A Review Focused Cloud Render System for 3D CGI Production

SW10 Semester Project
Group ds1010f20
Software
University of Aalborg
June 12, 2020

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfsvej 300
Phone: +45 99 40 99 40
Fax: +45 99 40 97 98
http://www.cs.aau.dk

Title:

A Review Focused Cloud Render System for 3D CGI Production

Project:

P10

Project period:

February 3 2020 - January 10 2020

Project group:

ds1010f20

Authors:

Kim Larsen
Martin Fabrin Karkov

Supervisor:

Brian Nielsen

Pagecount: 58
Finished: June 12, 2020

Abstract:

To improve the review process in 3D CGI production for creators, we investigate approaches of analysing a path-tracing render task, due to the stochastic nature of path-tracing. From this investigation we propose a design of a render farm system with additions targeting the review process. From the proposed design, we developed a minimal viable product. From evaluations of the minimal viable product, we find an improved system, to our previously developed render farm system, both in terms of its cost and accuracy. Additionally, we find the proposed system to be controllable and transferable to existing render farms. Thus, we show a system with both potential and attractiveness towards existing render farms, to improve upon the review process for creators.

Preface

This project has been created by group ds1010f20. The group is made up of two tenth-semester software students (forth semester of the Master's Degree) from Aalborg University. The rapport represents the last half of our Master Thesis.

The source code for this and ninth project can be found at: `https://github.com/thinhar/P9-reviewRender`

The report contains an appendix, which will be referenced within the report whenever relevant.

We will like to thank Brian Nielsen for supervising us on the project.

Martin Fabrin Karkov
<mkarko15@student.aau.dk>

Kim Larsen
<klars15@student.aau.dk>

Acronyms

AMQP Advanced Message Queuing Protocol
BVH Bounding Volume Hierarchy
CGI Computer-generated imagery
MVP Minimum Viable Product
VFX Visual Effects
WIP Work in Progress

Table of contents

Introduction 1

In the Information Age, the information available has become beyond the grasp of individuals. Thus, information is lost as a result of the seemingly unending stream of information. Meaning, to get consumers to pick up specific information, a vehicle to distribute information is vital. Images, in tandem with text and/or speech, is used as vehicles for sharing information. Images generated by a computer is called Computer-generated imagery (CGI). CGI is used in a multitude of different aspects in our society e.g. commercials, entertainment, documentaries and textbooks. The presence of CGI is so prevalent in today's society that most citizens in the developed countries sees CGI, in their everyday life.

The influential presence of CGI in today's society, has resulted in the establishment of CGI communities and enthusiasts, beyond the field of professionals. In addition to the advent of compute technology, tools have become available with free and open source programs e.g. Blender and GIMP, that makes the entry to CGI production even more accessible.

Because of the broad range of CGI developing entities, in both budget and developer experience, the production workflow for CGI varies among all entities. Henceforth we will refer to the people associated with CGI production as creators. Furthermore, any creation of CGI has to conform to its creators' expectations. A popular approach to conform to these expectations is to do multiple minor reviews of the Work in Progress (WIP). This allows creators to confirm the state of several aspects regarding the WIP, thus allowing fine-tuning in accordance to their expectations. Comprehendingly, a review process is generally incorporated into the production workflow, to maximise the efficiency of reviews.

1.1 Motivation

This section delineates the problem and gives the motivation for the project by establishing information on the traditional 3D production workflow. That is commonly used as the basis for industry workflows [1], to create CGI. Additionally, we discuss its difficulties regarding the review process.

Figure 1.1 illustrates the traditional stages in 3D production and Description 1.1 the corresponding description of each stages' implication. The last stage in 3D production is rendering, that computes the graphical representation of the scene, into images. Where a scene is a collection of graphical objects, each with its own attributes, i.e. colour and geometric shape. The generated images then goes through a review process, and upon approval they are forwarded to post-production [2, 3], that includes: i.a. compositing, sound editing, colour correction. However, if the images are not approved, they are sent

Layout: The conceptualisation of ideas and the creation of the storyboards that translate these ideas into visual form.
R&D: Preparing for future tasks in the production. E.g. the scene needs an explosion from a specific type of source, like a missile, the Visual Effects (VFX) artists study missiles to understand how the fire and smoke behaves. Once they understand the behaviours, they can create e.g. tools to efficiently work on the shots requiring an explosion from a missile.
Modeling: The process of moulding primitive shapes into completed 3D meshes. Such as shaping a cube into a 3D model of a door that can be refined and detailed subsequently.
Texturing: The process of overlaying images onto a 3D model by mapping which part of the images should overlay which part of the 3D model.
Rigging/Setup: The process of creating a controllable skeleton for a 3D model that is intended for animation. The design of the skeleton corresponds to the set of controls the 3D model are expected to perform e.g. jumping, running and walking.
Animation: The process of altering a 3D model, by changing its position in the scene or its graphical appearance e.g. size and form between frames.
VFX: The process of creating elements too complex and difficult to animate. E.g. hair where each strand of hair would require an individual animation and accompanying skeleton to allow for dynamic movement.
Lighting: The process of adding lighting objects to a scene to make objects visible for the camera. In addition to counter objects from appearing flat by simulating how lighting works in the real world.
Rendering: The process of computing the graphical representation of the scene.

Description 1.1: Each stages' implication in the traditional 3D production workflow

back to the previous stage in production, where the flaw was diagnosed. This repetitive process, repeated for each phase in the production workflow, is illustrated in Figure 1.2. Thus, in a traditional 3D production workflow, the review process is a repeated routine. This routine is crucial in validating the generated images, according to the standards of the creators' expectations. Consequently, this also means creators have to address three interconnected aspects of the repeating reviews; delay of rendering, the *visual quality* and the *computational requirement*. Delay of rendering has an intrinsic correspondence to the choice of the *visual quality* and the *computational requirement* required for the review. Where, the *visual quality* is required frame rate, resolution etc. for the review to bring the desired return value. And the *computational requirement* is the required CPU time, to uphold the *visual quality*.

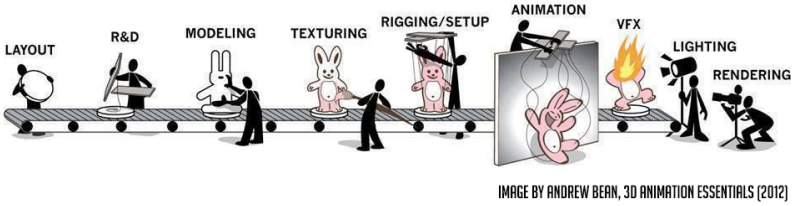


Figure 1.1: The traditional 3D production workflow

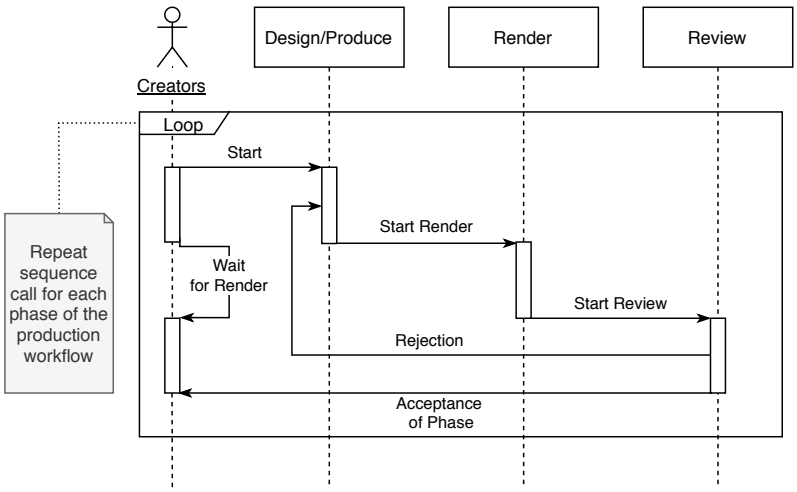


Figure 1.2: Sequence diagram illustrating the inner steps of the 3D production workflow

1.1.1 Delay of Rendering

The delay caused by a rendering of most stages in the traditional CGI workflow is often small enough that creators can review frames as they are getting rendered. Meaning, frames are rendered in a high enough frame rate, that the delay is not an obstacle for creators' review process. However, a rendering of some stages such as *Animation*, *VFX*, and *Lighting* requires computationally heavy techniques resulting in increased render times. Hence, delaying the review process and the entire workflow. To counteract this delay, render tasks with a too high *computational requirement* for a single machine are rendered using a render farm. A render farm is a cluster of networked machines, each devoted to rendering fragments of a task. Consequently, by using a render farm the intermediate results from rendering are not present on creators' local machines, but spread across multiple networked machines. Because of this, creators using a render farm do not get an usability-wise review experience similar to a local experience. To emulate the local review experience, render farms can sequentially stream the rendered frames to the creators' local machine.

1.1.2 Visual Quality and Computational Requirement

In extension to the delay caused by rendering, creators also must address two important aspects of the review process itself. In particular, the *visual quality* required for the review to bring the desired return value and the associated *computational requirement* of this *visual quality*. With high *visual quality* implying a high *computational requirement* e.g. a 4K ultra HD image would require more computational power than a Full HD image. Both aspects are essential for working efficiently in a 3D production environment [1, 4]. As too low *visual quality* may yield unacceptable returns, i.e. unusable for review. Where too high *visual quality* results in increased delays in the workflow, as illustrated in Figure 1.3a. Thus, finding the balance between the *visual quality* required for an acceptable review and corresponding *computational requirement* to meet production deadlines, is the key to working efficiently.

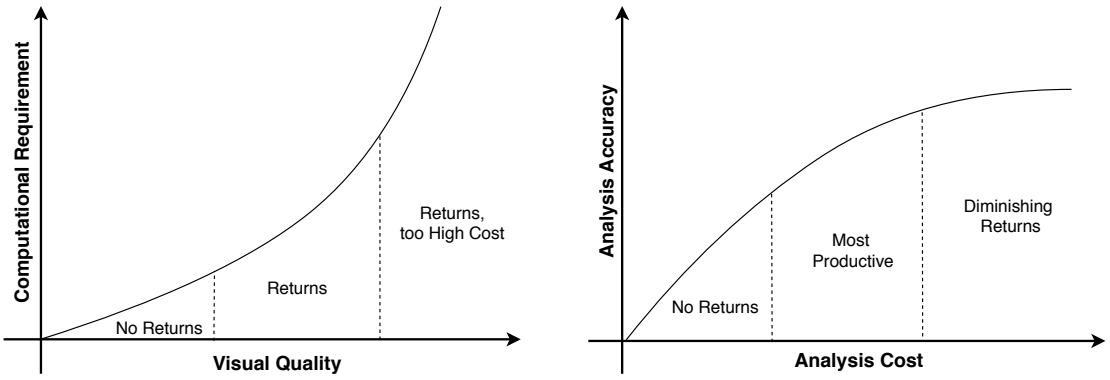
Additionally, as render farms normally work on multiple concurrent tasks, from different developing entities, the available resources varies. In addition to resources, in traditional

render farms, being distributed using simple scheduling disciplines such as first in first out (FIFO), and round-robin. Consequently, this means that the *computational requirement* of a task is not considered, during the distribution of resources. Thus, creators may receive unacceptable and unstable frame rates for their review.Hence, the render farm does not always uphold a *visual quality* that is adequate for the creators' review.

Therefore, a render farm that can determine the *computational requirement* of tasks before it distributes resources, will yield a better review experience for creators, similar to that of local rendering.

A method to determine the *computational requirement*, is to asses, via an analysis, what should be allotted to the task. For local rendering, the creator can do this from experience. However, due to the cloud's resources not being as stable, this can be more difficult for the creator. For this reason we suggest an additional feature for render farms, to ameliorate creators from deciding the *computational requirement*. This feature takes the form of an automatic analysis of each submitted task.

Consequently, such an automated analysis can not be automated without introducing a cost, either in terms of an additional delay of rendering, and/or an increase in resource usage. However, as render tasks are stochastic processes, as delineated in section 2.1, the task in its entirety needs to be rendered to accurately determine its render time. This would however defeat the purpose of an analysis, therefore, a balance between accuracy and cost should be found. We can model this dilemma as a case of diminishing returns as illustrated in Figure 1.3b. The figure shows the trade-off between cost and accuracy of the analysis becomes increasingly worse as the cost increases. Additionally, as this balance depends on the hardware, and the required accuracy is subjective. We submit that the system should be designed so that system operators can decide the preferred balance in their system.



(a) The relation between a tasks' *computational requirement* and *visual quality*

(b) The relation between an analysis' cost and accuracy

Figure 1.3:

1.2 Previous Work

This rapport is a continuation of our previous work [5]. In our previous work we design and implement software for a render farm specialised for the review process in 3D production.

In this section we summarise this work, and the state of the previously developed system.

This rapport is a continuation of our previous work [5]. In our previous work we propose a render farm specialised for the review process in 3D production. In this section we summarise this work, and the state of the previously developed system.

1.2.1 MVP Requirements

We started with an analysis of existing commercial render farms, and academic methods, or areas, of improvement. From this analysis we decided on creating a system that enables the creators to define how fast they want to be able to review their work. With the intent to create a proof of concept, requirements for an MVP were formulated. In addition, a MVP was developed in accordance to those requirements. Table 1.1 lists the requirements for the MVP, and their state of completion. The requirements are split, in accordance to the MoSCoW prioritisation method.

	Accepted state	Unaccepted state	Not started
(1) Must have functionality to distribute a render task between multiple workers	X		
(2) Must have functionality to combine rendered frames into a image sequence	X		
(3) Must have functionality to calculate the approximate render time of a task		X	
(4) Must have functionality to allocate resources based on the demand of a task.	X		
(5) Must have functionality to output a preview of rendered frames as an image sequence.	X		
(6) Must have functionality allow system operators to control the analysis' cost.		X	
(7) Should have an interface for users to playback an image sequence of rendered frames			X
(8) Should have functionality to subdivide single frames into an arbitrary number of sub images			X
(9) Should have functionality to merge sub images into single frames after being subdivided			X
(10) Could have functionality to intelligently select frames that should be analysed			X
(11) Could have functionality to perform the analysis and rendering concurrently			X
(12) Could have the render farm scale automatically based on resource requirements from active tasks			X
(13) Won't have account features			X
(14) Won't have encryption of internal or external messages			X

Table 1.1: State of MVP requirements

The created MVP supports most of the must have requirements. However, as shown in Table 1.1 two requirements were not accepted; approximation of a task and adjusting the

analysis settings.

From our experiments we learned that the accuracy of the approximation is only acceptable for a subset of render tasks. This variation in accuracy came from the preparation overhead required for rendering a task. As this overhead is repeated and included in the time measurement of the analysis. In addition, the overhead is not directly proportional with the render time of the task. We detail specifics regarding the preparation overhead in section 2.2.

As for allowing system operators to control the cost of the analysis, we designed the system to allow adjustments by choosing a target sample. Where the target sample is the amount of times, each considered pixel is rendered and measured during the analysis. The MVP's analysis approach is detailed in section 3.1. However, from our benchmarks we determined that the analysis cost were too dependent on the submitted task, and its preparation overhead. Thus, the control of the analysis was not adequate. Furthermore, from our experiments we learned that the analysis was limited to a single core, thus limiting the hardware utilisation for hardware capable of multi-threading. Hence, further limiting the system operators to control the cost of the analysis, based on their own render farms' hardware.

Because of the variation in the analysis' accuracy and the lacking control over its cost we will revisit the MVP.

1.2.2 MVP Components

To fulfil these requirements, we designed a system, as shown in Figure 1.4. The component diagram shows the relation between the components in the system. Description 1.2 provides a corresponding description of each and their responsibilities. The detailed description of the MVP, its requirements and design can be found in our previous work [5].

Browser: The creators' interface with the system. Not strictly limited to browsers, any interface that allows creators to send HTML requests.
API: Receives HTML requests to enable communication to the render farm. Its main endpoints are task submission and stream access.
Task Analyser: Responsible for analysing each task upon submission. From this analysis, calculate the correct number of resources that should be given to that task.
Cloud Platform: Responsible for exposing features that enable the spawning of workers and access to the tasks' file between components.
Task Manager: Responsible for management of tasks after analysis, this includes management of workers and the queue that details the remaining work for that task.
Blender: The program responsible for performing the rendering. With Blender's Cycles being the selected engine.
Render Workers: An entity whereupon the rendering is performed, using Blender.
Multimedia Framework: Responsible for encoding of the rendered frames, which are exposed to the creators.

Description 1.2: Each components' implication in the MVP

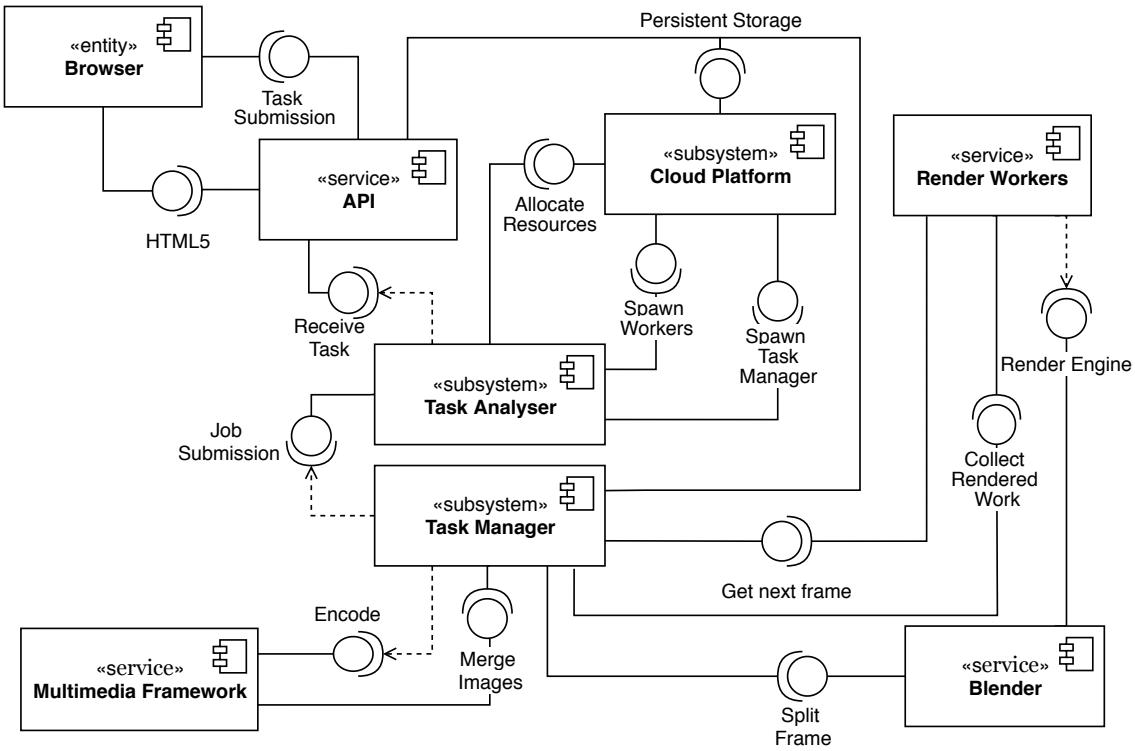


Figure 1.4: Component diagram of MVP

In terms of the MVP's overall design, we based it on existing cloud render farms. And with the objective of tailoring the CPU resource allocation based on the submitted tasks' *computational requirement*, we designed a Task Analyser. The Task Analyser analyses each task upon submission, by utilising the theoretical basis of how Cycles render its tasks, see

section 2.2. Therefore, we based the analysis on rendering a percentage of the task, spread evenly across its frames. Additionally, to address the delay of rendering, we introduced continuous encoding. Where frames are encoded, as they are rendered. In addition, we added an endpoint in the API that allows the creators to view the currently encoded frames.

