Linköpings universitet/Linköping University | IDA

Bachelor | Computer Engineering

Spring 2022 | **LIU-IDA/LITH-EX-G--22/033--SE**

Optimizing Environment Mapping in Redway3D

Oscar Davidsson, Jakob Karlsson Abay

Supervisor: Filip Strömbäck

Examinator: Jonas Wallgren

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: https://ep.liu.se/.

© 2022 Oscar Davidsson, Jakob Karlsson Abay

Abstract

Many contemporary types of 3D design software use some form of environment mapping, where the environment surrounding the 3D object is rendered to create a panoramic view similar to what we would see in the real world. This not only serves to make the scene look more realistic, but also helps to calculate light effects within the scene. Although environment mapping has been around for a while, and today's methods are highly optimized, they are still not trivially cheap to do, especially on lower end hardware.

Our thesis investigates how an environment mapping method currently in use by our client can be further optimized. This optimization intends to serve two main purposes - to improve performance for clients with lower quality hardware and to allow users to increase texture resolution without extra cost on performance. Two alternative methods are presented, both based somewhat loosely on the idea of occlusion culling. The two approaches are then tested and compared to the original solution in terms of speed, memory utilization and network performance.

Although both approaches show promise and outperform the original solution in some of the tests, they still lack the versatility of the original solution and suffer from some major flaws, making them less appealing alternatives for the general customer. The first approach managed to perform well in all three areas of measurement, but suffers a drawback which limits its use in a real-world scenario. The second approach did not have the same drawback which may make it a more viable option. However, the results of the second approach were not as positive as the first one. With that said, it showed some promise for users who do their rendering on a separate server. While this solution may not yet be viable for the general user, it may serve well for users with more unique needs. To make this approach a viable solution for the general user improvements in regards to rendering speed and GPU utilization will have to be investigated further.

keywords: environment mapping, occlusion culling, graphics

Abstrakt

Många moderna 3D-renderingsverktyg använder sig av någon form av *environment mapping.* Environment mapping är ett sätt att visualisera omgivningen i en 3D-scen, detta dels för att skapa en panoramabild som ger intrycket av en naturlig miljö runt om scenen, men även för att hjälpa till att rendera naturliga ljusfenomen såsom reflektion. Trots att environment mapping inte är någon ny teknik, och trots att de nuvarande tekniker som används för detta är väldigt optimerade, så medför de sig ändå en del prestandakostnader, framförallt för användare med sämre hårdvara.

Syftet med denna avhandling är att undersöka hur en metod för environment mapping, som för tillfället används av företaget Configura, kan optimeras. Poängen med denna optimering är främst till för att underlätta användningen av Configuras programvara hos användare med lägre hårdvaruprestanda, men även att tillåta användare med bättre hårdvara att uppnå bättre resultat utan märkbart sämre prestanda. Den här avhandlingen kommer att presentera två olika tillvägagångssätt, som båda är något löst baserade på en teknik som kallas *occlusion culling.* Dessa tillvägagångssätt utvärderas sedan på mått av renderingshastighet, GPU-minnesanvändning och nätverkspaketsstorlek, och ställs i kontrast mot orginallösningens prestanda.

Trots att båda tillvägagångssätten har potential och i vissa fall överträffar orginallösningen, så saknar de mångsidigheten hos orginallösningen då de lider av egenheter som gör dem svåra att tillämpa i alla situationer. Det första tillvägagångssättet presterar bra på alla prestandamätningar, men lider av ett problem som gör att den använda i många verkliga situationer. Trots att det andra tillvägagångssättet inte lider av detta problem så var dess prestanda inte på samma nivå, vilket gör att den kanske inte är ett självklart val för de flesta användare. Med det sagt så visade den en del intressanta egenskaper, som gör att den kan vara ett bra alternativ för vissa användare. Den behöver dock fortsatt arbete gällande renderingstid och minnesanvändning för att kunna tilltala den allmänna användaren.

Nyckelord: environment mapping, occlusion culling, grafik

Table of Contents

1. Introduction

When working with visual rendering software that strives to achieve photorealistic imagery, the detail of the surrounding environment is an important component. Environment mapping is a common method used for this purpose. Moreover, environment mapping can be used to effectively calculate natural visual phenomena such as light reflection and refraction. Most importantly, modern environment mapping techniques hold up very well in terms of performance. In fact, most modern GPUs can do environment mapping on the hardware level [1]. With that said, performance is still not a trivial issue, and although many of the mapping techniques are highly optimized, there are still performance costs associated with them.

Configura is a company developing a software product for interior rendering and design. They work with a wide range of users, each having varying degrees of computing power at their disposal. In order to allow for users on the lower end of the performance spectrum to use their product, further performance enhancement may be necessary. And to the users with decent hardware, quality could be further improved with little effect on performance. This thesis draws inspiration from previous work on occlusion culling and applies this idea to an environment map. Generally, occlusion culling is used more broadly to exclude geometry of little or no interest within a 3D environment. However, when used with an environment map, these areas can not simply be disregarded since they play an important part in the calculation of light phenomena. Thus, we intend to draw inspiration from the general idea of occlusion culling, but modify it to work together with the specific requirements of an environment map.

Since complete occlusion is not viable in the context of an environment map, areas not within direct view can instead use a lower resolution texture, while areas of high interest can use a higher resolution texture. We hypothesize that this can serve as an optimal balancing act, where on one hand, higher quality is achieved due to the high-resolution texture at the point of interest, and on the other, performance is not heavily affected since the areas of less interest are lowered in resolution. Thus, our aim is to investigate whether we can use the GPU memory in a more optimal way, by essentially prioritizing certain areas of the environment map. This would mean, in other words, that the GPU memory that was previously uniformly used to render a medium to high resolution environment map will now instead be used to render a high-resolution cut-out, as well as a low resolution environment map. Thus, the idea largely comes down to better distribution of hardware resources. Within this thesis, we will detail two approaches centered around this idea. They will each be compared to the non-occluded environment map currently in use by Configura. To better understand how these solutions compare to each other, they will be evaluated in terms of how they affect rendering time and GPU memory utilization, as well as network packet size when doing rendering on a separate server.

2. Aim

When increasing the resolution of the texture used to form a cube map from 4096 x 2048 px (4K) to 8192 x 4096 px (8K), some performance drops are expected. We intend to investigate how much this resolution increase affects performance. Furthermore, we will present two alternative approaches that both use a form of occlusion culling to only increase resolution in certain areas of high interest. We intend to see if these approaches serve as a viable or perhaps better solution than simply increasing resolution overall. These approaches will be tested in terms of rendering time and GPU memory consumption, as well as network packet size, in cases where rendering is done on a separate server. They will be tested against the performance of the original solution with a higher resolution texture. Thus, by first investigating the performance effects when simply increasing the resolution of the texture used for the environment map of the solution currently in use, we will establish a baseline which can later be used to evaluate the performance of the two approaches. As mentioned, evaluation will be done mainly in terms of the objective metrics described above. However, certain subjective measures will also be used to evaluate the quality of the result.

3. Research Questions

RQ1: How can occlusion culling be utilized to increase the resolution of the background image?

RQ2: By increasing the resolution of the background image, what are the effects on GPU memory utilization and data transmission amount?

RQ3: By increasing the resolution of the background image, how is the rendering speed affected?

4. Background

The background section describes the software or the application programming interfaces (APIs) used to conduct this thesis.

4.1 CET and Photolab

CET is a software tool used for space planning and design in either 2D or 3D which was developed by Configura. It has the built-in tool Photo lab which can produce photorealistic rendered images with natural light and shading. The rendering either occurs locally or is sent to a server via TCP. The rendering process is exactly the same whether it is done locally or on a server. The servers are provided by Configura, and any of their computers can be used as a server if the appropriate extensions are installed. This thesis will frequently refer to this rendering server. However, this does not necessarily mean that the rendering is done on a separate machine. In fact, unless otherwise specified, the render job is sent to a rendering server on the local machine.

CET can also be used as a programming environment to implement features and extensions for CET. It comes with its own programming language CM which streamlines the implementation of parametric, graphical and configurable objects in CET.

4.2 Redway3D

CET uses the 3D-engine Redway3D which comes with the C++ api called REDsdk for its rendering purposes. REDsdk provides two methods of rendering: software-rendering which can leverage the CPU for rendering and hardware rendering which uses the GPU.

Redway3D is a hybrid engine, supporting both software and hardware rendering. On the software side, ray tracing, a method that is described in more detail in 5.2, can be performed on the CPU, allowing for photorealistic light and shadow mapping [2]. On the hardware level, the GPU may be utilized to use more traditional graphical rendering techniques such as depth mapping. Although Redway3D may be configured to run only using one of these modes, it may also use a hybrid of the two, where different views can be rendered simultaneously using either hardware or software methods. By using the hybrid mode, additional capabilities may also be utilized, such as tone-mapping, which can be used to achieve a higher dynamic range of colors within the image. This is also the mode which CET uses to render the photo-realistic snapshots of the environment.

4.3 Magick++

Magick++ is an open-source C++ API for image-processing which will be used to manipulate panorama images. A detailed description of how this API is used is described in 7.1. The Magick++ API implementation is integrated with the standard template library (STL) making it possible to use with containers such as vector, map, list [3]. Additionally, the Magick++ api is also available in CET since the CM language has wrapper functions for the Magick++ API.

4.4 Computer Specifications

All evaluations are done on PC with Windows 10 operating system. The CPU is Intel i9-9900K with the ASUS motherboard ROG STRIX Z390-F GAMING. The graphics card is NVIDIA GeForce GTX 1660 SUPER.

5. Theory

This section describes relevant concepts which are important to be familiar with in order to grasp the content of this thesis.

5.1 Environment Mapping

Today there are many methods of doing environment mapping. While they all have some specifics and some are more widely used than others, they all serve two main purposes [4]. Firstly, environment maps act as a rendered environment surrounding a 3D scene. In video games, this is commonly referred to as a skybox. This mainly serves to give the user a sense of being immersed in a natural environment. This can, for example, give the illusion of a far away horizon or a distant mountain vista. The other main purpose served by an environment map is so-called reflection mapping. Briefly, pixel values may be sampled from parts of the environment map and can be used to calculate reflections and other light phenomena. Reflection mapping is described in more detail in 5.1.2. As mentioned, there are today a myriad of ways of doing environment mapping. However, the method most commonly used today is called cube mapping.

