Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Fakultät für Mathematik und Informatik
Institut für Informatik
Mensch-Computer-Interaktion

Masterarbeit

zur Erlangung des Grades
Master of Science (M.Sc.)
im Studiengang Informatik
an der Universität Würzburg

Enhancing Stereoscopic Image Quality in Virtual Reality by Means of High-Fidelity Texture Filtering

vorgelegt von
Till Wübbers
Matrikelnummer: 2560615

am 12. September 2023

Betreuer/Prüfer:
Prof. Dr. Sebastian von Mammen, Chair of Human-Computer Interaction, Universität Würzburg

Zusammenfassung

Die sphärische Fotografie ist eine leistungsstarke Technik zur schnellen und einfachen Erfassung großer Umgebungen. Die aufgenommenen Fotos können für immersive Umgebungen, virtuelle Touren und mehr verwendet werden. In dieser Arbeit zeigen wir die technischen Aspekte der Verarbeitung, die nach der Aufnahme eines sphärischen Bildes erfolgt, und wie die Bildqualität während der verschiedenen Schritte aufrechterhalten werden kann.

Wir erörtern Techniken zur Konvertierung des Bildes in verschiedene Projektionen, warum diese Projektionen überhaupt notwendig sind und wie man Methoden wie Supersampling anwendet, um Artefakte zu vermeiden. Im Zusammenhang mit der Stereofotografie werfen wir einen Blick auf Methoden zur Korrektur eines Stereopaars in eine einheitliche Perspektive und vergleichen drei Anwendungen. Anschließend werfen wir einen Blick darauf, wie Mip-Maps für die Anzeige von Bildern in einem Virtual-Reality-Viewer erzeugt werden.

Schließlich beschreiben und implementieren wir eine neue Technik, die Probleme mit Aliasing bei der Betrachtung von sphärischen Bildern durch ein Virtual-Reality-Headset verbessert, durch Anpassung der Parameter für anisotrope Filterung. Diese Änderungen sind sowohl für Mono- als auch für Stereobilder relevant. Wir vergleichen diese Technik mit einem von uns erstellten Referenzbild und stellen fest, dass unsere Technik Bilder erzeugt, die näher an dem Referenzbild liegen als Bilder, die standardmäßiges anisotrope Filterung verwenden.

Abstract

Spherical photography is a powerful technique to capture a large scene quickly and easily. The captured images can be used to create immersive environments, virtual tours and more. In this thesis we show the technical aspects of the processing that goes on after a spherical image has been taken, and how to maintain image quality throughout the various steps.

We discuss techniques for converting the image to different projections, why these projections are necessary in the first place, and how to apply methods like super-sampling to avoid artifacts. In the context of stereo photography, we take a look at methods for adjusting a stereo pair to a consistent perspective, and compare three applications. We then take a look at how mip-maps are generated for displaying images in a virtual reality viewer.

Finally, we describe and implement a new technique that helps alleviate problems of aliasing when viewing spherical images through a virtual reality headset by adapting the parameters for anisotropic filtering. These modification are relevant for both mono and stereo spherical images. We compare this technique to a high quality reference we created and measure that our technique creates images that are closer to the high quality reference than images using default anisotropic filtering.

Contents

List of Figures

List of Tables

Acronyms

API application programming interface. 26, 28, 33, 34

EWA Elliptical Weighted Average. 8, 27

glb glTF Binary. 33

MSE mean squared error. 37, 43, 46

PSNR peak signal-to-noise ratio. 37, 43, 46

SSIM structural similarity index measure. 37, 43, 46

VR virtual reality. 2, 4, 9, 11, 33, 38

1 Introduction

Spherical photography uses special camera arrangements to capture images from new and unique perspectives, utilizing a large field of view to capture a whole environment at once. One popular way to do so is by using wide-angle lenses to capture two 180-degree pictures in opposing directions and stitch them together. This is the method that will be explored in this thesis. There are other options, such as using more cameras or stitching a large number of pictures that are taken successively with the same camera at different angles. Using more cameras has the advantage of being able to create pictures with a higher resolution, at the cost of having more stitching seams within the image. Some smartphones allow the user to take panorama images by taking many individual pictures, without the need for any special hardware. However, this requires extensive processing to hide the transitions between the individual captures and also might create artifacts if the environment has elements that move over time.

The utility of spherical pictures is broad, and the comparatively low price of consumer hardware compared to more advanced environment capture tools like laser scanning allows them to be utilized in many situations.

A lot of utility lies in their capability to capture data of the environment in an instant, without having to move or rotate the setup. This not only saves time and allows for a larger number of captures, but also makes it possible to capture quickly changing environments. For example, some authors have used spherical photos to gather data about tree canopies (Ribas Costa et al., 2022), or to create reconstructions of three-dimensional environments based on the overlap of multiple captures (Pintore et al., 2018).

In this thesis, however, we will focus more on the further processing and display of spherical captures, in the context of displaying them inside virtual environments. Spherical photography has been used as a part of virtual environments for a long time, for example as cube-mapped backgrounds for large outdoor scenes in video

(a) Texture (b) Game

Figure 1.1. Example of an equirectangular texture (a) used as a background image, as seen in the screenshot of the video game Half-Life 2 (b).

games (figure 1.1). But a newer development is to display the resulting environment not as a backdrop, but to create an immersive scene that is experienced through a virtual reality (VR) headset. This allows a user to look around the image naturally and decide which parts of the image to focus on in an intuitive way.

For example, Walmsley and Kersten (2020) have used this to create a recreation of a historic location, using spherical photos in addition to laser scanned data and hand-crafted 3D modelling. This, as any application that allows the user to interactively view the image in VR, requires the spherical image to be rendered in real time, usually with a game engine. The authors used *Unreal Engine* for this purpose, but we will implement our own solution for this, using a simple graphics application provided by OpenXR (The Khronos Group, 2023) as a template.

This will require us to employ many techniques common to computer graphics, especially in the realm of texture sampling. Therefore, we will also discuss the process of getting a captured image displayed on a screen with the best possible quality, and all the preparation that is necessary.

Additionally, some cameras will take multiple images with stereo separation, allowing the image to have a sense of depth for the viewer. These scenes can be very immersive and create an appealing reason to use the technology in combination with a VR headset. However, stereo captures add another layer of complexity to the capturing process, as well as requiring additional processing to ensure correct depth perception when viewed. We will look at these difficulties and evaluate tools that help to overcome problems such as vertical stereo offset and misalignment.

Generally, the process of capturing and displaying any spherical image in an immersive way is a lot more complicated than regular photography. In this thesis we will follow the path of how an image is captured, processed and displayed. On the way, we will focus on three major areas of research: The vertical alignment of stereo

images, the pre-processing of the texture before sampling, and the texture sampling algorithm itself.

When working with stereo images, we want to trick the human visual system into perceiving depth through the use of two images that are slightly offset from each other. Any mismatch might throw off the viewing experience and create discomfort. Since these two images are captured by real camera systems that might not be perfectly aligned, we have to correct them and make sure they face in a consistent direction before we can do any further processing.

One important pre-processing step is the creation of mip-maps, as introduced by Williams (1998). These are required for our later texture sampling algorithm, and are created to pre-compute the removal of high frequency data in our our texture. Each mip-map is half of the resolution of the previous one, forming a chain down to the size of singular pixels. Using mip-maps is extremely common in graphics applications, and we will examine the use in the context of equirectangular textures.

Finally, we need to actually sample and display our textures. This requires understanding of all the previous steps, and differs depending on the choices that were made. Especially with equirectangular textures, the choice and shape of the mesh that the image will be projected on is an important factor. While authors like Hosseini and Swaminathan (2016) have researched spherical mesh generation for video streaming applications, we want to evaluate which resolution of spherical mesh we need for displaying our photographs. As an alternative, the cubemap approach promises a simpler choice of mesh but requires a different texture projection.

We will start at the point at which the photos are taken. This might be done by specialized cameras that use wide field of view lenses with angles of 180 degrees or more, or by taking multiple separate images with a single lens. These multiple images must be made into one, and often come in projected formats that are not simple to process with regular image software. Sometimes conversion between those projections is also necessary, which requires careful choice of resolution and anti-aliasing settings to maintain the best quality. Stereo images are more difficult still, as they require two aligned captures that are on the same height but horizontally separated. Any vertical offset that is still present after capture must be corrected for before any further processing.

