. Firstly, it guaranties full CPU utilization in all conditions because the render farm workers are always in a loop of “render, pick new task, render, pick new task, render ...”. As long as there are some unfinished tasks in the task pool of the render farm, the workers will continue to work with full CPU utilization and this means the system would be fully utilized. This idea may be correct theoretically but not practically. If it was correct in all cases, this thesis work would not exist since by using this approach there would be no need for further optimization. The problem starts with the fact that this approach works great only with fine-grained task splitting. This is illustrated in Figure 3.3 and Figure 3.4.

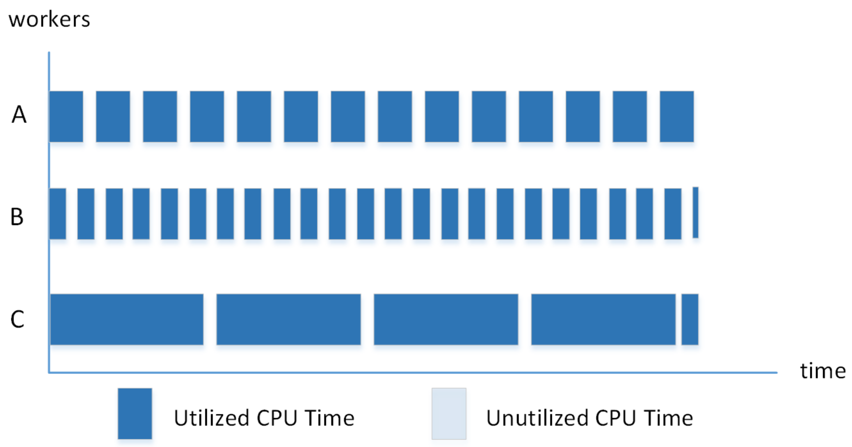


Figure 3.3: Fine Grained Task Splitting

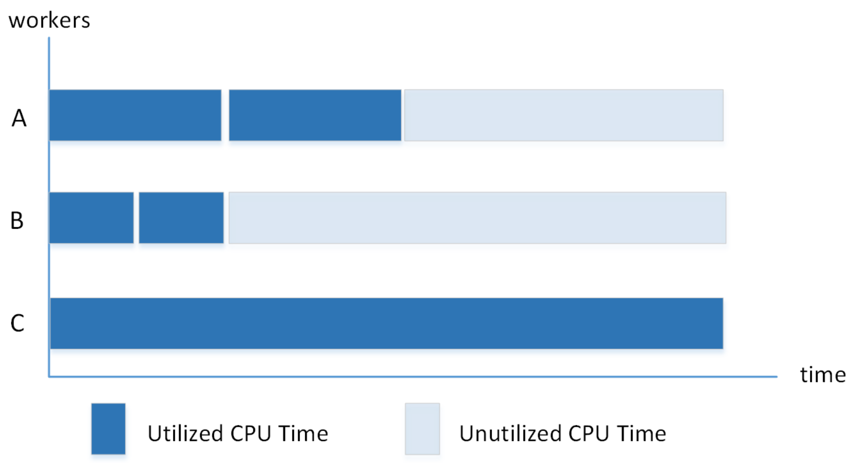


Figure 3.4: Coarse Grained Task Splitting

The figures illustrate a heterogeneous render farm with three rendering nodes (A, B, C). The processing power (i.e. speed) of A, B and C are $4p$, $8p$ and $p$ respectively (with the assumption that $p$ is a unit for processing power of a computer). Each rectangle represents the processing of one sub-task by a rendering node. In Figure 3.3 all tasks have equal cost which is $c$ (with the assumption of $c$ is unit for rendering task cost). In Figure 3.4 the sub-tasks have also equal costs but this time the cost is $8c$. In other words, in Figure 3.3 the rendering job are split into sub-tasks with 8 time higher granularity. The obvious problem in Figure 3.4 is the unutilized CPU times of A and B because of the unbalanced workload. Actually, if there were plenty of rendering jobs in the render farm and the only performance criteria was the overall elapsed time for all jobs, the case of Figure 3.4 could be considered as optimal, because the time durations marked as "Unutilized" in the figure would be used for the tasks of next job. However, if there is only one job or if each job belongs to a different user, per-job rendering completion times are very important. Per-job rendering completion times grow significantly for the case of Figure 3.4. Therefore, if the render farm controller does not have any control over matching tasks and workers (Demand Driven Approach), the system must definitely work with fine grained tasks.

At first glance, the fine grained task division seems to be a good solution. Unfortunately, this solution brings another problem which is task submission overhead. As can be seen from Figure 3.3 and Figure 3.4 there is a gap between each task execution. These gaps denotes the initialization time of the rendering software at the time of task submission. 3D editing software such as 3DsMax[1] or Blender accept the batch rendering task through a command-line prompt. This command contains various rendering settings with the path to job file and frame numbers to render. An example rendering command for 3DsMax is given in Appendix A. Each time this command is invoked the computer starts a 3DsMax process and loads some significant number of DLLs. Then the plug-in loading phase comes. There are plenty of plug-ins distributed with 3DsMax. In addition, there are also plenty of third party plug-ins available. For example, a plug-in for hair and fur movement, a plug-in for Physics calculations, a plug-in for procedural terrain generation etc. Even VRay, one of the most preferred GI renderer, is a plug-in for 3DsMax. Before starting to process a scene, all installed plug-ins are loaded and initialized. This causes an important amount of time. In addition, some 3D editing software such as 3DsMax does not have any built-in renderer. Instead, the renderers can be installed in 3D editing software as plug-ins. Both MentalRay and VRay accepts the models, lights or materials in their own data format. Thus, they translate the contents of the scene to their format before processing. This also requires some important amount of time depending on the complexity and size of the scene. In Table 3.1 the time spent for each phase according to the sample run in Appendix A is presented.

[1] 3DsMax is the commercial product of the software company AutoDesk. Although there is a powerful open source alternative named Blender exists, the 3DsMax dominates the market significantly. Since available and suitable testing scenes etc. are limited for Blender, throughout this research mainly 3DsMax is used while development of software and testing of the algorithms.

Table 3.1: Execution Phases and Time Durations of 3Ds-Max

| **Phase** | **Time (seconds)** |
|---|---|
| Startup of process “3dsmaxcmd.exe” | 20 |
| Loading of the plug-ins | 3 |
| Initialization of the renderer | 1 |
| Loading and translation of the scene | 5 |
| Rendering | 80 |
| Clean up and finalization | 11 |

Time spent for each phase of sample 3DsMax run presented in Appendix A

Table 3.1 shows that for the sample run presented in Appendix A approximately 30% of the total time is spent for processing other than rendering. Even though this ratio depends highly on the output size, hardware configuration, software configuration and the scene contents, it is still obvious that most of the times this overhead will be far from being negligible. Considering this overhead, there is a crucial tradeoff in granularity of task division. Although finest grained task division gives the best load balancing, it causes a significant increase in render time, because the task submission cost occurs over and over again for each small task.

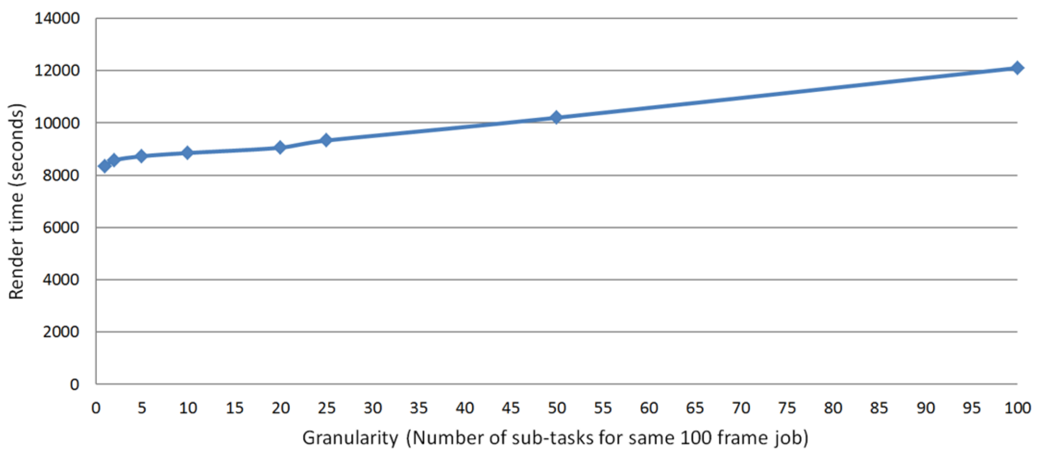


Figure 3.5: The Effect of Task Granularity to Job Completion Time

In Figure 3.5 the effect of overheads mentioned above can be observed. The test environment and the input is the same as the sample 3DsMax run given in Appendix A. The only differences are the start frame, end frame and every nth frame parameters. The given data set is acquired by rendering the first 100 frame of the job with different number of sub tasks on one render farm worker computer given in Appendix A. In other words, the 1 point on the horizontal axis shows the rendering time of job submitted to the renderer as a whole and the 50 point on the horizontal axis shows the rendering time of the same job with 50 separate sub tasks each having 2 frames. As the sub task granularity increases the job completion time increases linearly due to the time cost of the task submission. In this test setup this cost is approximately 35 seconds and if there is 50 separate sub tasks, the total non-rendering cost is approximately 1750 seconds. This phenomenon results in the requirement of decreasing the task granularity as much as possible.

After examining the Demand Driven Approach the need of control over task distribution becomes obvious. To deal with the tradeoff between granularity and initialization time, using Data Driven Approach seems to be indispensable. Namely, some smart decisions have to be made while dividing jobs into tasks and submitting tasks to the workers. To be more precise, the sub-problems which should be solved by the proposed algorithm are:

1. **Discovery of the cost map of job**: Since there is a huge overhead of the fine grained task division, task splitting must be coarse grained as much as possible. Otherwise the situation showed in Figure 3.4 would occur in this approach too. However, this brings the requirement of splitting the job according to the capacities of the workers available in the render farm at that moment [9]. To handle this requirement Reinhard et. al. proposed a technique [17]. However, the method of Reinhard et. al. is based on the knowledge about the geometries and the lights in the scene. Thus this method is not applicable for this research because almost all commercial software such as 3DsMax works with a proprietary file format which cannot be parsed outside of the application. In addition, even if there are alternatives such as Blender which has an open file format, dis-

covering the contents of the scene file is also a costly job. Consequently, a new approach is required to discover the cost distribution across to the rendering job without knowing the geometries and lights in the scene. In addition, it should be done with a smaller cost.

2. **Quantization of the processing power of workers in the farm**: As stated before, the Data Driven Approach with the coarse grained task division will be preferred in the proposed approach of this research. However, as the granularity of the tasks decrease, the risk of unbalanced workload increases as shown in Figure 3.4. To prevent this, the system must distribute the tasks to the workers in an intelligent way so that the completion time of the tasks of each worker is close as much as possible to improve utilization. Therefore, each worker should be assigned to a task having a cost proportional to the workers relative processing power. This brings the requirement of knowing the processing power of each worker in the farm.

   Moreover, the worker assessment methods should also be dynamic. Assessing a worker when it connects to the system for the first time might be an easy solution but the result of a non-dynamic assessment like this may be invalid after some time. Because there are possible cases like change in the software configuration of render farm worker, malicious software, a stuffed page file or registry etc. Each of these cases may decrease the performance of a computer over time. Therefore, the relative processing power point of each worker should be renewed dynamically without interrupting the execution of the system significantly.

3. **Dynamic assessment of the environment**: As mentioned in Section 1.2 the target environment for this research does not consists of dedicated servers having 24/7 up-time. Instead, it will probably contain staff workstations and laboratory computers that have variable up-times and workloads. Because of this fact, the system must deal with the momentary changes in the distribution of processing power among the workers of the render farm. For example, after submitting tasks of a job to the worker computers, a power loss may occur in some location where the most powerful computers reside. It is obvious that this invalidates the previous task distribution plan completely. In case of failure or

power loss for a worker computer, the assigned rendering tasks of this worker should be retract and assigned to other suitable workers. The opposite of this is also possible. In other words, when a powerful worker computer is connected to the system in the middle of a rendering job, it might be a good action to cancel the current task distribution plan and recreating it by including the new worker.

In Chapter 4, solutions to these problems are proposed by giving algorithms, detailed explanations and software architecture.

# CHAPTER 4

# PROPOSED APPROACH

The core of the proposed approach of this work is the task distribution algorithm together with render farm worker assessment techniques. To test these techniques a thorough software structure is necessary. Therefore, a piece of distributed computing software has been developed that satisfy every basic need of a render farm and the proposed algorithm is integrated in this software. Although the main concentration of this chapter is the algorithms explained in Section 4.3 and Section 4.4, the design of the developed software and the configuration of the hardware are explained in Section 4.1 and Section 4.2 to provide a deeper insight to the system.

## 4.1 Hardware Architecture

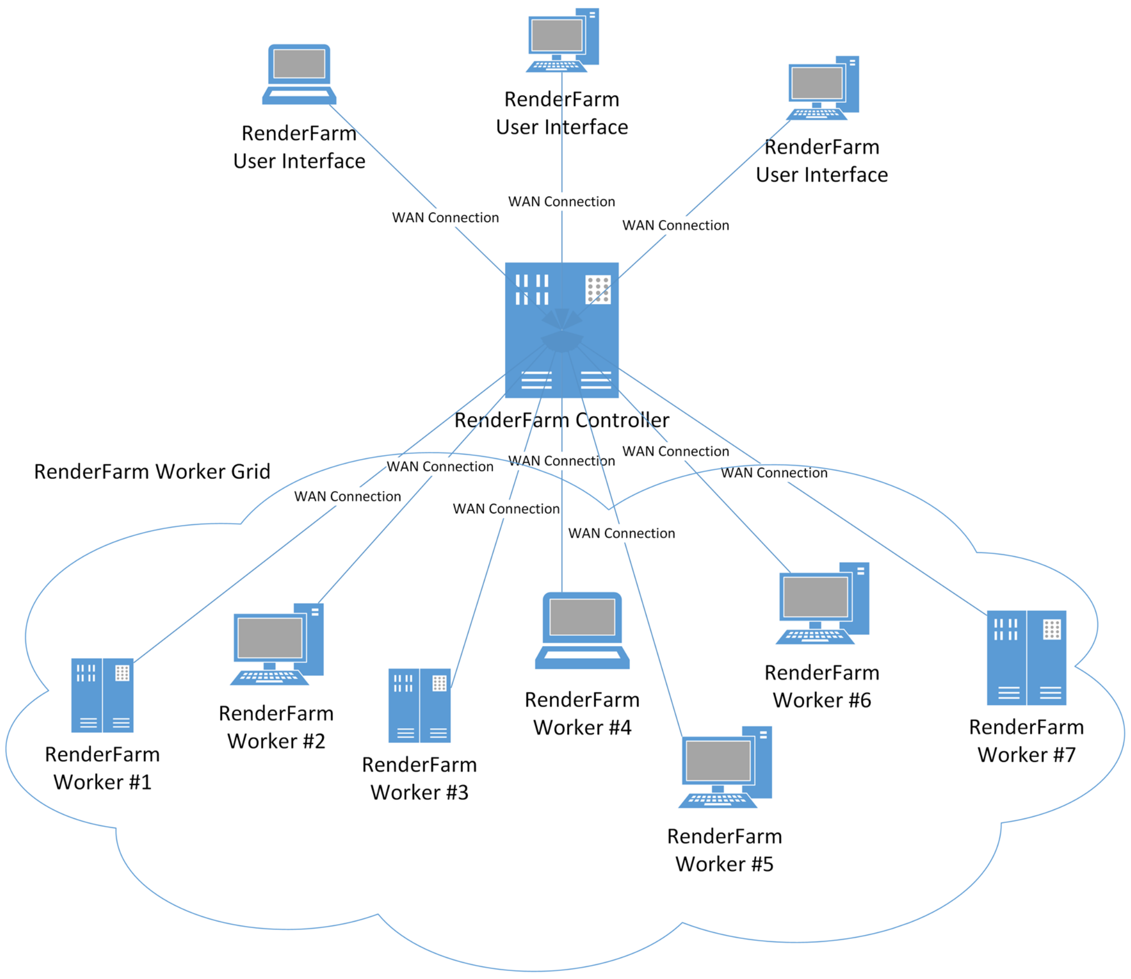


Figure 4.1: Hardware Configuration

In this research all tests are done with the hardware resources of the Modeling and Simulation Research and Development Center (MODSIMMER). The aimed system must work in campus network setting because in a campus network there is usually large number of relatively new computer which are not under load for 24/7. In other words, they are always on but not always used by students or staff. They are usually idle in weekends or at nights. This situation is very good for building a render farm. One of the secondary goals of this research is to utilize this wasted processing power of the campus network to perform some beneficial jobs. Decision of using the resources in a campus network settles the majority of the design choices about the hardware architecture.

Firstly, the resources will be divergent and heterogeneous. It is divergent, because different computers with different performance may be utilizable in different times. Let say there is a supercomputer and an Internet lab in a campus. In one moment all computers of the Internet lab may be available and the super computer may be in maintenance mode. In some other time the supercomputer may be available but all computers of the Internet lab may be occupied. Considering the processing power of the supercomputer is far more than the computers in the Internet lab, the total processing power available in the system and the distribution of it may change any moment by environmental factors.