To discuss the communication of the MVP we will start by distinguishing the different information sent to the system by the creators, and the various components that use it. For each task we separate the information into system meta, Blender meta and Blender data.

System meta: Denotes the metadata sent with the HTTP push method at task submission. This data instructs of the desired output for the rendering, i.e. the requested frame throughput of the system, which we will refer to as the requested frame rate. The Task Analyser uses this to determine the *computational requirement* of the submitted task.

Blender meta: Denotes the metadata for the rendering, i.a. number of samples, frame rate and resolution of the task. This data is situated within the Blender file and is read by our system to enable the system to understand the render task's *visual quality*. Hereby, enabling Workers to render, Task Analyser to analyse and the Task Manager to encode the submitted task in terms of its *visual quality*.

Blender data: Denotes i.a. objects, lights and textures in the scene. This information is handled by our chosen render engine Cycles, that is used by the Workers and the Task Analyser.

Additionally, in terms of the internal communication between the components, we observe the event of creators uploading a task to the system, as illustrated in Figure 1.5. The sequence diagram illustrates the Task Manager and Workers are created on a per task basis. The number of instances of these components are therefore scaled according to the number of concurrent tasks in the system. In contrast, the API and Task Analyser are not created on a per task basis. Instead, all incoming communication to both the API and Task Analyser gets distributed using a round robin policy, across their respective available instances. Thus, both these components are designed, so system operators can scale them automatically based on some metric, e.g. on CPU load above 80%.

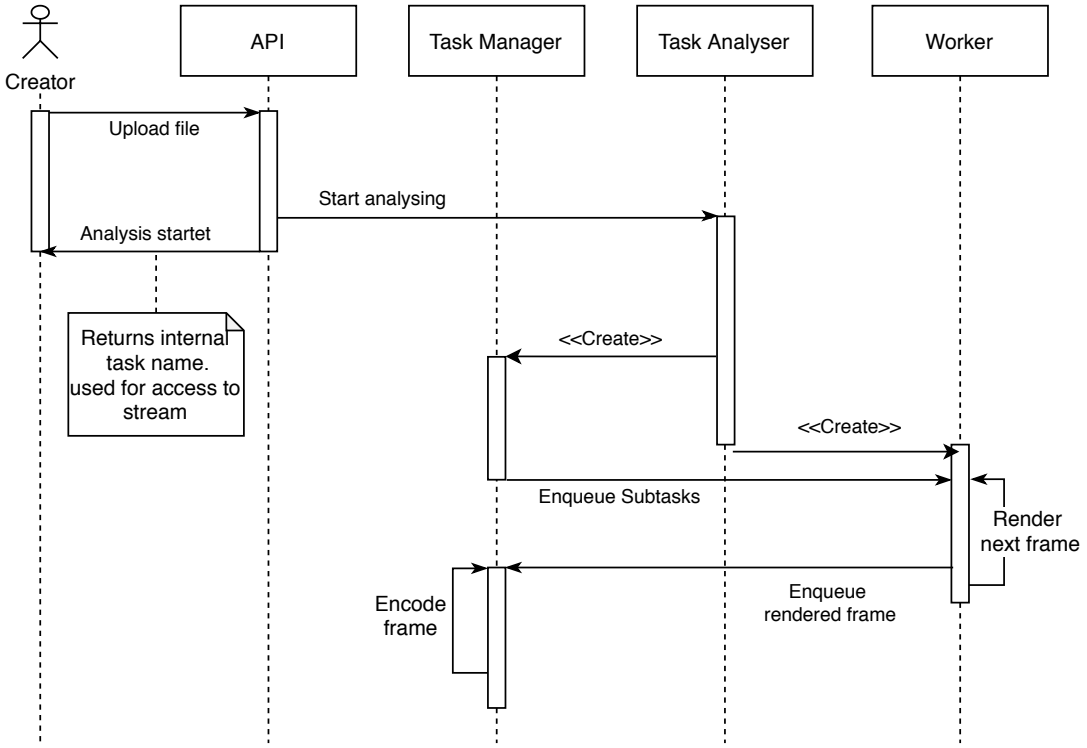


Figure 1.5: Sequence diagram showing the communication in the event of upload of a task file

Table 1.2 illustrates each components' responsibility in relation to the MVP requirements. The table lists the different components vertically and a number corresponding to a requirement horizontally. Where an X marks the component responsible for tending to that requirement.

Req. No. / Resp. Component	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Browser					X		X							X
API					X		X						X	X
Task Analyser			X			X				X	X	X		X
Cloud Platform	X			X		X					X	X		
Task Manager	X	X						X	X		X			X
Blender			X							X				
Render Workers	X	X		X							X			X
Multimedia Framework					X			X	X					

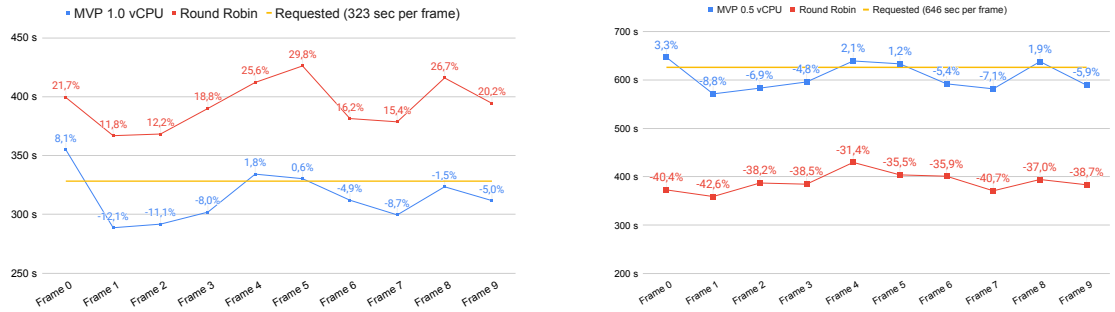
Table 1.2: Overview of the connection between requirements and components

The MVP was implemented with these components, and benchmarked to evaluate its performance of the three aspects: delay of rendering, *visual quality* and the *computational requirement* of a task.

1.2.3 MVP Benchmarks

For our benchmarks we compared our MVP to a system without an analysis of tasks. Instead, this other system used a round robin policy to evenly distribute resources between concurrent tasks. We then submitted two tasks, each with a different *visual quality*, and minimal impact from preparation overhead. Additionally, we employed strict limitations on the workers CPU's, as under normal circumstances they can utilise unused CPU's in the cluster. We placed this restriction on the workers, to better illustrate the accuracy of the analysis. For a more detailed description of the experiment setup, see [5].

In Figure 1.6 the *visual quality* have been simplified to represent a requested frame rate, illustrated by the straight yellow line. Additionally, two lines have been plotted, representing the timespan for each frames' relative render time in the two systems. Each frame is annotated by the percentage difference to the requested frame rate. As seen in Figure 1.6a and Figure 1.6b, the MVP are consistently closer to the requested frame rate for both submitted tasks. This demonstrates the satisfactory accuracy of our system at targeting a specific *visual quality*.



(a) Task 1

(b) Task 2

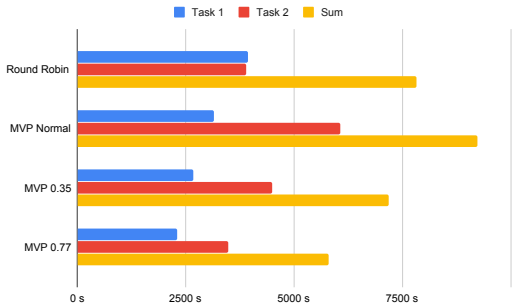
Figure 1.6: Comparison of *visual quality* between the MVP and the round robin system

Customising the resource requirement for each task comes at a cost, namely the cost of the analysis, found by comparing the absolute completion time of the two tasks. We benchmarked this cost and it can be seen in Figure 1.7. The two first clusters in Figure 1.7a shows the MVP takes a comparatively longer time to complete the two tasks. However, this is due to us strictly limiting the workers from using non occupied CPU resources. Thus, it is not fully utilising the available resources in the render farm. So to correct for this, we looked at the unutilised resources and factored them into the graph. We show this in the last two clusters of Figure 1.7a, with 0.35 extra CPU's corresponding to the lower bound of the unutilised resources and 0.77 CPU's being the upper bound. But as is visible in the graph, even with the lower bound, the MVP out competes the round robin system. However, this is a theoretical approximation.

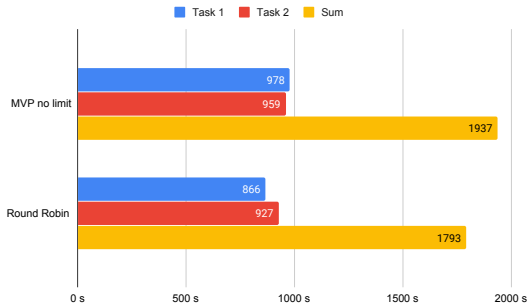
Thus, we performed a new benchmark with our MVP without limitations on the workers, the result is shown in Figure 1.7b. Noteworthy is that the render file differs from Figure 1.7a, where the difference in the sum of the two tasks is ca. 8%, which is to be expected. This discrepancy stems from the analysis time, and from the reduced hardware available when running a Task Analyser, and possibly some hardware variation.

Additionally, due to problems explained in section 2.2 it is not adjustable.

From these initial benchmarks we showed our MVP was able to target a requested *visual quality* in the form of a frame rate. But with an added ca. 8% cost to the overall render time. However, as previously stated, the analysis' accuracy varied when benchmarking a specific group of render tasks, which resulted in the analysis not being accepted.



(a) Comparison between the MVP with limit on workers and the round robin system



(b) Comparison between the MVP with no limit on workers and the round robin system

Figure 1.7: Comparison of cost between the MVP and the round robin system

1.2.4 MVP Summary

From the requirements, components and benchmarks we discussed, we observe a working cloud rendering system. However, there are some deficiencies in our system. First, the Task Analyser is limited in its control of the analysis' cost, thus limiting system operators of tailoring it to their system. Secondly, its accuracy is only acceptable for a subset of tasks. Thus, even though the benchmarks of the MVP showed the benefits of having an automated analysis upon submissions, in contrast to a similar round-robin system. It is not in an acceptable state, for which existing render farms can pick it up and use it for general tasks. To achieve an acceptable state for the MVP the two unaccepted requirements; approximation of a task and control of the analysis' cost, needs to be solved.

1.3 Project Goals

We have in this chapter highlighted two problems in regards to the review process in the traditional CGI workflow. Namely, the delay of rendering, and determining the balance between the *visual quality* required for an acceptable review and its corresponding *computational requirement* to achieve this.

We suggest moving the responsibility of determining the *computational requirement* for a given *visual quality* to the render farm, instead of relaying on creators' own experience. We base this suggestion on the varying hardware types and available resources in a render farm. Which increases the complexity of this decision, in contrast to local rendering. Thus, we discussed the current state of the MVP developed in our previous work [5], besides benchmarks of this MVP, to evaluate the potential of our suggestion.

From the evaluation we conclude that the developed MVP showed potential for a subset of tasks. However, for our suggestion to be applicable in the general scheme of things, we

need to solve the two requirements that was deemed unacceptable for the MVP. Therefore, the goal of this project is to solve these two requirements:

- Must have functionality to calculate the approximate render time of a task
- Must have functionality allow system operators to control the analysis' cost.

Furthermore, our evaluation criteria for the two requirements is for the system to be stable and consistent across all tasks supported by our chosen render engine Cycles. This means both the accuracy and cost of the approximation should be within reason across tasks. This leads us to the following problem statement.

Problem Statement

How can we establish a cloud rendering system, with a task analysis, to accommodate a specified visual quality from creators by determining the corresponding computational requirement in accordance to the creators expectations?

To answer this question in a conclusive manner we strive to perform a practical test. To perform such a test, we will create a design of a cloud render farm, and construct a minimal viable product of this design. After developing this system we will test it to conclusively answer our problem statement. Thus, the goal is to create a system comparable to the previous developed MVP in both cost and accuracy, but is not limited to a subset of render tasks.

1.4 Chapter Summary

In this chapter we provided the motivation for our project, by highlighting the possible improvement in the 3D animation workflow. Dividing the problem into two areas, each with their own aspects to consider. We then showed the component design of the previously developed system, discussed its communication and scaleability. In addition to its performance in terms of accuracy and cost, as we illustrated by benchmarks of the system. To conclude, we discussed the goals of the project.

Background 2

In this chapter we present the relevant rendering theory in relation to Blender and our chosen render engine Cycles, which uses path tracing. Additionally, we cover the theory behind the prevalent preparation, and the theoretical reasoning behind the analysis of render tasks.

2.1 Cycles

This section delineates on how path tracing works, in conjunction to our selected render engine Cycles. Additionally, we address the preparations performed by Cycles to speed up its rendering.

2.1.1 Render Preparations

Before Cycles starts rendering, it performs some preparations to improve render times. The two most dominant preparations are:

- Synchronising objects with the objective of creating virtual geometry of objects in the scene, and transferring these objects into memory [6].
- Building of the Bounding Volume Hierarchy (BVH) tree, which is a ray tracing acceleration structure and is useful for i.a. collision detection [7].

Both preparations are based on the contents of the task and are only necessary once per frame. Additionally, the results of preparations can in theory persist between frames, if no objects in the scene have changed position or been transformed. However, implementing an efficient way to validate no objects have been altered, is rather difficult, hence no such feature is present in Blender, except for non official release patches such as [8].

Appendix A.1 lists two logs of two different render tasks. In this example we see how the preparations impact the overall completion time of a task differently. The preparation takes 2.57 and 14.76 seconds respectively, before Cycles begins the actual rendering. These two timespans corresponds to a 22.54% and 86.26% of the overall completion time of a task, respectively. Hence, we see the preparation is not directly proportional to the render time of a task.

This preparation slowdown both the analysis, and the throughput of frames from the render workers. We should therefore consider this in the design of the system.

2.1.2 Path Tracing

Cycles is a fully path tracing render engine meaning its rendering is achieved by casting rays of light from each pixel of the camera into the scene. These rays then reflect, refract, and/or get absorbed by objects until they either hit a light source, reach a specified bounce limit, or extend beyond the boundary of the scene. This approach is illustrated in Figure 2.1, where we see how a light ray react according to the surface's attributes.

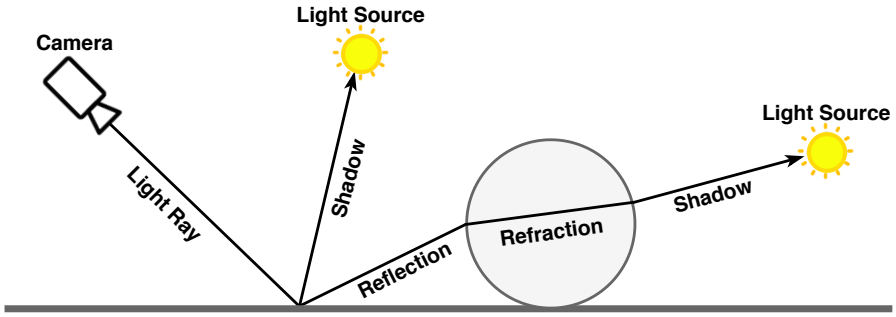


Figure 2.1: Illustrating a light ray cast from the camera, and its traced path

This means path tracing does the inverse of what reality does, tracing light rays from the camera into the scene and onto lights, rather than from the light sources into the scene and then into the camera. This has the advantage as to not calculate light rays that will not end up in the camera's view, but makes other aspects of lighting harder to simulate. E.g. finding the correct light path for pixels lit up by a reflection of a light source. In these cases finding the correct light path for pixels is not as feasible since the number of paths grows exponentially.

We can explain this exponential growth by looking at Figure 2.1. If we follow the light ray cast from the camera, we see it hits a surface and a reflection ray is created. However, we also see that a shadow ray is created. A shadow ray is the last ray in the light path, before hitting a light source. So essentially, the pixel initially hit by the light ray, cast from the camera, is getting lit up by two unique light sources. Thus, we need to backtrack the light path to both these light sources, to determine the correct colour for that pixel. However, since we are backtracking light paths, we do not know that two unique light sources affect that pixel. So on each surface bounce, we have to guess a direction and create a ray in that direction. Thus, path tracers uses Monte-Carlo simulation to randomise ray emissions from surfaces by different angles. This means for each bounce required in the light path, increases the amount of paths we need to check exponentially. Going back to the example in Figure 2.1, we see how the two light sources requires a different amount of bounces. And it is for this reason path tracing engines such as Cycles, uses the concept of samples. Samples refer to the number of light rays sent per pixel of the camera's view. Thus, given enough samples, the correct light paths can be determined and photo-realistic lighting can be achieved.

Since path tracing relays on Monte-Carlo simulation, it becomes a stochastic process, and thus making accurate approximations of render times difficult.

2.2 Analysis of Render Tasks

Because of the stochastic characteristic in path tracing, Cycles needs to render all samples of a pixel, to determine the computation time of that pixel. Additionally, because computation times also vary for each pixel, the actual render time of a task can first be determined once the entire task has been rendered. Thus, to achieve an acceptable approximation of a task's render time upfront, we need to take these observations into an account. For inspiration, and understanding of the basis of analysing a task, we look at Blender's own internal approximation of render times.