When using textures in a three-dimensional environment, they will naturally be displayed at varying sizes and angles depending on the point of view. Sampling such a texture so that it can be displayed in an artifact-free way is not trivial. We will look at the approach of mip-mapping and how it is applicable in the context of real-time computer graphics. We will also have a brief look at options to adapt the creation of such mip-maps to account for any inherent projections of the texture.

After all pre-processing is done, there must be an application that loads these images and displays them. While there are options to do so for a "flat" output device like a monitor, generally we assume the target device to be a VR headset, as it allows for the most natural experience of viewing spherical images. Such an application requires the use of 3D computer graphics to display a scene from the point of view of the headset. The image itself is projected as an environment around the point of view.

There are many different choices to be made, such as the shape of the model, the projection that is used and the details of how the texture is sampled. We will develop a reference application that displays the image in the best possible way known to us, in a non-interactive way. We will then implement and measure many different options, and compare them against that reference in terms of final image quality.

Lastly, we show a way to improve the rendering of equirectangular-projected images in an interactive 3D application. By understanding that general purpose texture sampling algorithms do not account for projected textures, we can adapt the texture filtering parameters to achieve better image quality. We will compare the results of this adapted code to unmodified images and show that in some cases the image quality can be improved.

2 Related Works

In this thesis, we look at the entire process of capturing and displaying spherical photography. Before we will describe our own process and methods for this purpose, we will look at the current state-of-the-art approaches. This involves many different processing steps and disciplines, so we will focus on the topics that are most relevant to our research. This chapter will also lay the groundwork for the techniques that are used throughout the thesis.

2.1 Uses of Spherical Photography

A large part of this thesis will focus on the display of spherical photos in virtual reality. Such applications usually aim to create new ways of experiencing the image, for example creating environments of historical locations (Walmsley & Kersten, 2020).

Using the spherical images as an environment that surrounds the viewer in virtual reality allows for the creation of more immersive experiences. The user can naturally decide which part of the environment to focus on simply by turning their head, eliminating a need for a user interface that would be required on a flat display. Additional interactive elements can be placed in the worlds, which allow for the user to move, receive additional information and more. This kind of interactivity usually requires an additional input device that the user can control.

Walmsley and Kersten (2020) supplement the spherical images with laser-scanned and hand-crafted 3D models, to create an experience that lets users explore the Imperial Cathedral in Königslutter. Interactive environments like this require the use of a game engine, such as Unreal Engine in this case. For such applications, spherical photography might only be a part of the whole experience and must be integrated accordingly.

To get the most out of such an experience and avoid adverse effects such as motion sickness, the experience should provide high quality imagery. Visual disturbances like flickering screens are known to cause virtual reality sickness (Chang et al., 2020), and it is likely that such flickering caused by rendering artifacts could create similar effects. Additionally, the experience might contain spherical stereo images, which require extra care to avoid a different cause of discomfort. Even small amounts of misalignment between the two stereo images, such as vertical offset or rotation, can cause discomfort (Jin et al., 2006).

2.2 Stereo Capture

Stereo images work by capturing two separate images at a fixed horizontal offset. The slight difference between the captures creates a perception of depth when the images are shown to the corresponding eyes. However, this effect needs to be within certain parameters to be comfortable to the viewer (Jin et al., 2006). If the images differ too much, the perception of depth is lost, and the viewing experience might become uncomfortable. As described by Hwang and Peli (2014), even if done perfectly, stereo images might still create some amount of discomfort.

It is also important to keep the vertical offset between the images to an absolute minimum. Jin et al. (2006) show that both rotation and vertical offset between the image pairs can cause discomfort in the viewer beyond a certain point. The image pair created by the two cameras is generally not perfectly aligned, but we can process the data to be closer to our defined bounds. If we can figure out a common space that the images are captured in, we can move and rotate them to be on the same plane with no vertical offset. Methods for finding such a reference space have been developed, but are also still an active area of research.

Papadimitriou and Dennis (1996) have developed an algorithm that takes a pair of stereo images and warps them to be vertically aligned. This transformation will make the two images epipolar, meaning that any matching point in the two images will be at the same vertical height. This is not only important for viewing comfort, but also for applications that will try to extract depth information, as it simplifies the search for correspondent points in the image. However, this approach is designed for perspective stereo images and does not work with spherical photos.

Fernandez Galaz (2021) has developed an application that can detect matching feature points in an image, extract the pose information from the location of those features, and then correct the image accordingly. We will later be using this application to apply such a transformation to some example images. This application only works with perspective photos as well, like the previously mentioned approach.

A method that can rectify spherical stereo images has been developed by Abraham and Förstner (2005). However, it requires a calibration board to perform the alignment, making it unsuitable for easy use with arbitrary stereo photos. Other authors have put further research into the field of aligning spherical stereo pairs. Ohashi et al. (2017) have shown that one might achieve better results by converting the images from the fisheye capture to an equirectangular projection first. The authors describe another approach to align the images: They detect matching feature points between the stereo pairs, as similar methods for perspective stereo images have done.

2.3 Texture Filtering

As we will later take our spherical image and display it on three-dimensional geometry, we need to sample it correctly. A pixel of our display device might cover a varying number of pixels on the texture, which have to be combined in a best possible estimation of what the texture looks like at this point. For this purpose, the process of mip-mapping has been developed by Williams (1998). Mip-maps are a series of lower resolution versions of the source image, shrinking in half with each step. They are generated before the texture gets sampled and are sometimes stored in special file formats like DDS (Microsoft, 2020). The simplest form used by the original paper averages the color of four pixels into one, but there are ways to generate higher-quality results, such as using a *Kaiser Filter* instead (Jonathan Blow, 2001).

If we just pick a single pixel of our sample area, the resulting image might contain aliasing artifacts such as moiré patterns. Mip-maps provide an easy way to reduce these, as the averaging process eliminates high frequency details from the source image. It is only necessary to calculate the fitting mip-map coordinates as described by Williams (1998). This means less calculations have to be done at run-time, as it is only necessary to sample the mip-map once at the correct location.

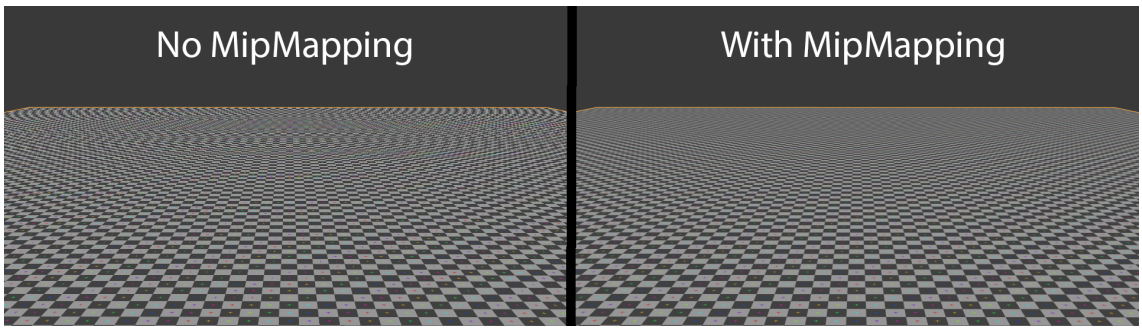


Figure 2.1. Comparison of a plane rendered with point sampling and mip-mapped sampling (Wikipedia contributors, 2023). The first image using point sampling shows visible moiré patterns.

However, the authors also mention that this process is not perfect. The averages are calculated in both axes equally, and therefore do not fit perfectly if the required area on the texture we want to sample is not square. This happens when looking at a texture with an oblique angle and requires more advanced sampling to take place. For this purpose, anisotropic filtering has been developed. By using an approach like the Elliptical Weighted Average (EWA) filter (Greene & Heckbert, 1986), we can create a better sampling result that neither creates moiré patterns nor blurs the texture too much. This method still requires the use of mip-maps, but does more than one sample at a time to get a higher quality result.