Secondly, because the campus computer network is the target hardware configuration, the network of the system will not be a closed local area network. Instead, it is an open and unsecured network. Therefore, the system must utilize some communication security tools. In addition, using the wide area network brings new possibilities such as using resources from very distant locations. Although, most of the network lines in a campus is very fast, this system does not have to be used with campus network only. It can also be used in a grid computing setting with narrow-band communication lines. Therefore, the communication overhead should be minimized.

## 4.2 Software Architecture

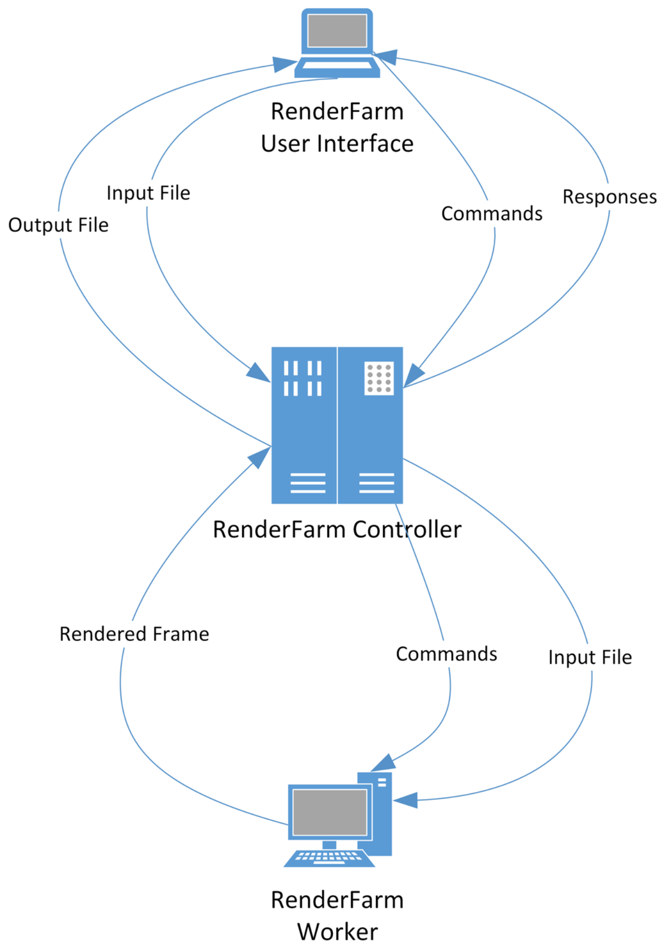


Figure 4.2: Software Configuration

After deciding the hardware architecture some important decisions have been made about the software architecture. The first thing known about the software is that it will be a piece of distributed computing software. Therefore, it needs a software unit on each worker computer to accept commands with inputs to process. This software unit is “RenderFarm Worker” which can be seen in Figure 4.2. Secondly, the render farm should have the ability to distribute inputs, execute some commands on inputs, and collect the outputs from the workers. This results in the need of a central controller which is named as “RenderFarm Controller”. Moreover, the user of the system is both the source of the input and the destination of the output. Therefore, the software should also provide a user interface for job submission and output retrieval. In addition the user should also be able to do some extra operations such as cancelling a job, restarting a job, observing the progress of a job or even previewing the output of an unfinished job. By considering these needs, there should be a third software unit which is the “RenderFarm User Interface” as can be seen in Figure 4.2. In the below parts the design of these three software units are explained in detail and a very basic communication sequence can be observed in Figure 4.3.

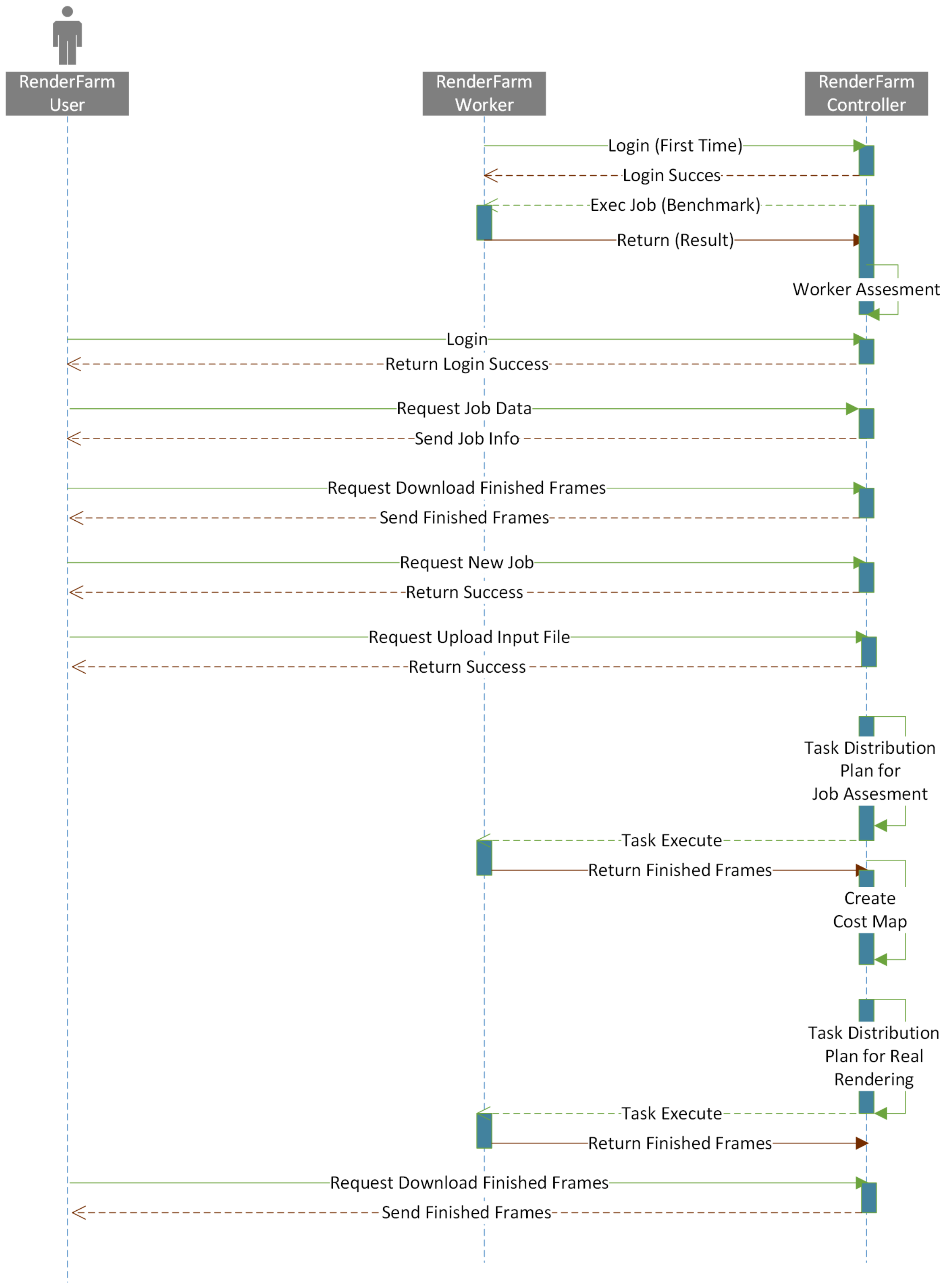


Figure 4.3: Sequence of a Basic Communication Between Software Units

### 4.2.1 Design of RenderFarm Controller

At this point of the development some important decisions have been made. The first thing was the programming language. Java has been chosen as the main programming language since platform independence is aimed. Although Windows is mainly used while development and the testing since 3DsMax needs it, the 3D software called "Blender" is completely platform independent and the software of this research can be completely portable also in case of using with Blender. After deciding the basic configuration of the distributed software and the programming language the communication scheme has been chosen. The alternatives were using third party off-the-shelf middleware software, using DDS (Data Distribution Service), using custom messages on bare TCP/IP and using RMI of Java. DDS and third party middleware alternatives are discarded because they are known with their extremely high configuration complexity. In other words, the solutions in these categories can be considered as too complex for this prototype work. On the other hand, using raw messages over TCP/IP can be over simple and also hard to implement since there are hundreds of commands in the interfaces of the system. If the raw messages with TCP/IP were used, encoder/decoder of each of those hundreds of messages would be implemented separately. Therefore, RMI has been chosen to be the communication layer of the system.

After the above decisions the remote interfaces are written according to the RMI standards and the implementation of the RenderFarm Controller is started. As mentioned above the controller is responsible for keeping all user accounts with the status of jobs, job input files and output files. It also responds to all commands coming from the "RenderFarm User Interface" instances. To increase the security of the communication via RMI, encryption enabled socket factories are used in the communication layer of the system.

The RenderFram controller has a database which is implemented with the technology called "H2" which is completely platform independent and free. The main entities in the database scheme are "Job", "Task" (which is a sub-part of a job having one or many frames), "Frame", "User" and "Worker". Moreover, there is a remote interface in the controller for each of these entities. The "User" entity is responsible for the authentication. In other words, login operations and every other method calls are done with a username and password. This way the security of the system is increased in an unsecured wide area network. In addition, the controller recognizes the caller of a method by using this credentials. The “Worker” entity and the interface is responsible for the communication of the controller and the “RenderFarm Worker” software unit. For example, a worker reports a problem to the controller through the “Worker” interface and the history and the performance information about a worker is stored in the “Worker” table of the database. The rest of the interfaces are straightforward. For example, when a user submits a new job, the “RenderFram User Interface” software unit calls the `"submitNewJob(String username, String password, Job job)"` method of the Job interface or when a worker finishes rendering a frame, it calls the `"frameRendered(String username, String password, Job job)"` method of the Frame interface. These calls can be observed in Figure 4.3.

### 4.2.2 Design of RenderFarm Worker

The software unit “RenderFarm Worker” is relatively simple than the others. Its main duty is to listen and apply the commands from the render farm controller. As can be seen in Figure 4.3 each worker instance also logs in to the system like a normal user while connecting. Each worker stores a global unique ID which is given by the controller at first login. At the time of this first login a special procedure is carried out called benchmarking. To assess the performance of the machine that the worker runs on, the controller executes a standard job on the worker. By measuring the task completion times the controller gives a performance point to that worker and matches it in the database with its unique identifier. Actually this is just the initial benchmarking and the whole assessment procedure is not limited to this. The details of this procedure are left to Section 4.4.

The other basic commands that can be executed on this software unit are:

- `newTaskAssigned(Task task, List<Frame> frames, RemoteInputStream remoteFileData, long jobFileSize);`
- `cancelTask(Task task);`

- `String getSystemInfo();`
- `void workerIsDeletedFromServer();`
- `List<FrameInterval> requestAvailableOutputOfTask(Task task);`
- `void workerUpdated(Worker worker);`
- `void cancelRenderUploadEncodeTask();`
- `void cancelRenderTask();`

As can be understood from these commands the controller assigns tasks to the workers by giving some necessary information about the task and the file handle to download the input file. After getting this command the worker starts to download the file from the controller server if it does not already have it. As mentioned earlier a "Task" entity is a subset of the frames of a job. The worker renders the frames listed in the "`List<Frame> frames`" parameter.

### 4.2.3 Design of RenderFarm User Interface

Platform independence is an important factor for this software unit and the best way of providing it to the users was choosing to develop a Web based interface. The most crucial decision has been made at this point. Although a Web based interface is the first thing that comes to mind, it is not used in this work. Instead, a desktop application is preferred. The most important factor for that decision was the download process of the output files. If a web based user interface was used, the user would only be able to download the output files together after the job is completed. This means that the user would have to wait some significant amount of time to download the output after the job is completed. However, by using a desktop application this problem has been solved. The developed desktop application has a synchronization module. This module continuously checks the account of a user for files waiting for download. By using this solution a rendered frame is downloaded to the computer of the user immediately and therefore the download of the output files and the rendering is overlapped.

Because the Java RMI is used for communication layer, this application is also developed with Java. It is a GUI application with a user session mechanism. In other words, the application asks for the credentials at start-up and displays the information of that user until log out. In addition, it has two modes which are "admin" and "user". If the logging in user has administrator privileges, the application displays all jobs of all users instead of only showing the jobs of the current user. In addition, it shows the status of all worker computers in a detailed view for maintenance purposes. Otherwise, if the logged in user is not an administrator, the user can only view the status of his/her jobs together with some detailed log information collected from the worker computers. The user has full control on his/her jobs such as cancelling, suspending, restarting, changing priority or previewing the partial results.

## 4.3 Communication Cost Optimization

As explained in Section 3.1 the network load and the network load imbalance is major problem for distributed animation rendering environments. Huge data transfers occur at the time of job submissions and at the time of collecting the output from a worker. The study of Chong et. al. [5] uses a lossless compression technique of the input files to deal with this problem. Although their technique is very efficient and elegant, it may not solve the problem in general. Because the majority of the input file size is usually occupied by the textures instead of the scene definition containing geometric objects, coordinates and directions. Moreover, the input file is transferred to the workers only once before the job begins. On the other hand, the output files flow to the render farm controller from the workers throughout the rendering process. Therefore, the optimizations for the network load of the output files may be more crucial especially if the resolution of the output files is high. Actually one may claim that the transfer of the output files is not a problem since it can be overlapped with the rendering. However, in general most of the render farms may not have dedicated network. In the local area network of an organization or in a campus network there may be so many other users using the same network lines. In this case if the render farm worker count is high, so many worker computers will make the lines busy with their output files. Consequently, this results in the deteriorated service quality for the other users. Another interactive application that depends on the network bandwidth may suffer from the high network load of the render farm. Therefore, a way to decrease the size of the output files should be found.

The biggest advantage of working with motion picture is the temporal coherence. The solutions proposed in this section and following one are highly dependent on the temporal coherence. (More detailed explanation about the temporal coherence will be given in Section 4.4.) The proposed solution for the network load problem in this work is actually from another field which is video compression.

Current video compression techniques use three main techniques called “transformation & quantization”, “entropy coding" and “prediction" which is the most important for this research since it is based on temporal coherence [14]. The technique “prediction” is important since its connection to the temporal coherence makes it a good tool for a research about motion picture rendering which also have temporal coherence. Therefore it is highly applicable to the problem in this work.

To be precise, the proposed approach of this work is to compress the output image files of the animation by using video encoding software. Actually, the outputs of the renderers, which are frame by frame and in still image formats, will eventually be combined into a video file since the aimed product of an animation is a motion picture in other words a movie. In the commercial render farm solutions such as Backburner of the Autodesk for now there is no option for getting the output as a movie if the rendering is distributed to multiple computers. If the rendering is distributed to multiple computers in Backburner, the output format can only be still image formats. That means that the user must convert the set of output frames into a video file manually after the rendering is completed for all frames. This requirement probably resulted by the fact that the video encoding software currently handles the parallel computing differently than the 3D rendering software. The 3D rendering software currently does not rely on the temporal coherence probably to increase the independence of the frames and therefore the ability to distribute the frames to different computers. On the other hand, the video compression techniques such as H.264 depend highly on the temporal coherence. Thus, video encoding software such as H.264 needs data from several frames at the same time [14]. Specifically a video encoder that will produce a single video file may be multi-threaded but must be one process working on a single computer with common memory and it must get the input frames in the correct sequential order while building the video. It cannot skip a frame and fill the gap of that frame later. Even though it is not impossible, this kind of software currently is not designed to handle these situations. On the other hand, the straightforward distributed rendering techniques are contradicting with these requirements since the finish times of the frames may be very divergent and the frames can be finished in mixed order on different computers that do not have a common memory. Therefore, the problem of reducing the network cost is now can be reduced to using the video compression on a distributed animation rendering environment with separate worker computers.

To reduce the network load, the video compression must be applied on the worker computer before sending the output files to the render farm controller. This contradicts with the requirement of having only one video encoder process running on a single computer and producing a single video file. Despite the differences in handling the parallel computing, the video encoding must be forced to work like rendering software. In other words, in some way it must be distributed to separate computers. With today's video encoders this is only possible by applying the video encoder separately to different continuous parts of the animation on different machines. In other words, the proposed technique of this work distributes the frames to the workers in a special way (which will be explained more detailed in Section 4.4) that each worker is responsible from a continuous part of the animation. For example, in a render farm with three workers, worker 1 may get the frames 1 to 100, worker 2 may get the frames 101 to 500 and the worker 3 may get the frames 501 to 1000 of an animation with 1000 frames. In the distribution scheme the workers creates a video encoding process for its continuous part and starts to render the frames sequentially. When a frame is completed, instead of sending the frame image file to the render farm controller immediately, it feeds that file to the video encoder. Then, when all frames in that continuous part are completed, the render farm worker software sends a close command to the video encoder to finalize the single video file that corresponds to the output of that continuous animation part. Then the render farm worker sends this compressed single movie file to the render farm controller. Hence, the network load because of the output of those frames is reduced as much as the compression ratio of the video compressor. Later when all workers have sent the video files that correspond to their part, the render farm controller merges the parts according to their sequences. Fortunately, this merging is possible with the video file containers such as “mp4" file which is also called “MPEG-4 Part 14”. By using the mp4 file format which supports streaming and merging, the render farm controller merges the output video without needing to re-encode them. In other words, the cost of this merging stage is almost identical to the cost of copying the final output video on the same disk.