5.1.1 Cube Mapping

Cube mapping, as the name implies, involves the mapping of a 2D texture onto a 3D cube. In essence, the camera is located at a fixed location at the center of the cube. It is thus positioned with an equal distance to each cube face. An environment texture is then mapped onto each of these six cube faces. Although there are several ways of doing this type of mapping, the most straightforward method involves the use of a texture laid out in the form of a cross. We could also imagine the texture as a cube unfolded to a flat surface. Cube mapping has the advantage of not suffering from many of the distortions present in earlier environment mapping techniques, such as sphere mapping [4]. For example, if correctly applied, the seams of the source texture will not be visible. Furthermore, another major advantage of cube mapping is that it is generally supported by most modern GPUs [1].

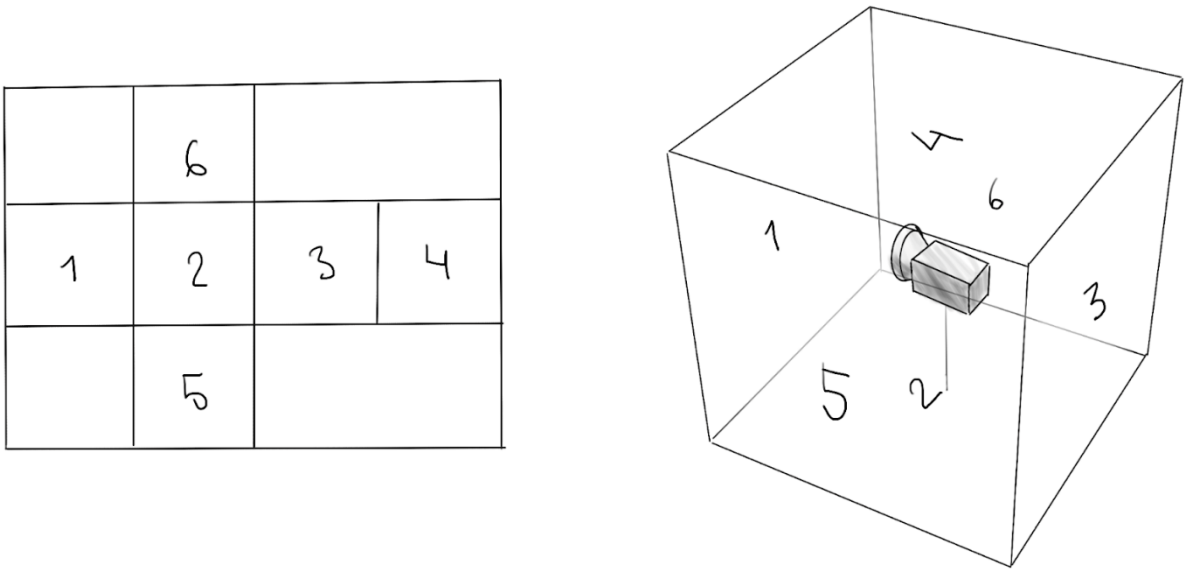


Figure 1. Typical layout of a cube map texture.

5.1.2 Reflection Mapping

Reflection mapping is a method of simulating the reflection of light within a 3D scene. By determining the vector with which the viewer views an object, the reflected color can be determined by calculating the reflection vector according to physical laws. As mentioned, a certain pixel coordinate on the cube map's surface can be sampled using a vector. By using the relationship between the light's incidence angle and the angle of reflection, an appropriate vector can be found that points in the direction of the reflected pixel. This allows for the environment to be accurately reflected on chromed surfaces within the scene. Moreover, this principle also applies to other phenomena such as refraction, where the appropriate pixel color can also be calculated in a similar fashion.

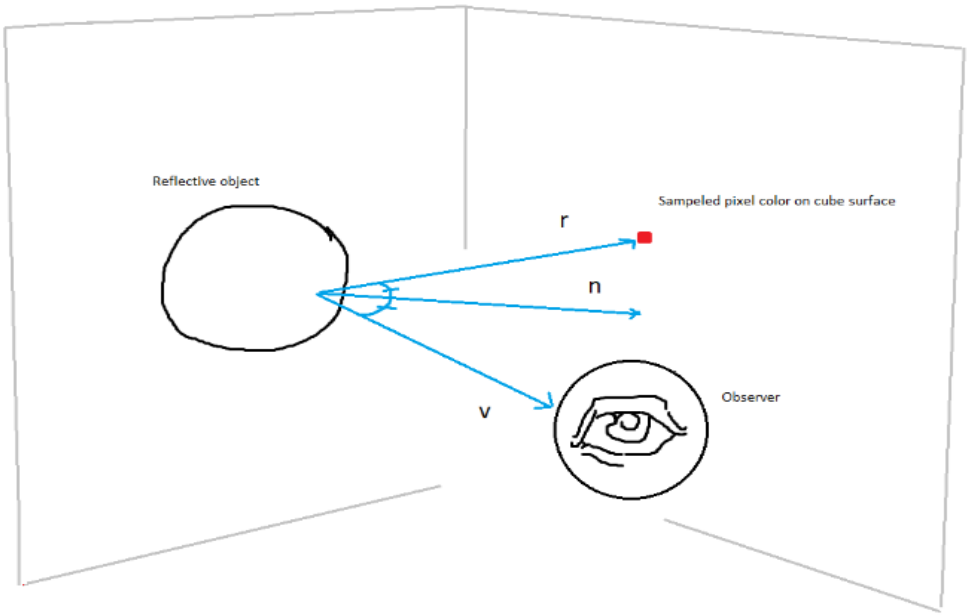


Figure 2. The law of reflection can be used with a cube map in order to model realistic reflections. V is the vector with which the light ray hits the camera or “observer”, r is the vector of the reflected light ray, and n is the surface normal of the reflective object.

5.1.3 Environment Types in Redway3D

Although the cross-shaped texture described in 5.1.1 may be the most common way of representing a cube map texture, the textures provided by CET use a different format. These images are generally high-resolution wide-angle photography of outdoor scenery. As such, the cube map must be constructed using somewhat different methods. Due to the wide-angle perspective of these images, they cannot simply be divided into six equal sized squares and pasted onto the faces of a cube. Instead, a projection method must be used. Projection is a large field in and of itself, and it contains a variety of different methods. The details of how projection works is, however, largely outside the scope of this thesis. The interested reader is instead referred to the related work on the subject, as referenced in 6.1. REDsdk provides support for three types of projection methods, in the documentation referred to as *Environment Types* [5]. The types of projection methods supported by REDsdk are: spherical, hemispherical and cylindrical. While we will not go into the details of their workings, it is

important to know that they all present a problem very much related to our work. This problem is common to all of these projection methods, but to illustrate, we will look at the case of cylindrical projection.

When doing cylindrical projection, the source texture is first projected onto a cylinder. This now cylindrical texture is then projected onto the faces of the cube. In REDsdk, this entire process is handled by the *createEnvironmentMap* method [6]. When the cylinder is formed out of a flat texture, the body of the cylinder presents little issue. Essentially, forming the body from a flat image can be likened to wrapping a paper to form a cylinder. However, the top and bottom lids present more of a problem.

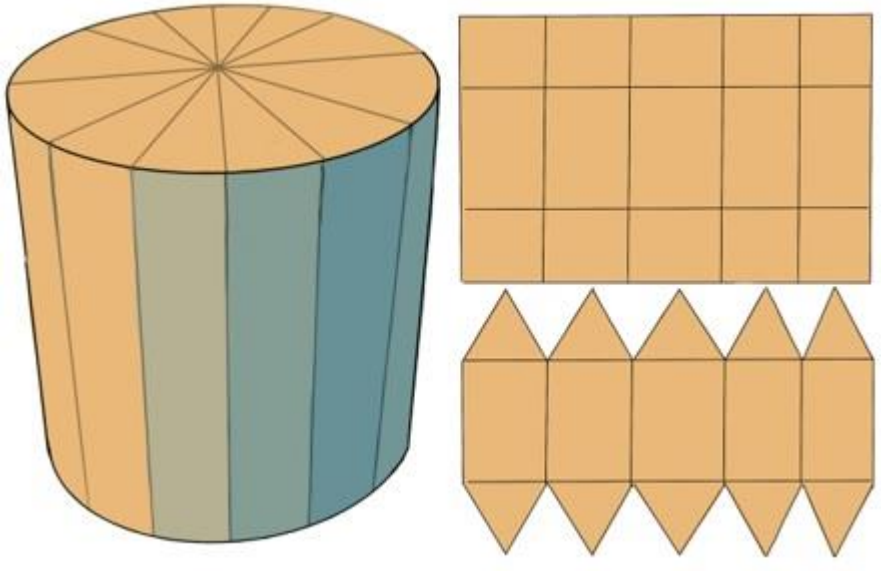


Figure 3. When folding a flat image into a cylinder, the top and bottom parts of the texture will be warped to fit these sector shapes.

As can be seen in figure 3, the top and bottom parts of the cylinder are constructed from a series of sector-shaped cutouts. These bits have to be constructed from the upper and lower parts of the source image (as seen in the image above on the right). The source image is stored locally on disk, and is in a standard square picture format. In the case of CET, these images are outdoor scenery photography images saved in a *jpg* format. This means that parts of the image that were originally in a square format now have to form a triangle. Furthermore, and perhaps most relevant to our work, the size of these bits will decrease towards the tip of the triangle. When these bits are then used to form the cylinder, the bottom edge of the triangle will be curved to form the sector shapes that make up the cylinder lid, as can be seen on the left in figure 3. When forming an environment map from this cylinder, this issue will not present any obvious aesthetic defects since the entirety of the lid will be projected onto the top of the cube. However, scale inaccuracies are still present, and may present issues in other fields where scale integrity is important, such as map projection. To see how these issues can be mitigated, we again refer to the work described in 6.1.