2.2.1 Blender's Approximation

Because of the stochastic nature of path tracing, Blender repeatedly calculates an approximation of the completion time, to notify creators of its progress. Blender's approach gives an approximation almost instantaneously as it compares the number of rendered samples to the total samples in the frame repeatedly. We show the approach in Figure 2.2. From the figure we see, Blender's approximation keeps measuring samples for the same tile, until it is fully rendered, then moves to the next tile. This approach is quite naive, as it assumes the rendered samples' *computational requirement* corresponds directly to the *computational requirement* of the samples not yet rendered. This can be seen in appendix A.1, which shows Blender's log. As such, a large discrepancy in accuracy presents itself when the task's *computational requirement* is not evenly distributed. E.g a frame containing an object in its center, and the rest of the frame being empty.

As our objective for the approximation differs from Blender's, we see clear a problem regarding Blender's approach, namely that the initial accuracy of Blender's approximation highly depends on how well the starting pixels' *computational requirement* represent the entire frame. We therefore look at selecting pixels that are a better representation of the entire frame, instead of indiscriminately rendering.

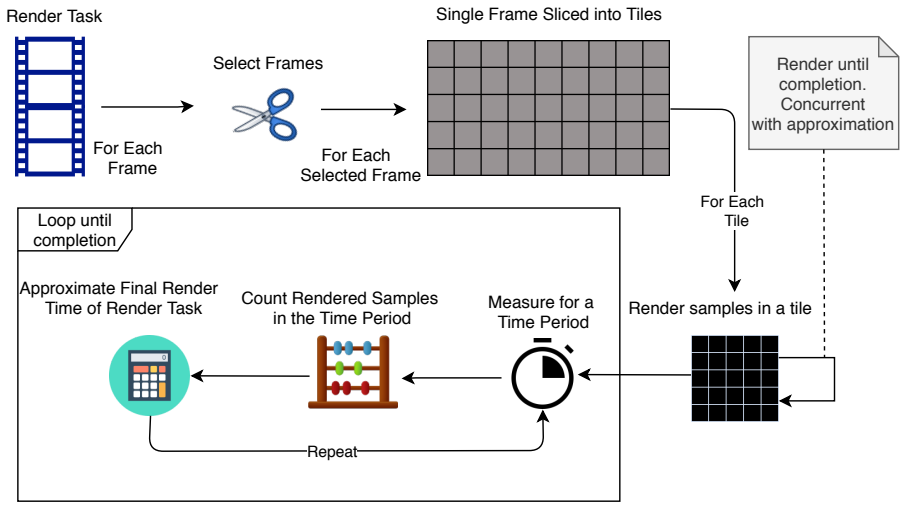


Figure 2.2: Illustration of Blender's internal approximation

2.2.2 Pixel Selection

Before we discuss pixel selection, we need to introduce the principle of 2D and 3D points in a scene. Objects in a scene consist of several edges and vertices connected to form the surfaces of objects. Surfaces of objects are therefore made of a number polygons, that each represent a 2D plane for an area of the object's surface. This 2D plane area consists of a number of 2D coordinate points. These 2D coordinate points are then transformed into 3D points in the scenes global space, by Blender, to simulate a 3D space. So, essentially a 3D point represents the smallest space Blender can work with. From this understanding, we examine 3D projection.

Because of how 3D projection works, that is to say, how the camera maps points in three-dimensions onto a two-dimensional plane, to represent a 3D scene. The camera compresses multiple 3D points of the scene into a number of 2D points to capture the scene's content as several pixels, as illustrated in Figure 2.3.

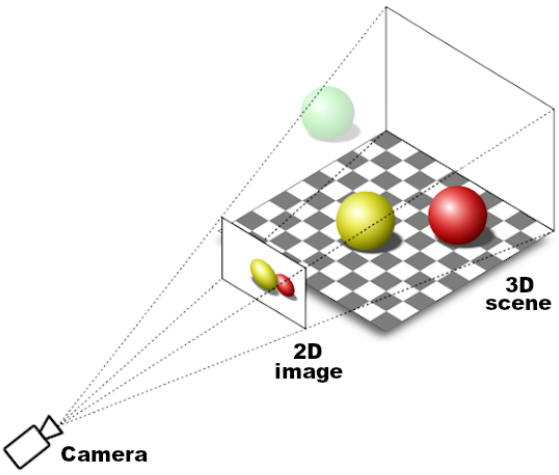
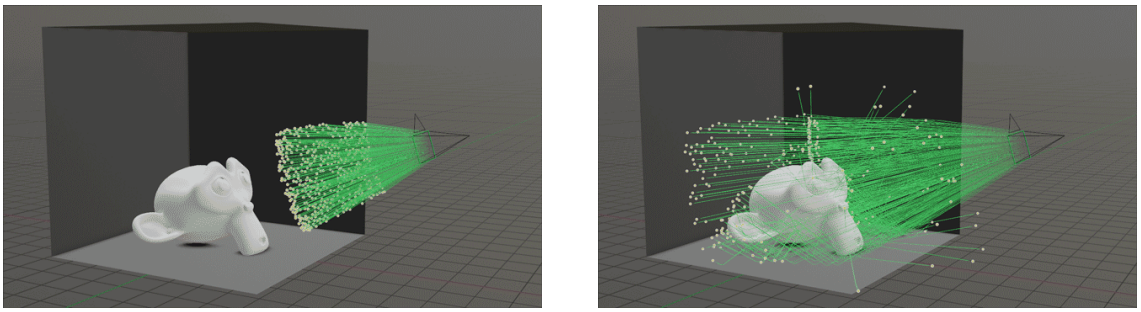


Figure 2.3: Illustration of a 3D projection

However, as a single pixel is generally too small to present an entire object, creators place the camera in such a way that each visible object gets presented by multiple pixels. Hence, casting light rays from neighbouring pixels are more likely to hit the same type of surface. This results in similar reflection and refraction patterns. Therefore, yield a similar computational complexity. In contrast to pixels distant from each other, in relation to the camera's view, with comparatively more dissimilar patterns. We illustrate this in Figure 2.4, where we see neighbouring pixels hits similar surfaces. From this, we make the assumption that analysing pixels, selected evenly across the camera's view, will yield a better approximation of the average pixel's *computation requirement*.



(a) Light rays is cast from each pixel of the camera's view

(b) The light rays have reached and reflected of objects in the scene

Figure 2.4: Illustration of path tracing with one sample

2.3 Chapter Summary

In this chapter we presented the essential background that govern our design. To cover this, we discussed the chosen engine Cycles and highlighted aspects of the engine that introduce complexity for the analysis. This includes the preparations of a task and the discussion of why analysis of path tracing is not trivial.

Design 3

In this chapter we present the design changes of the Task Analyser component and propose a new approach for the approximation. Additionally, we describe the internal communication of Blender and the required changes in its components, to support the new proposed approximation approach. Lastly, with the cloud platform being Microsoft Azure, we describe specific attributes of our system, and how we map the result of the analysis to a number of render workers.

3.1 Task Analyser

As mentioned in section 1.2 we found the Task Analyser previously developed did not fulfil our requirements. This section details the problems, and potential solutions, with the previous Task Analyser. Last, we explore optimisations of the analysis approach previously developed in the MVP.

3.1.1 The MVP's Approximation

The approach of the MVP's analysis is to render a percentage of a task, while measuring rendering time. These measurements are then scaled accordingly, meaning if 2% of the task was rendered, the measurements are multiplied by 50 to approximate the render time for the entire task. We visualise the different stages of this approach in Figure 3.1.

The approach renders a constant percentage of the task. Its analysis time is therefore entirely dependent on the render time of the task. Having the analysis time varying based on the render time is not optimal, as it may yield long analysis times. A more controllable analysis time would be preferable to limit the cost of the analysis. In addition to this, from our evaluations of the MVP, we found problems in our approach. These problem stems from the render preparation of a task. section 2.2 delineates on the render preparation. While some preparation is necessary to improve render times, the Task Analyser's design resulted in drastically increased timespans spent on preparations. Normally the preparations is performed once per frame, however, the Task Analyser performed the preparations multiple times for each frame. This is because the Task Analyser selects a portion of the frame to be rendered, and then instantiate a render instance for each part selected. However, Blender deliberately discards all results from the preparations, once the render instance has completed. So because the Task Analyser creates multiple render instances, instead of an unified one, it performed the preparations for each render instance. Hence, the preparations became detrimental for the Task Analyser in two notable ways.

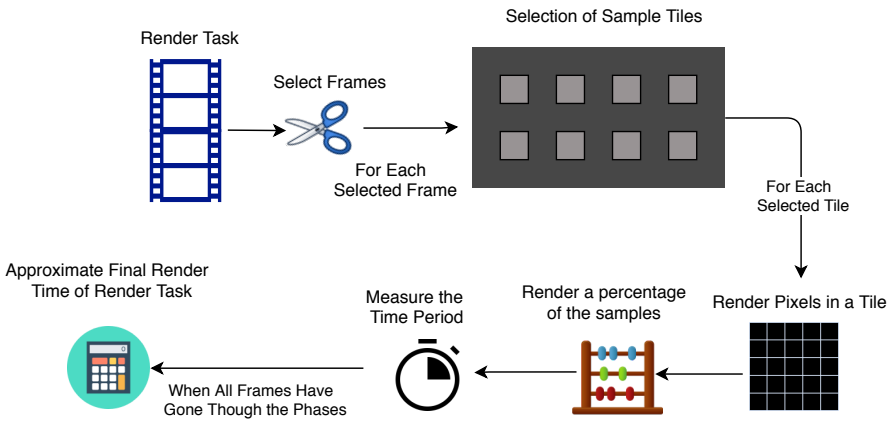


Figure 3.1: Illustration of the MVP analysis

Problem One: An Unreliable Accuracy

First, preparations influence the accuracy of the analysis, as we include the preparations in the analysis's measurement. This is due to the Task Analyser's timer starts before the preparations. A custom handler would allow a timer to be started after the preparations. A handler is a function for code that is to be triggered by the occurrence of a specific event, in this case the completion of the preparation phase.

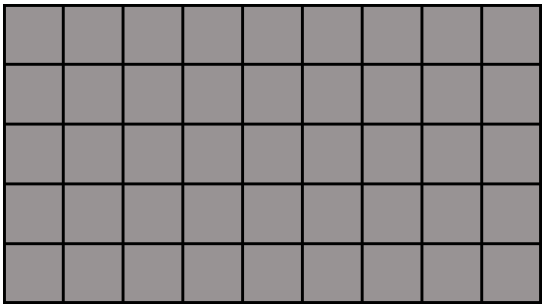
Problem Two: An Unnecessary Cost

Second, preparations increase the analysis' cost. As stated in section 1.1 we seek to balance the cost and accuracy of an analysis, thus this increase in cost is perceived as a non trivial decrease in the viability of our analysis. Thus, reduction of the unnecessary preparation is a priority. Our initial idea was to apply a patch for a previous version of Blender [8], as suggested by multiple people from the Blender community. This patch claims to keep scene data persistent across render instances, such that preparation only has to be performed once per frame. However, upon testing we found that only a limited part of the render preparation was persistent. While there were a visible reduction, it still was not acceptable for our purposes.

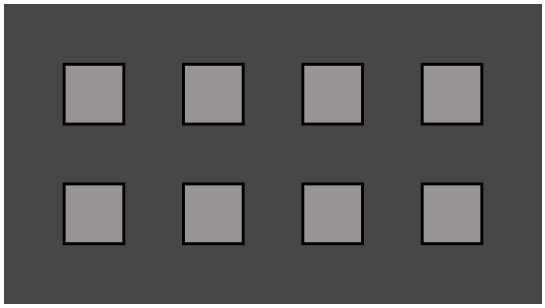
Alternatively, we can introduce the external tile selector, used in the MVP, into Blender as an internal tile selector. A tile selector is the method used to divide each frame into fragments, also called tiles. The benefit of an internal selector versus an external selector, is that a list of tiles are represented as one render instance. Thus, allowing Blender to reuse results from preparations across tiles and reduce preparations to once a frame. Additionally, as each render instance references a list of tiles, tiles can be rendered concurrently across threads. In contrast, an external selector is only able to render one tile at a time. Hence, an internal selector would both give the analysis better hardware utilisation, and reduce the unnecessary cost of preparations.

Reusing Blender's default selectors is not optimal as it selects the entire frame, however, our analysis requires a subset of the frame to be selected. Figure 3.2a shows a default internal selector in Blender, and Figure 3.2b shows the implemented external selector in the MVP. Thus, to keep the analysis cost proportionate to the task, our analysis requires

a specialised internal selector. Furthermore, as we argued in section 2.2, the importance of selecting pixels evenly across a frame. Because of this Blender's default internal selectors are not feasible for our purposes.



(a) The internal tile selector in Blender, illustrating the tile selection of a single frame



(b) The external tile selector implemented in the MVP, illustrating the tile selection of a single frame

Figure 3.2: Comparison between Blender's internal tile selector and the MVP external tile selector, with the same tile size set. Selected tiles are illustrated with a black border and grey interior

Reduction of Maintainability

These two purposed changes affect the maintainability of the system, as adding either a custom handler and/or specialised tile selector, requires changes in Blender's source code. Changes in Blender's source code detrimentally affects the maintainability of the project. Thus was not our initial choice. But, as the cost of the analysis is a concern, we see a reduction of the cost, by sacrificing some maintainability, as a positive trade off. Furthermore, as some maintainability has already been forfeit, by changing Blender's source code, it introduces the opportunity to make supplementary analysis optimisations.

3.1.2 New Proposed Approximation

Because a more controllable analysis time would be preferable to limit the cost of the analysis, we propose another approach for the approximation.

We considered using Blender's approximation, detailed in section 2.2, and extract the approximation after a given time period. However, we found it to be quite naive, as it is heavily reliant on the rendered samples directly corresponding to the samples not yet rendered. As such, a large discrepancy in accuracy can be found when the task's computational complexity is not evenly distributed. By using our selector, this problem could be mitigated, as a variety of different tiles all over the frame would be rendered. However, for this to work all selected tiles has to be visited, but not fully rendered, which raises some problems. Namely, we would also have to rework the method it uses for recording the current process, as normally the approximation assumes previously visited tiles have been fully rendered. Furthermore, as Blender's approximation is not designed for tile selectors that select only a subset of the frame. It would also require us to make the approximation believe it has to render the entire frame. It then becomes a rather messy implementation of tricking the approximation.

After some deliberation we had an idea of a more controllable analysis time; each thread

renders for a set time period, and upon reaching the end of the period, it stores the number of samples rendered and moves to the next tile. After repeating this for each selected tile, the total number of rendered samples is used to calculate the average render time of a single sample in the task. The average render time of a sample is then applied to the total number of samples in the frame which equates to the approximate render time of the frame. We illustrate this approach in Figure 3.3. Hereby, the time spent on the analysis is segregated from the render time of a frame. Instead, it primarily depends on the number of tiles that has to be analysed in conjunction with the set time period. The analysis time would therefore be, regardless of frames' varying computational complexities, consistent across frames.

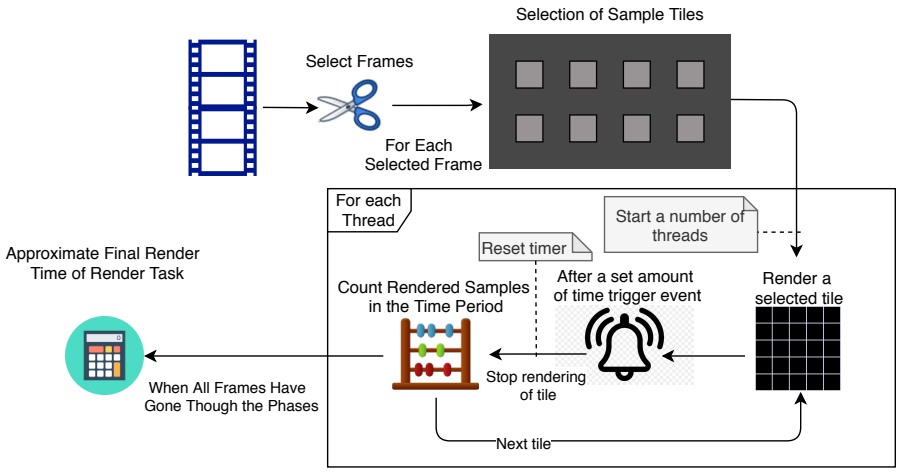


Figure 3.3: Illustration of the time period analysis

However, a naive implementation of this approach with a handler would require stretching the intended usage of handlers. The handler would have to keep track of how long each thread has been rendering and navigate to the correct thread to inform it to move to the next tile. This would break the core design of Blender's handlers and introduce extensive commutation overhead. Thus, it would be more proficient, in terms of performance, to add the timer internally in the thread. This would allow the thread to check every time a sample has been rendered, and determine if it should move to the next tile. Therefore, to benefit from the constant time analysis approach, we abandon the custom handler. Instead, we identify an alternative way to communicate between Blender's internal modules. Where the relevant communication is between the Cycles engine module, the module controlling the threads, and Blender's Python API module, the module exposing internal commands.

3.2 Blender's Internal Communication

In this section we describe Blender's internal communication, and the relevant components to support the new proposed approximation from subsection 3.1.2. Additionally, we will base this section on the newest release of Blender at the time of writing, Blender 2.82.

3.2.1 Threads in Blender

Before we can discuss the communication between the components of Blender, we first need to understand how information is transferred in Blender. Blender stores its information about the render job in three layers, specifically threads [9].

Blender Thread: is the one that receives render engine callbacks. From this thread it is safe to access the Blender scene, e.g. change active frame of the job.

Session Thread: sits between the Blender Thread and Device Threads, acting as the bridge for information between the two. From this thread, tiles are selected and distributed among the Device Threads.

Device Threads: performs the actual rendering, the number of Device Threads depends on the hardware. Blender supports several hardware types [10], in the case of a CPU device, one Device Thread are created for each of its physical cores.

In order to not break the core design of how Blender transfers its information, the additions to support the proposed approximation should stay within the boundaries of these three layers of threads. Meaning, if we want to extract information from the rendering, we need to pass it through the Session and Blender Thread. We illustrate the life cycle of the three layers of threads in Figure 3.4, which serves to show the hierarchy of the different threads, and their responsibility during a render job. The Blender Python API in the figure is the access point for which our Task Analyser component communicate information such as analysis settings and the render task itself.