Unfortunately, distributing the frames the way mentioned above makes the load balancing problem more complicated. It requires a special care to keep a balance between video compression ratio, load balance and fault tolerance. There is a crucial tradeoff between high video compression and fault tolerance that will be explained in Section 4.4. For example, the popular render farm solution Backburner of Autodesk distributes the frames with dynamically adjusted but usually very small chunks. That way it ensures the load balancing with its fine grained task distribution but the task submission overhead increases consequently. Moreover, that task distribution scheme makes the above mentioned video compression technique impossible because the frames assigned to a worker is not continuous. For video codecs such as H.264 that benefit from the temporal coherence having continuous big chunks is important. In other words, it is better to encode one video file having 100 frames than acquiring the same video file by merging 10 separately encoded video files having only 10 frames. With a task distribution scheme that does not consider the video compression techniques, the second case occurs. At this point the reason behind that should be examined. The video codec H.264 gives better compression ratio as the number of continuous frames increases. This effect of temporal coherence on the video compression ratio is illustrated in Figure 4.4.

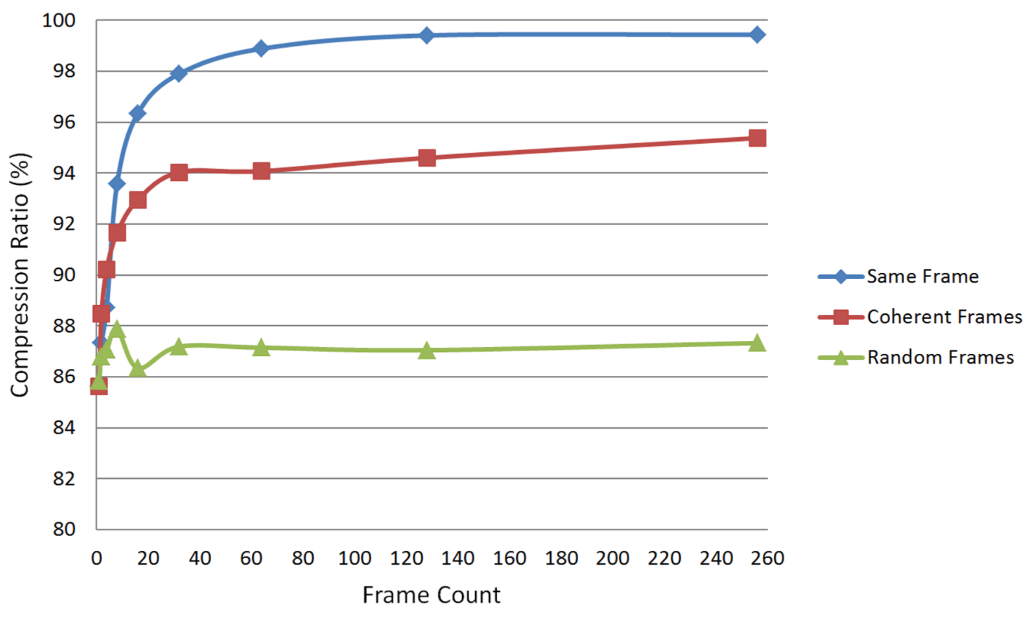


Figure 4.4: Effect of Temporal Coherence to Video Compression

The data on Figure 4.4 is acquired by 36 separate video encoding tests. In each test a video is encoded from several image files using H.264 codec[1]. The horizontal axis shows the number of images used in that test. The vertical axis shows the compression ratio. For example, if the total size of the 64 frames used in a test is 100 megabytes and the compression ratio is said to be 90% then the size of compressed video is 10 megabytes. The first 9 of these tests are illustrated with the curve having diamond marks. In these tests all frames are completely the same. A quickly rising and extremely high compression ratio can be observed in this graph. The second test group illustrated with curve having square marks. This curve corresponds to videos generated with different but coherent frames. Actually that frames belong to a real motion picture. In that case a drop can be observed in the compression ratio but it is still high and rising with frame count. Finally, the tests results illustrated on the curve having triangle marks belong to the tests made with completely different random images. In other words, there is no temporal coherence. In this case there is almost no rising trend on the curve. Instead it manifests a constant compression ratio which is a result of the other compression techniques of the H.264 codec called transformation & quantization and entropy coding which do not depend on temporal coherence [14].

[1] Please refer to Appendix B for compression presets of H.264 used in the test

## 4.4 Task Distribution Algorithm

As mentioned in Chapter 3 the challenge of rendering task distribution of an animation is the result of the per-task overheads. In other words, the initialization and finalization processes that have to be done for each task assigned to a worker. Although a fine-grained task division would provide a perfect load balancing, because of that overhead the task division should definitely be coarse grained so that the render farm deals with less number of per-task overhead. However, when the coarse grained task division is preferred, the problem of load imbalance may occur. Therefore, the ultimate goal of this algorithm is to achieve a smart task distribution where each worker has one task for one job and each worker finishes its task at the same time as much as possible. This problem results in the need of two fundamental data which are the cost distribution of the job and the performance metrics of the workers so that the algorithm assigns each worker to a task that has a cost proportional to the worker's processing power. In the following two sections the approaches used to assess the performance of the workers and to predict the cost distribution of the job are explained in detail. Later the steps of the task distribution algorithm will be presented in this chapter by combining the approaches in following two sections.

### 4.4.1 Measuring the Processing Powers of Render Farm Workers

It is crucial for the proposed task distribution algorithm to have a correct knowledge about the performance of the workers. Therefore some benchmarking mechanism is necessary. For this purpose the control software unit contains an embedded rendering task that is executed on each worker when it is connected to the system for the first time. Naturally, this task execution may give different results in different times depending on the other processes working on the machine. But it is assumed that the staff is responsible for ensuring the machine is completely idle while connecting it to the system for the first time. After this benchmarking the controller of the render farm has knowledge about the processing power of the worker machine. However the most important constraint about the approach proposed in this work is agility and dynamism. To satisfy this constraint the system should keep the performance point of the worker updated all the time. The need for this arises from the fact that the performance of a machine does not only depend on the hardware used in it. Instead, the health of the operating system, file system, the system up-time or even the disk usage may affect the performance of a system. Because of this, the controller should dynamically update the performance points of the workers in the render farm.

While assessing the performance of a worker, the controller uses ratios. For example, in a render farm containing hundred slow lap-tops and one powerful workstation the performance point of the workstation may be *"5.3"*. On the other hand the performance point of the same workstation may decrease to *"0.7"* if an extremely powerful mainframe is added to the render farm. Because of this fact the performance point of all workers are updated when there is any change in the statistics of the render farm. This change may be completion of a frame, failure of a worker or first-time connection of a new machine. After each of these events the performance point of each worker is updated according to the below formula.

$$P = (W(n_{frame}) \times \frac{T_{benchmarkRF}}{T_{benchmarkWorker}}) + ((1 - W(n_{frame})) \times \frac{T_{avgAllFramesRF}}{T_{avgAllFramesWorker}})$$

$$W(n) = 0.3 + 0.7 \times \frac{\arctan(\frac{300-n}{100}) + \frac{\pi}{2}}{\arctan(3) + \frac{\pi}{2}}$$

where:
$P$: The performance point of the worker
$W(n)$: The function calculating the weight of the benchmark with respect to the frame count
$T_{benchmarkRF}$: The average benchmark task completion time of the whole render farm
$T_{benchmarkWorker}$: The benchmark task completion time of the worker
$n_{frame}$: Number of frames successfully completed by the worker since the first connection
$T_{avgAllFramesRF}$: The average frame rendering time of the whole render farm
$T_{avgAllFramesWorker}$: The average frame rendering time of the worker

Basically this formula is a weighted average of the performance point acquired by the benchmarking and the future tasks. At the moment of first connection the performance point of a worker is determined primarily by the benchmarking results. In other words, the value of $W(n_{frame})$ is $1$. However, as the worker finishes some tasks, the effect of the finished tasks gains more and more importance by time but never gets more than $70\%$. The change of the ratio between benchmark and the statistical data with respect to the rendered frame count is illustrated in Figure 4.5.

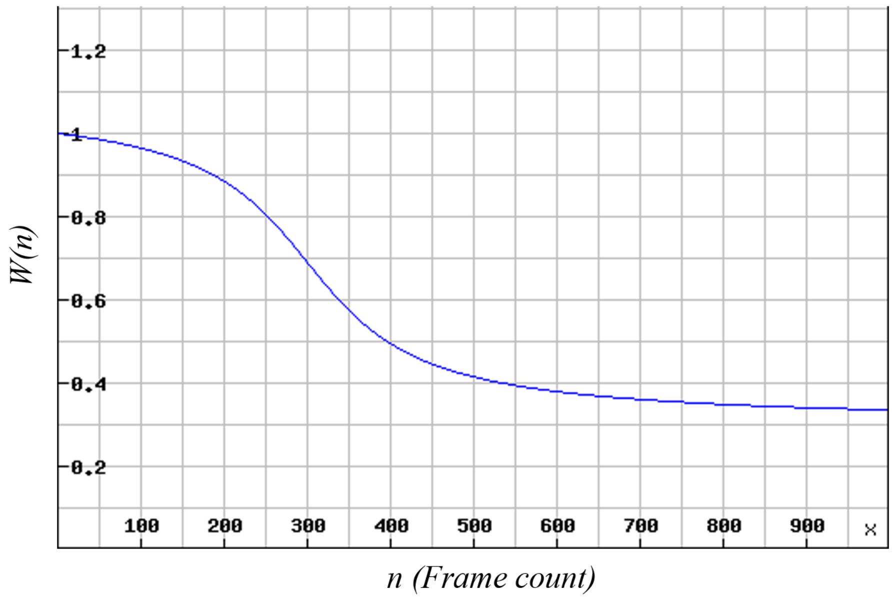


Figure 4.5: Visualization of Benchmark Weight Function

While designing the $W(n)$ function, smoothness has been taken as an important constraint. The $arctan$ function is used because of this. If the smoothness had not been provided, the sudden changes in the processing power of a worker would result in frequent task distribution changes while rendering.

There are two main constants in the function which are adjustable parts. The first element $0.3$ determines the minimum weight that the benchmarking point will take in the future which corresponds to $30\%$. Secondly, the $300$ value determines the frame count for which the slope of the weight function is highest as can be seen from Figure 4.5. Increasing this value results in a more horizontal curve. In other words, the $300$ term determines the change speed of the benchmark point weight in the processing power point. These two values can be fine-tuned according to the job characteristics of the render farm.

The notion of weighted average is also important because of another issue which can be explained with the example of an old render farm and a new worker connecting to it. If the render farm finished some extremely heavy jobs in the past and the contemporary jobs are relatively lighter, the new worker would get a performance point higher than it deserves because of the difference in jobs that contribute to the render farm average and the worker average. However, thanks to the weighted average this will not be a problem since the effect of the task averages to the performance point will increase after some significant amount of work.

The advantage of using the ratios for performance points will be clearer while explaining the cost prediction algorithm. To summarize, the aim of using ratios is to have ability to solve the following simple equation: *"How much time will it take for worker A to render frame n if the worker B has rendered it in t time where the performance points of the workers A and B are 1.3 and 0.8 respectively?"*. Answering this question is simply the first stage for a good task distribution.

### 4.4.2 Predicting the Cost Distribution of Job

As mentioned in Chapter 2 the problem of cost prediction is an important topic in distributed rendering optimizations because it is crucial for a good load balancing. Actually, if fine-grained task division was used and if the workers were just picking a small task when idle, this would not be a problem and naturally the work load of each worker would be proportional to the processing power of the worker. However because of the per-task overhead and the compression characteristics of the video compression algorithms mentioned in Section 4.3 coarse-grained task division must be used and therefore every task should be divided by the controller according to the processing power of the worker that it will be assigned to.

The problem arising at this point is that the performance of worker is not the only variable, but the costs of the frames of a job are also variable. Namely, in an animation movie there may be some high number of different scenes and each of them may have different complexity in terms of geometry and lights. Moreover, even in the same scene, as the camera moves the distance of a complex object to the camera may change and this also affects the cost of the frames. This situation can be illustrated with Figure 4.6[2].

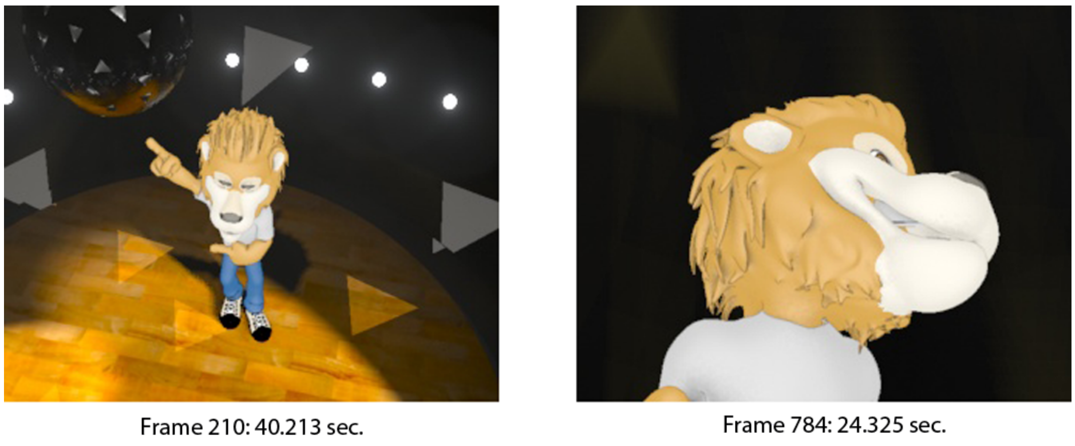


Figure 4.6: Cost Difference Caused By The Camera Angles

In Figure 4.6 two frames are presented from one of the test inputs of this work named "Aslan_dans_project.max". This input file contains only one scene but the camera and the actor moves throughout the movie. In frame 210 (on the left hand side) the actor is far from the camera compared to the frame 784 (on the right hand side). Thus, in frame 210 some other objects from the scene are also visible; whereas, in the frame 784 the only object rendered is the actor. As a result the rendering costs of these two frames are different. Namely, the frame 210 and 784 took 40.213 and 24.325 seconds to render respectively (with 400x320 resolution and on an Intel i7 8GB RAM machine). Therefore, the rendering time of frame 210 is 83% higher than the rendering time of frame 784. In case of a bad task distribution plan this difference may cause a significant load imbalance.

As mentioned in Chapter 2 Reinhard et. al. [17] proposed to build a spatial subdivision tree for cost prediction. This approach is not suitable for this work since it requires detection of the scene contents which is avoided in this research in order to create a more generic solution. Secondly, Gillibrand et. al. [7] use a profiling technique which renders a small subset of the scene for a few pixels. This method does not need to parse the scene file. However, this method is more suitable for a

[2] The pictures in figure are from an animation project provided by Zor Zanaat Animation Studio

fine-grained task division or for a real-time distributed rendering in which the closest dead-line is one frame instead of the whole job. Moreover, the methods of Reinhard et. al. [17] and Gillibrand et. al. [7] would be unnecessarily costly because the level of cost prediction precision they provide is unnecessary for an animation rendering job which may take extremely long times unlike the real-time distributed rendering. In this research, a lighter and cheaper algorithm should be used.

Although the method of Gillibrand et. al. [7] cannot be directly used for this work, it gives a clue for the solution. The proposed cost prediction approach of this work also uses the profiling notion but the main difference with the solution of Gillibrand et. al. [7] is that this technique profiles frames instead of pixels. To be more precise, before starting to render a job the system prepares a subset of the frames with constant gaps. For example, the profiling frame subset for an animation with 100 frames may be $1, 10, 20, \ldots, 100$. Each of these frames will be assigned to a worker. After rendering these frames the system will have a rough cost map of the job. As will be explained later in this chapter the first task distribution plan will be based on this rough cost prediction. Even though it is rough at first, as the frames are completed this map will be refined by time. In other words, the profiling is not a preprocessing stage in this approach. Instead it is the rendering itself and the profiling continues as long as the rendering continues by refining the cost map. This continuous refinement has a great benefit for the agility of the task distribution algorithm. Another advantage of this method is that the time spent for profiling is not wasted since the output files of the profiling stage is also the output of the rendering itself.

Besides the advantages mentioned above, a big question arises for this approach. In the early stages of the rendering, when there are only a few finished frames, will the cost prediction be reliable enough? The answer to this question does not only depend on the algorithm but also the characteristics of the scene. Namely, if the camera angles or the scene geometry changes dramatically in every frame, this approach will probably provide a bad estimation. However, this is not the case for most of the animations. An animation movie that is produced for humans should have some degree of continuity. Therefore, the term “Temporal coherence” is a reliable base for this approach. Temporal coherence can be illustrated in Figure 4.7.