5.2 Ray Tracing

Although ray tracing is a method that can be used to simulate a variety of different phenomena, it is primarily used to model the behavior of light rays within a 3D environment [4]. The general idea behind ray tracing is, as the name implies, to trace rays of light in order to determine what objects in an environment are within the reach of each ray, and which are not. Furthermore, ray tracing allows for the modelling of indirect light, making it possible to accurately simulate optical phenomena such as reflection, refraction, and ambient occlusion. Generally, the rays are emitted from the origin of the camera, as opposed to the light source itself. Since only a fraction of the rays emitted from the sun, for example, would ever hit a certain object and be reflected towards the camera, most rays are of no interest and may be ignored. Thus, we are only interested in modelling the rays that will in some way intersect the camera. For this reason, taking the camera as the origin makes more sense. Essentially, each ray of light may be seen as a vector originating at the camera's position and going forwards into the scene [7]. To figure out if a certain ray intersects a certain object within the scene, we can simply see if there is an arbitrary number t for which the vector can be extended with in order to intersect the object.

While ray tracing is a powerful tool for simulating the distribution of light within a scene, we may at times wish to increase the realism of the rendered image by simulating various light effects. For example, in reality light rays will scatter in various directions when hitting an object due to surface irregularities. Within a 3D environment, however, these effects need to be simulated. While adding actual irregularities to the surface of geometry within the scene may seem like an obvious solution, this is in most cases computationally much too heavy. Instead, we can approximate these irregularities with a random distribution. Redway3D specifically does this using a Monte-Carlo sampling method [8]. In essence, this means that each point on the surface of the intersected object will be sampled a number of times, each time allowing light to be reflected in a random direction according to the Monte-Carlo method [4].

5.3 Occlusion Culling and Level-of-Detail

Occlusion culling is the technique of selectively removing objects to render depending on if they are within the field of view or not [4]. Objects can for example be behind other objects or outside the field of view, in both cases there is no need to render the objects since they will not be seen after the rendering process. By culling these objects, less objects are essentially sent to the renderer, thus, the computational cost of the rendering is reduced. This is because when rendering objects, GPU and CPU resources are occupied to draw these objects. When occluding objects, the workload of the rendering process is essentially lessened.

REDsdk provides built-in functionality for both occlusion culling and level-of-detail rendering [9]. However, this functionality does not include the cube map since all information from the cube map is necessary to calculate the reflection of the cube map from all angles. Thus, when rendering a photorealistic image information outside the field of view is used during the rendering process.

Another approach to increase the efficiency of the rendering process is to use level-of-detail techniques [10]. This technique is especially favorable when objects have varying degrees of visibility. Because of the distance between the object and the camera, it will be hard to get a clear view of the object, hence there is no reason to render such an object with as much detail

as an object close to the camera. For example, if you have two objects with the same amount of detail, one object is placed closer to the camera and one object is placed further away from the camera. The object further away from the camera will be much smaller because of the distance from the camera, thus making it difficult to see all the same details as with the object close to the camera. Reducing the detail of the object further away will often be unnoticed on the appearance of the object due to the distance from the camera [4]. Thus, to increase the efficiency of the rendering process, a level-of-detail technique can be used so that objects which are viewed from a far distance, have a lesser amount of polygons to be rendered compared to objects close to the camera.

In many cases occlusion culling and level-of-detail techniques are used in real-time environments which are heavily dependent on computational cost in order to maintain an appropriate frame rate [11], [12], [13]. In this paper however, real-time rendering is not the focus, but the alleviation on the GPU by lowering the detail in unnecessary regions is still an efficient method to reduce rendering speed, GPU memory utilization, and data transfer amount

6. Related Works

This section describes work which in different ways are relevant to this thesis.

6.1 Projection and Angular Distortion

When doing environment mapping, some form of projection is generally involved [4]. The specific projection technique used varies widely depending on the type of environment map and its implementation, but common to all of these methods are some issues related to projection in general. The projection techniques used in environment mapping share many commonalities with general map projection, which has been a field of study for centuries if not longer. Lamberts investigates how various mathematical projection models can be used to mitigate some of these issues [1]. Specifically, Lambers uses angular distortion as a measure of how true the projection is to its source. To better understand angular distortion, we may look at the common example of trying to map a spherical object (such as the globe) onto a plane (such as a map). A problem that frequently occurs in such a case is the distortion of scale experienced near the poles, causing these areas to be rendered larger. When projecting onto a cube map, this issue is most evident at angles near the corners of the cube. As can be seen in figure 3, using equally distanced angles will produce equally distantly spaced sections on the surface of the sphere. However, since the distance from the sphere and cube surfaces varies depending on the angle, the distance of the projection on the cube surface will not always be equal. The circle projected near the corner of the cube (B) will appear larger than the one near the middle of the cube face (A). Lambers specifically looks at various projection techniques in the context of environment mapping, where computational complexity is also an issue. Thus, Lambers show how some of the methods have a high quality and stay true to the source image in terms of likeness and scale, but are computationally too heavy to be viable in all contexts. Other methods, on the other hand, are computationally cheap, but suffer heavily from artifacts caused by, for example, angular distortion. An example of such a method is the standard cube map commonly used today. Lamberts then evaluates the various methods based on these metrics - computational complexity and quality. Lambers finds that some of the methods hold up very well in both metrics.

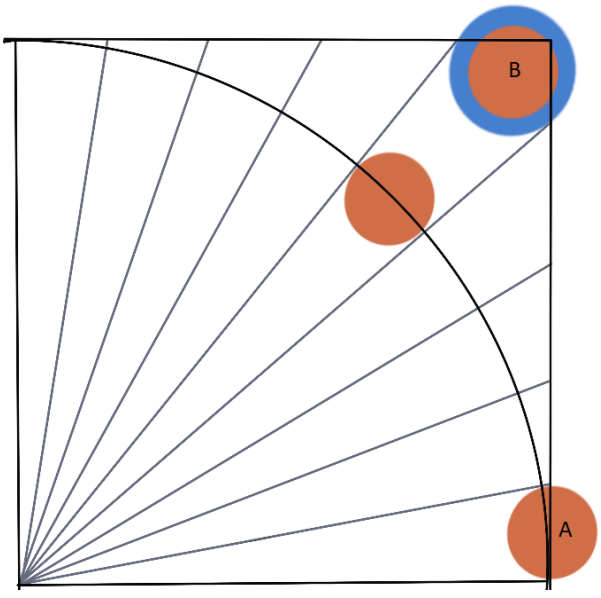


Figure 4. A 2D representation of a quadrant of a cube map. The lines represent 10 degree angle increments. As visible by the red and blue circles, scales appear larger near the corners of the cube. The red circles marked A and B have the same area. However, when projected onto the cube surface, the area of B will appear larger (as indicated by the blue circle) due to angular distortion.

6.2 Occlusion Culling and Rendering Performance

Bartz, D. et al. [13] uses occlusion culling with OpenGL to decrease the computational cost of rendering large polygonal datasets. By using parameters provided by openGL to calculate the viewpoint they managed to increase the rendering performance on polygonal models. The authors represent a scene with a moderately accurate space partitioning tree which is a hierarchical representation of the scene. A space partitioning tree is a data structure which divides the space of a scene into non overlapping subdivisions. By calculating what subdivisions of the tree are outside of the field of view, these regions are culled to increase performance.

El-Sana, J. et al. [11] integrate occlusion culling with level-of-detail rendering into their real-time rendering framework in order to lessen the computational cost of the rendering process. Level-of-detail rendering allows for changing the detail of objects to be rendered, which can be used to reduce the amount of polygons to render in objects that are, for example, far away. Since calculating precise visibility is expensive, an estimation of the visibility is performed instead. This visibility parameter is an approximation of occluded regions which is integrated into the level-of-detail algorithm which then calculates the detail that should occur in a specific region. Similarly, to our approaches, regions outside of the field of view are not entirely occluded, instead the level of detail in the occluded regions are diminished. This is because the rays from the areas that are not visible may still be reflected in other objects.

7. Method

In order to increase the quality of a rendered image while still maintaining high performance, we have two similar approaches. In both approaches increased quality will be achieved by using a higher resolution image. However, simply using a higher resolution image would result in performance loss. To circumvent this, an occlusion culling approach can be used [4]. Occlusion culling is the principle of not rendering objects that are outside the field of view, thus lowering the rendering time, increasing performance and lowering the amount of data processed. During the rendering process, different calculations are done such as reflection mapping (which is an essential part of environment mapping), thus, it is important to not fully occlude the non-visible parts of the cube map when rendering. Instead the resolution and pixel data of the cube map is lowered to reduce rendering speed, GPU memory utilization, and data amount.

In this section the two approaches will be described, followed by how they will be measured and compared to the original implementation that Configura currently uses. The first approach uses a high-resolution static image which is placed in front of the camera, using a function provided by REDsdk. The rendered image will thus only show the high-resolution image while still using the cube map for other purposes such as reflection. Similarly to the first approach, the second approach utilizes the same high resolution image but instead of using the provided function from REDsdk, it is pasted directly onto the cube map.

As mentioned, common to both methods is the use of a cropped image in high resolution. A cropped image basically means that the part of the source texture that is being viewed in the cube map is cut out to create a new image with high resolution. Typically when facing the cube map at an angle of zero degrees (when facing along the horizon line), the size of this crop will equal the resolution entered in CET. For our tests, we used a resolution of 1920 x 1080 px. What differs between the approaches is the application of this high-resolution image. To clarify, this image is a crop from the source texture, containing the area of the texture within direct view of the camera. To further reiterate, approach 1 will use this image together with the same cube map as used in the original solution, the difference being only that the cube map in this case will be of a lower resolution. Thus, the general thought behind this process can be summed up as a redistribution of resources. For example, by sending a low-resolution cube map and a high-resolution image to the GPU, instead of only a high-resolution cube map, more resources can be focused on the part that is truly important (the area in direct focus of the camera). This principle further applies to the case where rendering is done on a separate rendering server. In this case however, the idea is that packet size could be lowered since the total size of a high-resolution image plus a low-resolution cube map would be lower than only a high-resolution cube map. Approach 2, as mentioned, builds on the same principle. The main difference is that while approach 1 renders the low-resolution cube map and the high-resolution image using a method provided by the REDsdk, approach 2 does so by reconstructing the cube map itself.

Rendering time, GPU memory utilization and network packet size are metrics that will be used to determine the effectiveness of each of the approaches. This will be compared against the performance of the original solution with a higher resolution texture. In other words, we intend to investigate how the original solution (simply using a cube map) performs when increasing texture resolution. Thus, we will measure how time, memory and packet sizes vary between the original with a 4K resolution and with an 8K resolution. This will be put into

perspective by comparing how these metrics are affected when using the 8K resolution with each of the two approaches.