2.4 Spherical Mesh Generation

We have so far discussed different methods of storing spherical images, but we also need to display them. The obvious choice for displaying spherical images is a spherical mesh. However, the details of how to construct such a mesh and with which detail need to be understood. Hosseini and Swaminathan (2016) have created a system for dynamic 360 video streaming that generates a spherical mesh dynamically, creating detail where it is most needed.

Another method to display the image is ray tracing, which will be used later for creating reference images. Since we have a relatively simple scene - a sphere around a point of view - we can use the implicit sphere formula for deciding which point on the sphere we should display. This will remove any inaccuracies caused by low resolution sphere meshes.

3 Image Processing

To better understand the meaning and impact of choices that can be made along the way, we will follow the whole process of taking the spherical photo and processing it until it is shown to the viewer. This starts at the point where the image is captured by the hardware and ends at the pixels that get displayed inside the VR headset.

Since we are capturing a spherical area around our camera, we will need to think of our data in a spherical configuration as well. We need a system to specify the direction any point of data comes from. For that reason, we use a spherical coordinate system with its origin centered on the camera itself. The angles θ and ϕ describe the latitude and longitude of a point on the sphere. We can also use the radius r of the sphere to specify a distance from the origin, however, this information is usually not available for a regular camera capture.

However, in practice the data is not stored in a way that is parameterized by those coordinates. The camera chip, as well as most graphics applications produce and require image data in a rectangular format. Therefore, we need to map the surface of our sphere to a rectangle by applying some form of projection. Sphere-to-rectangle projections have been researched for a multitude of applications like cartography (Snyder, 1987), and a large number of possible projection methods exist.

The most important factors to consider for our choice of projection are the ease of computation and the compatibility with further image processing steps. Since the textures are often high-resolution photos, any transformations of the data might take a significant amount of time if no care is given to optimizing the process. This is less of a problem if this happens during a pre-processing step that might take place before the image is meant to be viewed. Even so, one might imagine applications like live-streaming the data directly from the camera to the headset where this would matter. Either way, when the image is loaded and needs to be displayed in the headset, the transformation of the loaded image into pixels on the screen absolutely needs to be efficient.

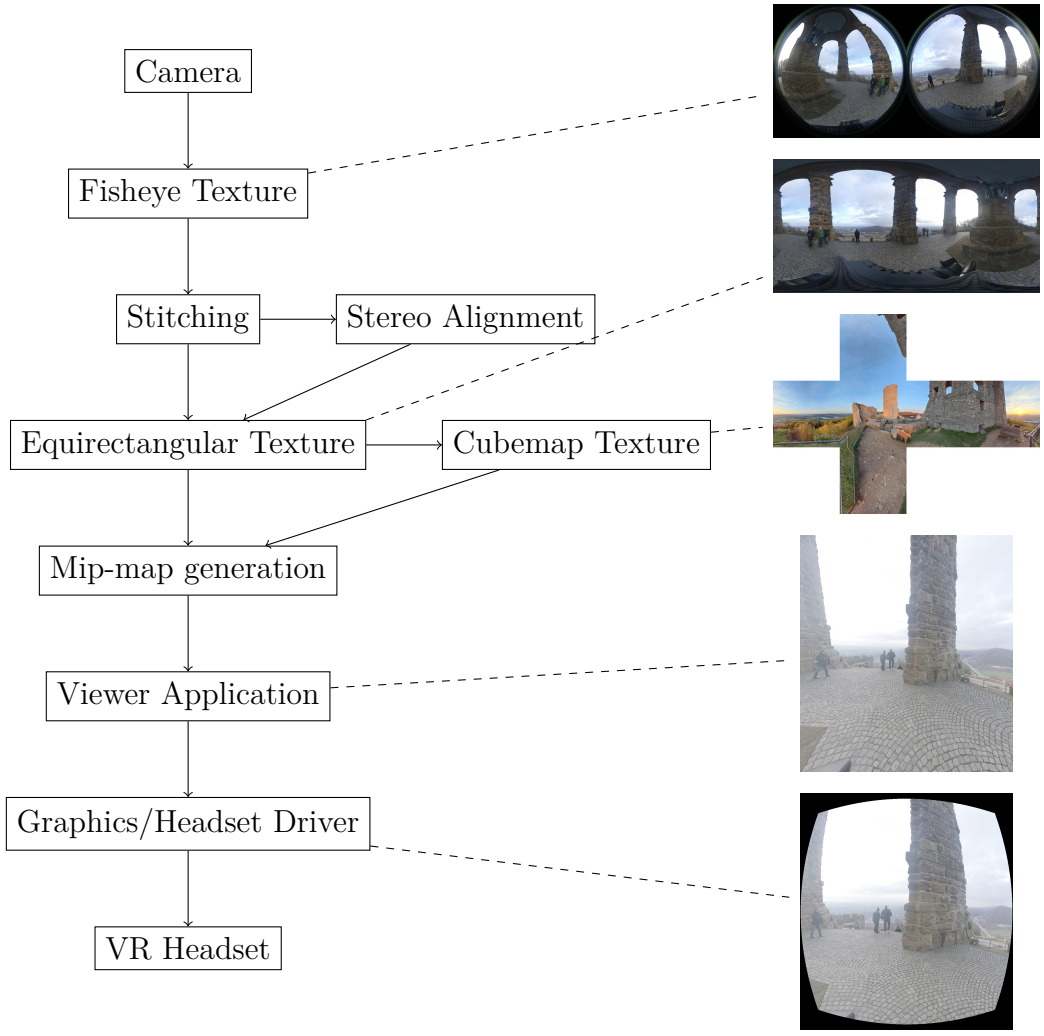


Figure 3.1. Process of capturing and displaying a spherical photo. The image undergoes multiple processing steps using different projections before being displayed in the VR headset.

Figure 3.1 gives a rough overview of all the required steps. We skip of most of the physical aspects of the camera hardware, and begin with the digital data that the camera provides to us. However, one aspect of the hardware is essential to know: The lens. Since the photo itself is not a simple perspective image, we need to know the field of view of the camera to correctly work with the image it creates. Our main focus will be on pictures taken by cameras with fisheye lenses, although it would also be possible to create spherical images by stitching a large number of simple perspective photos.

Such a camera captures the image through its optics and sensor, generating a fisheye-

projected texture. It might do so from multiple opposing angles, which means the images will need to be stitched together. Alternatively, the multiple angles can be used to create a stereo image, if they are taken in a parallel direction instead. At the end of the stitching process, the image is exported as an equirectangular projected texture. This format is a somewhat common ground between most cameras and methods, as it has a consistent way of covering the 360° area, and has all of the camera-specific processing already applied.

It is also commonly supported by VR viewer applications and some game engines. However, there is another projection format that is often used in game engines for this purpose: Cubemaps. We will compare the differences between those two formats later, but if we want to use cubemaps we will have to generate them from the equirectangular image at this point.

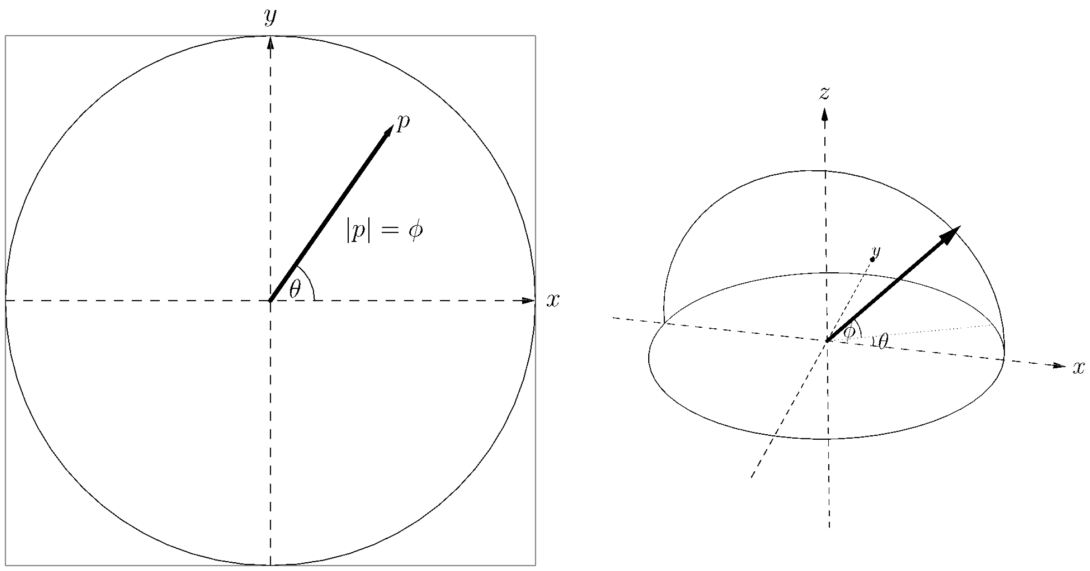
Finally, whatever format we decide to use will either go through a dedicated "offline" mip-map generation process, or have them created when loaded by the viewer application. This application will be the program that actually runs and displays the image on the VR headset. Once the texture with mip-maps is loaded, it will be rendered by the application using the pose of the headset to show the section of the texture that the viewer is looking at. Finally, the image will get post-processed so that it fits the display requirements of the headset used, which results in the final pixels the user actually gets to see.