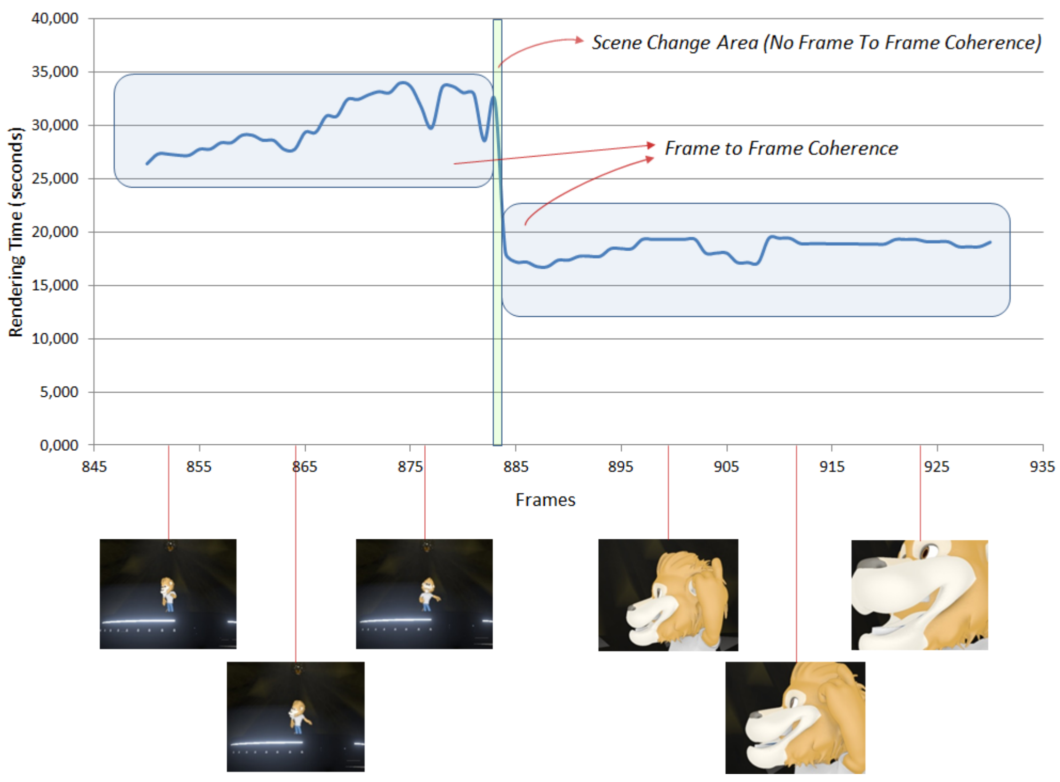


Figure 4.7: Effect of Temporal Coherence on Cost Distribution

In Figure 4.7 the cost effect of temporal coherence is shown. The data belongs to the frames between 850 and 930 from the test animation “Aslan_dans_project.max”. Throughout this animation the scene contents are always the same. In other words, the complexity of the scene contents does not change. However, the changes in the view angle and the position of the actor affect the frame costs dramatically. The sudden cost change at frame 880 is because of that fact. On the other hand, except this scene change area there is a significant coherence between the frames. This coherence is illustrated by selecting three random frames from each area. As mentioned before, animation files contain plenty of coherent areas and some inconsistent transition areas in between. Fortunately, as can be observed from Figure 4.7 the proportion of the coherent areas is far more that the inconsistent areas. For example, in this experiment there is only one inconsistent frame transition for 80 consistent frame transitions and this ratio may be even greater. As a result, counting on the temporal coherence is statistically reliable for estimating the frame costs.

### 4.4.3 The Algorithm

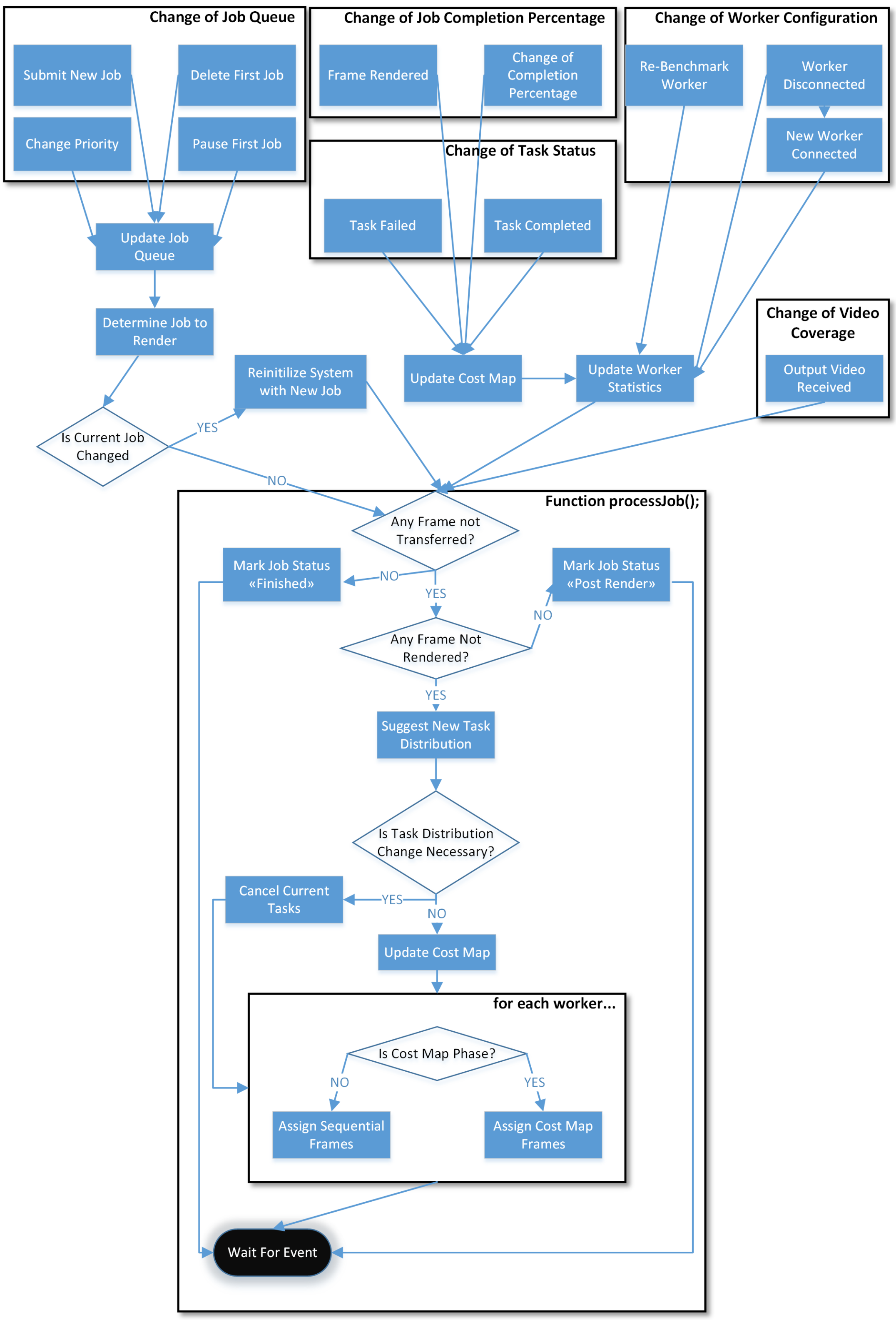


Figure 4.8: Basic Structure of the Task Distribution Algorithm

In this section of this chapter the core algorithm of the “RenderFarm Controller” software unit is presented. As mentioned at the beginning of this chapter the proposed algorithm aims to maintain the load balance between render farm workers and increase the video compression ratio by using the techniques explained in Sections 4.3, 4.4.1 and 4.4.2. The basic structure of the algorithm is illustrated in Figure 4.8. As can be seen from Figure 4.8 the algorithm is event based. After each event the algorithm carries out the necessary steps and halts until a new event is received. The events are as follows:

- **Change of Job Queue:** This event occurs when the first job of the job queue changes. This can be caused by inserting a job when there is no other job, inserting a job with higher priority than the job that is currently in the first order in the queue, pausing the first job of the queue, resuming a job having higher priority than the first job of the queue and finally deleting or cancelling the first job of the queue. The task distribution system needs this event to know which job to process.

- **Change of Job Completion Percentage:** This event is triggered when a frame is completed by its render farm worker. The task distribution system needs this event to ensure load balance during the whole progress of the job and know the completion of rendering.

- **Change of Video Coverage:** This event is triggered when the compressed video of a task is received by the render farm controller. The task distribution system needs this event to know the moment of completion of a job. Because of the video encoding approach, the job is not completed when its frame rendering percentage reaches 100%. In other words, the job is not finished until all encoded video files are received even though all frames are rendered by the workers. Therefore, after the “Change of Job Completion Percentage” event reports 100% frame rendering percentage, the state of job is set to “Post Rendering” and the task distribution system waits for the “Change of Video Coverage" event that reports all video files are received. Then the state of the job is set to “Finished”. After the status of current job is set to “Finished” the job queue is refreshed and this triggers the “Change of Job Queue” event.

- **Change of Task Status:** This event is triggered when one of the sub tasks of the current job changes. The possible states for tasks are “Pending”, “Rendering”, “Cancelled”, “Post Render”, “Finished” and “Failed”. When a task is created its task is “Pending”. When the worker starts its task the status is changed to “Rendering”. If any error occurs or the worker disconnects unexpectedly, the status of the task is changed to “Failed”. When the whole job is cancelled or a specific task is retracted from its worker the task status is set to “Cancelled”. When all frames are rendered, the status is set to “Post Render” before the task is marked as “Finished”. In this stage the worker encodes the result video and uploads this video to the render farm controller.

- **Change of Worker Configuration:** This event is triggered when the set of currently available workers is changed or when the relative processing power of the workers change. The set of currently available workers change when a new worker is connected, a worker is disconnected or the activeness of a worker is changed by the render farm administrator. The relative processing powers of the workers may change after the completion of each frame. When a worker reports that it has rendered a frame it also reports the elapsed time. With this value all relative processing powers are updated with the formula explained in Section 4.4.1. The task distribution system needs this event to maintain the load balance and ensure that the most optimal worker set is used throughout the rendering process.

Above mentioned above five events call the procedure named *processJob();* when triggered. The *processJob();* is given in Algorithm 1. The algorithms given below work on the following shared data:

**Variables:**

- $J$ is the currently rendering job
- $N$ is the number of frames in the current job
- $F$ is the array containing all frames of the current job having size $N$ and sorted by frame number

- $N'$ is the number of frames in the cost map where $N' < N$
- $C$ is the array containing the cost map frames where $C \subset F$ and sorted by frame number
- $W$ is the array containing all workers assigned to the current job
- $W_{all}$ is the array containing all workers of the render farm
- $T$ is the duration of the output animation movie
- $C_{percentage}$ is the ratio between cost map frames and all frames as percentage which is $100 \times \frac{N'}{N}$
- $R$ is a mapping denoting the task distribution regions. More precisely, $R = \{W \times (F \times F)$ | *Worker $w$ is responsible for rendering the region starting with $f_1$ and ending with $f_2$ where $w \in W$, $f_1 \in F$, $f_2 \in F$ and $f_1 < f_2$*$\}$
- $costPredictionEnabled$ is a flag indicating the cost prediction is possible
- $initialized$ is a flag indicating the initialization of the cost map is done

**Constants:**

- $N'_{min} := 10$ is minimum number of frames in $C$
- $C_{minPercentage} := 4$ is the minimum for $C_{percentage}$
- $C_{maxPercentage} := 15$ is the maximum for $C_{percentage}$
- $C_{costSamplePerSecond} := 1$ is the cost map frame count for one second of the animation.
- $n_{max} := 50$ is the maximum number of frame in a sub task
- $\lambda := 1.2$ is the cost increase acceptance factor when task distribution map changing

**Algorithm 1** The Main Task Distribution System Algorithm

```
1: function ProcessJob()
2:     if $\neg initialized$ then
3:         InitializeCostMap()
4:     end if
5:     if $\neg$anyFrameNotTransferred($F$) then
6:         SetJobStatus($J, Finished$)
7:         return                                                ▷ The job is completed.
8:     end if
9:     if $\neg$anyFrameNotRendered($F$) then
10:        SetJobStatus($J, PostRender$)
11:        return                          ▷ Wait until all rendered frames are transferred
12:    end if
13:    if MissingWorkerDetected($W$) then
14:        $R \leftarrow \varnothing$                      ▷ Task distribution plan is invalidated
15:    end if
16:    $R' \leftarrow$ SuggestTaskDistribution($F$)        ▷ Suggest optimal task distribution
17:    if ($R = \varnothing \vee$ isRegionChangeNecessary($R, R'$)) then
18:        if $W \neq \varnothing$ then
19:            for $i \leftarrow 1, |W|$ do
20:                CancelTaskOfWorker($W$)          ▷ Cancel current tasks of the workers
21:            end for
22:        end if
23:        $R \leftarrow R'$                        ▷ The new task distribution is accepted
24:        $|W| \leftarrow |R|$
25:        for $i \leftarrow 1, |R|$ do
26:            $W[i] \leftarrow$ GetWorkerOfRegion($R[i]$)     ▷ Reinitialize the worker list
27:        end for
28:    end if
29:    if anyFrameNotAssignedOrRendered($F$) then   ▷ While there is a frame to assign
30:        for $i \leftarrow 1, |W|$ do
31:            $w \leftarrow W[i]$                                ▷ For each worker w
32:            $r \leftarrow$ GetRegionOfWorker($w, R$)
33:            $i_{regionStart} \leftarrow$ GetStartIndexOfRegion($r$)
34:            $i_{regionEnd} \leftarrow$ GetEndIndexOfRegion($r$)
35:            $costMapPhase \leftarrow costPredictionEnabled \wedge$
    $\neg$anyFrameNotAssignedOrRendered($C$)        ▷ Cost map generation phase or not
```

```
36:             if $\neg$isWorkerBusy($w$) then
37:                 $j \leftarrow 0$
38:                                                   ▷ Determine the frames to assign
39:                                          ▷ Get each frame if not cost map generation phase
40:                                                   ▷ otherwise pick only cost map frames
41:                 for $i \leftarrow i_{regionStart}, i_{regionEnd}$ do
42:                     $f \leftarrow F[i]$
43:                     if $(costMapPhase \wedge f \in C) \vee \neg costMapPhase$ then
44:                         $F_{region}[j] \leftarrow f$
45:                         $j \leftarrow j + 1$
46:                     end if
47:                 end for
48:                 if $i_{regionStart} = 0$ then                          ▷ If left most region
49:                     $regionCenterFrame \leftarrow F[0]$
50:                 else if $i_{regionEnd} = |F|$ then                     ▷ If right most region
51:                     $regionCenterFrame \leftarrow F[|F|]$
52:                 else                                                    ▷ Middle regions
53:                     $regionCenterFrame \leftarrow F[\frac{i_{regionEnd} + i_{regionStart}}{2}]$
54:                 end if
55:                 SortByDistanceToRegionCenter($F_{region}, regionCenterFrame$)
56:                 for $i \leftarrow 1, \text{Min}(n_{max}, |F_{region}|)$ do
57:                     $F_{task}[i] \leftarrow F_{region}[i]$
58:                 end for
59:                 if $|F_{task}| > 0$ then
60:                     AssignTaskToWorker($w, F_{task}$)
61:                 end if
62:             end if
63:         end for
64:     end if
65: end function
```

In line 3 of Algorithm 1 the cost map of the job, which is a mapping from workers set to frame intervals, is initialized at the beginning of a job. The algorithm used in function *InitializeCostMap()* is given in Algorithm 2. Then at line 5 the algorithm checks whether the output of all frames is received by the render farm controller. If so the job is finished. As mentioned in Section 4.3, to be able to carry out video encoding, render farm workers do not send the output of the frames until the task is completed. When the task is completed, all image outputs of the frames will be encoded into a video file. Therefore, until the end of a task, the frames in that task may be rendered but not yet transferred. Because of that fact, at the very last moments of a job, all frames may be rendered but the uploading of the result videos may be in progress. In that period of time the state of the job is marked as “Post Render”. The if block at line 9 is responsible from this decision. The next if block at line 13 invalidates the task distribution plan when any worker of the job becomes unavailable. In that case becoming unavailable may be result of an unexpected failure, a network problem or intentional removal from the system by an administrator. In any way if there is previous task distribution plan $R$, it should be cancelled and a new one should be used. Then at line 16 an optimal task distribution plan is created by the function *SuggestTaskDistribution* and this mapping is assigned to a local variable $R'$ to be compared with the current task distribution plan $R$. This comparison is done by calling the function *isRegionChangeNecessary* at line 17. The purpose of comparing the currently working task distribution plan with the suggested optimal one is to detect any load imbalance. The fact that this comparison is done very frequently gives the dynamism to the system, which makes it possible to detect load imbalance immediately when it exceeds some threshold. The algorithm used in function *isRegionChangeNecessary*, which is developed for the decision of change the old task distribution regions with the new optimal ones, is given in Algorithm 4.

Then at line 30 there is the big for loop block which assigns the frames to the suitable workers. For carry out that task, the algorithm first determines whether the job is in cost map phase or not. (Line 35) The meaning of being in cost prediction phase is that assigning the equally spaced cost map frames to the workers before assigning all frames sequentially. By doing this the system tries to have an estimated knowledge about the frame cost based on the ideas given in Section 4.4.2. Thus the array $F_{region}$ is filled by all frames of the region or by the selected cost map frames in that region according to the variable $costMapPhase$. Later at line the algorithm finds the most secure frame of the region which is called the region center. The notion of being secure for a frame means that having the highest probability of belonging to the same worker till the end of job. This is one of the most important optimizations of this algorithm. For a frame belonging to the same worker is important because if that frame is rendered by a worker A and later its neighbor frame is rendered on a different worker B because of a task distribution plan change, this result with a break off in the output video. Consequently, the compression ratio of the result video file will decrease because of the facts explained in Section 4.3. Considering the dynamic load balancing approach of the algorithm, the frames close to the region boundaries is more prone to change region in the initial moments of the job since the reliability of cost map is lower at the beginning. Thus the rendering order of frame is not the frame number but the distance to the so called region center. In other words, rendering starts from the the most secure frame of the region.