7.1 Approach 1

In the first approach, a cube map will be used in conjunction with a static image. The high-resolution static image will be used for areas within direct visibility, while the cube map will be used for lighting calculations. Interestingly, Redway3D has a method that does just this. The method *setBackgroundImages* allows the user to add both a cube map and a static image [14]. Redway3D's ray tracing algorithm then uses the static image to sample all direct light, while all indirect light will sample the cube map.

In order to determine what part of the cube map texture to use for the static image, the area in view of the camera first needs to be determined. It is possible to calculate the pixel coordinates of the area on the cube map within view of the camera. Simply put, this can be done by first calculating the yaw and pitch of the camera and then using the relation between texture pixels and radians to determine which pixels of the texture are within view.

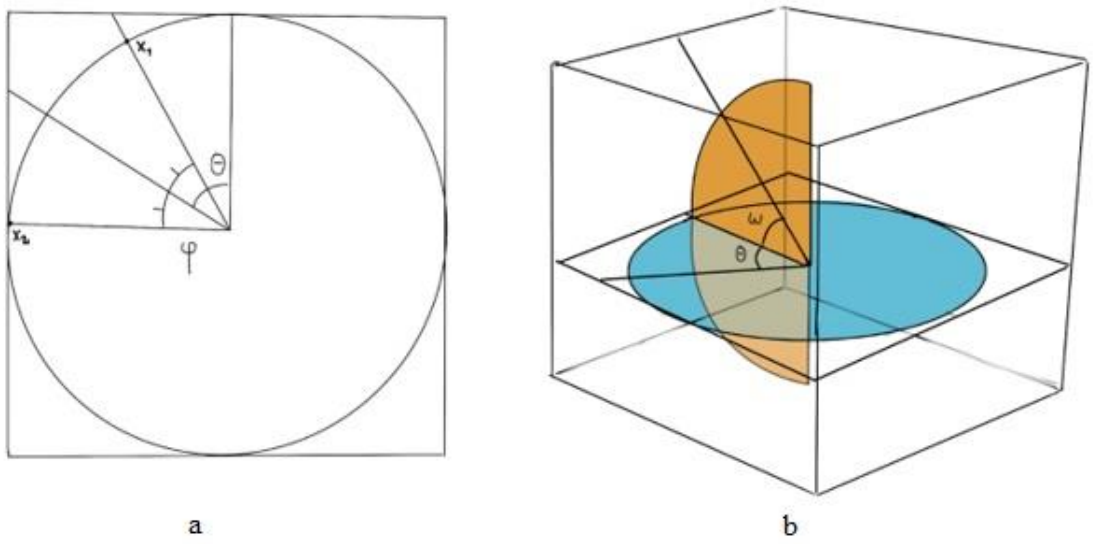


Figure 5. a shows the cube map as seen from above. The left - and rightmost pixels are marked as $x_1$and x_2. Figure 5.b, shows the cube map in perspective. Here we can see a unit circle (blue) superimposed on the xy-plane. Together with the half circle in orange, they make out the total range of motion of the camera. The angle Θ shows the yaw of the camera, while shows the pitch.

In figure 5.a, we can see the cube map from above. The cube is tangent to a unit sphere. Θ signifies the yaw of the camera (rotation around the xy-plane), while φ signifies the camera's field of view. The texture will be projected upon the sides of the cube tangent to the sphere. By knowing the texture width (*w*), pixels per radian (*ppr*) can be calculated as follows:

$$ppr_w = w / 2$$

The bounding pixels of the texture within view (marked by x_1 and x_2 in the image) can be determined by the following, where *a* is the aspect ratio:

$$x_1 = (\Theta - \underline{\varphi} / 2) \times ppr_w \times a \ , \qquad x_2 = (\Theta + \underline{\varphi} / 2) \times ppr_w \times a$$

In a similar fashion, the upper and lower bounding pixels can be determined with the same logic. However, as we can see in figure 2.b, pixels per radian for the height will be determined by:

$$ppr_h = h / \pi$$

Thus, the bounding pixels for the height can be calculated as follows:

$$y_1 = (\omega - \underline{\varphi} / 2) \ ppr_h \times 1 / a \ , y_2 = (\omega + \underline{\varphi} / 2) \times ppr_h \times 1 / a$$

Once the correct pixel coordinates are known, they can be used with the Magick++ API to crop the desired image. There is, however, an edge case to be mindful of. When pasting a flat texture onto a cube, the two ends of the texture (i.e. x-coordinate 0 and x-coordinate 4096 in the case of a 4K texture) will have to meet at some point on the surface of the cube. This point will have a seam. This is similar to a case where one would wrap a sheet of paper to form a cylinder. The seam in this case would be where the two ends of the paper meet. When facing this seam on the cube map, two images would need to be cropped from the source image. The width of the first image would be that of the width of the texture minus the leftmost pixel coordinate, x_1. The other image would span from zero to x_2. The Magick++ API can then be used to combine these images into one. Finally, the red3D method *setBackgroundImages* can be used to combine the cropped image and the low-resolution cube map.

7.1.1 Limitations when Ray Tracing

With approach 1 it will not be possible to get a correctly cropped image when viewing the top or the bottom of the cube map. Hence, this approach will only be a viable solution when viewing the different sides of the cube map. This limitation arises due to the way in which the cube map is constructed in CET. Essentially, as described in more detail in 5.1.3, this process involves a cylindrical projection, where the top and bottom parts of the cylinder are formed from sector shaped cut-outs sampled from the top and bottom sections of the source texture (see figure 3 for a more detailed image of how this is done). If, as in approach 1, a static image would be used to represent this area, this image would have to be constructed in a similar fashion. This process is complicated and is outside the limitations of our work. Thus, as it stands currently, approach 1 is limited to viewing the cube map along the horizon line.

7.2 Approach 2

In the second approach, only a cube map will be used. By determining what area of the cube map is within view of the camera, much in the same vein as in approach 1, the correct image can be cropped from the texture source image. In order to minimize network traffic load when sending data to the renderer, a low-resolution thumbnail is instead passed along with the cropped image. The thumbnail can then be scaled up on the render server and the cropped image pasted at the correct location.

7.2.1 Scaling the Image

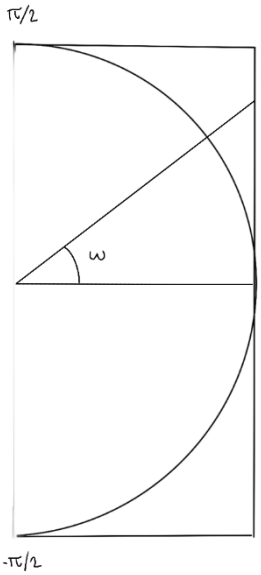
To scale the image, methods provided by the REDsdk API were used. By first allocating the appropriate amount of space to store the pixels for the scaled up image, these pixels could

then be accessed and written to. Memory could be allocated on the GPU with red3D's *setPixels* method [15]. Once enough memory had been allocated, the pixels in the source image (the thumbnail) could be used together with REDsdk's *resize* function [16] to resize the thumbnail and write the result to the allocated memory on the GPU.

7.2.2 Pasting the Image

The pasting of the image at the correct location on the scaled up image occurs on the rendering side. This rendering can be done either on the local machine or on a separate server, as described in 4.1. The pixels of the scaled up image can be accessed and written to with the *getPixels* method [8]. Each pixel of the cropped image is written to the correct pixel of the scaled image. The size of the cropped image and the location at which it will be pasted is further affected by its latitude. This is due to the projection method used, as described in 5.1.3. Essentially, the further up or down we look, the more we will sample of the sector shaped pieces described earlier. When going towards the tip of these sector shapes, we will experience size distortions. In order to mitigate these distortions, we use a modifier that samples more of the texture the further up or down we face. Two alternative modifiers will be presented and tested, one depending on the cos value of the camera´s yaw, and one depending on the average change in camera yaw. Their effectiveness will be evaluated based on the size of the area they crop. Essentially, we want to crop as small an area as possible while still covering the entire field of view of the camera. The following alternatives for modifiers will be tested:

- Cos(ω): By setting the modifier as cos(yaw) of the camera, the result would be a modifier that is near zero along the xy-plane and will either increase or decrease towards ± 1 near the poles. A modifier of 1 would mean the *1 * texture_width* should be used, while near the horizon this modifier would be close to *0 * texture_width.* The resulting modifier will then be clamped to a minimum of the size of the original crop, and a maximum of the size of the texture.
- The average yaw modifier: It will instead be based on the average change in yaw from the horizon line. By dividing this value with the angle of a quarter circle (as seen in figure 6), we can get a value ranging from 0 near the horizon line, to 1 near the poles. Since the texture size must always be a positive value, and may assume values between $\pm\ \pi / 2$, the modifier is assigned an absolute value. Again, the modifier will be clamped between the original crop size and the total texture size.



$$m = \Delta\omega \,/\, |\, \pi \,/\, 2 \,|$$

Figure 6. This figure shows how the change in angle in proportion to the angle of a quarter circle was used for the modifier m.

7.2.3 Building the Cube Map

Finally, as the desired image had been cropped, it was sent along with the low resolution thumbnail to the render server. Once on the render server, the thumbnail was scaled, the cropped image pasted, and a new cube was constructed using the REDsdk function *CreateEnvironmentMap* [6]. As described in 5.1.3, Redway3D provides three options for environment types - cylindrical, spherical and hemispherical. Cylindrical projection is the one currently in use by Configura, and since they all suffer from the issue described in 5.1.3, we decided to keep using cylindrical projection for the sake of consistency.

7.3 Measurements

Lambers, M. [1] analyzes different cube mapping techniques for optimizing cube mapping and measures their performance based on computational cost. Hence, to measure if the proposed solution is of value, two areas that will be considered are: rendering speed and GPU-memory. However, since it is possible to do the rendering of the cube map remotely, meaning the data will have to be transmitted via the internet, an additional measurement is required. Specifically, the amount of data transferred. Furthermore, since the main purpose is to increase the resolution of the background of the environment, the image resolution will also be evaluated. When switching between the implementations for measuring, the CET Developer command “clean” will be used. This command resets the software, clearing all buffers and unsaved work. This will be done to minimize the effect previous renderings will have on upcoming ones.