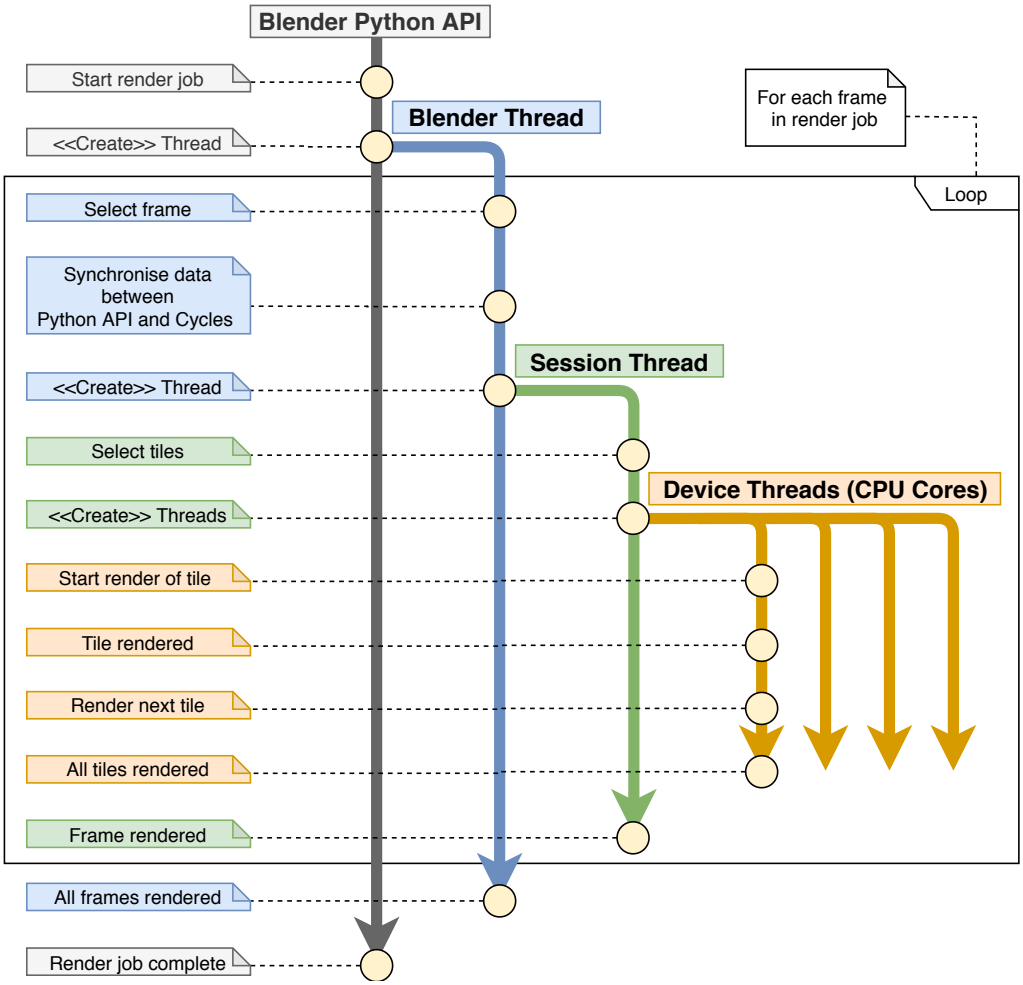


Figure 3.4: The life cycle of the three main threads in Blender during a render job

3.2.2 Components in Blender

Given the overview of Blender's three layers of threads, that handles a render job, we move on to the components of Blender. To highlight the relevant components, we created the component diagram in Figure 3.5. The diagram is created based on Blender's source code, and the documentation of its components[11].

Figure 3.5 highlights two areas of Blender's internal components, specifically *Python & Add-ons* and *Cycles*. These areas contain the components that handles communication between the three layers of threads. Thus, to add support for the new proposed approximation, we need to modify these components. A description of each components' responsibility in relation to the render job, can be found in appendix section A.2.

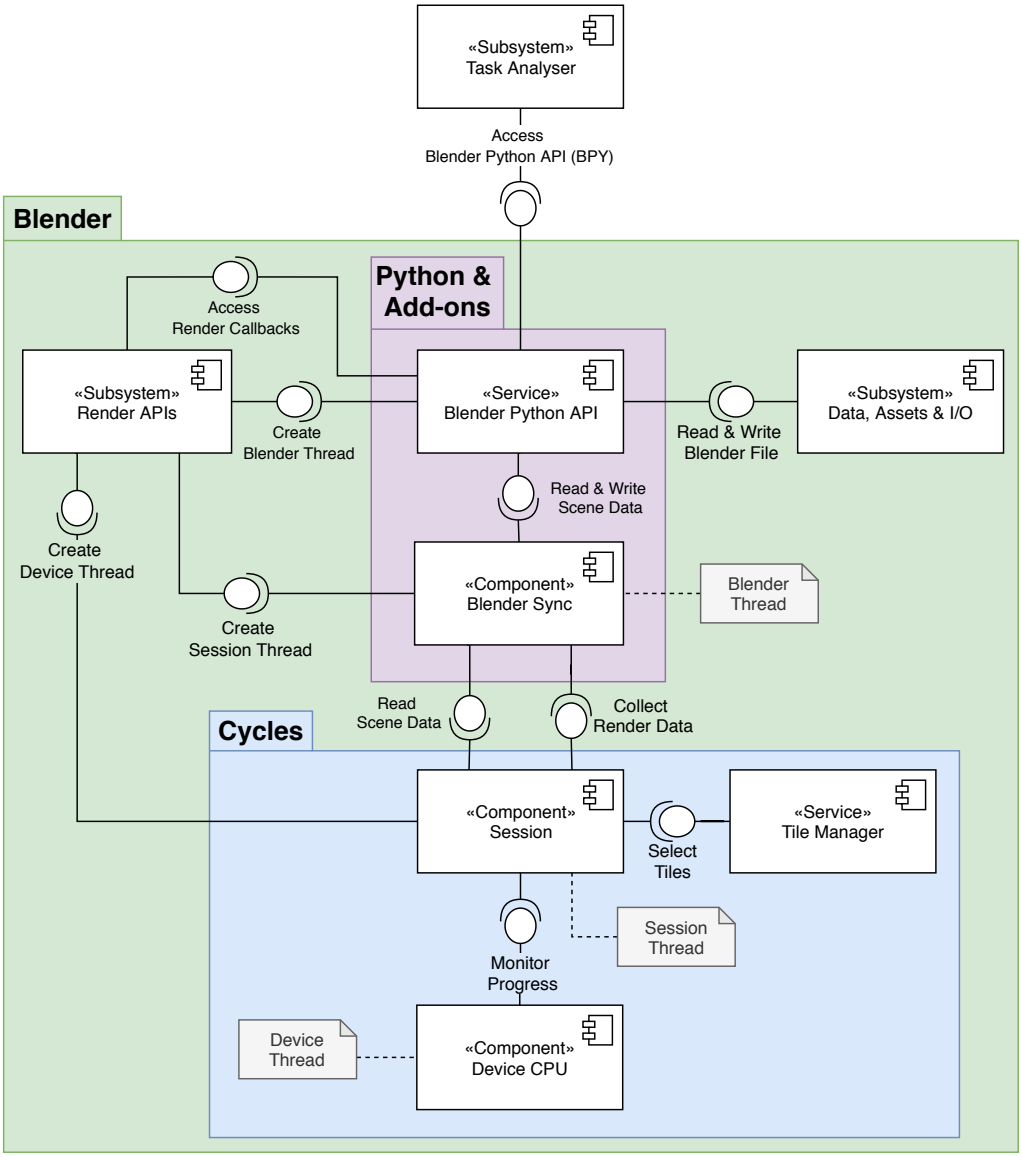


Figure 3.5: Component diagram illustrating Blender's internal communication

3.2.3 Blender Modifications

From the thread overview Figure 3.4, and the internal components of Blender Figure 3.5, we now have an overview of Blender's internal communication. Given this overview, we introduce the modifications necessary to perform our new proposed approximation. We start with the top layers of our Task Analyser component communicating with the Blender Python API, and end with the low levels of Device Threads performing the actual rendering.

In order to make the analysis' cost controllable without having to rebuild Blender, we first introduce a number of parameters, in particular: the frames and number of tiles to be considered for the analysis, the analysis time limit of each Device Thread, and the chosen tile selector. Second, we modify the Blender Sync component to keep these parameters synchronised between both the Blender Thread and the Session Thread. We can achieve this by extending the scene data struct maintained by Blender Sync component. This

struct contains all the necessary information of the render job and is the main container for information sharing, between the three layers of threads.

Given the synchronisation, we are able to adjust the parameters of the analysis at runtime, as the synchronisation takes place on each frame change. Hereby, we can adjust the analysis during its intermediate results, making the cost of the analysis controllable.

Additionally, with the analysis parameters present in the scene data struct, we can use Blender's intended path for information sharing, to inform the Device Threads on runtime to change its behaviour. This is used to state the Device Threads' time limit for rendering a tile. However, the Device Thread also has to be modified such that it accesses this new information, and stops rendering accordingly. This requires changes in the backend of Cycles CPU rendering.

Lastly, we need to extend the Session Thread's monitoring of Device Threads, in order to transfer render data back to the Blender Python API component. Hereby, giving our Task Analyser component access to the data, so it can approximate the task's render time, and determine its *computational requirement*.

3.3 Expansion of MVP

In this section we explain changes made to the MVP system. However, we limit descriptions as there were many smaller changes that are not impact-full enough to warrant description.

As we perform a task analysis upfront, we discussed at what point, after task submission POST, we should return data to the creator. There are two points that we discussed, sending the return value when the upload of a file is done, or when the analysis is done. If we give the creators data when the analysis is done, we can provide more relevant data, for instance a boolean to signal if there is currently enough capacity to get the required results. However, compared to returning data after upload, we would not be able to provide the same information. As such we made an endpoint for both, so the system operators decide which they prefer.

To provide an controllable analysis, we had to modify the structure of the system. We made the changes in structure to avoid static variables, for among other things analyser settings. We use environment variables that can be set at deployment in the yaml file. Where the yaml file is a collection of settings for the creation of a pod. This approach provides the system operators the option of changing those attributes at runtime.

In the MVP we have three different queues, all using the Advanced Message Queuing Protocol (AMQP). First, we have the task queue, which details the tasks not yet started. Second, we have the frame queue, which contains the non started frames of a task. Lastly, we have the encode queue which contains the frames not yet encoded. This communication is largely unchanged for the iteration of the system. However, our render workers communicate with the AMQP frame queue to receive a new frame after completion of a frame. This makes the render workers receive their tasks in a sequence, thus they only know their current frame. Because of this, the MVP called Blender in a sequence for each frame. Meaning that Blender terminated between frames, which is detrimental to the

throughput of the MVP. This problem was mended by performing the sequential selection of frames and communication from within the Blender instance, calling the render callbacks with the Blender Python API. Thus, not closing Blender except on completions.

3.4 Mapping Analysis to Render Workers

This section describes the mapping between the analysis' approximation of a task's render time, i.e. its *computational requirement*, and assigning a corresponding amount of CPU resources. We measure these CPU resources in terms of vCPU's which is an abstraction layer for hardware handling. This hardware abstraction is available to us as we use the Azure cloud platform. In the cloud we will have several nodes, meaning a number of connected virtual machines. Each of these nodes will have resource capacities, i.e. memory, processor and storage. In Azure the processor resource is abstracted to vCPU, which dictates how much access a given process has to the processor.

A necessary assumption needed for the analysis to yield usable results is that: any one vCPU is equivalent to any other vCPU, meaning their computational powers are equal. Without this assumption, converting the approximation to render workers is not feasible. In theory this assumption holds, however, in practice there is slight variations in the hardware that can not be avoided.

As for the calculation of required vCPUs, it depends on the units of the approximation's output and the requested frames. Depending on the relation between these units, transformations might be required. We transform the output of the analysis so it has the unit *seconds per frame*. *Seconds per frame* meaning the amount of time it would require to render a frame. Similarly, the requested frame rate provided by creators is in the form of *frames per seconds*. *Frames per seconds* meaning the number of frames rendered each second. Multiplying these units cancels each other out, therefore additionally multiplying the analyser's allotted vCPU's will yield the required vCPUs needed to get the requested frame rate. We show this formula in Equation 3.1, with units noted in subscript.

$$worker_{vCPU} = (analysisoutput_{SPF} * analyser_{vCPU}) * requested_{FPS} \tag{3.1}$$

The required vCPUs serves as the sum of vCPUs allotted to all a task's workers. However, splitting the vCPUs is not a trivial task. There are multiple approaches. But, some of them go in direct violation to the goal of our system. As mentioned, a central commandment for the system is to get the result to the creators as fast as possible. An example of an approach that does not provide this feature is to measure the available vCPU on a node and create a worker with that size. This approach would enable the creation of unbalanced workers, which could enable some pieces of the task to complete much faster than the rest. This would delay the review of the malnourished worker's frames. Thus, we want the workers for a single task to be balanced, to ensure that some frames won't be much slower than the rest. Following this philosophy, our approach is as follows; scan available nodes, to determine if there is enough vCPUs for the task. If there is room for the task, store the capacity of each node. Determine if there is room for a single worker with the required vCPU, if so create that worker. If not increase the number of workers, observe if there are

space for each of the proposed workers. This repeats until a configuration is found or the number of workers exceed the number of frames or exceed a defined upper bound. We then create the workers one by one, each being placed on the node with the highest available capacity.

3.5 Chapter Summary

In this chapter we discussed the problems from the previously developed Task Analyser and using the knowledge gained to propose an improved design of it. Culminating in an alternative approximation approach that provides more control of the analysis' cost and accuracy. Furthermore, we explained Blender's internal communication design, and proposed some changes to facilitate the new approximation approach. Additionally, we presented the impact-full changes made to the system, compared to the previously developed MVP. Lastly, we discussed our approach for mapping the approximation to a number of workers with their corresponding CPU resources, to uphold the specified *visual quality* of the submitted task.

Evaluation 4

From our problem statement, see section 1.3, we posed whether we could develop a system that can accommodate a specified *visual quality* from creators by determining the corresponding *computational requirement*.

In this chapter we evaluate the system by using the new proposed design from chapter 3, that builds upon the design of our previous work detailed in section 1.2. We use this design to implement a MVP that we use to verify if we were able to fulfil the requirements for the system in accordance to the problem statement. In particular, if the system can accurately determine the *computational requirement* of a task. In addition, to provide system operators with enough control of the analysis' cost, to be feasible in their existing render farms. Thus, we evaluate the system to identify if its capable of handling general tasks in Cycles and be controllable in terms of balancing the ratios between cost and accuracy.

We start with a component evaluation of the Task Analyser, to observe its accuracy and cost for three, specifically chosen, different tasks. We designed this test to observe the Task Analyser performance for average tasks. We are interested in its performance for average tasks, as the analysis approach we detailed in section 3.1, does not provide guarantees. As there is no lower bound for the analysis approach's accuracy. Thus, we aim to observe the average case performance of the Task Analyser.

Additionally, to test if the analysis' performance scales with the number of active physical cores of a CPU. We analyse the same task multiple times, but with a different number of active cores, that each have the same computational power. This is to verify if an additional cost presents itself, by increasing the number of active cores. In addition to if the relation between the preparations of a task and its render time affects the analysis' accuracy, as with the previous developed MVP, see section section 1.2.

This is then followed by a system test, meant to observe the performance of the entire system. We do this to observe how the system in its entirety performs, on perhaps less stable hardware; a public cloud. It is done by submitting four consecutive tasks, each with a different submitted *computational requirement*. We then repeat this test three times to observe if the output of these are comparable, such that we can evaluate if the system's performance is unstable. This test is done to determine if the fact that it is running on a public cloud, which might not have the same stability of local hardware, impacts the results of the system.

4.1 Component Evaluation: Task Analyser

In this section we perform a component evaluation of the Task Analyser. We setup an experiment to verify the analysis' accuracy and cost based on the number of active CPU cores, by adjusting the number of cores the CPU can utilise during the analysis. We are specifically interested in its consistency across a variety of tasks, both in terms of its accuracy and its cost. This is done to observe the analysis' average case performance on given different amounts of processing power and given a variety of tasks.

4.1.1 Test Setup

To evaluate the Task Analyser component's performance across a variety of tasks, we examine the demo files provided by our selected render engine Cycles [12]. Cycles provide five different demo files to showcase their engine. We deem these demo files to be a good representation of what tasks may look like, as they are made to showcase different aspects of the engine. Thus, they use various techniques within Cycles to achieve photo-realistic imagery. However, upon rendering the five demo files, numerous errors was printed to the Blender console, by two of the demo files. On further inspection we found this to be the cause of i.a. textures not loading properly, objects using deprecated Cycles features, and internal python scripts within the files, calling deprecated calls to the Blender Python API. Both demo files was created back in 2015, meaning they originally was created in Blender 2.7, which got an overhaul in July 2019 [13] in the form of Blender 2.8. Thus, we suspect the numerous errors from the two demo files to be caused by not being compatible with Blender 2.8, which is further supported by the fact we could render them on a Blender 2.79 version. However, as our modifications and Blender build is based on Blender 2.82, we discarded them from the test. Therefore, we select the three remaining Cycles demo files, with a preview image from each illustrated in Figure 4.1. We will refer to these three tasks as: pavillion, classroom and barbershop.

(a) Task 1: Pavillion

(b) Task 2: Classroom

(c) Task 3: Barbershop

Figure 4.1: Preview image for each of the three tasks' first frame

We performed the tests by first analysing a task and then fully rendering the tasks while logging actual rendering time, the analysis' approximation, time and cost. We ran the tests on a single computer, with the specifications listed in Appendix A.3.

To confirm if the analysis' accuracy depends on the preparations of a task, we test the relation between the computational power of a single CPU core and up to eight CPU cores. To achieve this a multitude of tests were performed with different number of active CPU cores. Noteworthy for this test, the analysis is done on the same hardware as the rendering, where in normal circumstances the computational power of the analysis and render worker is not equal. We do it in this manner to show its relation to the rendering time of the task.

Lastly, we use the same analysis settings across all tests, to verify the analysis' consistency in accuracy and its cost, given a variety of tasks and active CPU cores. The analysis settings used for the tests: a tile selection of 11 tiles, a tile size of 64x64 pixels, and a tile render limit of one second.

4.1.2 Results

After performing the tests, we analysed the data and constructed multiple graphs. In this section we showcase the two tasks with the best and worst results. The final task can be found in appendix A.3.

Analysis Accuracy

Figure 4.2 and Figure 4.3 shows the most accurate and least accurate tests, respectively. The graphs compare the accuracy of eight tests with a different number of active CPU cores for each task. With each cluster of bars representing an approximated render time and the actual render time. In the graphs we note some variation in the analysis' accuracy. With Figure 4.2 showing the span of accuracy for classroom going from -1.63% to 1.80%, and Figure 4.3 the span of accuracy for pavillion going from 5.70% to 9.13%. We can explain some of this variation by the inconsistency in hardware's executions.