In the next sections, we will discuss each step in more detail, and describe which path we took for our image processing setup.

3.1 Image Capture

There are many different ways to create a spherical image. If one does not require a full sphere, setups that capture a sphere segment in a single shot with just one sensor and lens are possible. This is usually done with fisheye lenses, as they allow for high field-of-view angles of 190 degrees and more.

To capture a full sphere, multiple images need to be stitched together. These images can be created by either capturing them at the same time with a multi-camera setup, or by capturing them one-by-one with the same camera at different orientations. The advantage of the first method is that the images will be easier to combine, as they



(a) Texture View (b) Sphere View

Figure 3.2. The relation between the fisheye texture coordinates in (a) that map a point (θ, ϕ) on the texture to a directional vector on the sphere (b).

capture the whole environment at the same time. Changes in the environment might not only cause artifacts, but can also cause worse stitching results. A common capture setup is to use two 180+ degree fisheye cameras that point in opposing directions. This is also the setup we will focus on in this thesis. The overlap between the two cameras is used to stitch the image in to one final result later.

The camera used for most of the example images in this thesis is the "Insta360 EVO" camera. It uses two fisheye lenses to create either 360° mono or 180° stereo images. For our captures the camera is attached to a tripod, while making sure that the two lenses are the same distance from the ground. This is especially important for stereo images, as we will later need to correct any vertical offset between the images. For mono 360° images, this will also make sure the horizon is mostly level.

When the photo is taken, the camera will take a picture with both sensors simultaneously. The raw image data is captured as a single texture with two fisheye-projected parts side-by-side, at a resolution of 6080 by 3040 pixels. Each fisheye capture has an angle of slightly more than 180 degrees. The camera stores this data as an ".insp" file, which is actually a file in JPEG format with extra data attached. This serves as our source for further processing.

As the spherical area of the image is reduced to two circles on a rectangular texture,

Figure 3.3. Side-by-side fisheye capture. Each half spans an area of slightly more than 180 degrees.

there are large areas outside the circles that are not used to store any information. While image compression will likely mitigate the effect on disk space used, it is still a large waste of memory once the image gets loaded on to a graphics card. It is therefore advantageous to use a different format for displaying the image in a viewer application.

As seen in figure 3.3, the images feature a small amount of overlap around the edges (for example the arm of the statue). This will be used for aligning and stitching the images in the next step. One can also see a small black border around the far edges of the projection, which is the face of the camera itself. This needs to be eliminated in the stitching step.

The camera comes with a proprietary software, that according to its user interface can perform stitching, horizon-leveling and exposure matching. After applying these steps, it will export an equirectangular image with the same resolution as the fisheye source file.

3.1.1 Angular Fisheye Projection

The angular fisheye projection is a format commonly used by spherical cameras to initially store the image in. It can be considered the “raw” format of a photo as

it directly generated from the captured light hitting the sensor of the camera. The projection maps a section of a sphere on to a circular area of the "flattened" two-dimensional texture. While the projection is capable of storing a full range of 360 degrees, camera lenses are usually not able to fill that area. Therefore pictures are often taken by two separate cameras, and combined into a single texture with two fisheye sections side-by-side. Regardless, for each fisheye-projected area, the origin is assumed to be in the center of the circle.

For the mathematical description of the projection we can create a bijective mapping of texture points to sphere points. As described by Ying et al. (2006), a point (θ, ϕ) on the sphere is mapped to the texture coordinates (x, y) in the following way,

$$x = c\theta \cos \phi$$

$$y = c\theta \sin \phi$$

with c being a constant scaling factor. The inverse mapping can be seen in figure 3.2, where a point p is mapped to sphere coordinates. The distance of the point to the center of the fisheye represents the vertical angle at which the light hits the lens, while the angle in relation to the x axis on the texture maps to the same angle on the lens. It is noteworthy that real fisheye lenses used on cameras do not exactly follow this formula as they might be somewhat non-linear. The distortions caused by the lenses can be measured and corrected for, as described by Schwalbe (2005).

3.2 Image Stitching

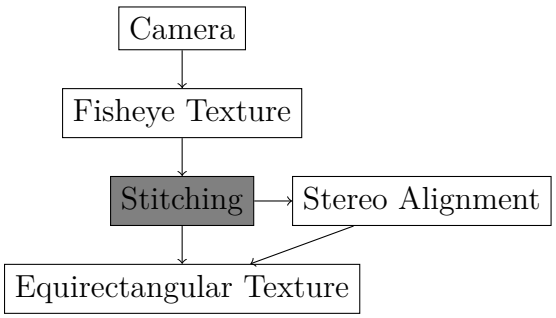


Figure 3.4. If the image is a stereo capture, we first need to align it.

With the proprietary software supplied by the camera manufacturer the two fisheye sections can be fused to a single 360° photo. This process is not well documented, but

performs image stitching at the transition between the two images, as well as match the exposure between them. The image stitching process has been researched in many ways (Ho & Budagavi, 2017; Lo et al., 2018), and it is likely that the software uses a similar approach. The process requires the unwrapping of the fisheye images, with the most common target projection being the equirectangular projection. The result is a single equirectangular image that smoothly transitions between the two fisheye sections from the source image.

3.3 Stereo Alignment

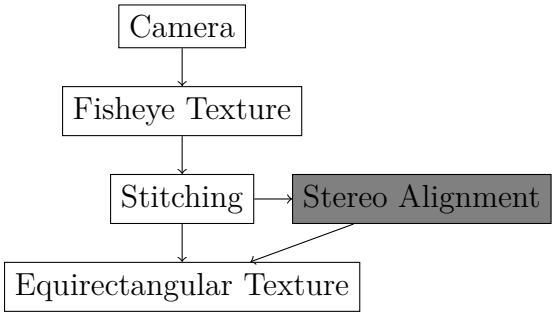


Figure 3.5. If the image is a stereo capture, we first need to align it.

When aligning perspective images, we have to figure out common features in the image pair. By measuring the different location of these points in the image pair, we can gather information about the pose each camera has in relation to the scene. Tools like stmani3 (Fernandez Galaz, 2021) use feature matching algorithms to detect similar points in both images. However, for our example image (figure 3.6) this did not work, and points had to be manually configured.

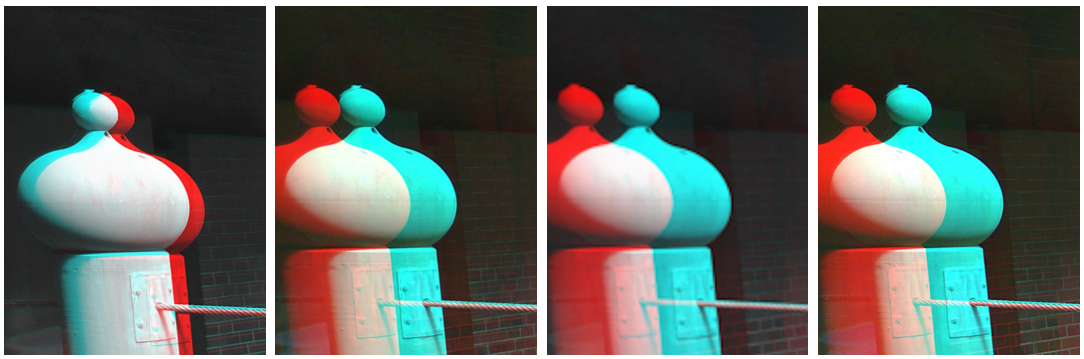
Other, closed-source applications managed to fully automate the process. StereoPhotoMaker managed to align the image correctly in the vertical axis, and so did Cosima3D. All applications calculated a vertical offset of about 11 to 13 pixels as seen in table 3.1, which seems to be correct when visually inspecting figure 3.7.