Approach 1 and 2 have some varying cases - when viewing the cube map at the “seam”, as described at the end of 5.1, as well as when viewing the cube map “normally”. Hence, those two cases will have to be measured separately for each approach. However, we will not present this as an alternative for the original solution since it does not need to crop the images as described in 5.1. Additionally, an edge case will be measured for approach 2. Namely, as described in 7.2.2, when facing the cube map straight up or down, more of the texture has to be cropped and pasted onto the new image. Thus, there is good reason to believe this will negatively affect performance and rendering time, potentially serving as a worst-case for approach 2.

Below is a breakdown of the cases we will investigate.

- *Original 4K.* This is the original solution, as was used by Configura before this thesis. This method renders the entire cube map in 4K.
- *Original 8K.* Same implementation as above, apart from using a texture image of 8192x4096 pixels (8K).
- *Approach 1.* This is the solution described in 7.1. The cube map has a resolution of 4K while the static image is cropped from an 8K image.
- *Approach 1 Seam.* This is the same solution as *Approach 1,* but both of the edges of the texture are viewed. A more detailed description of what this entails is found in section 7.1.
- *Approach 2 8K.* This is the solution described in 7.2. The cube map is scaled up to 8K from a 128x64 image, while the static image is cropped from an 8K image.
- Approach 2 Seam. This is the same solution as Approach 2, but when both the edges of the image are viewed.
- Approach 2 UP. Same solution as Approach 2, but when viewing the top of the image.

7.3.1 Rendering Speed

The functionality to measure the time from when the *render* button is pressed and to when the rendered image is received is already built into Photolab. This functionality will be used to measure the rendering time of the approaches compared to the original implementation.

Firstly, the original solution will be measured when using a 4K resolution texture in order to establish a baseline. Texture resolution will then be increased to 8K, where rendering time will again be measured, first for the original solution and then for the two approaches. The increase in rendering time as compared to the baseline will then be presented for each of the solutions.

The rendering time was calculated as the average time of 20 renderings done with the same cube map texture viewed from the same position - either looking straight at a cube face, or at the seam (as described in 7.3). Additionally, the edge case described in 7.2 was tested by taking five separate time measures and calculating the average between them. This edge case will be referred to as UP, since it occurs when viewing the cube map straight up or down.

7.3.2 GPU Memory

The GPU memory consumption is also a very relevant measurement for determining appropriate solutions. This is because of the limited amount of video memory available on GPUs. The clients using CET can either render images locally, in which case the GPU memory utilization can not be too high since the hardware used is an unknown variable. Otherwise, the rendering occurs on a server, which receives multiple requests and rendering a single image can not take up too much of the GPU memory. In any case, the GPU memory utilization is an important measurement to determine appropriate solutions.

Samuel, T. K. et al. [17] leverages the built-in NVIDIA monitoring tool to measure the GPU memory utilization of NVIDIA GPUs when developing the Keeneland initial delivery system. By using the command *nvidia-smi* in the terminal window, statistics on the GPU will be outputted. This tool was used to measure the GPU memory utilization while rendering images.

As with the rendering speed measurement, the benchmark measurements from the original implementation are using both 4096x2048 and 8192x4096 images. The benchmarks are compared against the two approaches which were tested with 8192x4096 image resolution.

To measure memory utilization, five consecutive renders with each setting (resolution, if viewing at the seam or not) was done using *nvidia-smi*. The average was then calculated. We will present both *peak memory utilization* - the highest peak in memory use as compared to the baseline, as well as *total memory utilization* - the memory utilized during the entire rendering session. Finally, the edge case described in 7.2 was tested for memory utilization as well. Since the baseline value varies depending on many other factors, such as background processes, this value was first established before each render. This was done by observing the *nvidia-smi* output without interaction for a few outputs.

7.3.3 Data Transmission

As mentioned in the background section, users can choose to render images on a server. This means that the image to be rendered and all its corresponding metadata will have to be transmitted via the Internet. Since the data transmitted depends on what implementation is used, this metric is valuable to establish to determine if the two approaches developed for this paper are viable.

Saxena, P. and Sharma, K., S. [18] proposes in their article the packet sniffing tool called *Wireshark* for capturing and analyzing packets sent using protocols such as TCP, UDP, DNS etc. Hence, *Wireshark* was used in this paper to compare the data transmitted.

When CET prepares and requests a rendering job the data is always sent using the TCP protocol, the data is not prepared and sent differently if it is done locally or at a server. To get an accurate comparison between the implementations, the round-trip-time (RTT) is excluded from the comparison. This is because RTT invites unnecessary unknown variables which might affect the result, such as congestion control. The measurements were done locally and included all packets involved in the rendering process, specifically, both to and from the renderer. The original solution is measured with both a 4096x2048 image resolution and 8192x4096 image resolution. The two approaches on the other hand, are only measured with an image resolution of 8192x4096. The data transmission amount will be calculated to an average of 20 renderings for each case of all the different implementations. Furthermore, there is no need to measure approach 1 when looking at the seam of the cube map. That is because the two cropped images are combined into one image before it is sent to the rendered. However, this will still need to be measured for approach 2.

7.3.4 Image Results

Since one of the main purposes of this thesis is to increase the quality of the background image it is also important to evaluate the outcome of the rendering. However, simply increasing the image resolution from 4096x2048 to 8192x4096 will naturally increase the image quality of the background. Thus, it is unnecessary to present the rendered images for each implementation. Instead this measurement is mainly used to analyze if something went seriously wrong with the rendering and the resulting image is not what was expected. The observation measurement between images in this thesis is a subjective measurement, meaning the comparison is done with the human eye.

Furthermore, walls and windows were placed in the scene to see if any of the solutions were negatively affected by these objects when rendering. If any problems arise due to these objects, they will be presented in the results section. Additionally, any visible artifacts that occur while the camera is facing straight up, will also be presented in the results section.

8. Results

In this section all measurements for each implementation and their varying cases will be presented.

8.1 Rendering Speed

Figure 7 shows the outcomes for measuring rendering time for the original solution as well as the two approaches, when rendered 20 times with the same settings.

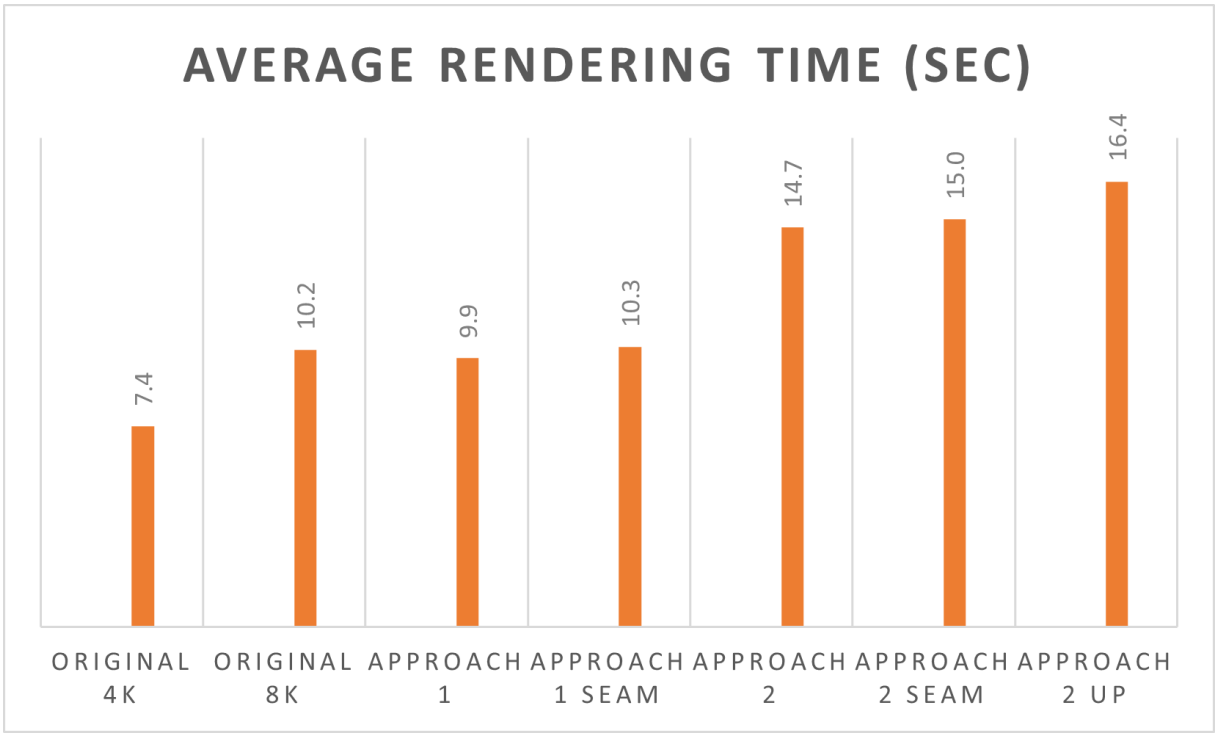


Figure 7. Average rendering time over 20 different renders with the different solutions.

As can be seen, increasing the texture resolution affects the time it takes to render a scene. Increasing the texture resolution of the original solution from 4K to 8K meant an increase in rendering time of roughly 2.75 seconds. Rendering time for approach 1 was the lowest, with an increase of roughly 2.48 seconds compared to the original 4K. When using approach 1 and viewing the texture at the seam, this increase was about 2.84 seconds. Approach 2 had the highest rendering times overall, with an increase of about 7.25 seconds when viewing the

texture at a cube face, and 7.58 seconds when viewing the texture at the seam. When facing either of the poles of the cube map (UP), the time increase was just over 9 seconds, making it the largest increase in rendering time out of all cases.

8.2 GPU Memory Utilization

Figure 8 shows average GPU memory utilization, as measured over five different renders. All renders were done with the same settings, and all using the same background image. As mentioned, there is some baseline variance in memory utilization, as can be seen by the first and last data points of each curve.