After sorting the frames according to their distance to the region center, the algorithm chooses the first $n_{max}$ of the frames to assign to the worker. The task size in this system is limited to $n_{max}$ because of a tradeoff. As mentioned above the video encoding approach requires holding the rendering result until the end of the task to encode the image files into single video file. However, because of that behavior, if the worker fails in the middle of a task, the frames which are rendered but not yet transferred becomes un-rendered since it is now impossible to reach their outputs. As a result, instead of assigning all regions to the worker at once, the size of a task is limited to $n_{max}$.

The specific value 50 for the $n_{max}$ is chosen according to the main conclusion that is derived from the experiments demonstrated in Figure 4.4 and the details of the video compression technique called “Prediction" [14]. According to the experiments the compression ratio increases as the number of coherent and adjacent frames increases. The rising trend of the curves of same and coherent frames (diamond and square mark) in Figure 4.4 is a result of that feature of H.264 video codec. Therefore, one of the main constraints of the task distribution algorithm is to assign big continuous parts to the workers. In other words, it should implement a coarse grained task distribution. However, as mentioned before, there is a tradeoff and it should not relinquish the load balance to enlarge these task parts since bigger sub-tasks means higher possibility of load imbalance. Fortunately, the effect of temporal coherence on compression ratio does not necessarily require the sub-tasks to be as big as possible. As can be seen from Figure 4.4, the rise of the compression ratio slows down significantly after 50 frames in a video. In other words, a video with 1000 frames and 50 frames have very similar compression ratios because of the size of the prediction window size of the H.264 video compression. Therefore "50 frames" is a good maximum sub-task size to choose. Therefore, the maximum task size in the algorithm is also set to be 50 frames. In other words, even if a huge region having 2000 frames is assigned to a worker, the worker can get at most 50 frames in one sub task. Therefore, that region with 2000 frames may be rendered with 40 separate sub tasks.

**Algorithm 2** Cost Map Initialization Algorithm

```
 1: function InitializeCostMap()
 2:     $N' \leftarrow T \times C_{costSamplePerSecond}$
 3:     $C_{percentage} \leftarrow 100 \times \frac{N'}{N}$
 4:     if $C_{percentage} < C_{minPercentage}$ then
 5:         $C_{percentage} \leftarrow C_{minPercentage}$
 6:         $N' \leftarrow T \times C_{costSamplePerSecond}$
 7:     end if
 8:     if $C_{percentage} > C_{maxPercentage}$ then
 9:         $C_{percentage} \leftarrow C_{maxPercentage}$
10:         $N' \leftarrow T \times C_{costSamplePerSecond}$
11:     end if
12:     if $N' < N'_{min}$ then
13:         $costPredictionEnabled \leftarrow false$
14:         $N' \leftarrow T \times C_{costSamplePerSecond}$
15:     else
16:         $costPredictionEnabled \leftarrow true$
17:         $costmapGap \leftarrow \frac{(N-N')}{N'-1}$
18:         $|C| \leftarrow N'$
19:         $C[1] \leftarrow F[1]$
20:         for $i \leftarrow 2, N'$ do
21:             $C[i] \leftarrow F[(i \times costmapGap) + i]$
22:         end for
23:         $C[N'] \leftarrow F[N]$
24:     end if
25:     $initialized \leftarrow true$
26:     SetJobStatus$(J, Rendering)$
27: end function
```

As mentioned above Algorithm 2 is responsible from creating the cost map which is actually a subset of all frames whose elements are equally spaced frames. To be more precise, the algorithm first tries to determine how many of the frames will be in the cost map $C$. The key factor in this calculation is the constant $C_{costSamplePerSecond}$. The algorithm determines how many frames correspond to one second of the animation using the frame rate value. Then it calculates the $C_{percentage}$ and checks whether it is between the allowed minimum and maximum. As long as $C_{percentage}$ is between $C_{minPercentage}$ and $C_{maxPercentage}$ values, the system tries to render $C_{costSamplePerSecond} \times T$ frames in the cost prediction phase. For example, for an animation with 25 FPS and 10000 frames (400 seconds) there will be 400 cost map frames which results $C_{percentage}$ to be $4$ which is in the limits. FPS value may change the $C_{percentage}$ value, however if the calculated $C_{percentage}$ exceeds the limits, it will be recalculated with a different $n_{costSamplePerSecond}$ to keep it in limits. Unfortunately, there is another minimum which is the minimum number of frames in the cost map. If an animation is so small that cost map contains less than $N'_{min}$ frames even with the maximum $C_{percentage}$, the cost prediction phase will be disabled since the cost map will be unreliable. Straightforward rendering approach will be used in that case.

The purpose of Algorithm 3 is to generate the most load balanced and optimal task distribution plan at any moment. As mentioned before, the algorithm 4 decides to use that task distribution or not. At line 2, the workers are sorted according to their relative processing power. The purpose of that is to choose the most powerful machines if there are more workers than the frame count even though this is a small probability. Then the total processing power and the total remaining cost of frames is calculated. Rest of the idea is straight forward. The list of all remaining frames is partitioned into continuous regions having costs proportional to the processing power of the worker that it is assigned to. As a result this creates a worker-region mapping list $R$ which is expected to be load balanced according to the current knowledge about the job and the workers.

**Algorithm 3** Task Distribution Suggestion Algorithm

```
function SuggestTaskDistribution($F$)
    SortByProcessingPower($W_{all}$)
    $totalWorkerPower \leftarrow 0$
    for $i \leftarrow 1, \text{Min}(|W_{all}|, |F|)$ do
        $W_{optimal}[i] \leftarrow W_{all}[i]$        ▷ Get the most powerful workers
        $totalWorkerPower \leftarrow totalWorkerPower + \text{GetPowerOf}(W_{all}[i])$
    end for
    $totalRemainingCost \leftarrow 0$
    for $i \leftarrow 1, |F|$ do
        if $\neg\text{IsRendered}(F[i])$ then
            $totalRemainingCost \leftarrow totalRemainingCost + \text{GetCostOf}(F[i])$
        end if
    end for
    for $i \leftarrow 1, |W_{optimal}|$ do
        $w \leftarrow W_{optimal}[i]$
        $powerProportion \leftarrow \frac{\text{GetPowerOf}(W_{all}[i])}{totalWorkerPower}$
        $regionTargetCost \leftarrow totalRemainingCost \times powerProportion$
        $i_{regionStart} \leftarrow 0, i_{regionEnd} \leftarrow 0, cost \leftarrow 0$
        for $j \leftarrow i_{regionStart}, |F| \land cost < regionTargetCost$ do
            $f \leftarrow F[j]$
            if $\neg\text{IsOutputRequested}(f) \land \neg\text{IsOutputReceived}(f)$ then
                $cost \leftarrow cost + \text{GetCostOf}(f)$
            end if
            $i_{regionEnd} \leftarrow i_{regionEnd} + 1$
        end for
        if $cost > 0$ then
            $R_{result}[i] \leftarrow (w, (i_{regionStart}, i_{regionEnd}))$
            if $i_{regionEnd} > |F|$ then
                break
            end if
            $i_{regionStart} \leftarrow i_{regionEnd} + 1$
            $i_{regionEnd} \leftarrow i_{regionStart} + 1$
        end if
    end for
    return $R_{result}$
end function
```

**Algorithm 4** Task Distribution Change Decision Algorithm

```
function IsRegionChangeNecessary($r, r'$)    ▷ r is the current and r' is the suggested regions
    if $r' = \varnothing \lor$ AreRegionsSame($r, r'$) then
        return false
    end if
    if $r = \varnothing$ then
        return true
    else
        $t_{remainingCurrent} \leftarrow$ GetCurrentRemainingTime($r$)
        $t_{remainingNew} \leftarrow$ GetRemainingTimeAfterRegionChange($r'$)
        if $t_{remainingNew} \times \lambda < t_{remainingCurrent}$ then
            return true
        else
            return false
        end if
    end if
end function
```

At any moment of the job, the system generates an optimal task distribution plan and the Algorithm 4 decides whether to cancel the current task distribution and switch to the new one. The key point in this decision is the estimated remaining times which is calculated by Algorithm 5. As can be seen between lines 8 and 14, the algorithm gets the remaining time estimations for the current task distribution and the suggested one. Then, it decides to switch to the new one if the new one will make the current job to finish earlier. At this comparison the estimated remaining time for the new task distribution plan is multiplied with $\lambda$ to decrease the probability of wrong decisions that can increase the rendering time caused by the task cancellation penalty which is low video compression ratio and increased task submission costs.

**Algorithm 5** Remaining Time Estimation Algorithm

```
1: function GETCURRENTREMAININGTIME($r, afterRegionChange$)
2:     if ¬ANYFRAMERENDERED($F$) then
3:         return -1
4:     end if
5:     $t_{Job} \leftarrow 0$                                   ▷ Remaining time of job
6:     $w \leftarrow GetWorkers(r)$
7:     for $i \leftarrow 1, |w|$ do
8:         $i_{regionStart} \leftarrow$ GETSTARTINDEXOFREGION($r[i]$)
9:         $i_{regionEnd} \leftarrow$ GETENDINDEXOFREGION($r[i]$)
10:        $remingCost \leftarrow 0$                      ▷ Remaining scaled cost of region
11:        $n_{unassignedFrames} \leftarrow 0$             ▷ Number of unassigned frames
12:        for $j \leftarrow i_{regionStart}, i_{regionEnd}$ do
13:            if afterRegionChange then
14:                if ¬ISOUTPUTREQUESTED($F[j]$) $\wedge$ ¬ISOUTPUTRECEIVED($F[j]$) then
15:                    $remingCost \leftarrow remingCost +$ GETCOSTOF($F[j]$)
16:                    $n_{unassignedFrames} \leftarrow n_{unassignedFrames} + 1$
17:                end if
18:            else
19:                if ¬ISRENDERED($F[j]$) then
20:                    $remingCost \leftarrow remingCost +$ GETCOSTOF($F[j]$)
21:                end if
22:                if ¬ISASSIGNED($F[j]$) then
23:                    $n_{unassignedFrames} \leftarrow n_{unassignedFrames} + 1$
24:                end if
25:            end if
26:        end for
27:        $t_{Region} \leftarrow \frac{remingCost}{GetRelativeProcessingPowerOf(w[i])}$
28:        $n_{taskSubmission} \leftarrow \frac{n_{unassignedFrames}}{n_{max}}$
29:        $t_{taskSubmission} \leftarrow n_{taskSubmission} \times$ GETSTARTUPTIMECOSTOF($w[i]$)
30:        $t_{Region} \leftarrow t_{Region} + t_{taskSubmission}$
31:        if $t_{Job} < t_{Region}$ then
32:            $t_{Job} \leftarrow t_{Region}$
33:        end if
34:    end for
35: end function
```

Together with the techniques given in Section 4.4.2, algorithm 5 is one of the most important parts of the task distribution system since good remaining time estimation is the key of load balancing. The main idea of this algorithm is that the remaining time for a job is the maximum of the remaining times of workers. The if block at line 31 corresponds to this maximum calculation. But before that the challenge is to calculate the remaining time for each worker according to its given region. To do that, firstly, total scaled cost and number of the remaining frames should be calculated. However, the term "remaining" gets two different meanings according to the use case of this function. The if block at line 13 emerges because of this. The $afterRegionChange$ parameter of the function determines this use case as can be seen at line 8 of Algorithm 4. If the function is used to calculate the remaining time of the suggested regions after canceling the current one, the $afterRegionChange$ variable is set to be $true$ and the first part of the if block at line 13 works. In case of cancelling the current task distribution, a frame which is rendered but not yet transferred to the render farm controller becomes both un-rendered and un-assigned after the task cancellation. On the other hand, in case of calculating for the current task distribution without cancelling, even though the output is not transferred yet, it will arrive in the future. Therefore, a rendered but not transferred frame is not considered to be remaining. After collecting the remaining scaled cost information, to change the scaled cost information into remaining time information, the cost is divided by the relative processing power of the worker. (Line 27) Unfortunately, the rendering time does not correspond directly to the remaining time of worker. There is also $t_{taskSubmission}$ which is the total task submission overhead that will be occurred in the future. To calculate this value total number of future task submissions is multiplied by the renderer start-up time of worker which is detected while initial benchmarking. Naturally, the renderer startup time does not depend on the job.

### 4.4.4 Complexity Analysis of the Algorithm

The time complexity of Algorithm 1 depends on the other algorithms because all other algorithms are used in Algorithm 1. The time complexity of Algorithm 2 is $O(n)$ where "$n$" is the frame count of the animation. The complexity of Algorithm 3 is $O(wn)$ because of the nested loop at line 19 where "$w$" is the worker count. The complexity of Algorithm 4 equals to the complexity of the Algorithm 5 because it directly calls the function *GetCurrentRemainingTime()*. Algorithm 5 has also complexity $O(wn)$ since it loops on frames inside a loop running on workers list. Fortunately, the function *processJob()*, which implements Algorithm 1, does not call the other algorithms in loops. However, at line 55 of Algorithm 1 the frame list is sorted in a loop running on the workers list. Because the complexity of this sorting operation is $O(nlogn)$ the complexity of Algorithm 1 is $O(wnlogn)$.

# CHAPTER 5

# RESULTS AND DISCUSSION

After implementing the algorithms presented in Chapter 4 together with the rest of render farm software some tests have been carried out on the render farm built in MODSIMMER. The hardware configuration of the render farm is given below:

**Hardware Configuration of Workers 1-6:**

CPU: Intel Xeon E5620 2.40GHz
Memory: 24 GB
HDD: 7200 RPM
Network: Gigabit Network Adapter

**Hardware Configuration of Workers 7-10:**

CPU: Intel Core 2 Duo E7400 2.80GHz
Memory: 2 GB
HDD: 7200 RPM
Network: Gigabit Network Adapter

All computers are connected with gigabit network switches. Throughout the rest of this chapter, the worker numbers 1-10 mentioned above will be referred for each test.

The software used during the tests is Autodesk 3DsMax Design 2013 with MentalRay Renderer.

As mentioned in Chapter 3 this research has two main problems to solve. The first is reducing the network communication load of the render farm and the solution proposed for this is to compress the renderer output before sending from render farm worker to the render farm controller. The test results to measure the efficiency of this approach are evaluated in Section 5.1. The second main problem addressed in this research is to ensure load balancing while increasing the video compression ratio as much as possible and keeping the task submission overhead minimum. The results of tests about the approach addressing this problem is presented in Section 5.2

## 5.1 Test Results of Network Load Reduction Approach

To measure the network usage during this test a piece of network monitoring software is used. After running the network monitoring software on the machine that the render farm controller is running on, two test jobs have been submitted to the render farm controller and the rendering started. The jobs have been submitted to workers 1, 2, 3, 4, 5, 6 with the following adjustments:

**Job adjustments for test run #1:**

Video Encoding: Disabled
Output format: bmp
Output resolution: 1920x1080
Frames: 1-250

**Job adjustments for test run #2:**

Video Encoding: Enabled
Output format: mp4 (Codec: H.264[1])
Output resolution: 1920x1080
Frames: 1-250

For the first run, the video encoding is disabled while submitting the job and for the second run the video encoding was enabled. As can be seen from the above adjustments the only difference in these two test runs is the enable/disable status of the video encoding optimization. The reason for making the same test with this difference is to see the effect of video encoding approach on the network utilization.

[1] Please refer to Appendix B for compression presets of H.264 used in the test

Figures 5.1 and 5.2 are acquired from the network monitoring software and show the network load on the controller machine.