These mentioned tools only work for non-spherical images. With an equirectangular projection, the method of matching features to the same height no longer works, as horizontal lines in the scene are no longer horizontal in the picture. Instead, algorithms such as the one by Ohashi et al. (2017) previously discussed in section 2.2 must be used instead. For our fisheye image, the provided software by the camera manufacturer takes care of processing and aligning the stereo image.

Tool	vertical disparity
StereoPhotoMaker	11
stmani3	10.9
Cosima 3D	13

Table 3.1. Calculated vertical offset between the left and right stereo image of the picture in figure 3.6.

Figure 3.6. Perspective stereo image.



(a) unaligned (b) aligned (SPM) (c) aligned (stmani3) (d) aligned (Cosima 3D)

Figure 3.7. Difference between the stereo pair as an anaglyph image. The aligned image in (b) was created by StereoPhoto Maker, the image in (c) was made by the stmani3 software and (d) was using Cosima 3D.

3.4 Equirectangular Projection

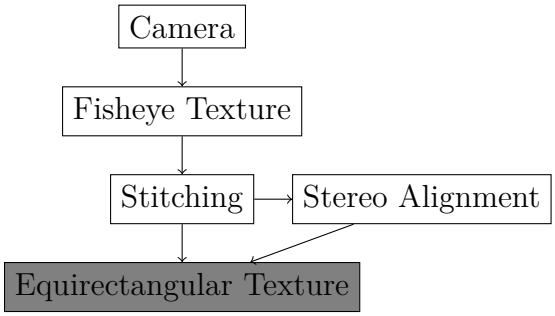


Figure 3.8. The image has been stitched and is now stored as a texture in equirectangular format. This is a common way to store spherical photos regardless of capture method.

Finally, regardless of which tool we use or if stereo adjustment was applied, our result will be an equirectangular texture (or two in the case of stereo images). If we choose to, we do not need to reproject this again, as it can serve directly as a texture to sample from. However, we will see why we might want to change the projection anyway in the next section.

The equirectangular projection is a commonly used for storing spherical images, as it is relatively easy to compute. Let ϕ, λ be the latitude and longitude of a point on a sphere with radius R. λ_0 is the central meridian, which will be mapped to the left edge of the texture. ϕ_1 is the standard parallel, which is the latitude with the least amount of distortion. It will be mapped to the center of the texture. For spherical photography purposes this is usually set to the equator of the sphere, as this allows the highest quality to be where people will look most often. The poles of the sphere will be straight above and below the point of view, which are less likely to be of interest.

According to Snyder (1987), the texture coordinates x, y are calculated as follows:

$$x = R(\lambda - \lambda_0)\cos\phi_1$$

$$y = R\phi$$

This will result in x coordinates in the $[-\pi, \pi]$ range and y coordinates in the $[-\pi/2, \pi/2]$ range. To use these as texture coordinates, we will remap both ranges into $[0, 1]$.

Figure 3.9. The stitched equirectangular image that was generated from the fisheye projection.

This step is done by the camera when taking or converting the original image. When a texture with this projection is rendered, a mesh with a UV layout that inverts this projection is used. Each vertex of the sphere model is mapped to a location on the texture, which is linearly interpolated between the vertices. This interpolation doesn't account for the spherical surface we are trying to approximate, which causes distortions, as seen in 4.3c. To reduce the distortions, the amount of vertices can be increased, at the cost of rendering performance.

3.5 Conversion to Cubemap

Cubemaps are commonly used in real-time computer graphics, often used as a distant background image or as a quick and effective estimation of environment reflections (Greene, 1986). A cubemap stores a 360 degree image as six individual textures, each of which is a simple 90° perspective projection taken in one of the cardinal directions. The edges of each texture seamlessly connect to another, and can be arranged as the net of the cube as seen in figure 3.11.

Cubemaps have some key advantages to other projections: The use of perspective projection for the individual faces means we can rely on the default behavior of the graphics pipeline and don't require custom texture filtering solutions like we will

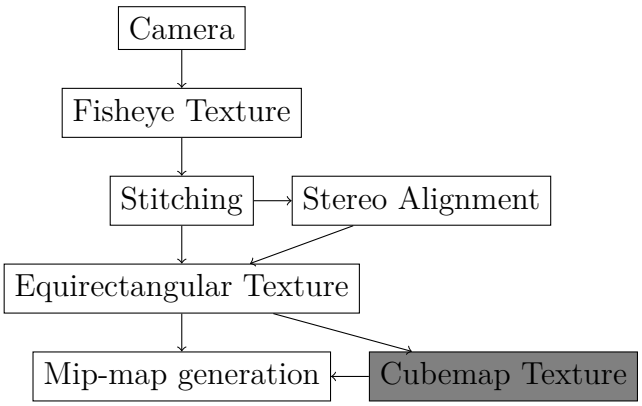


Figure 3.10. The conversion from equirectangular projection to a cubemap is optional, but might be more fitting for applications like game engines that expect this format.

show for other projections. Graphics cards also have specialized hardware that is optimized to sample from cubemaps in an efficient manner.

As described by Wan et al. (2007), a problem that cubemaps share with equirectangular projections is that they do not sample the area of the sphere equally.

If we want to display our image using a cubemap projection, we will have to convert it from the current equirectangular format at this point. The result of the stitching process forces the equirectangular format first, since most stitching algorithms work within that projection. As we are converting the image, we will be sampling the equirectangular texture to create the new cubemap. Therefore, as always when sampling a texture, we will have to worry about aliasing.

For the actual creation of the cubemap, one can imagine six perspective camera views within a sphere, pointing along the positive and negative direction of each axis. The cameras have a field of view of 90 degrees. Each of the cameras creates an individual texture, the edges of which match up exactly with its neighbor. With this assumption, we can map each pixel of the camera to a point on the sphere, which we can then map back to our equirectangular texture and sample.

To get a resolution that is capable of containing close to the same amount of information as the original texture, we set the size of the cubemap faces to a quarter of the width of the equirectangular texture. This should result in a roughly equal amount of pixels per area, assuming the following: The equator on the equirectangular texture maps to the equator of the sphere. In the horizontal direction, it covers the most space on the sphere, and therefore also contains the most information,

Figure 3.11. Cubemap projection with the six individual faces laid out in a cross pattern.

as the equirectangular projection stretches the horizontal information further apart the closer it gets to the poles. Since the equator is a straight horizontal line on the equirectangular texture, we should use the same amount of pixels to represent it on the cubemap. To cover the whole equator on the cubemap, we need the front, right, back, and left faces. Therefore each of these faces should require at least a quarter of the width of the equirectangular texture.

As we are generating pixels on the cubemap, we will need to sample from the equirectangular texture in a way that avoids aliasing. Simply matching the resolution as described before does not solve this problem, but allows us to use a fixed amount of supersampling. With 2× supersampling, each pixel gets sampled four times instead of one, in an evenly distributed pattern. Since we made sure that we have a roughly one-to-one mapping of the pixels, 2× supersampling should be enough to avoid aliasing. To make sure this holds ture, we will later evaluate this approach in chapter 4.

For our process, the conversion was done with the sphere2persp application by Bourke (2023). It can perform the conversion from equirectangular to perspective view with our given parameters, and also implements supersampling.