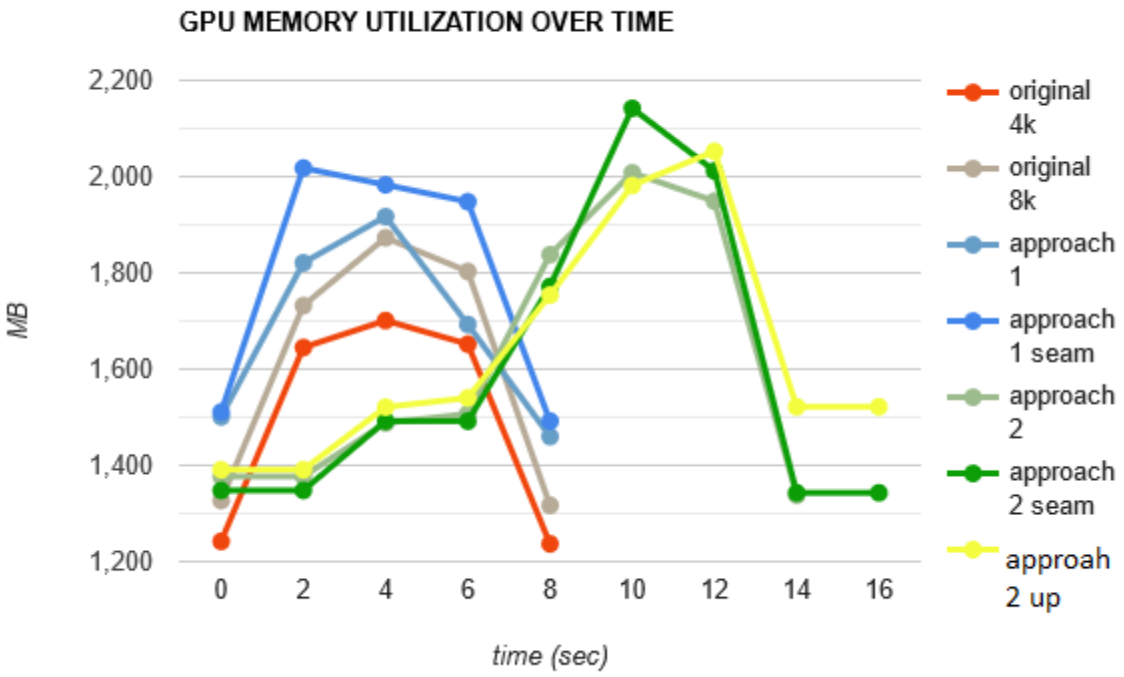


Figure 8. GPU memory utilization measured at two-second intervals over rendering duration for the different solutions.

Memory utilization usually peaks somewhere in the middle of the rendering process, and then subsides back to baseline. For approach 2, however, this peak generally occurs near the end of the process.

8.2.1 Total Average Memory Utilization

Since rendering times vary between implementations, it is interesting to examine the total average GPU memory utilization. To do this, an estimate was calculated using the accumulation of all outputs during the rendering period. Baseline values were disregarded, and only values during the actual rendering process were used. Additionally, the baseline was subtracted, yielding only the accumulation of memory allocation above the baseline level. The results can be seen in figure 9.

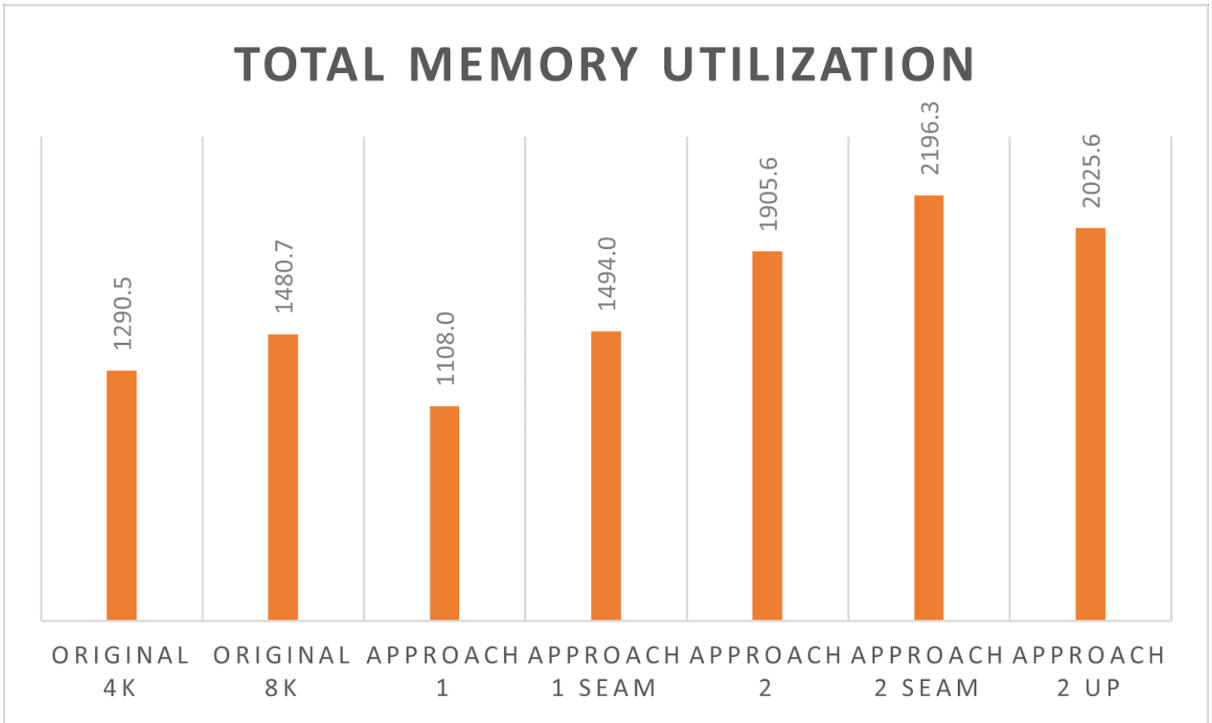


Figure 9. Total GPU memory utilization in megabytes for the different solutions. Total GPU memory utilization is calculated as the accumulated value of above-baseline memory utilization during the duration of a render.

As expected, increasing texture resolution of the cube map yields higher memory utilization, as can be seen by comparing original 4K and original 8K. This increase was about 190 MB on average. Approach 1 used the least amount of total memory out of all the solutions. In fact, it even outperformed the original 4K solution by 182 MB. When facing the texture at the seam, however, approach 1 used up more total memory than both original 4K and original 8K. Approach 2 used the most total memory out of all the solutions, with the most expensive case being when viewing the texture at the seam. In this case, it used a total of 716 MB more than the original 8K, and 702 MB more than when facing the texture seam with approach 1. Somewhat surprisingly, approach 2 up did better than approach 2 seam, and only used 120 MB more than approach 2.

8.2.2 Peak Memory Utilization

Peak memory utilization is an important measure for hardware with limited memory resources. Essentially, peak memory utilization is the maximum increase in GPU memory utilization over baseline level during a render. These values were measured as the average utilization over five consecutive renders. The peak was then calculated as the maximum value minus the baseline value. The results can be seen in figure 10.

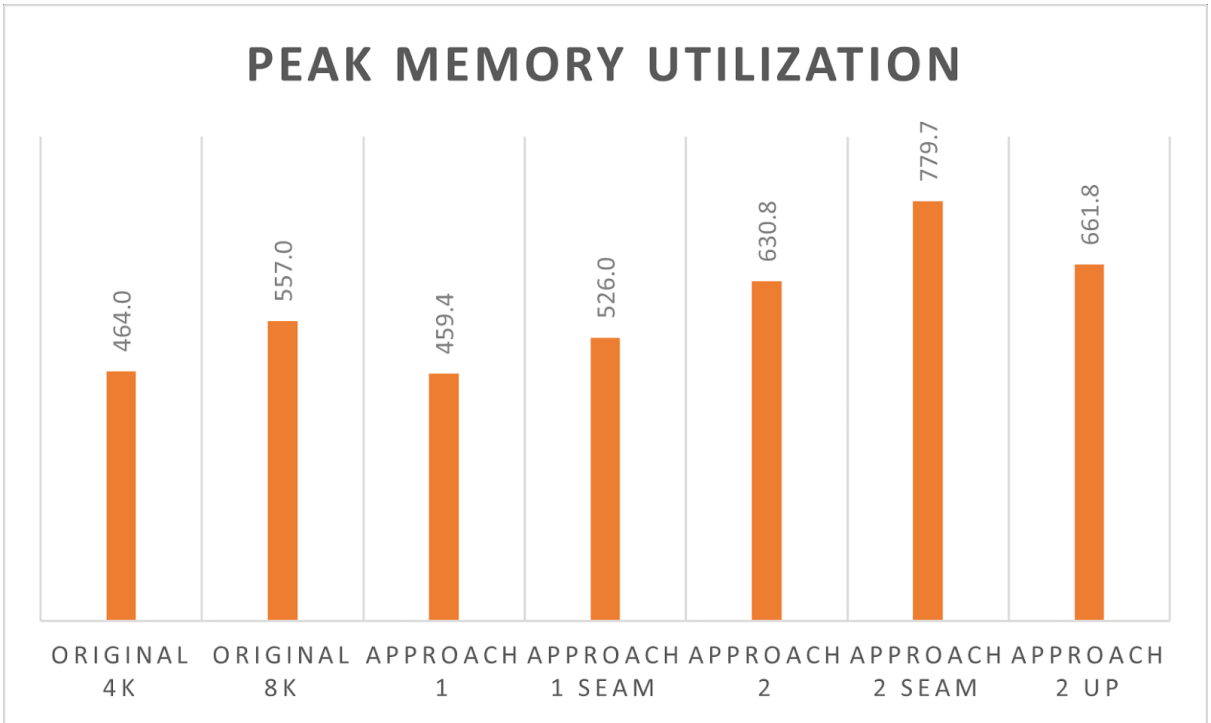


Figure 10. Peak memory utilization for the various solutions.

As expected, increasing the resolution of the original meant increased peak memory utilization. This increase in texture resolution resulted in a peak memory utilization that was 93 MB higher. Again, approach 1 had the best performance with the lowest peak utilization of all the solutions, beating the original 4K by 4.6 MB. Even when viewing the texture at the seam with approach 1, it outperformed the original 8K by 31 MB, and had a peak utilization of 62 MB above the original 4K. Approach 2 had the overall highest peak memory utilization, measured as 223 MB over the original 8K solution, and 320 MB over approach 1. When viewing the cube map at one of the poles (approach 2 up), we saw an increase of 30 MB as compared to approach 2. When viewing the texture at the seam with approach 2, this increase was 149 MB.