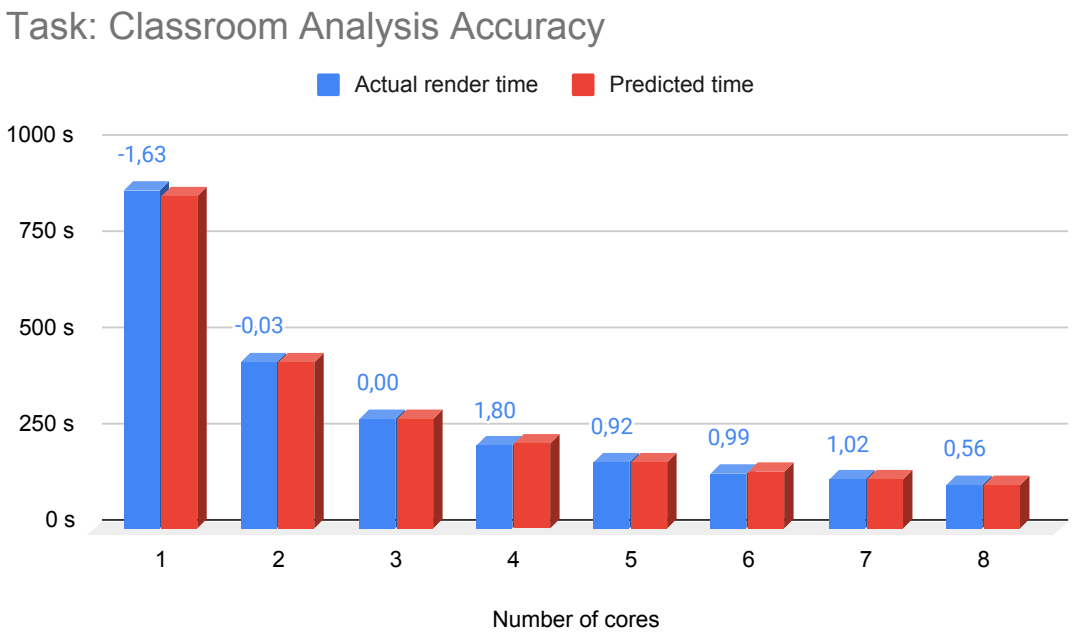


Figure 4.2: Graph showing the analysis' accuracy of task classroom

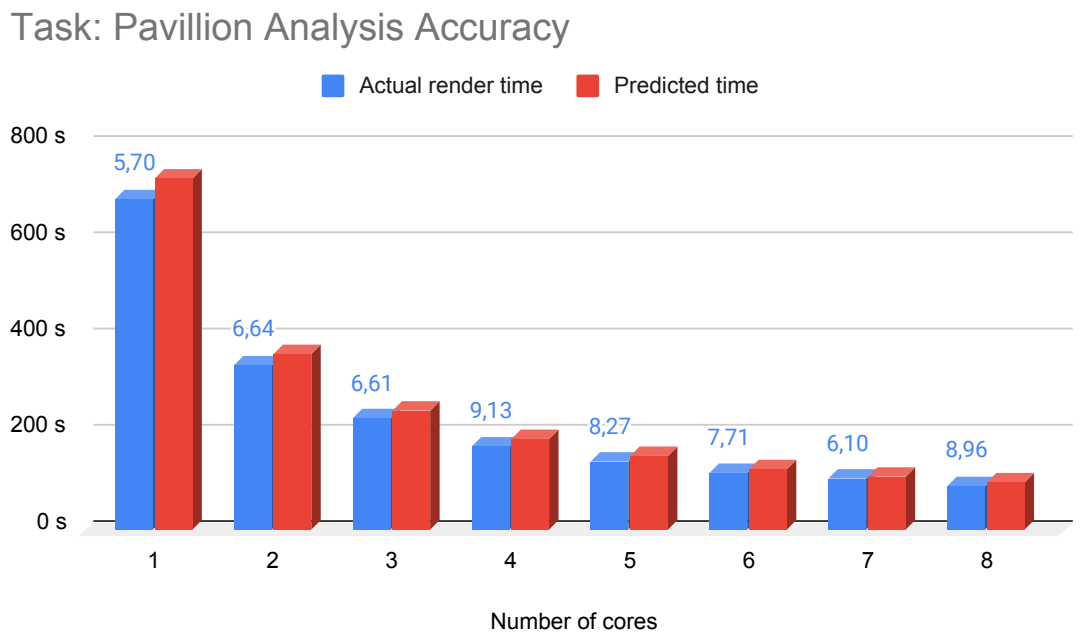


Figure 4.3: Graph showing the analysis' accuracy of task pavillion

However, since we see no apparent patterns of accuracy deceasing or increasing based on the number of active cores across the three tasks. We suspect the primarily cause for the slight variations in the analysis' accuracy are caused by inconsistency in the hardware's executions. Noteworthy is across all three tasks, the accuracy of the analysis using one core is the best. Like-wise the analyses using one to three cores appear to be more accurate in contrast to the analyses using six to eight cores. This is, however, not a consistent pattern throughout all tests, and we require more extensive tests to verify if the analysis's accuracy decreases with the number of active cores.

Furthermore, from the tests we see the accuracy of the analysis no longer depend on the timespan of preparations, since we no longer include them in the approximation, with our new implemented analysis approach. We verify this by calculating the average absolute variation in accuracy for each task. With classroom being at 0.87% and pavillion 7.39% with the last test, barbershop, being at 2.94%. In addition, their respectively average preparation timespan across all tests, being 2.60, 37.87 and 4.75 seconds. A figure of the preparation timespans can be found in section A.3. Looking at these numbers, with the old analysis approach the task barbershop would have been the least accurate. However, when comparing the variation in accuracy, it is more than 4% more accurate than the pavillion task which has a lower preparation time. This confirms the fact the analysis' accuracy is no longer depended on preparations, however, it still depends on the task itself. Which is why we seek to observe the average case. If the demos, provided by Cycles, is any indication of the average case of a render task then we have an average absolute accuracy variation of 3.7%.

Our previous MVP had an average absolute accuracy variation 5,5%, thus, we have improved the accuracy variation by around 2%. Additionally, the previous MVP could

only take tasks with minimal preparations. Thus, we see a clear improvement in the new approximation approach, because it has both showed a noticeably higher accuracy, and works on a higher variety of tasks.

Analysis Cost

Figure 4.4 and Figure 4.5 shows the most and least costly tasks, respectively. With the graphs each comparing the actual render time and the analysis time with different number of active cores. The graphs show the comparative size of rendering compared to the analysis time. Furthermore, the preparation of the analysis has been highlighted, as the preparation of the task is a major factor for the cost of the analysis. However, as discussed in section 2.1 this preparation is necessary to achieve faster render times. Additionally, Blender's current design limits the preparation to a single core. From this it is clear that the analysis does not incur a static percentage cost and depends both on the task and the active core count. With Figure 4.4 showing the span of cost for classroom going from 1.24% to 3.31%, and Figure 4.5 the span of cost for barbershop going from 2.62% to 15.58%.

If the demos, provided by Cycles, is any indication of the average case of a render task then we have an average analysis cost of 5.1%.

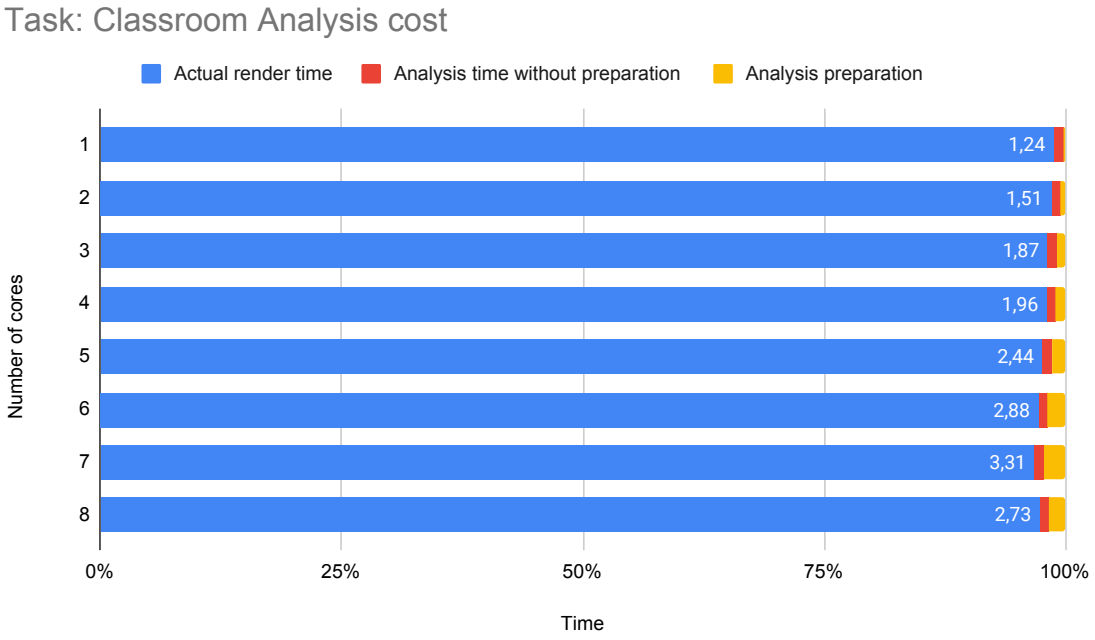


Figure 4.4: Graph showing accuracy of analysis of task classroom

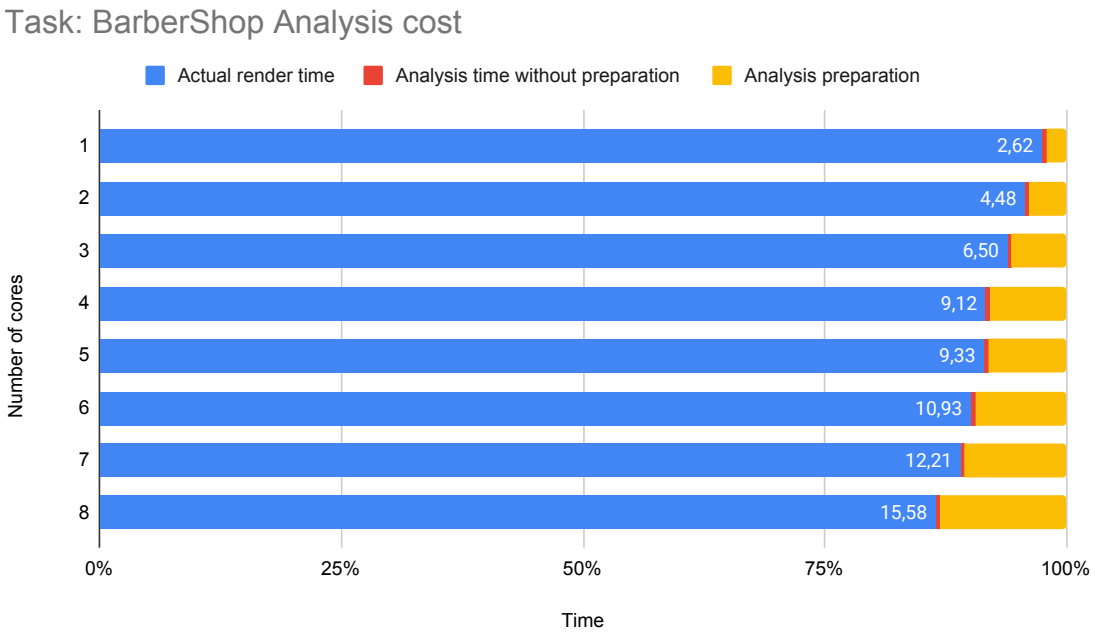


Figure 4.5: Graph showing accuracy of analysis of task barbershop

From the graphs we note some visible patterns in the analysis' cost. To better highlight these patterns, we disregard the preparation time from the analysis' cost, and focus on the actual time spent on analysing the task. We illustrate this in Figure 4.6, with each cluster of bars representing the actual time spent on analysing with the number of cores active. From the figure we see that the analysis time is fairly consistent between the three tasks, and across all clusters.

Additionally, we see the analysis time decreases with the number of cores active. This decrease acts as expected with the analyses using one core being close to half the analysis time of analyses using two cores. Like-wise, the analyses using four cores are close to half the analysis time of analyses using two cores. Noteworthy, this pattern in the decrease of cost stops after the number of cores becomes higher than five. This is because the analysis settings used in the tests are set to consider 11 tiles, and since tiles are distributed across cores, the number of inactive cores starts increasing after five active cores. E.g. analyses using eight cores, in this case after the first three cores finishes their first tile, all 11 tiles have been considered, and the remaining cores becomes inactive.

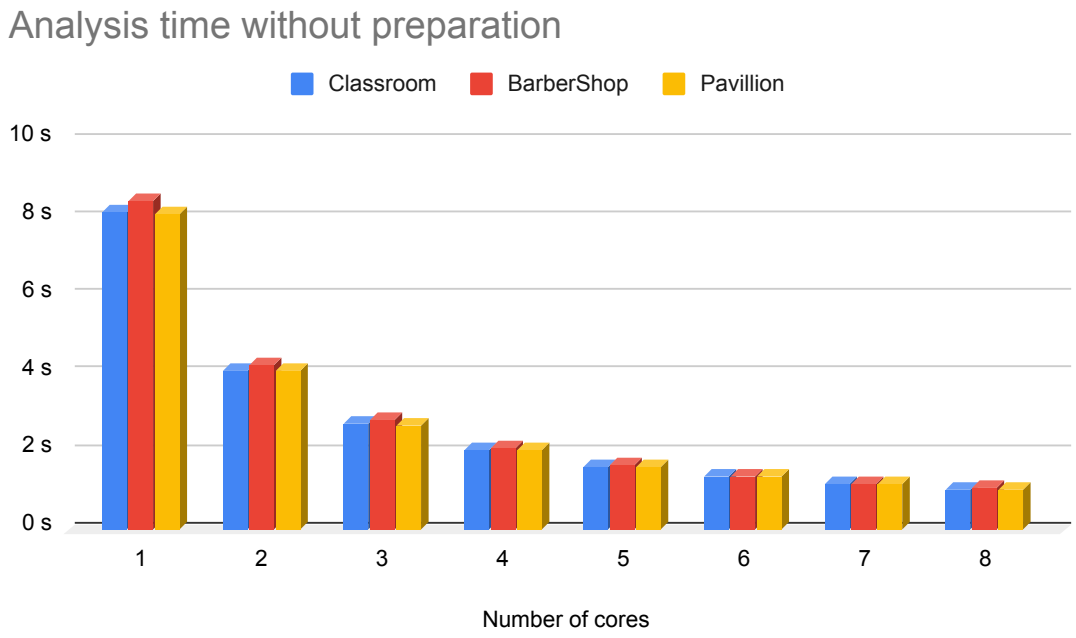


Figure 4.6: Graph showing analysis time without preparation of the three tasks

Our previous MVP had a cost of around 8%, which compared to the new 5.1% marks a ca 3 % reduction in cost. Notably the test on the previous MVP was on tasks with low preparation, which only serves to emphasise the gained a reduction of cost and versatility of the analysis. However, as Cycles is currently limited to perform preparations on a single core, the analysis' cost is still depended on the size of the task's preparations. Additionally, the previous MVP could only take tasks with minimal preparations. Thus, we see a clear improvement in the new approximation approach, in that it has both showed a noticeably higher accuracy, and works on a higher variety of tasks.

Furthermore, the performance benefits from increasing the number of active cores conforms to our expectations. Since each additional active core has the same throughout, of analysed tiles. However, from our tests, it is noticeable that the improvements from increasing active core becomes almost nonexistent when above six cores, due to idle cores. Which results from the number of cores and the number of tiles not lining up. An extreme example of this is: eight cores and nine tiles, after each core complete a tile, seven cores are idle waiting for the only active core.

4.1.3 Conclusion

The basis for our evaluation of the Task Analyser is that the three selected Cycles demo files represent general tasks submitted to render farms using Cycles as an engine.

From our evaluations we see the accuracy of the analysis no longer depend on the size of preparations, since they are now only included in the approximation once per frame, with the new implemented analysis approach. Additionally, we see the analysis' accuracy still depend on the task itself, which, given our approach is to be expected as we detailed in section 2.1, about how Cycles work.

Furthermore, as our evaluations all used the same analysis settings to isolate its performance, our results may become better by finding the optimal analysis settings. However, this would require more extensive testing, and we leave this for system operators, as this depends on the underlying hardware of the render farm. Additionally, we note further testing are required to verify the impact of increasing the number of active cores on the analysis's accuracy, as we discovered no apparent patterns during our tests.

In terms of the analysis' cost we conclude that it primarily depends on the preparations of the task, rather than the analysis of the task itself. Additionally, we state this as a limitation in Cycles current implementation of its preparations stage, limiting it to a single core. Thus, we deem the cost analysis to be fairly consistent, disregarding the preparations timespan, across different tasks and hardware. Hence, we deem the analysis to be in an acceptable state with an opportunity to increase or lower the cost and accuracy of the analysis. Hereby, giving system operators the option to calibrate their analysis for their hardware.

We can also conclude that the accuracy and cost of the analysis is noticeably better compared to the previous developed MVP. Even though the former MVP was tested on a favourable task. We therefore accept the analysis as fulfilling its requirement.

4.2 System Test

In this section we evaluate the cloud render system in its entirety. We setup an experiment to verify the consistency of the system's performance while handling multiple concurrent tasks. We do this to observe, if a deployment in a public cloud introduces variations in its performance.

4.2.1 Test Setup

To evaluate the stability of the system, we setup an Azure cloud cluster with four nodes, with the specifications of the nodes listed in appendix A.4. Given these specifications, the Task Analyser and render workers will therefore use 16 cores.

Next, we submit the classroom task four times simultaneously, with different requested frame rates, and observe the behaviour of the system. With the analysis settings set to select 32 tiles, a tile size of 64x64 pixels and a render limit of five seconds. This setup was repeated three times. Notably, for this test we use strict CPU limits for the workers, similar to the tests described in section 1.2. We use strict CPU limits for the same reasons; to test the visibility of the system's accuracy in terms of the specified visual quality, simplified to the form of a requested frame rate.

4.2.2 Results

After analysing the results of the tests, by plotting and inspecting multiple graphs, we selected two representative of the tests to present. The rest can be found in appendix A.4.