3.6 Mip-Map Generation

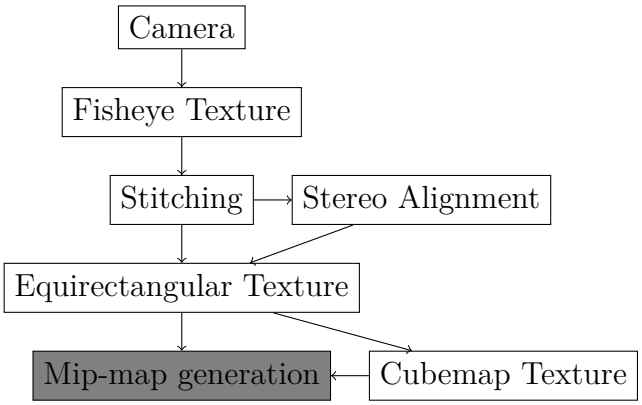


Figure 3.12. Before the image can be displayed by a viewer application, mip-maps have to be generated to allow for high quality and efficient sampling.

When rendering a three-dimensional mesh with a texture applied to its surface, we need to choose the color of each pixel that this mesh occupies on screen. With typical rasterized rendering we can map the position of our screen pixel to a position on the texture by interpolating between a set of texture coordinates (also called UV-coordinates) that are supplied with the mesh. This lets us choose the texture pixel closest to the pixel on the screen.

This will create a recognizable representation of the texture, but also unwanted artifacts. Notably, this does not take care of the case when a screen pixel maps to a point somewhere between two texture pixels. Especially when a texture is close to the screen and one texture pixel takes up more than one screen pixel, it creates a visible edge between the pixels. This can be solved by bilinear filtering, which linearly interpolates between the four closest pixels around the point on the target texture.

Another problem appears when the texture is far away from the point of view and therefore appears smaller: Even tiny movements of the point of view can create sudden changes in the output. This will happen when the texture itself contains high-frequency details which are sampled at a lower rate. For example, if our point of view is far away from the surface the texture is mapped onto, we might only sample the texture every ten pixels. According to the Nyquist Sampling Theorem (Landau, 1967), we need to sample at twice the frequency of the data to get a correct result. This leaves us with two options: First, increasing our sampling rate, also known as supersampling. We will later use this approach for our ray traced renderer,

but in this example it would require twenty samples per source pixel. This is not suitable for real-time applications. Therefore we use the second option: Filtering the texture we are sampling so that it does not contain frequencies higher than half of our sampling rate.

To filter a texture in this way, multiple methods have been developed. One such option often used in real-time computer graphics is mip-mapping. Mip-maps are lower resolution versions of the source texture, usually created in intervals of half of the previous resolution. The lower resolution allows us to remove high-frequency information in the image with a limited amount of memory space. We can now pick an appropriate level of mip-map for each pixel we sample, accounting for the amount of pixels of the source texture that we would have needed to sample instead. However, if we sample only the closest mip-map level, visible seams will appear at the distances at which the required sample jumps to another level. To fix this we can interpolate between different mip-map levels to get a smooth transition, which is known as trilinear filtering.

To actually create the mip-map levels, we need to combine the information present in an area of the source texture into a single pixel. For this, we will average four pixels of the previous texture into one pixel of the output, a technique known as the *box filter*. This will result in a half-resolution texture. We could continue this pattern to get progressively smaller textures that can be used for further distances. But to get higher quality results when creating these further levels, we will sample a larger amount of pixels from the source texture instead.

The box filter is only an approximation of the perfect filtering operation. As described in Pharr et al. (2016), we actually want to band-limit the high frequency information and convolve it with a pixel filter (e.g. a Gaussian). However, the authors also claim that "a box filter may be used for the band-limiting step, and the second step is usually ignored completely". Therefore, we have opted to not implement more advanced techniques.

3.6.1 Adapted Mip-Map Generation

In our tool we generate the mip-maps offline, before we run the viewer application. This lets us save and compare the generated images later. For our purpose of

sampling equirectangular textures, a special case emerges: When a pixel of the mipmap gets sampled, it is intended to serve as a representation of the lower frequency details of the source. The box filter as described before will create these lower frequency details with equal weight of all pixels. This is fine for flat surfaces, but for equirectangular textures it does not reflect the actual image information that is mapped to the sphere. The top and bottom rows of the texture span many pixels, but map to a small area of the sphere. As an example, a box filter that is not adjusted would sample four pixels into one, ignoring the fact that the pixels closer to the edge of the texture should have a slightly smaller weight as they span less area on the sphere.

One idea would be to sample a larger area, instead of "merging" exactly four pixels into one. However, increasing the sample area of a pixel would mean overlap of the sample areas for neighboring pixels. We cannot offset the next sample point to compensate for this without changing the overall resolution. Therefore, we can only work with adjusting the weights each pixel has in the filtering process. Lowering the weight of a pixel should make it less influential in the sum, reflecting the lower real area this pixel covers on the sphere.

3.7 Projecting the Texture on a Mesh

We now have a texture that we want to look at from a virtual camera perspective. This requires transforming the texture itself to the correct shape on the display, so that a person can look around and experience the image as if it was surrounding them. When using the rasterization pipeline of a graphics card to render the image, we will need to choose a mesh to project our textures on. In the case of our equirectangular projection, this will be a sphere. If we want to be able to efficiently sample the texture on the graphics card, we will have to assign a texture coordinate (also known as UV coordinate) to each vertex of the sphere. These coordinates represent the inverse equirectangular mapping from the sphere to the texture.

With our sphere mesh centered on the origin, each of our vertices lies at a world position (x, y, z). We can convert these coordinates to spherical coordinates (r, θ, ϕ) in the following way:

$$r = \sqrt{x^2 + y^2 + z^2}$$

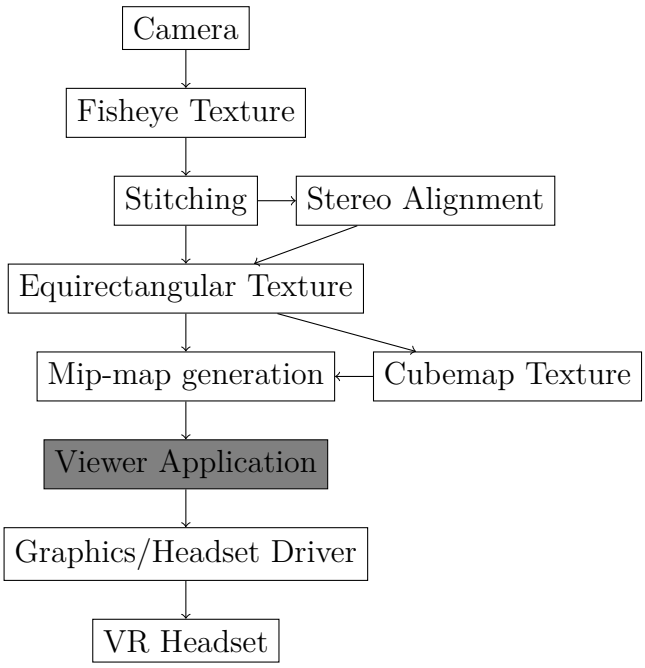


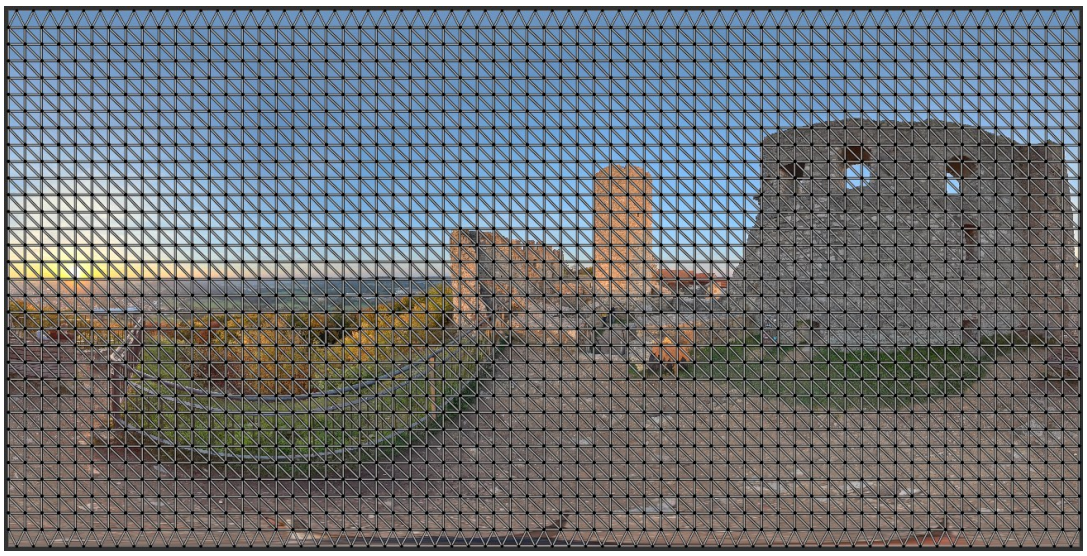
Figure 3.13. All pre-processing steps are done. The image now needs to be loaded by the viewer application so that the scene can be rendered.