8.3 Data Transmission

The calculated average for each implementation and their appropriate cases can be seen in figure 10. The increase of the data amount between 4K and 8K is approximately doubled for the original solution. Approach 1 is slightly more expensive than original 4K, but it transmits approximately half of the total data compared to original 8K. Approach 2 and Approach 2 Seam are the two cases that perform best out of all scenarios with a data amount of 41.3 MB and 38.2 MB respectively, differing about 3 MB. Disregarding the original implementation and looking only at the two approaches and their cases, Approach 2 UP transmits the most amount of data.

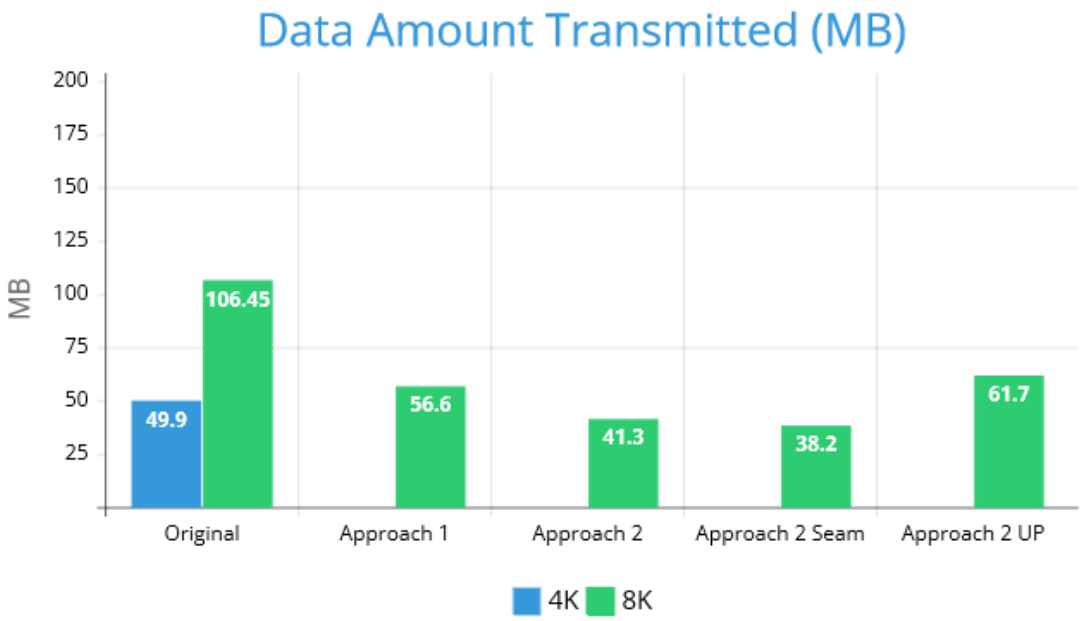


Figure 11. The average data amount transmitted both to and from the renderer for each implementation. For the original solution, 4K stands for a 4096x2048 image resolution and 8K means an image resolution of 8192x4096 was used.

8.4 Image Results

In this section limitations that occurred with increasing the image resolution from 4094x2048 to 8192x4096 will be presented.

8.4.1 Limitations when Ray Tracing with Approach 1

As stated in at the beginning of section 7.1, as well as in the documentation on the *setBackgroundImages* method, all indirect light will sample the cube map for its pixel values [14]. This proved to be a major limitation of approach 1 when using it to design interior environments. Since glass windows in CET mimic the light refracting and reflecting characteristics of a real glass window, the rays traveling through the window will sample the cube map, and not the static image. This means that when facing the window and rendering, the low-quality cube map will instead be sampled for the area outside the window, diminishing the quality improvements intended with this method.

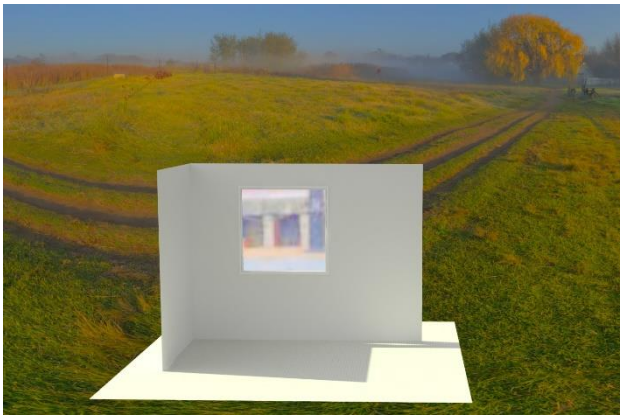


Figure 12. A screenshot from CET, illustrating the issue with rendering through a glass window with approach 1. As can be seen, the image sampled through the window is not the same as the actual background. In this case, a separate texture was used for the background and the cube map in order to illustrate the problem.

8.4.2 Yaw Modifiers

As described in 7.2.2, two alternatives for the yaw modifiers were proposed. When not using any modifier and facing the cube map near one of the poles, we would get a result where the sector shape described in 5.1.3 was clearly visible, as in figure 13.

Figure 13. A rendered image from CET, where we can see the top shapes that make up the cylinder lid.

The two proposed methods were cos(ω) and the average yaw. To evaluate, we let each modifier crop sections of the texture as the camera pitch increased.

Figure 14. Cos(ω) on the left and average yaw on the right. As visible by the red outlines, cos(ω) tended to yield larger crops, which would result in lower performance.

As visible by the red outlines in figure 14, the area of the cropped image varies between the modifiers. In figure 14 on the left, we can see how cos(ω) tended to crop large areas. For example, when facing the cube map at an angle of 75 degrees (just below the top edge of the texture, as marked by the top red outline), cos(ω) cropped roughly 96% of the width of the texture. When facing the same angle with the average pitch modifier, this crop would be only 79% of the width of the texture. Thus the average yaw modifier managed to crop smaller areas while still completely covering the camera's field of view.

9. Discussion

In this section the method and the results will be discussed and analyzed. Firstly, possible improvements to the method which would be interesting to investigate if we had more time and resources will be discussed. Secondly, the results from the previous section are discussed and analyzed.

9.1 Method

The computer that all measurements were performed on had fairly powerful hardware since it is used for graphics software. Since an important detail is that the approaches need to be developed with the computers' performance in mind, it would have been a good idea to perform the measurements on an additional computer with lower performance. Thus, it would have been possible to see how much the results changed depending on the hardware.

Regarding approach 1, there are changes that could have been done to the implementation which could possibly affect the result of the rendered image. The problem of approach 1 was that when rays travelled through windows, refraction occurs, and rays which would otherwise be interpreted as direct rays are interpreted as indirect rays. Thus, the cube map is displayed instead of the high-resolution image. This is because of how the function *setBackgroundImages* provided by REDsdk works. Instead of using this function, a wall could be placed at an appropriate distance from the camera. The high-resolution image can then be projected upon the wall. This could possibly achieve the same effect as *setBackgroundImages* without the drawback of the function. This is something that could not be tested due to time-constraints but could hold potential for future work on the subject.

There are also design changes to approach 2 which would have been interesting to test. Currently, the resize operation is done using the REDsdk API. It would have been interesting to see how the rendering speed and GPU memory utilization would have been affected if the resize operation was done using the Magick++ API instead. This was not easy to implement due to the design of the software and because of the time constraint of the thesis.

9.2 Result

The results from the data transmission measurement were mostly in accordance with our expectations. The increase of the data transmitted for the original implementation between 4K and 8K was expected. The data transmission amount from approach 1 was also in accordance with our expectations since the implementation transmits the same data as original 4K, but with an extra image. Thus, it makes sense that it transmits slightly more data than original 4K. Furthermore, we expected approach 2 and approach 2 seam to transmit the least amount of data which it in fact did.

This is because the cube map is not transmitted as with original 4K/8K and approach 1, instead, 128x64 thumbnail image is transmitted, scaled and converted to a cube map. Additionally, it makes sense that approach 2 up has the second highest data transfer amount, since the cropped image in that scenario is larger compared to the other approaches that sends a cropped image. Slightly unexpected was that approach 2 Seam transmitted less data than approach 2, since it has to transmit two cropped images instead of just 1 which approach 2 does. However, the size of the two cropped images is the same as the single cropped image, which possibly means that the meta-data that comes from sending two images is negligible. Thus, the difference between

those two cases can be regarded as a random occurrence which is caused by the natural variation between the samples.

The results of the rendering time measurements held up fairly well to our expectations. However, there was only a minimal increase in average time when viewing the texture seam, as opposed to facing the cube face directly. Although we had expected this increase to be larger, it would be reasonable to assume that cropping two smaller images instead of one larger does not affect rendering time to a significant degree. When facing one of the cube map poles on the other hand, as seen in the test results for approach 2 up, the increase in rendering time was much larger. This is not surprising, since, when using a 8K resolution texture and facing up, the entire length of the texture will have to be cropped, as opposed to a much smaller section (which depends on CET settings, but generally ranges between 800x600 and 1920x1080 px) when facing the cube map at the horizon.

As can be seen in figure 8 in 8.2, when measuring GPU memory utilization, both the original solution and approach 1 tend to peak their memory utilization around the midpoint of the rendering process. For approach 2, however, this peak occurs much later, near the end of the process. Furthermore, the early stages of approach 2´s rendering process is marked by a very low utilization. This could be explained by the large portion of GPU-heavy operations performed on the rendering server. During the early stages of the rendering process, which always occurs on the local machine, approach 2 only handles a very low-resolution thumbnail and a small to mid size crop (size here again depends on settings in CET). When passed to the rendering server, however, much of the heavy lifting is done, such as image manipulation and construction of the cube map. Although the memory utilization for approach 2 is generally much higher than both the original and approach 1 (as seen in 8.2.1), it may still prove advantageous to users with limited hardware who choose to do their rendering on a separate server. Since this server likely has much better hardware resources, the user can be relieved of the heavy memory requirements of the late stages of the rendering process.