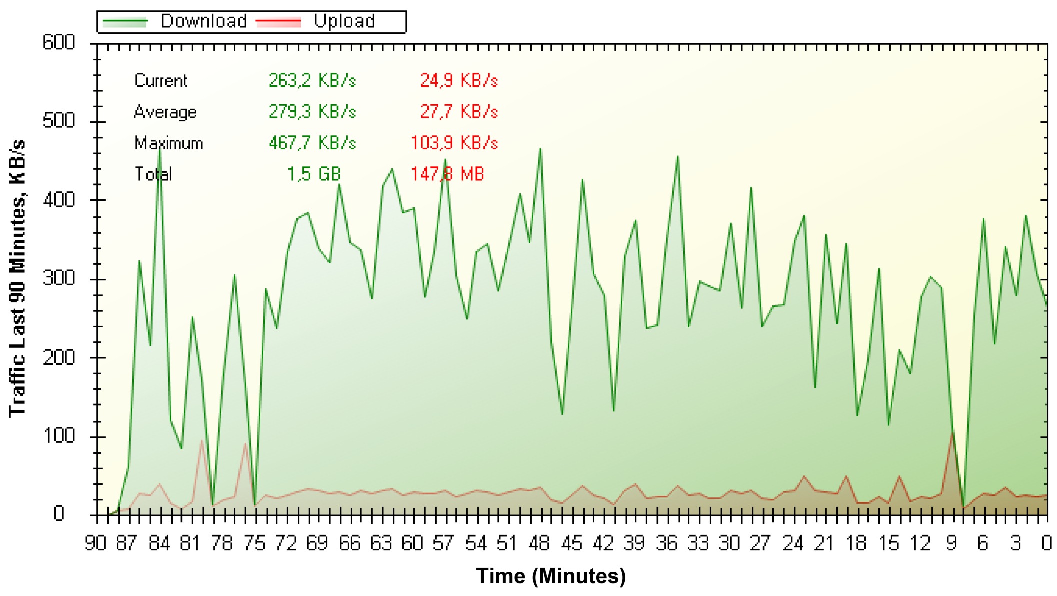


Figure 5.1: Network Load When Communication Cost Optimization is Disabled

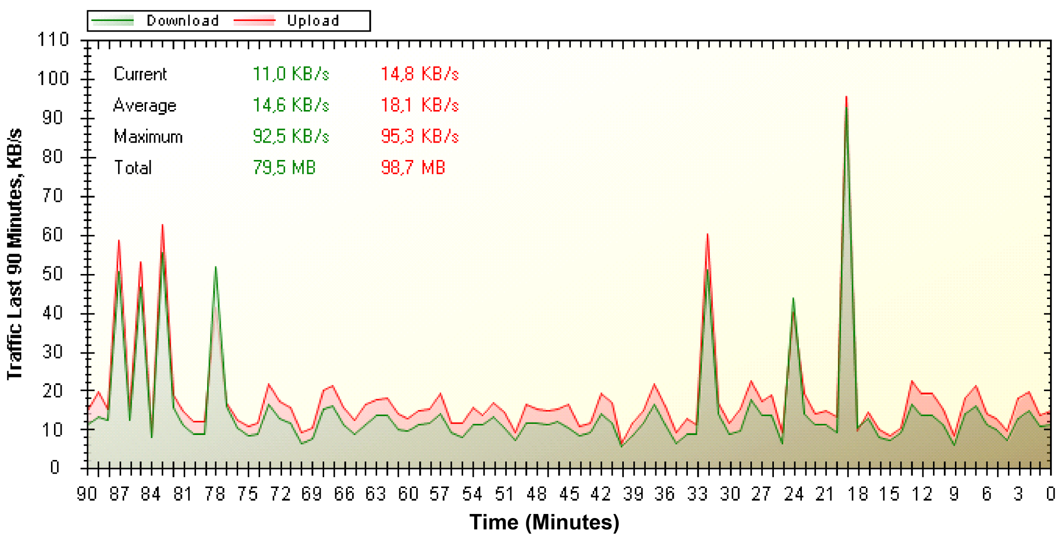


Figure 5.2: Network Load When Communication Cost Optimization is Enabled

The data shown in Figure 5.1 belongs to the test run #1. According to the results the total data flown from the workers to the controller is 1.5 gigabytes. When the output files, which are sent as separate images, are examined the total size of the files is calculated as 1.42 gigabytes (average bmp image file size was 5.9 megabytes), which means that 82 megabytes of network load is created by the procedural communications between the workers and the controller. This procedural communications are primarily status reports of the tasks from the workers to the controller. When Figure 5.2 is examined, it is observed that the total data flown into the controller machine is 79.5 megabytes in test run #2. The total size of the output files, which are sent as compressed videos, is calculated as 7.7 megabytes. This results show that the proposed approach for decreasing the communication cost has a great success. The outgoing network load caused by the render farm workers is decreased by 94.8% when the total data flow is considered. When the data flow caused by the procedural communication is subtracted from the total data flow, the success of this approach manifests itself more dramatically. In this way, it is observed that the data flow caused by the output file uploading is decreased by 99.5% by the video encoding approach. When the test run #1 is repeated with jpeg[2] as output format instead of “bmp” the percentages 94.8% and 99.5% becomes 53.5% and 90.4% respectively. However, the results acquired by bmp files are more important because the jpeg compression decreases the quality of the image files. After the rendering when the user applies the video encoding to the compressed jpeg files, the final quality of the video file will be lower than the quality of a video acquired by encoding the uncompressed bmp files. Therefore, in a production rendering bmp output file format would probably be more preferable since the quality is very important.

The optimization technique for decreasing the network cost in this research can be extremely beneficial in the following cases. First of all, if the render farm is built on a wide area network with a narrow band network connection, the overall performance will increase with this optimization. Because the output upload operations of the workers may accumulate to the end of job because of the low data transfer rates, even if the rendering is finished, the arrival time of the final product is delayed and thus the overall rendering time increases. Secondly, even if the network connection is not narrow band, high network load of a render farm can still be a problem. A campus network with a render farm distributed on it may be a good example for this issue. The worker computers of this render farm might be very distant when compared to a local area network. Therefore, the packets of the render farm do not only pass

[2] Compression quality parameter was set to 100 (best quality) and smoothness was set to 0 in the jpeg settings of 3DsMax

through switches but also through routers which might be affected by congestion. Moreover, so many other people might use the same network at the same time for the interactive applications such as video streaming or remote desktop connections. If the network load of the render farm is so high, these interactive applications will probably be affected. Therefore, decreasing the network load provides significant benefits for these kind of render farm setups too.

## 5.2 Test Results of Load Balanced Task Distribution Approach

The task distribution system tries to provide a better compression ratio to the video encoding approach used for network load optimization. This is done by applying a coarse grained task distribution. In other words, the frames of the animation are assigned to the workers as continuous groups which are as big as possible. However, as mentioned in Chapter 3 there are some tradeoffs. As the sub tasks get bigger, the system becomes more prone to load imbalance. The tests presented in Sections 5.2.1 and 5.2.3 have been done to verify that the proposed approach deals well with this tradeoff. The test presented in Section 5.2.2 is to evaluate the precision of cost prediction based on the temporal coherence. The test presented in Section 5.2.4 is to show the scalability of the system. Finally, section 5.2.5 shows a comparison of the overall rendering times of this system and the popular commercial render farm product “Backburner” with a case study.

### 5.2.1 Video Compression Ratio

The success of the approach proposed in Section 4.3 mainly depends on the compression ratio of the video encoding. Because of the tradeoffs mentioned in Section 3.1 the system may not always provide the most optimal compression ratio. Considering the nature of the task distribution algorithm, the output images of the animation are not combined into a single video at once. While producing a single video file at once provides the best compression ratio, the task distribution system produces several video files to be merged later. In other words, the workers produce video parts for the region that they are responsible from. For example, if a worker is assigned to a region from frame 100 to frame 130, at the end of the task it uploads a video file like “100-130.mp4”. Later, at the end of the job, these video files are merged to get the final video file. Consequently, the break offs in the videos might decrease the compression ratio. The question is how much it decreases. To measure this, the input file used in this case study is rendered on the render farm. Workers 1, 2, 3, 4, 5, 6 are used for this test. The test is repeated two times with the following adjustments:

**Job adjustments for test run #1:**

Video Encoding: Disabled
Output format: bmp
Output resolution: 1920x1080
Frames: 1-1000

**Job adjustments for test run #2:**

Video Encoding: Enabled
Output format: mp4 (Codec: H.264[3])
Output resolution: 1920x1080
Frames: 1-1000

After the output files of the test run #1 have been acquired, the total size of the output images is calculated as 6.22 gigabytes. On the other hand, the size of the final output file from the test run #2 is 16.3 megabytes. That means that the proposed approach of this research have provided 99.744% compression ratio. To see how much compression ratio is relinquished for load balanced task distribution, the output files acquired from the test run #1 are compressed into a single video file by a piece of video encoding software with the same quality level. The output size of this video file was 15 megabytes. That corresponds to 99.764% which is the maximum possible compression ratio for that output and quality level. As a result, it is observed that because of the break offs in the video files, the system loses 0.02% compression ratio.

If the test run #1 is repeated with jpg[4] as the output format, the total size of the output

[3] Please refer to Appendix B for compression presets of H.264 used in the test
[4] Compression quality parameter was set to 100 (best quality) and smoothness was set to 0 in the jpeg settings of 3DsMax

images is 377.5 megabytes. After encoding these jpg files into a single video file, same video with size 15 megabytes is acquired. This means that with jpg output file format the optimal video compression ratio is 96.02% and the video compression ratio of the render farm is 95.68%. Thus the loss in compression ratio for the jpg case is 0.35%.

As a result, in both cases the loss in compression ratio is negligible. This proves the idea mentioned in Section 4.4. According to that idea, the effect of temporal coherence on compression ratio does not necessarily requires the sub tasks to be as big as possible. As can be seen from Figure 4.4 the rise of the compression ratio slows down significantly after 50 frames in a video. In other words, a video with 1000 frames and 50 frames have very similar compression ratios because of the prediction window size of the H.264 video compression. Therefore, the success of the proposed approach in providing good video compression has been proved.

### 5.2.2 Cost Prediction and Remaining Time Estimation

Before assessing the load balancing performance of the proposed approach, the precisions of the cost prediction and the remaining time estimation should be tested, because the success of the load balancing approach proposed in this research directly depends on the precisions of the cost prediction and the remaining time estimation. As can be seen in Algorithm 2 in Chapter 4, the system first initializes a cost map containing a subset of the frames which are equally separated. After rendering these frames and interpolating the unknown costs, the initial cost map is acquired. The idea of interpolating this cost list is completely based on the temporal coherence. Using these costs estimations of this interpolated cost map, the remaining frames of the animation are distributed to the workers and later the load balance is continuously tested with the estimated remaining time comparison on the line 8 of Algorithm 4.

Firstly, the question is how reliable it is to decide cost of a frame from the known costs of the neighbor frames. To assess this approach the cost distribution graphs produced periodically by the render farm software have been used. Figures 5.3 and 5.4 are two of these graphs. They are produced from the test carried out with the workers 1, 2, 3, 4, 5, 6 and with the following job adjustments:

**Job adjustments for the test run:**

Video Encoding: Enabled

Output format: mp4 (Codec: H.264[5])

Output resolution: 1024x768

Frames: 1-3000

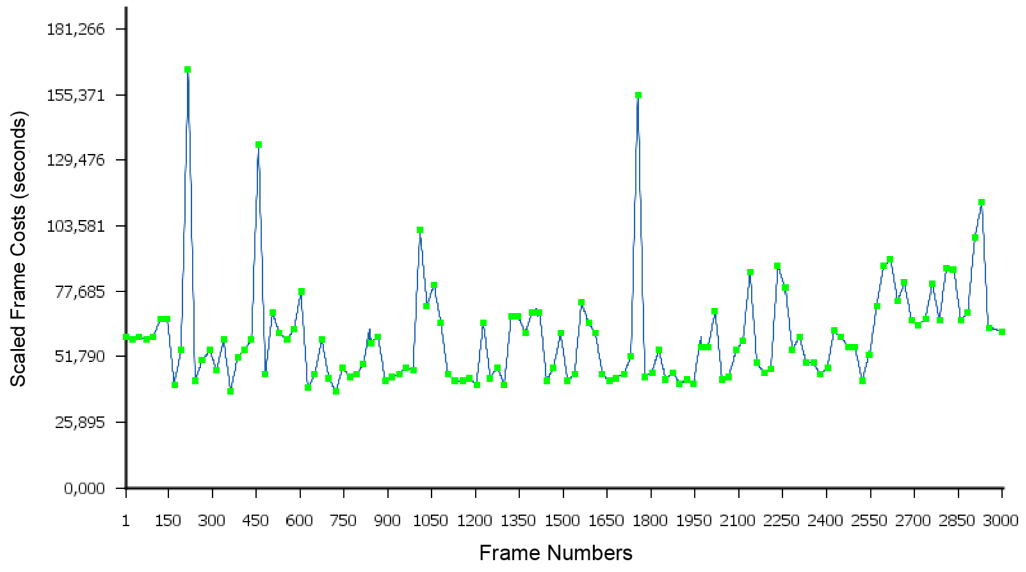


Figure 5.3: Initial Cost Map

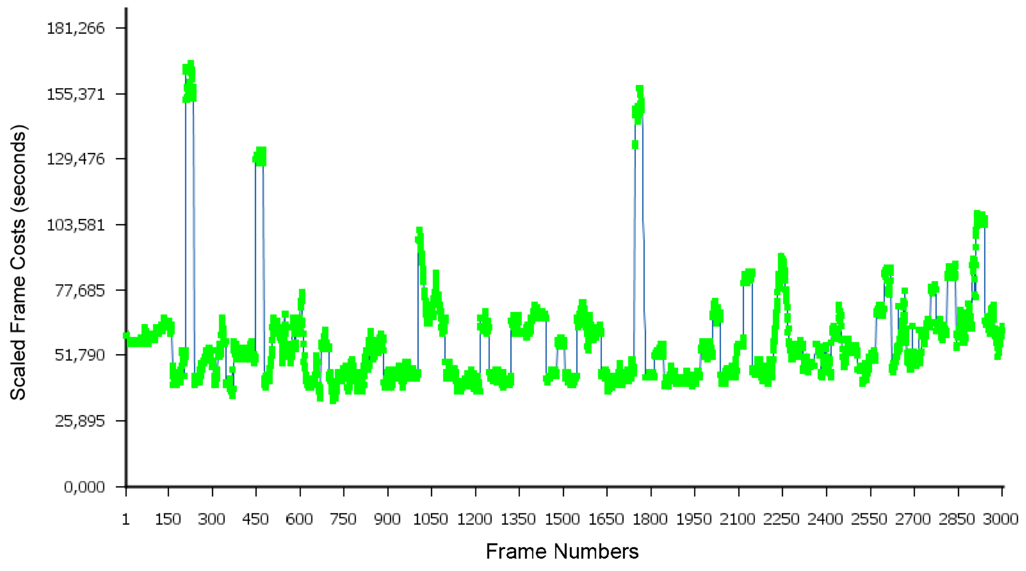


Figure 5.4: Cost Map at the End of the Job

[5] Please refer to Appendix B for compression presets of H.264 used in the test

Figure 5.3 shows the state of the cost map at the moment that the cost map frames are rendered and the costs of the remaining frames are determined by interpolation. On the other hand, Figure 5.4 shows the cost distribution at the end of the job. In other words, all cost data in Figure 5.4 is real. When these two graphs are compared visually, the consistency is very obvious. For example, as can be seen in Figure 5.4, the costs of the frames increase significantly around frames 200, 500 and 1800. The cause of this increase is the changes in the scene and the camera angle. When Figure 5.3 is examined, it is observed that the cost prediction system have predicted these increased costs in the early stages of the rendering. Besides these significant cost changes, the cost prediction approach have predicted even the smaller changes which can be observed around frame 1050.

Besides the visual assessment, the success of the cost prediction approach has also been tested numerically. For this assessment the deviation for each frame is determined. The deviation for each frame has been calculated with the following formula:

$$D = 100 \times \frac{|C_{real} - C_{predicted}|}{C_{real}}$$

where:
$D$: The deviation of cost for a frame
$C_{real}$: Real cost of the frame determined by rendering
$C_{predicted}$: Predicted cost of the frame determined by interpolation

These deviations are illustrated in Figure 5.5.

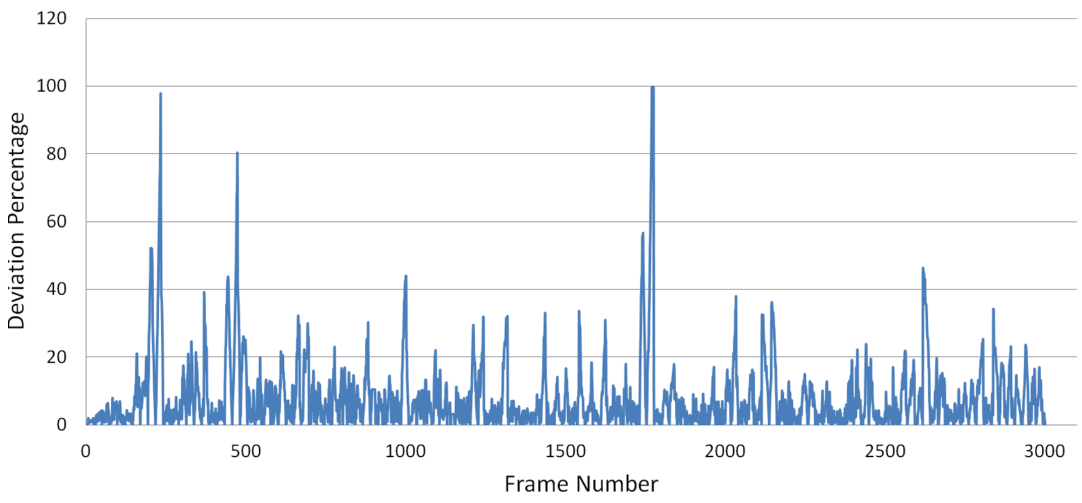


Figure 5.5: Deviations of Predicted Costs from Real Costs

In Figure 5.5 the majority of the deviations can be considered as small. The average of these deviations is 8.17%. Only at the scene change regions such as the regions around frames 200, 500 and 1800, there are some frames whose costs are predicted with a very high deviation. However, since this cost prediction method does not interested in the costs of individual frames, this will not affect the task distribution significantly. This can be explained with the following scenario. In this scenario there are two identical render farm workers with the same processing powers. The above mentioned job is assigned to these workers. Naturally, the task distribution system will try to divide the frames into two parts whose total costs are equal. The system will use the interpolated cost map for the first task distribution plan. According to this cost data, the boundaries of regions are 1-1523 and 1524-3000. On the other hand, if this partitioning is done by the real costs, the boundaries of the regions are exactly the same, which are 1-1523 and 1524-3000. This concludes that the proposed cost prediction is fairly reliable for this application because while creating a load balanced task distribution plan the cost of frame groups are considered instead of individual frame costs.