$$\theta = \arccos\frac{z}{r}$$

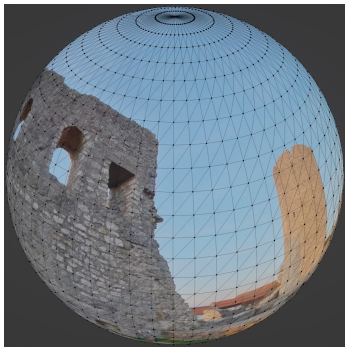
$$\phi = arccos\frac{x}{\sqrt{x^2 + y^2}}$$

We can now apply the equirectangular formula from section 3.4 to get our texture coordinates (u, v). This can be pre-computed and stored as part of the mesh file itself. Some 3D-graphics applications like *blender* (Blender Documentation Team, 2023) already provide functions to create such a layout, including parameters for the distribution of vertices on the sphere.

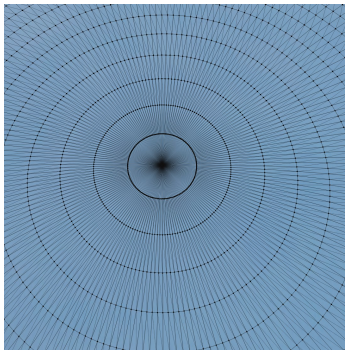
If we decide to not use the equirectangular texture and pick a cubemap instead, the shape of the mesh is obvious. Since we are no longer approximating a curved surface, we can represent the shape with just a few triangles, as seen in figure 3.15. The UV coordinates do not matter for the cubemap either, as we will be calculating our texture sample locations based on the view direction alone. This makes the mesh very simple to render, and we do not have to worry about mesh resolution or any pole regions.



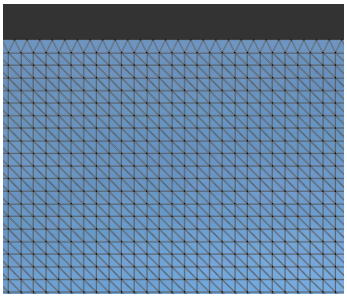
(a) Low-Res UV-Layout



(b) Low-Res Sphere Mesh



(c) Pole of High-Res Sphere



(d) Section of High-Res UV-Layout

Figure 3.14. Views of the selected meshes for equirectangular rendering. Images (a) and (b) show the layout of the low-resolution sphere. The high-resolution sphere shown in (c) and (d) has a similar layout, but with smaller triangle sections at the pole.

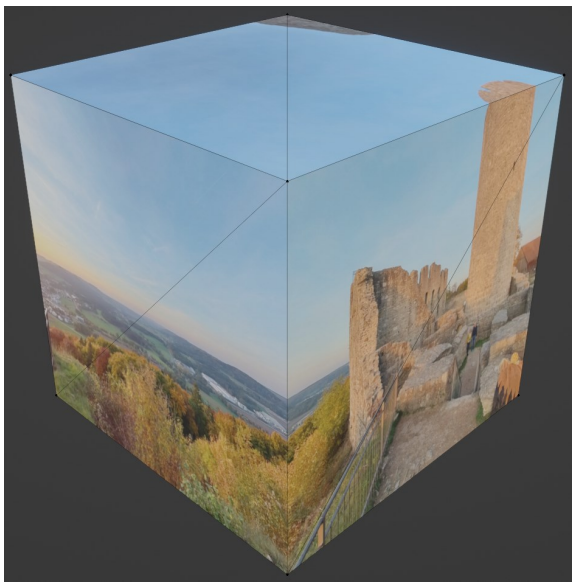


Figure 3.15. The cube meshe used for rendering the cubemap image. It uses significantly less polygons than even the low-res sphere model and does not have any pole regions.

3.7.1 Sampling the Cubemap

Sampling the cubemap does not use regular UV-Coordinate rendering. Instead we calculate the sample point on the texture directly based of the angle of the ray from the camera. The code in 3.16 works as follows: We get the view direction of the camera by subtracting the point on the cube mesh from the camera origin. This gives us a direction vector that points directly at the pixel we want to shade. We can then do an optional rotation to match our cubemap orientation to the same orientation that our equirectangular renders have. Finally, we sample our cubemap texture with the provided direction vector.

This is implemented by the graphics application programming interface (API), but roughly works in the following way: We intersect the direction vector with our hypothetical sphere, getting world coordinates x, y, z. These can then be mapped to our texture coordinates with the mapping defined in section 3.7.

3.7.2 Anisotropic Filtering

But still, a problem remains. Mip-maps create lower frequency versions of the source image, but they do so in both axes equally. If a texture is at an oblique

```
float4 MainPS(PSVertex input) : SV_TARGET
{
    float4x4 _90degRotation = { 0, 0, 1, 0,
                                0, 1, 0, 0,
                               -1, 0, 0, 0,
                                0, 0, 0, 1 };
    float3 dir = normalize(input.WorldPos - CameraPos);
    dir = mul(float4(dir, 1), _90degRotation);
    return tex.Sample(cubemapSampler, dir);
}
```

Figure 3.16. HLSL shader for sampling the cubemap. The image has to be rotated by 90 degrees to match the equirectangular image orientation due to the layout of the UV coordinates of the cube.

angle to the camera, it needs varied amounts of sampling in different directions on the texture. We can calculate how much by taking the partial derivatives of the screen coordinates in texture space. This gives us two vectors that have magnitudes corresponding to the required sampling range in texture space. This allows us to sample from different mip-map levels depending on the required amount of filtering, which is called anisotropic filtering. This whole process is usually implemented by the graphics card, with the only requirement being to enable the anisotropic filtering mode and providing mip-maps for the texture that needs to be sampled. A commonly used implementation for this that describes the exact pixels to pick from each mip-map level is EWA, as described by Greene and Heckbert (1986).

3.7.3 Adjusted Equirectangular Sampling

As described before, the default approach to anistoropic filtering does not account for the distorted mapping of equirectangular textures. Therefore, we will need to update the parameters for our filtering approach until we sample the correct area of pixels. Since the actual anisotropic filtering is implemented by the graphics card, we will need to influence this indirectly. The easiest way to do so is through changing the texture derivatives. Anisotropic filtering uses these two vectors to decide which area of pixels needs to be sampled. They are generated from the angle the camera is facing the particular triangle that the pixel lies on, but one can supply any value to the texture sampling function.

To get the partial derivatives of the texture UV-Coordinates in screen space, the *ddx* and *ddy* functions from the DirectX12 API can be used. Other graphics APIs might provide similar functions. When supplied with the UV coordinate for the current texture pixel, the partial derivative in form of a vector in screen space will be returned. The unmodified value of this can be given to the SampleGrad function to sample the texture with regular anisotropic filtering. However, we want to modify this value for our purposes first. As our texture is distorted, we want to adjust both values to counter this. The resulting values should still be usable to perform anisotropic filtering, but create a broader sampling area in the regions that have become stretched by the equirectangular projection.