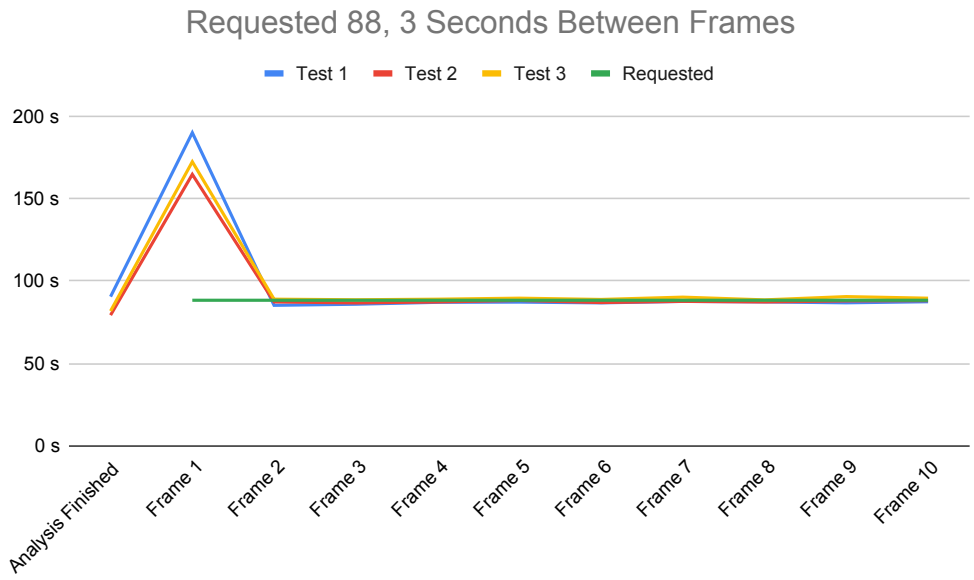


Figure 4.7: Graph showing frame rate of three tasks of "classroom" each requested frame rate of 88.3 seconds per frame

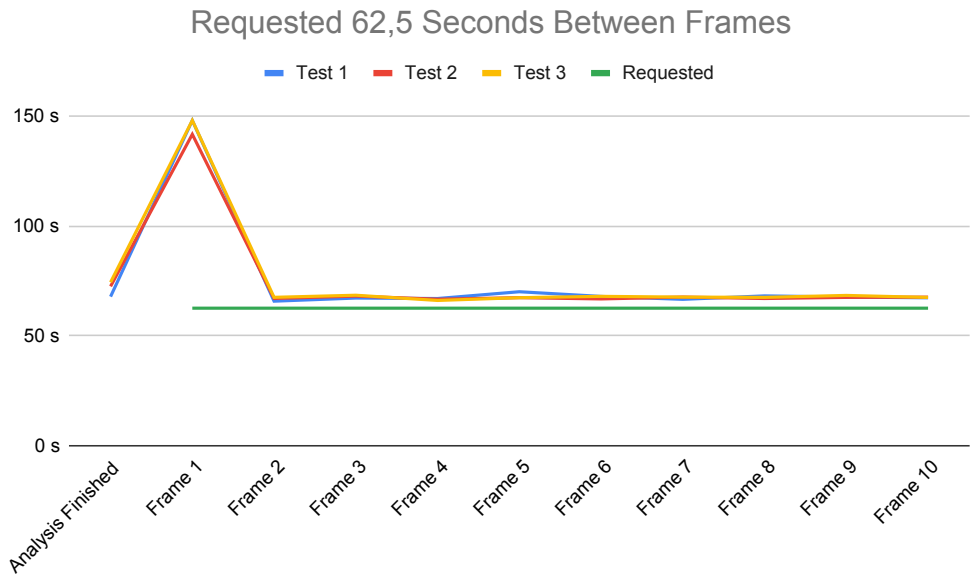


Figure 4.8: Graph showing frame rate of three tasks of "classroom" each requested frame rate of 62.5 seconds per frame

In Figure 4.7 and Figure 4.8 the horizontal axis marks the completion of milestones; analysis completion and completion of each frame. Whereas the vertical axis graph marks the amount of time since last milestone was completed. Noteworthy, we measure the analysis' completion from the task's submission. The startup of Blender, loading of task file, and communication within the system is thus added to the completion measure of the analysis.

From Figure 4.7 and Figure 4.8, we observe there is a relatively large delay before the first frame is completed. We separate this delay into two distinct parts: before completion of the analysis and after. Both parts share some primary sources for the delay: startup of Blender, reading task file and communication. And some unique sources: performing the analysis and startup of the workers' containers, for before and after the analysis respectively. After analysis completion is the delay that gives the graph the initial spike.

From the test results we observe that the delay before analysis' completion is not the dominant delay before the first frame is completed. We estimate the increased render time for the first frame, by taking the average render time of the remaining frames and subtracting it with the completion time of the first frame. From this calculation, we discovered that among our tests, that this delay from after the analysis completion averages 15.65% greater than the delay before analysis completion.

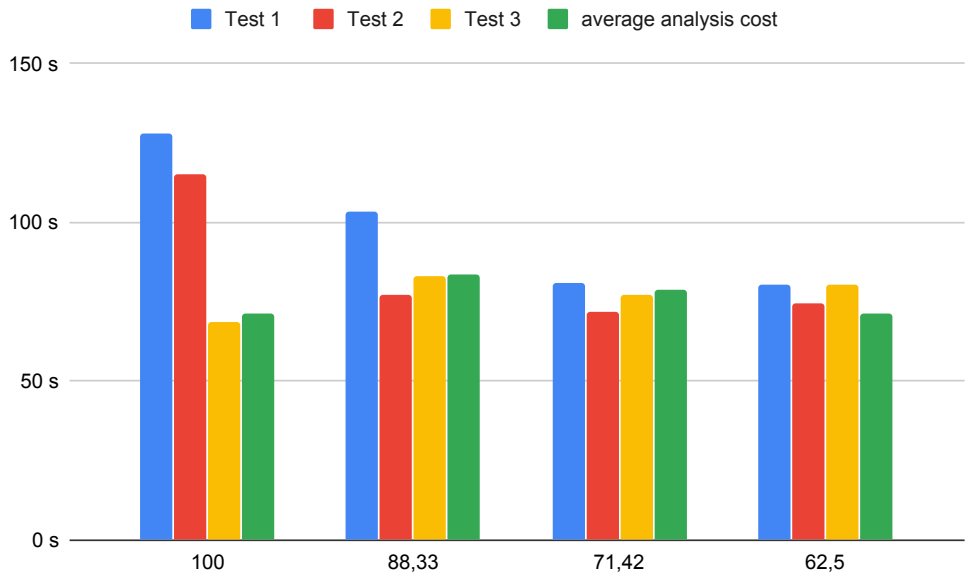


Figure 4.9: Illustration of relation between initial startup thing and the analysis time

In Figure 4.10 we remove the two delays, to observe the accuracy of the system. A graph that includes the delays can be found in appendix A.4. Figure 4.10 is separated into four clusters, each with a different requested frame rate, and each bar is labelled with the difference to the requested frame rate in percentage. We see in the graph that there is some inconsistency between different requests. With the cluster containing the requested frame rate of 88.33 showing a higher frame rate than requested, while all other tests have a slightly lower frame rate than requested. We suspect this variation is caused by rounding numbers in our system, we will discuss this further in section 5.3.

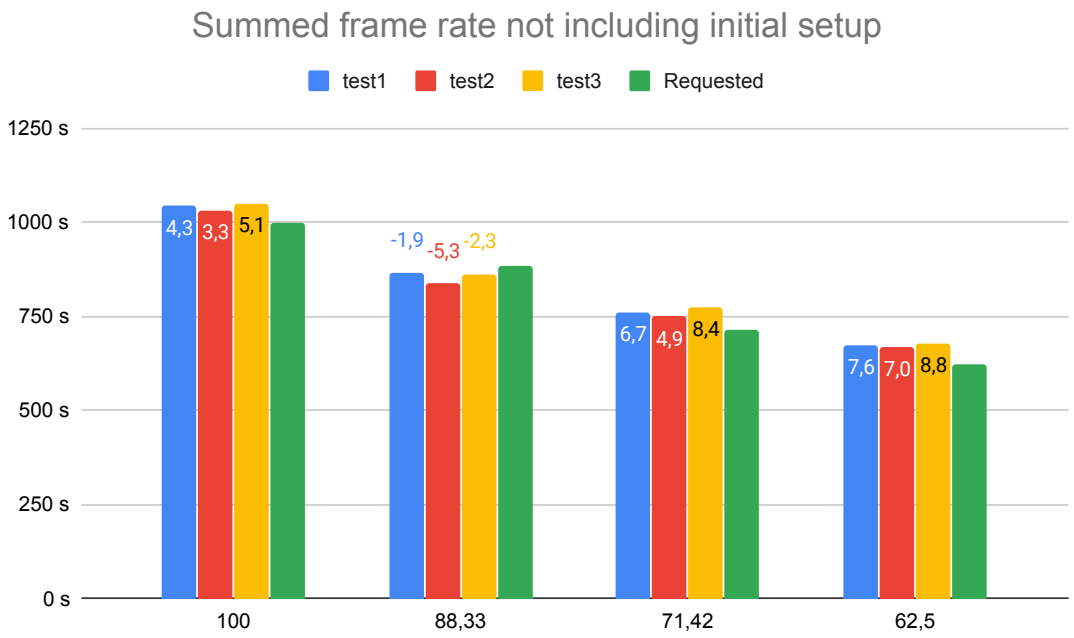


Figure 4.10: Graph showing the sum of frame times for three tests of classroom with different requested frame rates

4.2.3 Conclusion

From this evaluation we confirm a relatively consistent performance of the system, while handling multiple concurrent tasks. It did have some minor inconsistencies within the same frame rate, additionally, a single frame rate. 88,33, was noticeably far from the other frame rates. We did not expect the scaling of different sizes would have such a visible variation. We suspect it to be a result of rounding, however, we need further testing to confirm the validity of this theory. Additionally, while we were very focused on maintaining a low analysis cost, from the testing it appears that the startup time of the workers have a larger impact. This much like the analysis time detrimentally affects the analysis of a task and maybe should be considered in the future. However, the startup is not dependent on the task, but on the computational power of the worker.

We can conclude that the system is consistent, at least within our expectations of consistency. However, it does have some variations that was outside out expectations namely the change in accuracy based on the requested frame rate. However, as discussed in section 5.3, this variation does not seem to indicate a problem with the approach.

4.3 Threats Against Validity

In this section we discuss elements that might have a negative impact on the results credibility.

First, we can dispute our choice of tasks, as they are not inherently average tasks. However, we chose them because we believe that they are a good representation of the average task. As finding definitively average tasks would not be trivial, as it would require extensive

data. A possible method of approaching tests of an average task would be to perform tests on a large randomly selected number of tasks. However, finding a set of randomly selected tasks that are large enough is not simple. Additionally, this would require renting a large cloud cluster for a large amount of which is not within our budget.

Additionally, we know Intel's hyper threading technology [14] changes the performance of the Task Analyser component. Hyper threading changes the performance because we rely on all cores being equal in their computational power, hyper threading introduces some problems with that assumption. By enabling multiple threads to run on each core, which we suspect causes each core to work on multiple Device Threads in Blender. This means cores computational power fluctuate according to how much of the processor is used.

There is also a threat from the use of a public cloud. As we do not have guarantees of consistency of load on cloud by other users. This could be mitigated by performing the tests at multiple points of the day repeatedly. To confirm the influence from the possible public cloud variance.

Furthermore, we performed the system test on specific hardware. Multiple system tests, with different cluster hardware, is needed to observe if this affects not only the Task Analyser's performance but also other components as well.

4.4 Chapter Summary

In this chapter we evaluated the Task Analyser, which was observed as an improvement compared to the previous developed MVP. Out competing the former MVP on both its accuracy and cost, even though the previous MVP was tested on a favourable task. An evaluation of the system, in its entirety, then followed this, where we discussed how consistent its performance was, while handling multiple concurrent tasks. Finally, we discussed possible threats to the credibility of the evaluation.

Discussion 5

In this section we discuss the system relative to our problem statements and requirements, listed in section 1.3. This includes the current state of the Task Analyser in relation to its accuracy and cost. Additionally, we discuss the maintainability of our Blender modifications. Lastly, we discuss the system in its entirety and the future of the system in relation to its potential and attractiveness towards existing render farms, to pickup and extend their feature set for creators.

5.1 Task Analyser

In this section we discuss problems related to the Task Analyser's accuracy and cost, in addition to what we can do to improve it.

5.1.1 Accuracy

When discussing accuracy there are two kinds of inaccuracy that negatively affect separate entities. If the analysis over approximates the *computational requirement* of a task, it results in the task being finished before its deadline. Conversely, an under approximation results in the task being finished after its deadline. Finishing after the deadline negatively affects the creator, while before its deadline means the render farm provided unnecessary computational power for the task, thus negatively affecting the capacity of the render farm.

For our use case, guarantees of not under approximating is preferable, as the system is meant to improve the animation workflow for the creators. However, because of the nature of path tracing, no method of providing guarantees was found. As such, perhaps the analysis should be complemented with direct monitoring that adjusts the allocated resources if the workers are lagging behind. Or maybe the analysis should be more pessimistic, to help prevent under approximations. This however would increase the over approximation of tasks. Our current implementation and design support a more pessimistic approach, by looking at the highest render time of a sample, instead of taking the average of all samples. However, we have not tested this pessimistic approach, thus; we leave it up to the system operators, to determine if this pessimistic approach is preferable for them.

Furthermore, since a hardware feature such as Intel's hyper threading can affect the analysis' accuracy. We know that the accuracy depends on some hardware features. However, we cannot definitively say that the accuracy is not somewhat dependent on the actual hardware as well. If this is the case, system operators might have to calibrate the hardware running the analysis in their system.

5.1.2 Cost

We can confirm that the relation between the render tasks' render time and the analysis' cost depends on the hardware. However, we do not see a genuine way of avoiding this. Some hardware has more powerful cores, whereas others have a larger number of cores. Powerful cores make the preparations faster, whereas additional cores make the rendering faster.

From our component evaluation of the Task Analyser, we found the preparations to be the primary cost of the analysis. Thus, we discuss possible ways of reducing this.

One approach is to not perform the preparations. However, as preparations are a measure to lower render times, by not doing preparations the render time of each sample increases. Hence, skew with the results of the analysis. Furthermore, with Cycles recently moving to the Embree framework [15], the preparations have become faster. Thus, improvements to the preparation stage in Cycles are continuously improved. Additionally, there exist work regarding algorithms to build BVH trees in parallel on a CPU [16, 17]. Additionally, as detailed in section 2.1, BVH building is one of the most time-consuming preparations preformed by Cycles.

The work of [16] propose an algorithm that yields superior BVH tree build performance compared to high-quality builders implemented in the Embree framework while closely matching their ray tracing performance. We cannot conclude on the impact of theses algorithms in terms of the analysis' cost. However, as separate BVH builders can be implemented in Cycles, primarily for the analysis, this may yield a lower analysis cost. Hence, we leave the discussion of parallel BVH building as a potential future improvement.

5.2 Maintainability of Blender Modifications

In this section we discuss the maintainability of the changes we introduced into Blender's source code.

All the changes we have introduced, except the additions in the backend of Cycles CPU rendering, fit Blender's core design. Thus, even if there are major changes, future developers should be able to follow Blender's new design philosophy. However, the changes in the backend of Cycles CPU rendering is an addition to Cycles's design. This might therefore be harder to maintain. But since it is located in the core of Cycles, it should be a fairly stable region, thus not exposed to updates commonly.

5.3 System in Its Entirety

This section discuss the system in its entirety in relation to its transferability, performance, and attractiveness for existing render farms.

5.3.1 Transferability

A transferable design can be used by a larger set of systems, without making major changes. This is a focus point as the use cases of the design is limited by how many entities can

implement a feature according to our design. Which is important as the main contribution of this project is the proposed design and an evaluation of this design.

As we wanted the approach of the system to be usable for as many entities as possible, the transferability of our system was a focus. The design of the system is transferable as a result of this focus. However, the transferability of the analysis is limited to path tracing based engines, as it is based on the theoretical basis of path tracing.

There exist three primary types of engines in today's render tools, rasterization, ray tracing and path tracing. However, rasterization based engines avoid the computational heavy techniques required for ray tracing and path tracing engines. Therefore, its appeal for render farms is lesser, In addition, many rasterization engines are real-time engine, such as Blender's second engine EEVEE [18]. Thus, it is primarily an analysis approach for ray tracing based engines that is appealing. Furthermore, an analysis approach for ray tracing engines can most likely be inspired by our proposed approach, as the two types of engines are similar in their theoretical basis. Therefore, even though our design of the analysis is limited to path tracing based engines, we predict designing an analysis for ray tracing tasks to be fairly similar.

5.3.2 System Performance

From our evaluations, the system appears to produce an accurate frame rate. However, we have some concern as to why different requested frame rates affects the accuracy, see Figure 4.10, given a similar analysis approximation. It is unclear how high a variation in accuracy is possible. However, we theorise that it is primarily due to rounding errors, when using the equation Equation 3.1 from section 3.4. As the only one showing this problem has an infinite repeating decimal; 88,33. As we calculate with doubles, which then has to be rounded into 1000th parts of a vCPU, as that is the smallest vCPU unit in Azure. However, this theory is unproven and would require some additional testing to verify.

5.3.3 The System's Attractiveness for Existing Render Farms

We intend our system for either mid to high end hobbyist or small to medium firms. We expect smaller entities are more lenient with their deadlines, thus they won't be as focused on improving their workflow. While big corporations, for one, have their own render farm and their tasks requiring large rendering times. Thus, these corporations such as Pixar [1] have people dedicated to provide better reviews by analysing their render tasks.

We believe that we have proposed a design for a render system using an analyser that might be used in existing render farms. However, the designed system can not be directly used as a standalone render system as it lacks some features. The design of the system could therefore instead be used to implement an optional feature in existing render farms for creators. Additionally, with our evaluation we showed an accuracy deviation of 4% from the actual render time and an added 5% cost to the overall render time. And together with an adjustable cost and accuracy, that can be tailored for the specific existing system. We showed the system's potential and attractiveness towards existing render farms.

5.4 Future Works

In this section we discuss possible future development of the system.

5.4.1 Startup Reduction

From our evaluation we observed a startup time that eclipsed the analysis time. We suspect this is primarily caused by the startup of the render workers' containers and opening the task file in Blender. This startup time might be mitigated by having the Task Analyser serve as a render worker after it finishes the analysis. This would enable at least one worker to start rendering without the initial delay. However, this is not without its drawbacks, as this require a Task Analyser to be started for each task. This would add an initial startup to the analysis instead. Additionally, it would not eliminate the startup delay for the other workers.

Another approach could be to take a snapshot of the Task Analyser's Blender state once it has fully opened the task file. The workers could then use this snapshot, meaning instead of waiting for Blender to read the task file, each worker starts in a state where the task file have already been opened. This is, however, only theoretical, it is unclear if this is possible or faster than reading the task file. Furthermore, there are ways of optimising the startup of a container. E.g. rigorously evaluating the Docker images' files, removing unnecessary data to reduce its size. It is unclear how much this would reduce the initial startup, however, any reduction would benefit the system. Lastly, if we are intent on reducing the delay, we might have workers on standby that can take tasks without needing container creation or Blender startup. This would however incur additional resource costs, as maintaining several workers on standby requires resources. Thus, it is a judgement call, as to how much to sacrifice for a reduction in the delay.