The second important aspect about cost prediction is the estimation of remaining time which is done by Algorithm 5 in Section 4.4. To assess the precision of this algorithm a test run has been carried out on the render farm workers 1, 2, 3, 4, 5, 6. The job adjustments of the test run are the following:

**Job adjustments for the test run:**

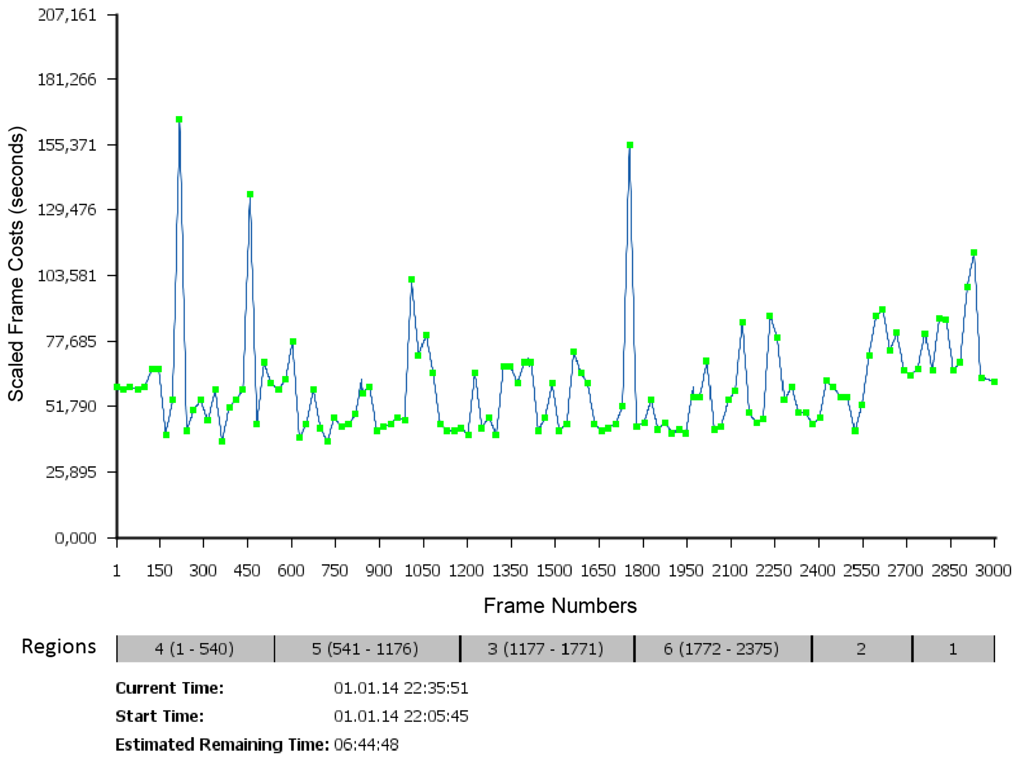
Video Encoding: Enabled

Output format: mp4 (Codec: H.264[6])

Output resolution: 1024x768

Frames: 1-3000

The render farm software starts displaying the estimated remaining time of the job when the first cost map is completed. The job status graph created by the render farm controller software at that time is given in Figure 5.6. The time is recorded as 22:35:51 and the software estimated the remaining time as 6 hours and 44 minutes.



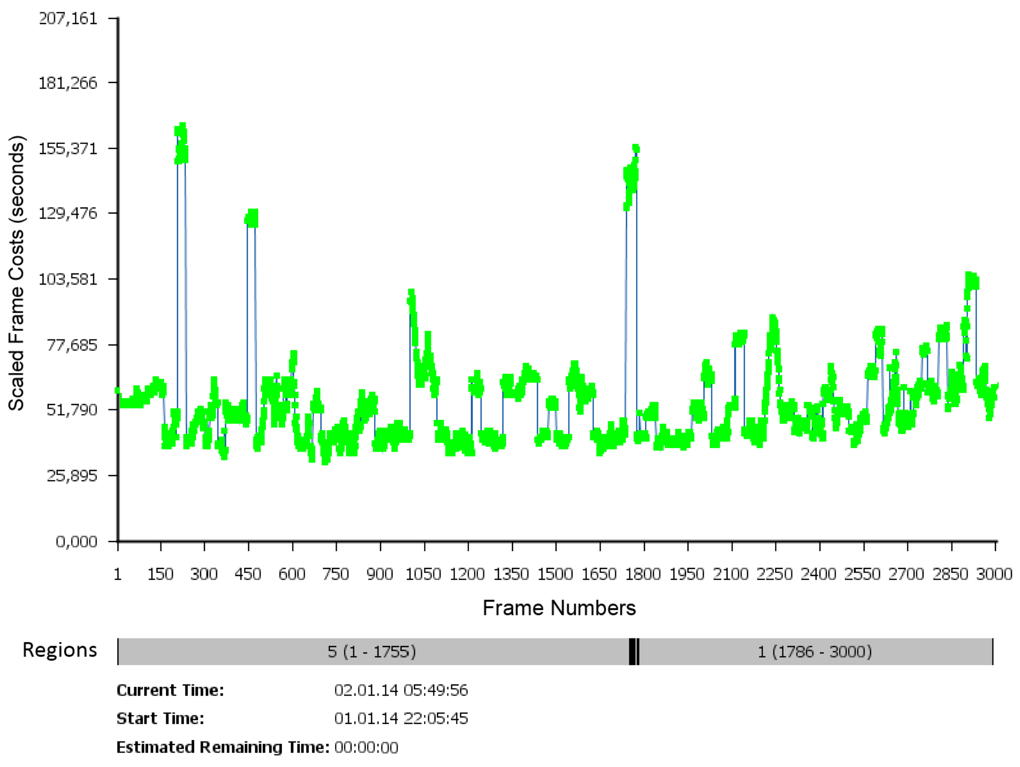
**Current Time:** 01.01.14 22:35:51

**Start Time:** 01.01.14 22:05:45

**Estimated Remaining Time:** 06:44:48

Figure 5.6: Estimated Remaining Time Test (After First Cost Map)

[6] Please refer to Appendix B for compression presets of H.264 used in the test



**Current Time:** 02.01.14 05:49:56
**Start Time:** 01.01.14 22:05:45
**Estimated Remaining Time:** 00:00:00

Figure 5.7: Estimated Remaining Time Test (End of Job)

When the job is finished the job status graph created by the render farm controller software is recorded again (Figure 5.7). The job is completed at 05:49:56. The real elapsed time is 7 hours and 15 minutes. Therefore, the error rate in remaining time estimation is 6.89% which is close to the 8.17% error in cost prediction.

### 5.2.3 Load Balance and Utilization

The utilization of a render farm can be discussed with two different criteria. The first one is the individual CPU usage percentages. The renderer named “MentalRay" is used for the tests of this study. During the tests it is observed that MentalRay uses the CPU with 100% utilization almost always during the rendering. Second criterion for utilization is about the load balance. As can be explained by Figure 3.4 in Section 3.2, if the tasks of a job are distributed without balancing the load, the workers which finish their tasks will wait for the workers which continue rendering. The amount of this unutilized time determines the render farm utilization. More precisely it has been calculated as follows:

$$U = 100 \times \frac{\sum_{i=1}^{w} t_i}{w \times T}$$

where:

$U$: The utilization of the render farm

$w$: The worker count

$t_i$: The total utilized time for worker $i$

$T$: The time difference between start of the job and finish of the last frame

**Job adjustments for the test run (With workers 1, 2, 3, 4, 5):**

Video Encoding: Enabled

Output format: mp4 (Codec: H.264[7])

Output resolution: 1920x1080

Frames: 1-1000

Table 5.1: Task Start Completion Times in Test Run

| Worker | Task ID | Task Start Time | Task End Time |
|---|---|---|---|
| 1 | 1 | 06:21:41 | 06:30:21 |
| 2 | 2 | 06:21:42 | 06:27:30 |
| 3 | 3 | 06:21:42 | 06:26:27 |
| 4 | 4 | 06:21:43 | 06:29:54 |
| 5 | 5 | 06:21:43 | 06:26:52 |
| . | . | . | . |
| . | . | . | . |
| 1 | 20 | 06:34:06 | 09:36:31 |
| 2 | 21 | 06:34:06 | 09:33:14 |
| 3 | 25 | 09:28:46 | 09:30:12 |
| 4 | 23 | 06:34:07 | 09:29:48 |
| 5 | 24 | 06:34:08 | 09:30:45 |

[7] Please refer to Appendix B for compression presets of H.264 used in the test

In Table 5.1 start times of the first tasks and the finish times of the last tasks are presented. As can be seen on the table the job start time is 06:21:41 and the job finish time is 09:36:31 which is the completion time of the last task. Therefore, the $T$ value is 11690 seconds. Total rendering time of workers $t_1$, $t_2$, $t_3$, $t_4$, $t_5$ are 11690, 11493, 11311, 11286, 11344 seconds respectively. Therefore, the individual utilization percentages of the workers are 100%, 98.31%, 96.75%, 96.54%, 97.04% respectively and the utilization of the whole render farm for this job is 97.73%.

### 5.2.4 Scalability

As mentioned before, the centralized control structure of render farms results in tendency to low scalability when the resources of controller machine is not increased as much as the resources of worker machines. In other words, if the count and the total processing power of the worker machines are increased while controller stays the same, the system will eventually suffer from some bottlenecks on the controller machine.

In this case study, the reaction of the system against the scaling is measured. Same job is rendered by various processing power configurations and the same controller machine as shown in Figure 5.8

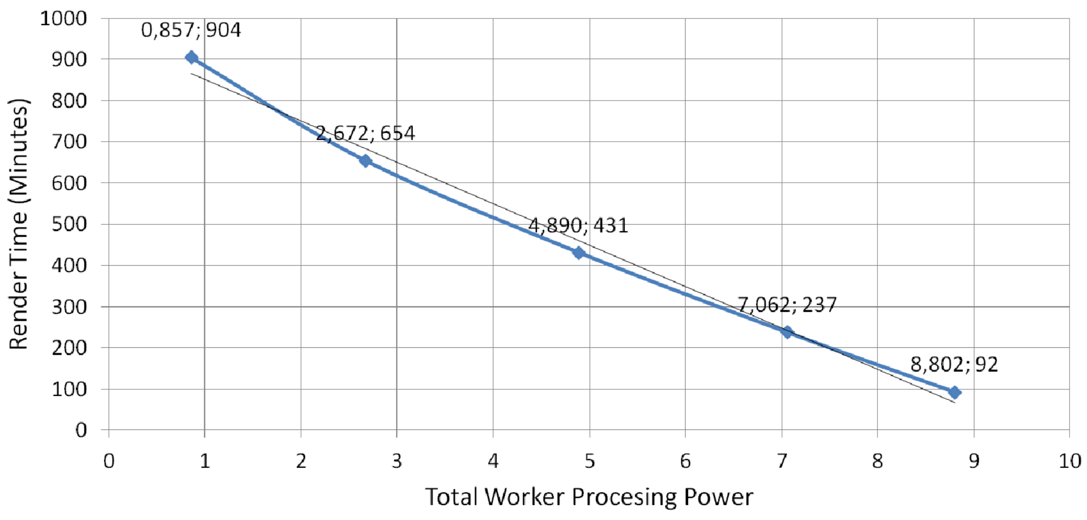


Figure 5.8: Change of Rendering Time with Increase in Worker Resources

The five test run in Figure 5.8 have been carried out by the 5 worker configurations shown in Table 5.2. The first column shows the number of the experiment, the second column shows the total processing powers of the workers which also corresponds to the horizontal axis of the graph in Figure 5.8, and finally the third column of the table shows the workers (Workers 1-10) in the experiment and their individual processing power points (PPP) determined by the render farm controller software.

Table 5.2: Worker Configurations in Scalability Tests

| **Exp.#** | **Total PPP** | **Workers (Individual PPP)** |
|---|---|---|
| 1 | 0.857 | 8(0.44), 9(0.41) |
| 2 | 2.672 | 3(1.34), 4(1.33) |
| 3 | 4.890 | 2(0.84), 3(1.34), 4(1.33), 5(1.37) |
| 4 | 7.062 | 1(0.85), 2(0.84), 3(1.34), 4(1.33), 5(1.37), 6(1.31) |
| 5 | 8.802 | 1(0.85), 2(0.84), 3(1.34), 4(1.33), 5(1.37), 6(1.31), 7(0.43), 8(0.44), 9(0.41), 10(0.45) |

The curve shown in Figure 5.8 is very close to linear as expected for this level of worker count. As the worker computer count increases this curve will probably start to be more horizontal because of the ratio of task submission overhead to the rendering time as explained in Figure 3.5. As long as the controller is not the bottleneck, the rendering time will continue to decrease while the worker count increases. However, it is obvious that a bottleneck will eventually occur in a centralized controlled distributed computing system if the communication bandwidth and processing power of the controller computer is not scaled proportional to the worker computers. Fortunately, since the communication costs are decreased significantly by the video encoding optimizations, the worker count at which this bottleneck occurs is very high and could not even be seen by the hardware resources of this case study.

### 5.2.5 Comparison with Backburner

Backburner is one of the most popular distributed rendering solutions. It is created by the company Autodesk which is also the creator of the major 3D design software

3DsMax. Actually, there is not any known optimization in Backburner for decreasing the network load. Therefore, there will not be any comparison about the network load optimizations. Instead, the job completion times will be compared. For this comparison the environmental factors should be eliminated. In other words, the tests have been done on the same hardware and software configurations. In addition, during the test there was not any other user on the computers who can share the CPU power with the renderers. The test has been carried out on workers 1, 2, 3, 4, 5, 6 and the same input file was used with the following adjustments:

**Job adjustments for the test run with the software of this research:**

Video Encoding: Enabled
Output format: mp4 (Codec: H.264[8])
Output resolution: 1024x768
Frames: 1-3000

**Job adjustments for the test run with Backburner:**

Video Encoding: (not available)
Output format: jpg[9]
Output resolution: 1024x768
Frames: 1-3000

The first test has been carried out with the software developed for this research. The job completion time was 7 hours 44 minutes and 38 seconds. The second test, on the other hand, has been carried out by using Backburner. The job completion time with Backburner increased up to 8 hours 10 minutes and 3 seconds. This corresponds to 5.47% speedup. A plausible explanation of this speedup is that Backburner maintains the load balance by using fine grained task distribution and thus the task submission overhead increases the job completion time. This explanation is stated as plausible instead of exact since the task distribution algorithm of Backburner is not known completely. However, the fine grained task distribution behavior can be observed on

[8] Please refer to Appendix B for compression presets of H.264 used in the test

[9] Compression quality parameter was set to 100 (best quality) and smoothness was set to 0 in the jpeg settings of 3DsMax

its monitoring interface.

Since the amount of task submission cost does not depend on the rendering time, the 5.47% speedup varies when the input job or adjustments changes. More precisely, if the rendering time increases the speedup ratio should decrease because the task submission overhead does not depend on rendering time. To verify this, two additional tests have been carried out. The environment and the job was exactly the same. The only difference was that the output resolution is raised up to 1920x1080 pixels. The software of this research achieves 12 hours 17 minutes and 37 seconds as rendering time. The rendering time of Backburner, on the other hand, was 12 hours 33 minutes and 33 seconds this time. As a result, the above idea has been verified since the speedup ratio was decreased to 2.16%.

The proposed approach of this research provides another speedup resulting from the video encoding. The resulting images of the above rendering with Backburner have to be encoded into video format on a single computer after the rendering is completed. The worker #1 of the render farm which has an Intel i7 CPU has completed this encoding process in 21 minutes and 40 seconds. That means the speedup provided by the proposed approach of this research is more than 2.16%. The final product of rendering was delayed almost 38 minutes with Backburner when the final video encoding is considered.

## 5.3 Usage of GPUs with the Proposed Approach of This Research

With the advances in the general purpose GPU computing, the number of GPU renderers also increased. The products “NVIDIA Optix”, “ARION”, “LIGHTWORKS”, “V-RAY” are good examples in this area. At this point a valid question arises: “Is GPU rendering technology applicable to the proposed approach of this research?”. The answer is 100% positive. Because the techniques explained in this study have an entity called render farm worker. The work is distributed among the render farm workers which are separate computers connected with network. The system does not require the information about what type of renderer is inside the worker computer as long as it gets the result. Therefore, even the processing power measurement technique is independent from the usage of GPU or CPU. It only requires the result creation time of a computer.