In 8.2.1, when looking at total memory utilization, we were somewhat surprised by the excellent performance of approach 1. Although expected to perform well, we did not imagine it to outperform the original with a 4K resolution texture. This may even at first glance seem counterintuitive. The original uses only a 4K resolution cube map, while approach 1 uses the same cube map, but with an additional cropped image. It would be reasonable to assume that this extra image would negatively affect GPU memory utilization. However, we need to keep in mind that approach 1 uses a built in REDsdk method, as described in 7.1. This method not only affects the appearance of the rendered image, but the ray tracing process itself. In fact, the rays directly hitting the static image will only be interpreted as direct rays. While it is also true that Redway3D does its ray tracing on the CPU, it does not do so exclusively when running in hybrid mode, which is the default setting for CET [2]. Hence, it is possible that using the REDsdk method in approach 1 affects the rendering process, and may positively impact memory utilization.

These results are largely mirrored in 8.2.2 - Peak Memory Utilization. While peak memory utilization may be the most important measure for users with low end hardware, it is important to keep in mind that the distribution of memory utilization looks very different between the various solutions. As previously discussed, much of approach 2´s memory utilization occurs at the end of the render, which has the potential of being performed on a separate machine. Thus, while the results presented in figure 10 hold true when running on a local machine, they should

be interpreted with the caveat that the actual timing of when peak utilization occurs differs between implementations. As a result, approach 2 may still be a viable alternative for users with very limited video memory.

When determining the outcome of the test results overall, we need to keep in mind the thesis aim as well as the priorities of our client. Previous feedback from Configura's customers has deemed rendering time to be one of the biggest factors that affect their opinion of CET. Moreover, from Configura's perspective, memory utilization is one of the biggest concerns, since CET is used by customers with very large variances in terms of computing power and graphics memory. In terms of these metrics, we can surmise that approach 2 yielded results that were generally unfavorable as compared to both approach 1 and the original solution. With that said however, we should again mention the potential for approach 2 when running on a separate server. Approach 2 not only outperformed the other solutions in terms of data packet size, but also has a unique distribution of memory utilization where much of the GPU memory utilization can happen on the rendering server. Thus, approach 2 could be a viable alternative for customers choosing to do their rendering on a separate server, which is an option provided by CET.

When examining the results of approach 1, we can see that it fared well in all of the metrics. Although it was outperformed by the original in terms of rendering speed, it did perform better on average when it came to packet size. Also, approach 1 performed best in both peak memory utilization, which may be a very important factor for customers with a very limited GPU memory, as well as total memory utilization. The biggest drawback with approach 1, however, is the effect shown in 8.4.1. This is of course a major flaw when designing interiors with windows. A future solution might look at how these two factors can be reconciled, as to either change the parameters of the glass material to allow the light to be sampled as a direct ray, or to add the static image not using the *setBackgroundImages* method, but perhaps instead adding the image as a separate object at the appropriate location in the environment. Furthermore, approach 1 may appeal to certain customers, such as those dealing more with open area or outdoor environment designs.

To sum up, the original solution still holds up very well in most metrics. Although slightly behind approach 1 in terms of memory and data packet size, it does have the advantage of being very versatile. It does not suffer from the performance drops experienced by both approach 1 and 2 when viewing the texture at the seam, and it does not experience the light refraction effect that approach 1 does. In this sense, the original solution is likely the best option for most customers. However, both approach 1 and 2 have their strengths, and may appeal to customers with more unique needs. For example, customers soley doing their rendering on a separate server may benefit from the smaller data packet sizes and peak memory distribution of approach 2, while customers with very limited hardware resources may choose to use approach 1, if their design needs allow for it.

Finally, when determining the subjective quality of the images produced by the various methods, we can conclude that they show very similar results in most cases. However, both approach 1 and approach 2 suffer from peculiar effects that the original does not. As mentioned, approach 1 suffers from the refraction property of glass windows, and approach 2 suffers in performance when viewing the cube at an angle. However, if we choose to overlook these issues, the overall image quality is improved for all three solutions. In other words, image quality can be enhanced by increasing the cube map texture resolution, and the results are more than noticeable. As a reference, this is an image taken first with the original 4K on the left, and

with approach 1 on the right. The improved resolution largely affects image quality. Furthermore, we may also keep in mind that both original 4K (left) and approach 1 (right) showed very similar performance in terms of both memory utilization and rendering speed, the two factors deemed most important to Configuras customers.

10. Conclusion

With an overview of the results as presented in the previous section, and the methods as described in depth in section 7, we can draw some conclusions about our work and use these to give answers to the questions posed in section 3. Firstly, our work explored an area rooted both in environment mapping and occlusion culling. While drawing from ideas from both fields, we also explored the limitations of this combination. More concretely, we found that occlusion culling can not be used in its complete form, where certain areas are completely disregarded from rendering, since this would interfere with the reflection mapping process of the environment map. Instead, the general idea of occlusion culling would instead be used in a somewhat different manner, lowering quality of some areas instead of completely disregarding them. This of course, meant a balancing act between the two fields. On one hand, areas could be heavily occluded, resulting in lower quality but with better performance. On the other hand, performance could be sacrificed for higher quality. In this case, as in many others, there is not one solution that would appeal perfectly to all users. Instead, it is up to each user to determine what is important for them, and what may fit their specific situation. With that said, however, we did see that a form of occlusion culling, much in the vein as described in 6.2, where areas are selected to be rendered in high quality, and others are in low quality, can be an effective way to improve performance, and thereby allow for higher quality textures in selective areas. Thus, we believe that, although the approaches presented in this thesis are far from perfect and would require further work to be used optimally, there is potential to use a method of occlusion culling together with environment mapping to improve performance, and quality by extension.

A large part of our thesis concerns the performance of environment maps as their source textures increase in resolution. This performance was measured as rendering time and memory utilization, as well as packet size when using a separate rendering server. Not surprisingly, we saw how increasing resolution of the texture used for the environment map resulted in increased rendering time, memory usage and packet size. Our method of improving quality relied largely on the opportunities that would be possible with better performance. Better quality, in this case, mainly refers to the higher resolution textures that would be permissible with increased performance. When looking at the outcomes of the two approaches presented, we can conclude that they both hold potential in terms of increasing texture resolution, but do so at the expense of certain other functionality. In essence, both approach 1 and 2 hold potential for improving resolution in very specific cases. For instance, approach 1 outperforms the original solution in terms of memory utilization, and approach 2 is highly suitable for use on a dedicated render server. As presented in 8.2, they may both appeal to niche customers, and may in these cases be viable alternatives. For the general customer, however, the original solution may still prove the best option.

While this thesis can be seen as an early exploration into some alternative methods of environment mapping and occlusion culling, each of the approaches described would need further refinement and work to become viable alternatives used in the industry. For example, time constraints forced some design decisions that we believe could be improved, which would likely bring further improvements in terms of performance. Most of all, we hope that this early look into these alternative solutions may at least shed some light on what is possible, and which alternatives hold promise if further developed. For instance, further development of approach 1 could potentially fix the issue now brought to light in 8.4.1. Furthermore, we see potential to also further minimize memory usage in approach 2, which could make it a viable alternative in the future. Most of all though, we hope that what we have presented in this thesis should serve

as a guide on how to go forth in trying to optimize performance, and show which paths are worth investigating, and which may lead to a dead end.

11. References

[1] Lambers, M. “Survey of cube mapping methods in interactive computer graphics.” Visual Computer 36, 1043–1051 2020, doi: 10.1007/s00371-019-01708-4

[2] Redway3D (2021). *Redway3D Documentation*. http://www.downloads.redway3d.com/downloads/public/documentation/bk_re_ray_tracing.html [2022-04-06]

[3] Magick++ *Magick++ Documentation* https://imagemagick.org/Magick++/ [2022-04-11]

[4] Akienne-Möller, T., Haines, E., Hoffman N. (2018) “Real-time rendering. 4th edition.”, Boca raton, FL, USA.

[5] Redway3D documentation (2021) *RED::ENV_TYPE* http://www.downloads.redway3d.com/downloads/public/documentation/APInamespaceRED.html#i4592 [2022-04-06]

[6] Redway3D documentation (2021) RED::IImageCube::CreateEnvironmentMap http://www.downloads.redway3d.com/downloads/public/documentation/APIclassRED_IImageCube.html#i1091 [2022-04-06]

[7] Gambetta G. (2021) “Computer graphics from scratch” 1st edition, San Fransisco, CA, USA

[8] Redway3D (2021). *Redway3D Documentation* http://www.downloads.redway3d.com/downloads/public/documentation/bk_re_monte_carlo_sampling.html [2022-04-20]

[9] Redway3D documentation (2021) *Culling method* http://www.downloads.redway3d.com/downloads/public/documentation/bk_sg_culling_methods.html

[10] Andújar, C., Saona-Vázquez, C., Navazo, I. and Brunet, P. (2000), Integrating Occlusion Culling and Levels of Detail through Hardly-Visible Sets. Computer Graphics Forum, 19: 499-506. https://doi.org/10.1111/1467-8659.00442

[11] J. El-Sana, N. Sokolovsky and C. T. Silva, "Integrating occlusion culling with view-dependent rendering," *Proceedings Visualization, 2001. VIS '01.*, 2001, pp. 371-575, doi: 10.1109/VISUAL.2001.964534.

[12] Pantazopoulos, I., & Tzafestas, S. (2002). Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent and Robotic Systems*, *35*(2), 123-156.

[13] Bartz, D., Meißner, M., & Hüttner, T. (1999). OpenGL-assisted occlusion culling for large polygonal models. *Computers & Graphics*, *23*(5), 667-679.

[14] Redway3D (2021). *Redway3D Documentation*. http://www.downloads.redway3d.com/downloads/public/documentation/APIclassRED_IViewpointRenderList.html#i2128 [2022-04-012]

[15] Redway3D documentation (2021) Manipulating images http://www.downloads.redway3d.com/downloads/public/documentation/bk_im_manipulating_images.html [2022-04-06]

[16] Redway3D documentation (2021) *RED::ImageTools::Resize* http://www.downloads.redway3d.com/downloads/public/documentation/APIclassRED_ImageTools.html#i1252 [2022-04-06]

[17] Samuel, T. K., McNally, S., & Wynkoop, J. (2012, July). An analysis of gpu utilization trends on the keeneland initial delivery system. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond* (pp. 1-6).

[18] Saxena, P., & Sharma, S. K. (2017). Analysis of network traffic by using packet sniffing tool: Wireshark. *Int. J. Adv. Res. Ideas Innov. Technol*, *3*(6), 804-808.