5.4.2 Missing Requirements

The next step in the development of this system would be to convert it to a standalone rendering system. This would mean putting a larger emphasis on some of the requirements. This is requirements already implemented in current existing render farms, thus, were de-emphasized for the project. The system would need the following things:

- **Interface** to provide a GUI for the users, important as we cant expect all users to be technological.
- **Subdivide frame functionality** to enable greater possible granularity for tasks.
- **Merge of sub frames functionality** merging is needed to utilise subdivision of frames.
- **Encryption of communication** artifacts used for commercial purposes has to be secured in some form.
- **Account features** common practice to have features that allow access to all that is available for the creator without having an individual key/password for each artifact.

Beyond this there are features that would improve the analysis' performance.

- **A simultaneous analysis and rendering:** by performing the analysis in smaller chucks, workers can be started earlier. Additionally, the accuracy of the analysis

may also increase. E.g. given a task consisting of 100 frames, and assuming we want to analyse every 10th frame. Then instead of analysing the 10 frames upfront, we analyse one frame, and begin rendering the first 10 frames. While the first 10 frames are rendering, we analyse the next frame, and the result of this analysis is used to determine the resources given to the workers rendering the next 10 frame. Hereby, the variation between frames have less of an impact on the analysis' accuracy. This, however, comes with the drawback of more communication between components in the system, thus, more overhead. Additionally, to support this repeated analysis approach, we have to change the allotted CPU resources for the workers. However, changing the allotted CPU of workers in our system requires a termination of workers and the startup of new workers, which would infer a new delay. Another method of changing the CPU resources allotted for the task is to scale the number of workers, while maintaining the life of existing workers.

- **Selection of frames for analysis:** just like the selection of pixels can affect the accuracy of the system, the frames chosen for analysis can too. Thus, choosing frames based on an assumption might yield better results, such as selecting the frames with the most key-frames. Key-frames representing either the start transformation or end transformation of an object's animation. Hence, frames with most key-frames have most moving objects at once. Thus, it can be used to approximate the complexity of a frame.

Conclusion 6

In this project, we have sought to improve the review process in 3D computer-generated imagery production for creators. We have proposed a design of a render farm with additions targeting the review process. We developed a minimum viable product to evaluate and showcase the system's potential and attractiveness towards existing render farms.

As described in section 1.3 we came to the following problem statement.

Problem Statement

> *How can we establish a cloud rendering system, with a task analysis, to accommodate a specified visual quality from creators by determining the corresponding computational requirement in accordance to the creators expectations?*

To answer the question, we explored the theoretical basis for Cycles' path tracing, Blender's internal approximation, and the importance of pixel selection for an approximation. Given that Blender is in rapid development and new improvements arise, meant creating a render system that both is maintainable and transferable to new solutions.

We incorporated the concept of *visual quality* though a requested frame rate specified by the creators. Additionally, we introduced an automatic analysis of render tasks to determine the *computational requirement* corresponding to the requested *visual quality*. We created a specialised tile selector, to both, reuse the results from the preparations and get a better average pixel selection. Furthermore, we proposed an improved approximation approach of tasks' render times. By combing this new approximation with the tile selector, and modifying Blender's source code, we achieved an analysis with a controllable and consistent cost across a variety of tasks. However, because of Blender's limitation of its single core preparations, a non controllable delay are introduced, before the analysis can begin.

Additionally, the analysis achieved an accuracy deviation of 4% from the actual render time of the tasks and a cost of 5%. Which compared to our previous developed minimum viable product considerably outperforms it in both accuracy and cost. From this, we accept the analysis as fulfilling our requirement.

From our system tests we showed our system can accommodate a specified *visual quality*, with multiple concurrent render tasks running. However, as presented there are some threats to the viability of the tests. The most pressing of which is the choice of the average tasks. As such, testing as proposed with a large randomly selected set of render tasks,

would serve as a final validation of the system. But our tests showed a consistency that was within our expectations.

Finally, we conclude we were able to create a render system that fulfils all the *must have* requirements, specified in subsection 1.2.1. Furthermore, we conclude we have created both a maintainable and transferable design. In addition, we implemented and evaluated a new minimum viable product to showcase the system's potential and attractiveness towards existing render farms, to pickup and extend their feature set for creators.

Bibliography

[1] Leif Pedersen. *Pixar's USD Pipeline*. Visited: 27/02/2020. https://renderman.pixar.com/stories/pixars-usd-pipeline, Dec. 2019.

[2] Aldric Chang. *The Process of 3D Animation*. Visited: 11/12/2019. https://www.media-freaks.com/the-process-of-3d-animation/, 2019.

[3] Payam Adib. *All you need to know about 3D animation pipeline*. Visited: 17/11/2019. https://dreamfarmstudios.com/blog/all-you-need-to-know-about-3d-animation-pipeline/, 2019.

[4] Payam Adib. *The final step in 3D animation production: 3D Rendering*. Visited: 23/12/2019. https://dreamfarmstudios.com/blog/the-final-step-in-3d-animation-production-3d-rendering/, Dec. 2019.

[5] Kim Larsen and Martin Karkov. *Cloud Render System Specialised in Animation Workflow*. Can be emailed upon request. Jan. 2020.

[6] Blender. *Subdivision Surface Modifier*. Visited: 18/12/2019. https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/subdivision_surface.html, 2019.

[7] Blender. *Bounding Volume Hierarchy*. Visited: 18/12/2019. https://wiki.blender.org/wiki/Source/Render/Cycles/BVH, 2019.

[8] Lukas Stockner. *Cycles: Implement persistent storage for all scene data*. Visited: 17/03/2020. https://developer.blender.org/D2613, 2017.

[9] Blender. *Cycles: Threads*. Visited: 12/04/2020. https://wiki.blender.org/wiki/Source/Render/Cycles/Threads, 2018.

[10] Blender. *Devices*. Visited: 25/05/2020. https://wiki.blender.org/wiki/Source/Render/Cycles/Devices.

[11] Blender. *Modules*. Visited: 25/05/2020. https://wiki.blender.org/wiki/Modules.

[12] Blender. *Demo Files*. Visited: 25/05/2020. https://www.blender.org/download/demo-files/.

[13] Blender. *Blender 2.80 Release Notes*. Visited: 01/06/2020. https://wiki.blender.org/wiki/Reference/Release_Notes/2.80, 2019.

[14] Intel. *What Is Hyper-Threading?* Visited: 01/06/2020. http://intel.com/content/www/us/en/gaming/resources/hyper-threading.html, 2020.

[15] Blender. *Bounding Volume Hierarchy*. Visited: 02/06/2020. https://wiki.blender.org/wiki/Source/Render/Cycles/BVH, 2020.

[16] J. Hendrich, D. Meister, and J. Bittner. "Parallel BVH Construction Using Progressive Hierarchical Refinement". In: *Comput. Graph. Forum* 36.2 (May 2017), pp. 487–494. ISSN: 0167-7055. DOI: 10.1111/cgf.13143. URL: https://doi.org/10.1111/cgf.13143.

[17] Brian Fischer and Ian Huang. *Parallel BVH Construction on a CPU*. Visited: 02/06/2020. http://www.bwfischer.com/oculusRayTracer.html, 2010.

[18] Blender. *Eevee: Introduction*. Visited: 02/06/2020. https://docs.blender.org/manual/en/latest/render/eevee/introduction.html, 2020.

Appendix A

A.1 Analyser Logs

The following is three edditied logs from rendering files that ilustrate the actions taken before actual rendering is started and Blenders approximation.

File:/Classroom.blend:

Fra:1 Mem:250.76M | Time:00:00.08 | Synchronizing object | door_window

[...]

Fra:1 Mem:277.09M | Time:00:00.52 | Initializing

[...]

Fra:1 Mem:394.65M | Time:00:01.02 | Updating Meshes

[...]

Fra:1 Mem:406.03M | Time:00:01.15 | Updating Scene BVH | Building

Fra:1 Mem:406.04M | Time:00:01.15 | Updating Scene BVH | Building BVH

[...]

Fra:1 Mem:442.83M | Time:00:01.49 | Updating Images

[...]

Fra:1 Mem:443.95M | Time:00:01.66 | Rendered 0/240 Tiles, Sample 0/100

Fra:1 Mem:443.96M | Time:00:04.91 | Remaining:12:08.87 | Rendered 1/240 Tiles, Sample 100/100

Fra:1 Mem:443.96M | Time:00:08.14 | Remaining:12:02.24 | Rendered 2/240 Tiles, Sample 100/100

Fra:1 Mem:443.96M | Time:00:11.39 | Remaining:11:59.69 | Rendered 3/240 Tiles, Sample 100/100

Fra:1 Mem:443.96M | Time:00:14.64 | Remaining:11:56.87 | Rendered 4/240 Tiles, Sample 100/100

Fra:1 Mem:443.96M | Time:00:18.00 | Remaining:11:58.95 | Rendered 5/240 Tiles, Sample 100/100

Fra:1 Mem:443.96M | Time:00:21.30 | Remaining:11:56.95 | Rendered 6/240 Tiles, Sample 100/100

[...]

Fra:1 Mem:443.96M | Time:03:39.55 | Remaining:09:59.19 | Rendered 60/240 Tiles, Sample 100/100

[...]

Fra:1 Mem:443.96M | Time:06:51.74 | Remaining:05:58.82 | Rendered 120/240 Tiles, Sample 100/100

[...]

Fra:1 Mem:443.96M | Time:09:46.24 | Remaining:03:09.12 | Rendered 170/240 Tiles, Sample 100/100

[...]

Fra:1 Mem:443.96M | Time:13:19.49 | Remaining:00:00.88 | Rendered 239/240 Tiles, Sample 100/100

Fra:1 Mem:443.96M | Time:13:20.15 | _mainScene, interior | Rendered 240/240 Tiles

Fra:1 Mem:443.95M | Time:13:20.15 | _mainScene, interior | Finished

Fra:1 Mem:221.77M | Time:13:20.17 | Sce: _mainScene Ve:0 Fa:0 La:0

file: pavillo_barcelone_v1.2.blend:

Read blend: /pavillo_barcelone_v1.2.blend

Fra:1 Mem:122.44M | Time:00:00.81 | Synchronizing object | Cube.024

Fra:1 Mem:136.77M | Time:00:01.19 | Synchronizing object | tree_scatter

Fra:1 Mem:148.77M | Time:00:01.22 | Initializing

Fra:1 Mem:109.41M | Time:00:01.25 | Waiting for render to start

Fra:1 Mem:109.41M | Time:00:01.26 | Loading render kernels (may take a few minutes the first time)

Fra:1 Mem:109.41M | Time:00:01.31 | Updating Scene

Fra:1 Mem:123.28M | Time:00:01.48 | Updating Mesh BVH Cube.091 1/13 | Building BVH

Fra:1 Mem:123.28M | Time:00:01.48 | Updating Mesh BVH Plane.006 8/13 | Building BVH

Fra:1 Mem:126.00M | Time:00:01.48 | Updating Mesh BVH Plane.006 8/13 | Packing BVH triangles and strands

Fra:1 Mem:127.14M | Time:00:01.49 | Updating Mesh BVH Plane.006 8/13 | Packing BVH nodes

Fra:1 Mem:127.29M | Time:00:01.49 | Updating Mesh BVH Plane.006 8/13 | Packing BVH triangles and strands

Fra:1 Mem:127.54M | Time:00:01.50 | Updating Mesh BVH Cube.035 10/13 | Packing BVH triangles and strands

Fra:1 Mem:127.55M | Time:00:01.50 | Updating Mesh BVH Cube.035 10/13 | Packing BVH nodes

Fra:1 Mem:127.72M | Time:00:01.51 | Updating Mesh BVH Cube.028 11/13 | Building BVH

Fra:1 Mem:127.75M | Time:00:01.51 | Updating Mesh BVH Cube.028 11/13 | Packing BVH nodes

Fra:1 Mem:129.94M | Time:00:01.51 | Updating Mesh BVH Cube.028 11/13 | Building BVH

Fra:1 Mem:129.97M | Time:00:01.52 | Updating Mesh BVH Cube.028 11/13 | Packing BVH nodes

Fra:1 Mem:129.98M | Time:00:01.52 | Updating Mesh BVH Cube.028 11/13 | Packing BVH triangles and strands

Fra:1 Mem:129.81M | Time:00:01.52 | Updating Mesh BVH Cube.033 12/13 | Building BVH

Fra:1 Mem:130.65M | Time:00:01.53 | Updating Mesh BVH Cube.033 12/13 | Packing BVH nodes

Fra:1 Mem:129.17M | Time:00:01.53 | Updating Mesh BVH Branches 13/13 | Building BVH

Fra:1 Mem:134.59M | Time:00:01.54 | Updating Mesh BVH Branches 13/13 | Packing BVH triangles and strands

Fra:1 Mem:134.04M | Time:00:01.54 | Updating Mesh BVH Branches 13/13 | Packing BVH nodes

Fra:1 Mem:135.29M | Time:00:01.57 | Updating Mesh BVH Branches 13/13 | Packing BVH triangles and strands

Fra:1 Mem:141.33M | Time:00:01.59 | Updating Mesh BVH Branches 13/13 | Packing BVH nodes

Fra:1 Mem:137.86M | Time:00:01.62 | Updating Scene BVH | Building

Fra:1 Mem:138.01M | Time:00:01.70 | Updating Scene BVH | Building BVH

Fra:1 Mem:144.78M | Time:00:01.85 | Updating Scene BVH | Packing BVH triangles and strands

Fra:1 Mem:150.01M | Time:00:01.95 | Updating Scene BVH | Packing BVH nodes

Fra:1 Mem:162.26M | Time:00:02.07 | Updating Scene BVH | Copying BVH to device

Fra:1 Mem:272.34M | Time:00:02.56 | Updating Device | Writing constant memory

Fra:1 Mem:272.34M | Time:00:02.57 | Updating Device | Writing constant memory | Compiling render kernels

Fra:1 Mem:272.34M | Time:00:02.57 | Updating Device | Writing constant memory

Fra:1 Mem:272.65M | Time:00:02.57 | Rendered 0/28 Tiles, Sample 0/1000

Fra:1 Mem:274.34M | Time:00:03.62 | Rendered 0/28 Tiles, Sample 110/1000

Fra:1 Mem:274.34M | Time:00:04.68 | Rendered 0/28 Tiles, Sample 231/1000

Fra:1 Mem:274.34M | Time:00:05.73 | Rendered 0/28 Tiles, Sample 352/1000

Fra:1 Mem:274.34M | Time:00:06.79 | Rendered 0/28 Tiles, Sample 473/1000

Fra:1 Mem:274.34M | Time:00:07.84 | Rendered 0/28 Tiles, Sample 594/1000

Fra:1 Mem:274.34M | Time:00:08.90 | Rendered 0/28 Tiles, Sample 715/1000

Fra:1 Mem:274.34M | Time:00:09.95 | Rendered 0/28 Tiles, Sample 836/1000

Fra:1 Mem:274.34M | Time:00:11.01 | Rendered 0/28 Tiles, Sample 957/1000

Fra:1 Mem:272.65M | Time:00:11.40 | Rendered 1/28 Tiles, Sample 1000/1000

File:/mr_elephant.blend:

Read blend: /mr_elephant.blend

Fra:1 Mem:1786.70M | Time:00:03.39 | Synchronizing object | Circle.015

Fra:1 Mem:2698.49M | Time:00:05.98 | Synchronizing object | Plane

Fra:1 Mem:2698.49M | Time:00:05.99 | Initializing

Fra:1 Mem:1619.33M | Time:00:06.11 | Waiting for render to start

Fra:1 Mem:1619.33M | Time:00:06.11 | Loading render kernels (may take a few minutes the first time)

Fra:1 Mem:1619.33M | Time:00:06.17 | Updating Scene

Fra:1 Mem:2070.06M | Time:00:07.01 | Updating Mesh BVH Cube.211 1/1 | Building BVH

Fra:1 Mem:2081.73M | Time:00:07.09 | Updating Mesh BVH Cube.211 1/1 | Packing BVH triangles and strands

Fra:1 Mem:2092.62M | Time:00:07.10 | Updating Mesh BVH Cube.211 1/1 | Packing BVH nodes

Fra:1 Mem:2086.38M | Time:00:07.12 | Updating Scene BVH | Building

Fra:1 Mem:2086.38M | Time:00:07.12 | Updating Scene BVH | Building BVH

Fra:1 Mem:2774.98M | Time:00:09.24 | Updating Scene BVH | Building BVH 0%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:09.49 | Updating Scene BVH | Building BVH 7%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:09.75 | Updating Scene BVH | Building BVH 16%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:10.00 | Updating Scene BVH | Building BVH 26%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:10.27 | Updating Scene BVH | Building BVH 30%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:10.52 | Updating Scene BVH | Building BVH 32%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:10.78 | Updating Scene BVH | Building BVH 34%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:11.04 | Updating Scene BVH | Building BVH 39%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:11.29 | Updating Scene BVH | Building BVH 52%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:11.55 | Updating Scene BVH | Building BVH 65%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:11.80 | Updating Scene BVH | Building BVH 77%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:12.06 | Updating Scene BVH | Building BVH 89%, duplicates 0%

Fra:1 Mem:2774.98M | Time:00:12.36 | Updating Scene BVH | Packing BVH triangles and strands

Fra:1 Mem:3414.12M | Time:00:12.69 | Updating Scene BVH | Packing BVH nodes

Fra:1 Mem:3063.78M | Time:00:14.03 | Updating Scene BVH | Copying BVH to device

Fra:1 Mem:3503.25M | Time:00:14.75 | Updating Device | Writing constant memory

Fra:1 Mem:3503.25M | Time:00:14.76 | Updating Device | Writing constant memory | Compiling render kernels

Fra:1 Mem:3503.25M | Time:00:14.76 | Updating Device | Writing constant memory

Fra:1 Mem:3503.26M | Time:00:14.76 | Rendered 0/1 Tiles, Sample 0/2500

Fra:1 Mem:3503.38M | Time:00:15.80 | Remaining:00:01.24 | Rendered 0/1 Tiles, Sample 1104/2500

Fra:1 Mem:3503.38M | Time:00:16.83 | Remaining:00:00.26 | Rendered 0/1 Tiles, Sample 2208/2500

Fra:1 Mem:3503.26M | Time:00:17.11 | Rendered 1/1 Tiles

A.2 Blender Components Description

This section will describe the components of blender. Description A.1 is a description of Blenders components

A.3 Analyser Tests

Hardware specifications: Processor: Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz (8 CPUs), $\tilde{4}$.6GHz

Render APIs: This component contains render callbacks, such as starting and canceling render jobs. Additionally, it is responsible for creating the three layers of threads.