## 5.4 Usage of Hadoop and MapReduce with the Proposed Approach of This Research

As mentioned above even though the amount of data transmitted through the network connections is decreased by the video encoding optimization, there will be a bottleneck eventually at some point as the render farm scale increases. In their study of Liu et. al. [19] propose a method for a similar I/O bottleneck problem. Specifically, they proposed an approach for building a render farm on HDFS (Hadoop Distributed File System). The conventional method for storing the scene data is keeping it on a media with NFS (Network File System) and accessing directly via network paths. This method is very prone to bottlenecks [19]. By using the distributed file system HDFS, Liu et. al. have achieved optimizing the process of accessing the scene data. Proposed approach of Liu et. al. distributes the data over a number of data nodes on different machines. Thus, the I/O load for this data is also distributed. However, even though there won't be local bottlenecks with this approach, the load on the network interfaces may still be very high. As a result, applying both the video encoding optimization and HDFS may provide even better scalability for render farms.

# CHAPTER 6

# CONCLUSION

This research aimed to improve the utilization, load balance and scalability of render farms by solving the high communication load and load balancing issues. The high communication load issue occurs because of the centralized control in the render farms. Throughout the rendering job the render farm workers upload the result images to the render farm controller. While the networking hardware of a worker machine only deals with its own network packets, the networking hardware of the render farm controller deals with all the network packets coming from large number of workers. This results in network traffic imbalance between the controller and the workers. This imbalance results in a bottleneck on the controller machine if the worker count is high enough. Unfortunately, with a centralized render farm controller probability of this bottleneck cannot be overcome. Instead, the problem may be solved by trying to decrease the size of the data that is sent over the network. This research proposes to reduce the output size by merging the output images of the rendering into a video file compressed by H.264 codec. Although compressing the data before sending to controller is a solution which is applicable to any other distributed processing environment, the case of render farm is special because of the temporal coherence. The H.264 codec compresses the motion picture by several different techniques. One of those techniques is called "Prediction" and it is the most important one among them. The prediction technique takes advantage of the temporal coherence in motion pictures and encodes only a small portion of the images. By using H.264[1] encoding the output images are compressed up to 99.5%. The technique of Chong et. al. [5] addresses this high network load issue by hashing the geometric objects. Naturally the

[1] Please refer to Appendix B for compression presets of H.264 used in the case study

hash value of an object will be very small compared to the object itself. Therefore, by sending only one copy of the object and the hash values whenever necessary reduces the network traffic significantly. According to the test results presented in the paper of Chong et. al. [5], this technique can achieve compression ratios up to 97% for the scene input. The technique of this research, on the other hand, compresses the output data. As a result using the both methods together can achieve even better results.

The case study of the research has been carried out by the H.264 compression settings presented in Appendix B. Although these settings provide images that are nearly lossless to the eyes, theoretically they are not 100% lossless. Although, the H.264 video encoder provides a lossless compression too, this decreases the compression ratio significantly. If the lossless compression is applied (when "crf" parameter of H.264 is set to 0), the proposed approach may not be able to provide a valuable optimization level for communication cost. However, according to the official guide of the ffmpeg for video encoding, setting the "crf" value as 18 instead of the default value 23 gives results which are extremely close to lossless and it still provides a significant compression. Therefore, even though lossless compression is not possible with this approach, by providing a user interface for adjusting the "crf" setting of the encoder, the user may obtain output data according to his/her needs, as long as completely lossless output is not required.

Without the temporal coherence this high compression ratio could not be achieved. For example, a web server also has a similar structure which is basically a server and some number of connected clients. The difference is that the centralized web server feeds the clients with the data this time. Therefore, the same bottleneck problem may occur and the same problem may be addressed by compressing the data. Although, data compression solution will help with this problem too, it may not be as efficient as the compression in the render farm because there is no temporal coherence in the data of web servers unlike the render farms.

The second problem addressed in this research is the computational load balancing of the render farm workers. Although the fine grained task distribution provides perfect load balancing naturally, in this research it cannot be used because of the solution of the first problem. The video compression technique which is based on the temporal coherence gives better compression ratio when the number of consecutive frames that feed the video increases. Therefore, to get a good video compression ratio, bigger continuous sub tasks should be assigned to the workers at once. That task distribution scheme is called coarse grained task distribution. Unfortunately, coarse grained task distribution brings the high possibility of load imbalance. To solve this problem the proposed approach of this research tries to partition the job into sub tasks which have costs proportional to the powers of the workers that they will be assigned to. At that point the load balancing problem is reduced to two sub problems which are determining the processing powers of the workers and cost distribution of the job. The processing powers of the render farm workers are measured by a hybrid technique which both utilizes the benchmarking results and the statistical background of the workers. The cost distribution of the job, on the other hand, is predicted by and interpolation method based on the temporal coherence. More precisely, the system first renders a subset of the job which consists of equally separated sparse frames. Then the costs of the frames between these frames are predicted by interpolating the cost graph. For this problem also the temporal coherence is crucial. Without the temporal coherence property of the motion pictures, this cost prediction solution may not be possible.

Although, the other researchers like Reinhard et. al. [17] and Gillibrand et. al. [7] propose some extremely elegant cost prediction techniques, they are suitable for the load balancing in a real time distributed rendering environment, because their techniques reveal the cost distribution inside a frame. Whereas, the cost distribution inside a frame is not important for a render farm working on a non-real-time animation job, because in this kind of applications the deadline is the end of the job instead of a frame. Therefore, while rendering an animation the cost distribution in the whole animation is important. This fact results in the requirement of a cost prediction method like proposed in this research. The proposed cost prediction technique in this research achieved 91.83% correctness in the case study. When the results of the cost prediction are used together with the processing power information of the workers, the system acquires the ability to estimate the remaining time of a job. In the case study, the proposed approach estimated the remaining time with 93.11% correctness.

Another successful outcome of the case study was the comparison with Backburner which is the popular product of the Autodesk for render farms. Even though the speedup of the rendering process was small (approximately 2%-5%) the proposed approach of this research has a great advantage over Backburner. There is currently no render farm solution that provides distributed video encoding for the animation jobs. They can only produce still image files for each frame and the merging process of these frames is left to user. The output images acquired from one of the test runs of this case study has taken 21 minutes to merge into a video on an Intel i7 CPU. That means user of the Backburner will have to wait another 21 minutes before getting the final product. On the other hand, the software of this research distributes the video encoding process to the render farm workers. Therefore, the final product video becomes ready as soon as the rendering finishes.

## 6.1 Future Work

One of the most important parts of this research is the estimation of the remaining time and cost prediction. Having the ability to estimate the remaining time of a rendering job, so many great advances in this area can be achieved. One of these advances is about the fair distribution of the rendering resources in a multi user render farm. Today's regular render farms work as rendering queues where only one rendering job uses the resources at the same time. However, a company servicing with a render farm to other customers may want to distribute the resources according to the job rendering time, due date and payment. In other words, a customer with a high payment and tight due date may have a bigger portion of the rendering resources than other users, if other users pay less and their due dates are not soon. In this area there is the important study of Abramson et. al. [2] that discuss their efforts in developing a resource management system for scheduling computations on resources distributed across the world with varying quality of service (QoS). With the knowledge of rendering cost of an animation this approach may be applied to a render farm too.

# REFERENCES

[1] F. Abraham, W. Celes, R. Cerqueira, and J. L. Campos. A load-balancing strategy for sort-first distributed rendering. *17th Brazilian Symposium on Computer Graphics and Image Processing*, pages 292–299, October 2004.

[2] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, October 2002.

[3] D. P. Anderson, C. Christensen, and B. Allen. Designing a runtime system for volunteer computing. *SC 2006 Conference, Proceedings of the ACM/IEEE*, page 33, November 2006.

[4] A. Appel. Some techniques for shading machine renderings for shading. *FIPS Conference Proc.*, pages 37–45, April 1968.

[5] A. Chong, A. Sourin, and K. Levinski. Grid-based computer animation rendering. *GRAPHITE '06 Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 39–47, November 2006.

[6] J. Ericson. Processing avatar. *Information Management Magazine*, December 2009.

[7] R. Gillibrand, P. Longhurst, K. Debattista, and A. Chalmers. Cost prediction for global illumination using a fast rasterised scene preview. *AFRIGRAPH '06 Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 41–48, 2006.

[8] C. González-Morcillo, D. Vallejo, J. Albusac, L. Jiménez, and J. J. Castro-Schez. A new approach to grid computing for distributed rendering. *International Conference on P2P, Parallel, Grid, Cloud, and Internet Computing*, pages 9–16, October 2011.

[9] C. González-Morcillo, G. Weiss, L. Jimenez, and D. Vallejo. A multi-agent approach to distributed rendering optimizationlti agent approach to distributed rendering optimization. *Proc. 19th Annual Conf. on Innovative Applications of Artificial Intelligence (IAAI 2007)*, pages 1775–1780, July 2007.

[10] C. González-Morcillo, G. Weiss, D. Vallejo-Fernández, L. Jiménez-Linares, and J. Albusac-Jiménez. 3d distributed rendering and optimization using free soft-

ware. *Proceedings of the International Conference on Free/Libre/Open Source Systems (FLOSS, pp. 52–66)*, pages 52–66, March 2007.

[11] S. L. Gooding, L. Arns, P. Smith, and J. Tillotson. Implementation of a distributed rendering environment for the teragrid. *Challenges of Large Applications in Distributed Environments, IEEE*, pages 13–21, 2006.

[12] Z. Ji and B. He. A dynamic load balancing method for parallel rendering and physical simulation system based sort-first architecture. *International Conference on Computer Science and Network Technology*, pages 1792–1796, December 2011.

[13] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE*, 14(4):23–32, July 1994.

[14] N. S. Narkhede and N. Kant. The emerging h.264/advanced video coding standard and it's applications. *Proceeding ICAC3 '09 Proceedings of the International Conference on Advances in Computing, Communication and Control*, pages 300–305, 2009.

[15] L. Neumann and A. Neumann. Radiosity and hybrid methods. *ACM Transactions on Graphics*, 14(3):233–265, July 1995.

[16] M. Z. Patoli, M. Gkion, A. Al-Barakati, W. Zhang, P. Newbury, and M. White. An open source grid based render farm for blender 3d. *Power Systems Conference and Exposition, PSCE '09 IEEE/PES*, pages 1–6, March 2009.

[17] E. Reinhard, A. J. F. Kok, and F. W. Jansen. Cost prediction in ray tracing. *Eurographics*, pages 44–50, June 1996.

[18] T. Whitted. An improved illumination model for shaded display. *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, 23(6):343–349, June 1979.

[19] W. Yiu, B. Gong, and Y. Hu. A large-scale rendering system based on hadoop. *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference*, pages 470–475, 2011.

[20] Y. zhen Cao, B. feng Lu, and Q. Wang. Asymmetric distributed animation rendering system based on wan. *IEEE International Conference of Cognitive Informatics (ICCI)*, pages 415–420, July 2010.

# APPENDIX A

# SAMPLE RUN OF 3DSMAXCMD.EXE

The developed render farm application invokes the renderer software with terminal commands. Below is a sample command to invoke 3DsMax to render frame 1 of the scene “input.max” with 800x600 resolution.

```
>   3dsmaxcmd

         -workPath:C:\workspace

         -v:5 -showRFW:0

         -continueOnError

         -outputName:C:\out.png

         -width:800 -height:600

         -start:1

         -end:1

         C:\input.max
```

The command also tells that continue if possible when an error occurred (`-continueOnError`), do not show render frame window i.e. work in background, (`-showRFW:0`) and work with verbosity level 5 (`-v:5`)

The below sequence is the output of 3dsmaxcmd.exe called from command-line with the command presented above. The timestamps in the sequence can be useful to see the proportions of times spent to initialize the software itself, spent to initialize plug-ins, spent to load and translate the scene and the rendering itself. The moment of invocation and the real time of the process termination is written to the first and last lines.

The time difference between the last log at 28.07.2013 16:53:43 and the line "`At 28.07.2013 16:53:54 control process is terminated.`" shows the time spent to cleanup and finalization process. This time also has an effect on the total processing time of a render job.

```
At 28.07.2013 16:52:00 below command is executed...

>   3dsmaxcmd

         -workPath:C:\workspace

         -v:5 -showRFW:0

         -continueOnError

         -outputName:C:\out.png

         -width:800 -height:600

         -start:1

         -end:1

         C:\input.max

28.07.2013 16:52:01; Parsing scene:  C:\input.max
28.07.2013 16:52:01; 1 frames initialized
28.07.2013 16:52:01; Max install location:  C:\Program Files (x86)
\Autodesk\3ds Max Design 2013
28.07.2013 16:52:01; Max file being rendered:  C:\input.max
28.07.2013 16:52:01; Renderer:  NVIDIA mentalray
28.07.2013 16:52:19; [V-Ray] ===========================================
28.07.2013 16:52:19; [V-Ray] Console created, V-Ray A for x86 Feb 16 2013
28.07.2013 16:52:19; [V-Ray] ===========================================
28.07.2013 16:52:19; [V-Ray] Compiled with Intel C++ compiler, version 12.1
28.07.2013 16:52:19; [V-Ray] Host is 3dsmax, version 15
28.07.2013 16:52:19; [V-Ray] V-Ray DLL version is 2.00.01
28.07.2013 16:52:22; Max is ready
28.07.2013 16:52:23; Frame 1 assigned
28.07.2013 16:52:28; MENTAL RAY LOG: 0.4 94 MB warn 082058:
```

```
jitter and contour rendering are incompatible, using jitter 0.0
28.07.2013 16:53:43; Frame 1 completed; Elapsed time 00:01:20
28.07.2013 16:53:43; Job Complete - Results in C:\
28.07.2013 16:53:43; Send End of Job command to Max
28.07.2013 16:53:43; Job Completed with Warning(s) - see above
28.07.2013 16:53:43; Total elapsed time 00:01:21
At 28.07.2013 16:53:54 process is terminated.
```

The software configuration of the test platform is as follows:

- Operating System: Microsoft Windows 7 Service Pack 1 x64
- 3DsMax: AutoDesk 3DsMax Studio 2013 x86
- 3DsMax Installed Third Party Plug-ins: Only ChaosGroup VRay x86 2.00.01
- 3DsMax Used Renderer: MentalRay

The hardware configuration of the test platform is as follows:

- CPU: Intel Core i5 2.5 GHz
- Memory: 8 GB 1600 MHz DDR3
- Storage: Solid Stated Drive with 430 MB/s read speed and 330 MB/s write speed

# APPENDIX B

# H.264/MPEG-4 AVC ENCODING SETTINGS

Following presets are used when encoding a video during the case study.

| | |
|---|---|
| profile : | Not Set |
| preset: | medium |
| tune: | Not Set |
| slow-firstpass: | Not Set |
| keyint: | 250 |
| min-keyint: | Auto |
| no-scenecut: | Not Set |
| scenecut: | 40 |
| intra-refresh: | Off |
| bframes: | 3 |
| b-adapt: | 1 |
| b-bias: | 0 |
| b-pyramid: | Norrmal |
| open-gop: | None |
| no-cabac: | Not Set |
| ref: | 3 |
| no-deblock: | Not Set |
| slices: | 0 |
| slice-max-size: | 0 |
| slice-max-mbs: | 0 |
| constrained-intra: | Not Set |

```
pulldown: none
fake-interlaced: Not Set
frame-packing: Not Set
qp: Not Set
bitrate: Not Set
crf: 23
rc-lookahead: 40
vbv-maxrate: 0
vbv-bufsize: 0
vbv-init. 0.9
crf-max: Not Set
qpmin: 0
qpmax: 51
qpstep: 4
ratetol: 1.0
ipratio: 1.40
pbratio: 1.30
chroma-qp-Offset: 0
aq-mode: 1
aq-strength: 1.0
no-mbtree: Not Set
qcomp: 0.60
cplxblur: 20
qblur: 0.5
me: ’hex’
merange: 16
mvrange: -1 (Auto)
mvrange-thread: -1 (Auto)
subme: 7
psy-rd: 1.0:0.0
no-mixed-refs: Not Set
no-chroma-me: Not Set
no-8x8dct: Not Set
```

```
trellis: 1
colorprim: undef
transfer: undef
colormatrix: undef
chromaloc: 0
nal-hrd: None
pic-struct: Not Set
crop-rect: Not Set
muxer: Auto
demuxer: Auto
input-csp: i420
output-cspt: i420
input-range: Auto
psnr: Not Set
ssim: Not Set
```

**TEZ FOTOKOPİ İZİN FORMU**

**ENSTİTÜ**

| | |
|---|---|
| Fen Bilimleri Enstitüsü | ☐ |
| Sosyal Bilimler Enstitüsü | ☐ |
| Uygulamalı Matematik Enstitüsü | ☐ |
| Enformatik Enstitüsü | ☐ |
| Deniz Bilimleri Enstitüsü | ☐ |

**YAZARIN**

Soyadı : ..........................................................................................................................
Adı : ..........................................................................................................................
Bölümü : ..........................................................................................................................

**TEZİN ADI** (İngilizce) : ..........................................................................................................
..........................................................................................................................................
..........................................................................................................................................
..........................................................................................................................................
..........................................................................................................................................

**TEZİN TÜRÜ** : Yüksek Lisans ☐ Doktora ☐

Tezimin tamamı dünya çapında erişime açılsın ve kaynak gösterilmek şartıyla tezimin bir kısmı veya tamamının fotokopisi alınsın. ☐

Tezimin tamamı yalnızca Orta Doğu Teknik Üniversitesi kullancılarının erişimine açılsın. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.) ☐

Tezim bir (1) yıl süreyle erişime kapalı olsun. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.) ☐

Yazarın imzası ............................ Tarih .............................