Data, Assets & I/O: Responsible for storing and managing all data and I/O related tasks.

Blender Python API: The interface used by the Task Analyser component to give Blender commands.

Blender Sync: Responsible for keeping information in sync during the render job. This is archived though a synchronisation process of information from the Blender Python API and the .blend file. This component is the basics for the Blender Thread.

Session: Responsible for distributing tasks across Device Threads, in addition to collect and monitor the process of each Device Thread.

Tile Manager: The component containing all tile selector algorithms, it is responsible for dividing a frame into tiles and return them to the Session Thread. Each tile are represented by two coordinates corresponding to a square of the frame.

Device CPU: The backend for CPU rendering, it is responsible for rendering of tiles, and the component for which Device Threads are created from.

Description A.1: Each component' implication in Blender's internal communication

Ram: DDR4-3000 C15 - 16GB
Storage: Samsung 860 EVO 2.5" SSD - 500GB (Read: 560 MB/s, Write: 271 MB/s); (Read: 73974 IOPS, Write: 55664)

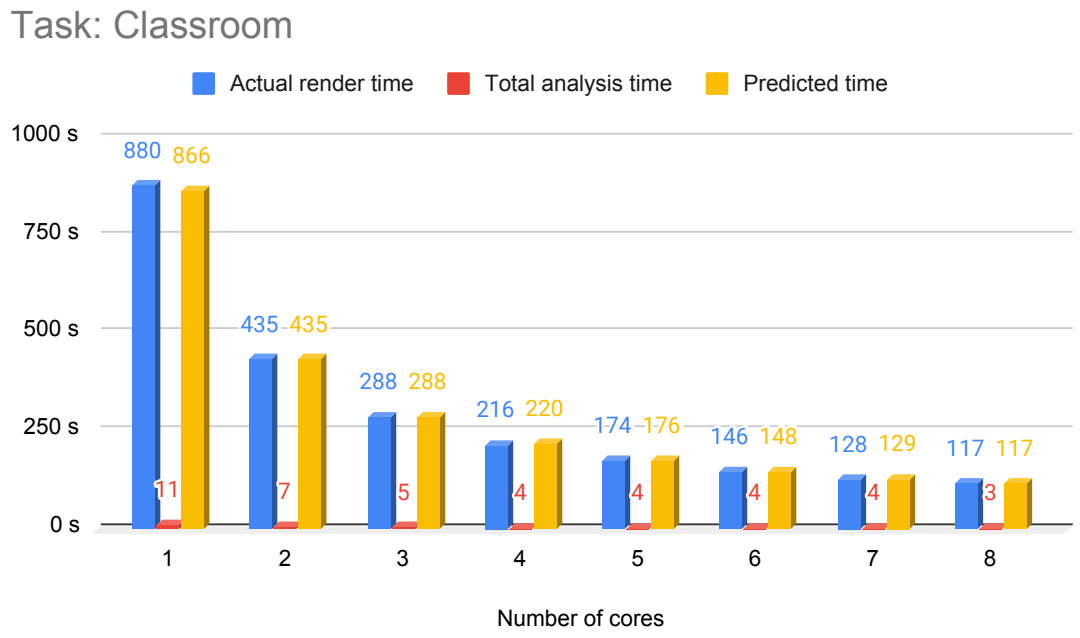


Figure A.1: Graph showing overview of analysis cost and accuracy of of task classroom

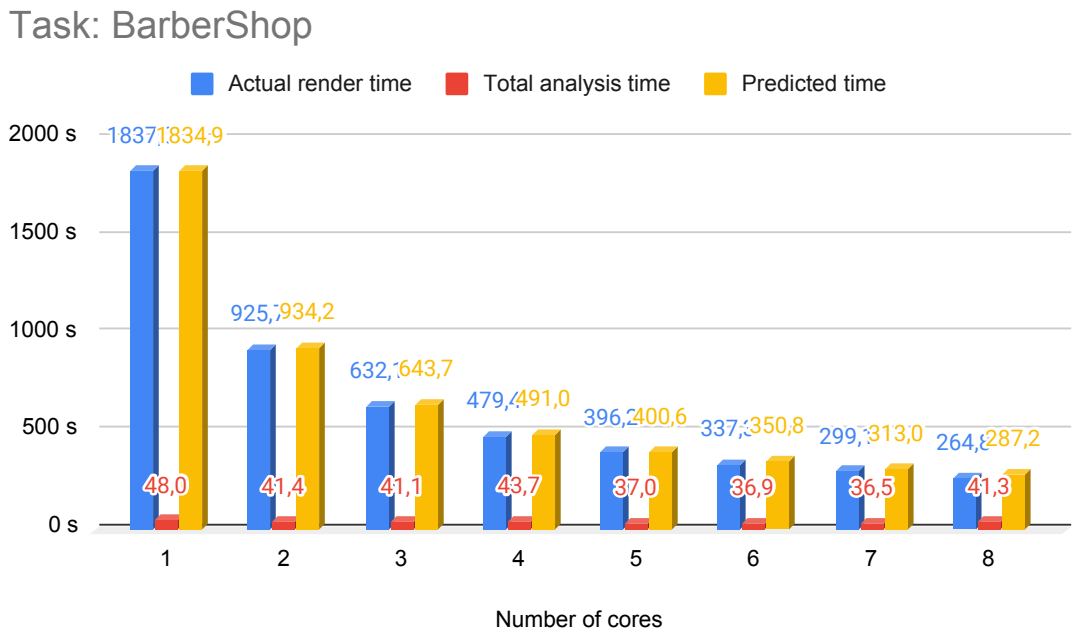


Figure A.2: Graph showing overview of analysis cost and accuracy of of task Barbershop

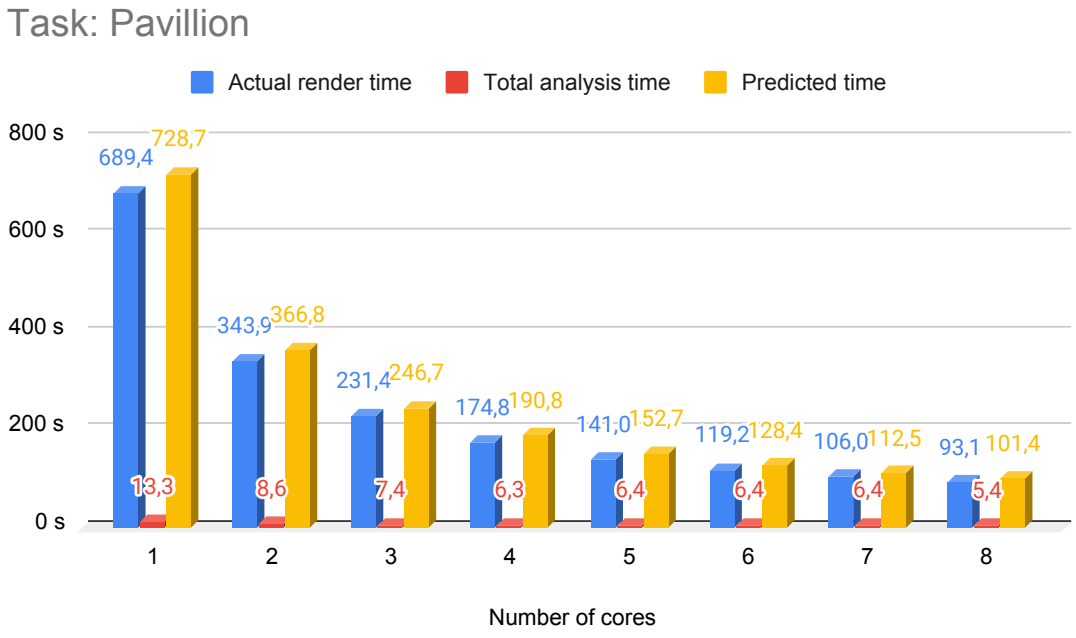


Figure A.3: Graph showing overview of analysis cost and accuracy of of task pavilion

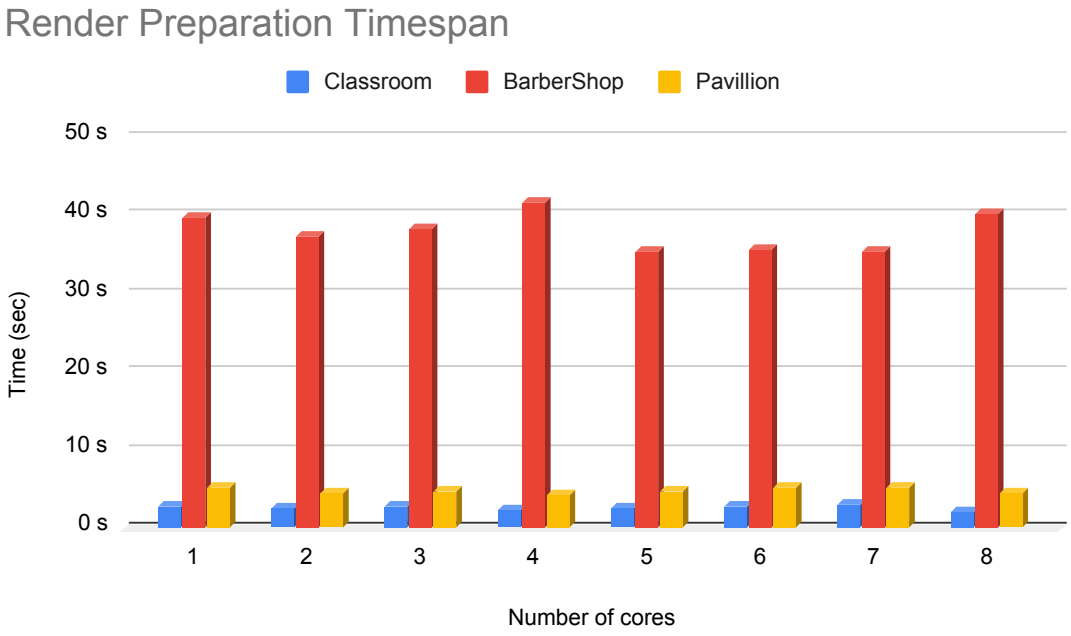


Figure A.4: Graph showing the render preparation timespan for all three tasks

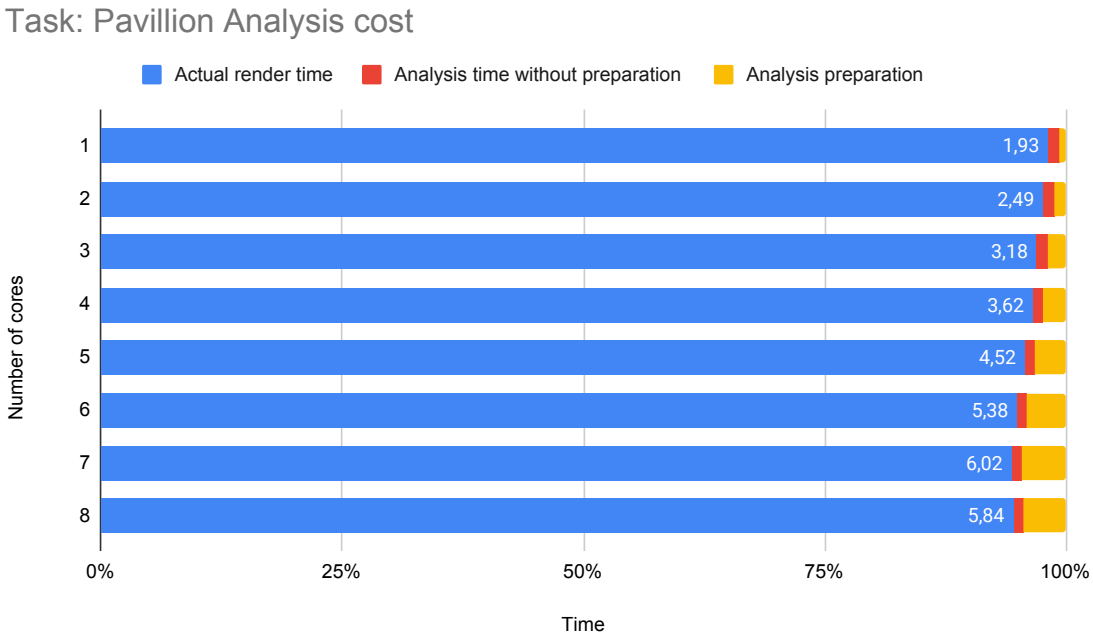


Figure A.5: Graph showing accuracy of analysis of task Pavillion

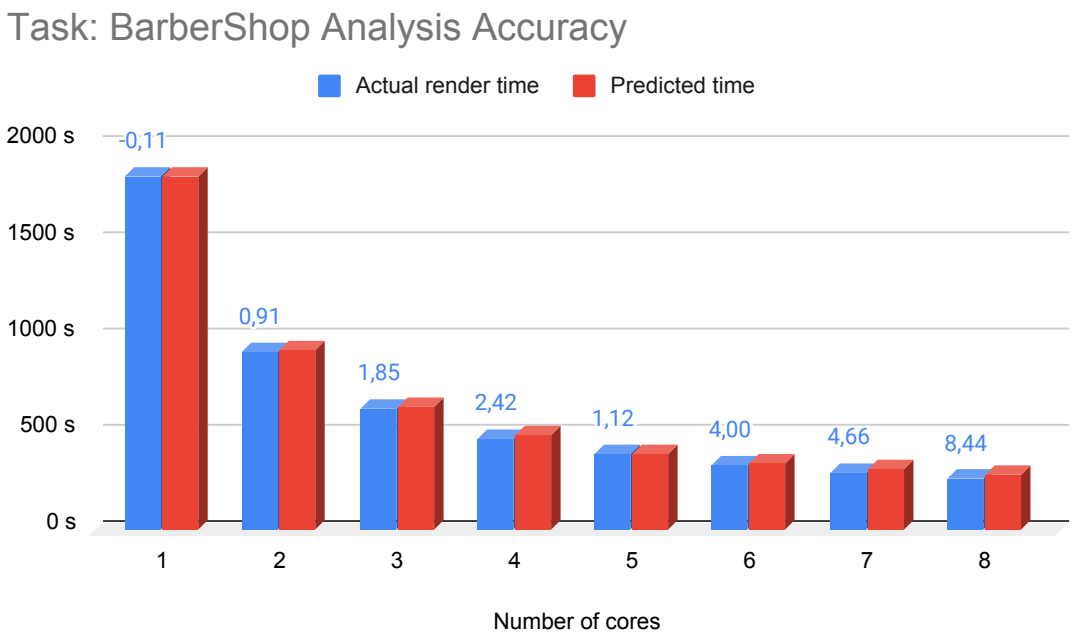


Figure A.6: Graph showing accuracy of analysis of task barbershop

A.4 Stability Test

4 nodes of Azure h16
Processor; Intel Xeon E5 2667 v3 3,2 GHz
Memory: 112 GB
Memory bandwidth: 80 GB/s

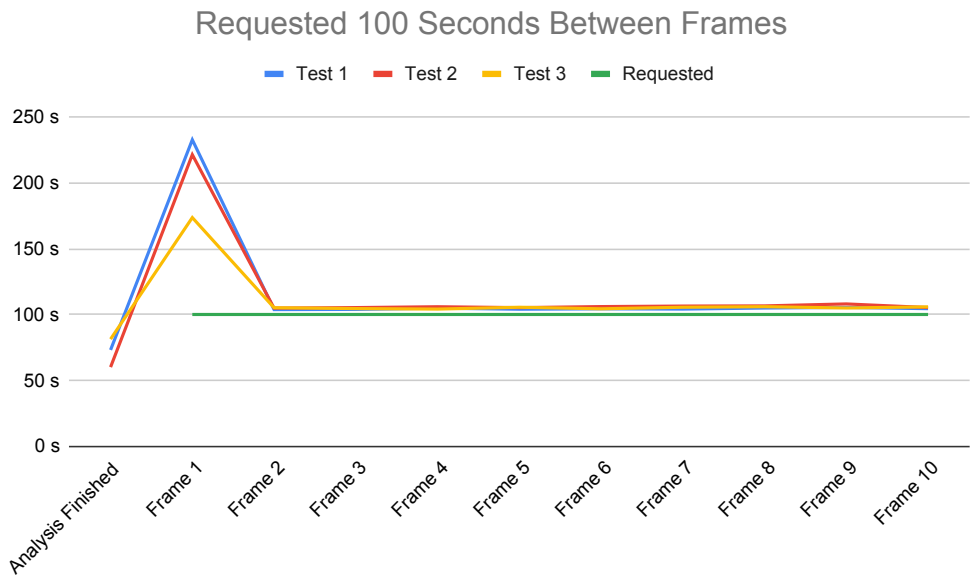


Figure A.7: Graph showing frame rate of three tasks of "classroom" each requested 100

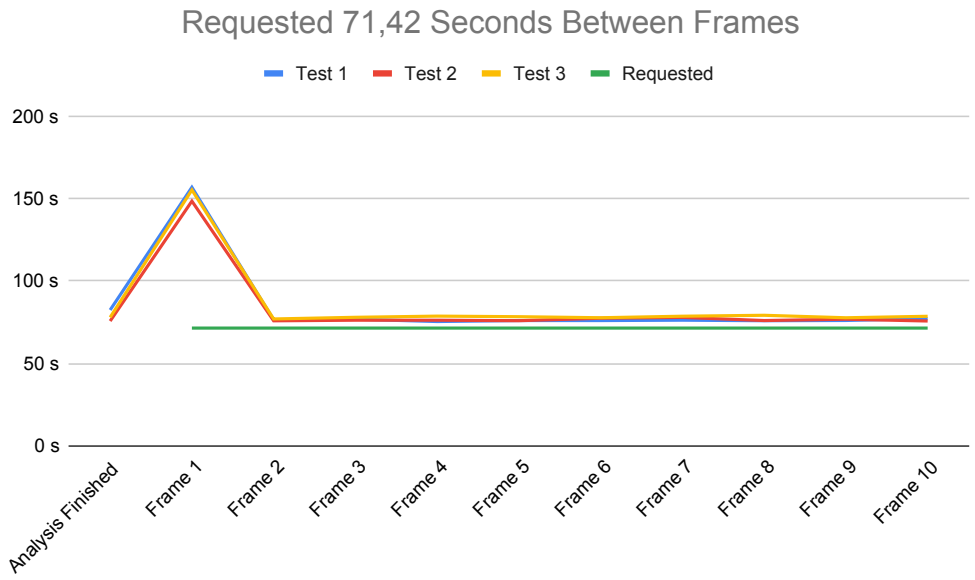


Figure A.8: Graph showing frame rate of three tasks of "classroom" each requested 71.42