The distortion of an equirectangular projection only affects the horizontal direction, and becomes stronger at the top and bottom of the texture. The center line of pixels on the projected texture perfectly maps to the equator of the sphere. As we go further from the center, the same amount of pixels in a row maps to smaller and smaller latitudes around the sphere. Therefore, we will need to widen the area of our anisotropic filter in the horizontal direction. The circumference c of a sphere with radius r at latitude l follows the formula of $c = 2\pi r * cos(l)$. The ratio between the texture width w and the circumference c determines by how much we need to extend the filter. As this formula expects the latitude ϕ, we need to calculate it from the y texture coordinate by reversing the equirectangular projection (Snyder, 1987):

$$\phi = y/R$$

$$\lambda = \lambda_0 + x/(R\cos\phi_1)$$

We need to remember that our texture y coordinate ranges from 0 to 1 instead of $-\pi/2$ to $\pi/2$, so we will need to map it accordingly. We can now multiply the horizontal components of the vectors from the *ddx* and *ddy* functions by the calculated ratio, resulting in an adjusted filter. This is limited by the maximum amount of anisotropy that is set by the graphics API (usually limited to a ratio of 1:16).

Applying the shader from figure 3.17 should take the ddx and ddy values, modify them according to our calculations, and then return the filtered result of the pixel. This implementation should be compatible with any DirectX application, and should

```
float4 PixelShader(PSVertex input) : SV_TARGET
{
    float2 xDerivative = ddx(input.UV);
    float2 yDerivative = ddy(input.UV);

    float latitude = input.UV.y * PI - PI / 2;
    float circumference = cos(latitude);
    float circumferenceRatioToNoDistortion = 1.f / circumference;
    xDerivative.x *= circumferenceRatioToNoDistortion;
    yDerivative.x *= circumferenceRatioToNoDistortion;

    return tex.SampleGrad(texSampler, input.UV,
                          xDerivative, yDerivative);
}
```

Figure 3.17. HLSL shader for sampling the equirectangular sphere mesh. The anisotropic filter gradients are adjusted according to our calculations to allow for correct sampling of the projected texture.

also be easily portable to other APIs or game engines, as long as an equivalent SampleGrad function is available.

3.8 Ray Traced Reference Implementation

To evaluate our resulting images and be able to apply objective measures to them we will need a reference image. As we are comparing the rendered frame buffer generated by our image viewer application, we will need to recreate an image that comes as close to a perfect result as possible.

For this purpose, a ray tracing solution was developed. We take our reference pose - the view and projection matrix of the application - and generate a rendered image manually on the CPU. Instead of using a mesh that is rasterized, we can utilize the possibilities of ray tracing and instead calculate our results with the implicit sphere formula. This gets rid of any inaccuracies caused by low-resolution mesh data. We generate an image with the same resolution as our application and project the rays through our virtual camera with the following algorithm:

Algorithm 1: Screen Coordinate to Ray

Input: Screen coordinates (screenX, screenY), screen dimensions (screenWidth, screenHeight), projection and view matrix (projectionMatrix, viewMatrix)

Output: Sphere intersection point

$clipSpaceX \leftarrow \frac{2\times}{screenWidth} - 1$;
$clipSpaceY \leftarrow \frac{2y}{screenHeight} - 1$;
$clipSpacePositionNear \leftarrow (clipSpaceX, clipSpaceY, 0, 1)$;
$clipSpacePositionFar \leftarrow (clipSpaceX, clipSpaceY, 1, 1)$;
$viewSpacePositionNear \leftarrow clipSpacePositionNear \times (projection)^{-1}$;
$viewSpacePositionFar \leftarrow clipSpacePositionFar \times (projection)^{-1}$;
$worldSpacePositionNear \leftarrow viewSpacePositionNear \times (spaceToView)^{-1}$;
$worldSpacePositionFar \leftarrow viewSpacePositionFar \times (spaceToView)^{-1}$;
$worldSpaceDirection \leftarrow$ Normalize($worldSpacePositionFar - worldSpacePositionNear$);
$sphereRadius \leftarrow 500.0$;
$sphereOrigin \leftarrow (0, 0, 0)$;
return RaySphereIntersection(worldSpacePositionNear, worldSpaceDirection, sphereOrigin, sphereRadius);

This takes the coordinates of the pixel on the screen and converts them to a ray that extends from the near plane of the camera at the matching location. RaySphereIntersection is a simple implementation of the line/sphere intersection formula, taking a ray origin and direction as well as a sphere origin and radius. It returns between zero and two resulting intersections. In case of no intersections nothing is rendered, as this will only happen when the view is moved outside the sphere. With two intersections, the one that is in the positive direction of the *worldSpaceDirection* vector is picked. We should now have a single point in world space on the surface of the sphere.

Our sphere is centered on the origin and has a radius $r = 500$. We can simply convert the world location (x, y, z) that we got from our intersection to spherical coordinates (r, θ, ϕ) as previously described in section 3.7:

$$\theta = \arccos\frac{z}{r}$$

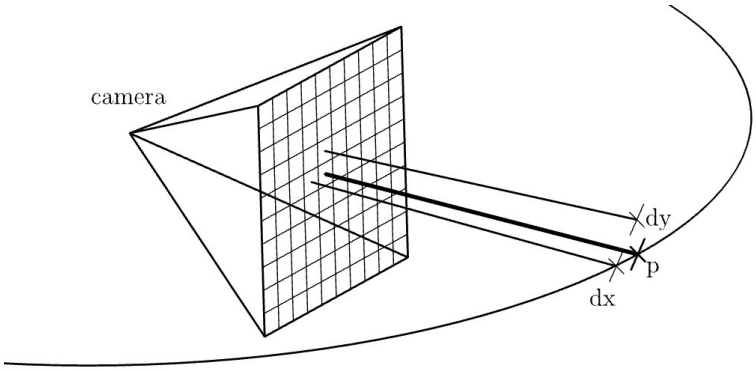
$$\phi = arccos \frac{x}{\sqrt{x^2 + y^2}}$$

We can then map these sphere coordinates to our texture by using the (inverse) equirectangular projection formula. This gives a precise point on the texture that we need to sample. It would also be easy to sample from a texture with any other type of projection defined in terms of sphere coordinates, as we simply need to switch out the formula.

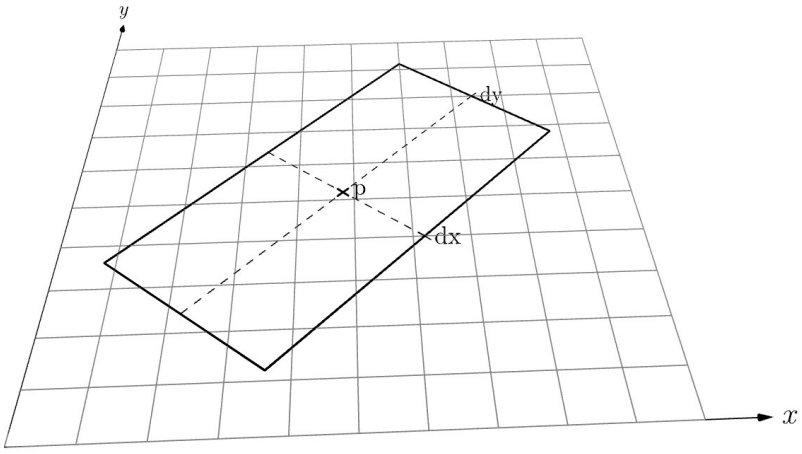
However, to sample the texture correctly we will again need to sample it in a way that avoids aliasing. Therefore, we employ the technique of ray differentials (Igehy, 1999) to get the bounds of the area that will need to be sampled. Since our ray tracing setup is relatively simple, we can just send an extra ray for the horizontal and vertical direction. The rays are offset by one screen pixel, and use the same intersection algorithm (1). Wherever that ray hits the sphere, we can transform it into texture space. We then use the difference between those texture space positions and the position of the main ray to calculate the derivatives. With that we can span an area on the texture which needs to be sampled, as seen in figure 3.18. We then sample this area with a sampling rate that is at least twice the amount of pixels contained on the longest axis, and use linear interpolation to blend between pixels at the sampling points. Ray differentials assume that the surface that is hit is planar, which the sphere is not. However, since the rays are only spaced a single pixel apart, the difference to the real required sampling area is negligible.

We now have the bounds of the area that we need to sample in the form of a quadrilateral. At this point in the GPU pipeline, techniques like mip-mapping (Williams, 1998) and elliptically weighted averages (Greene & Heckbert, 1986) would be used to efficiently sample this area. Since efficiency does not matter for our purpose as a reference image, we can instead sample each pixel within the area individually and average the result. This avoids having to generate and choose mip-map levels and will generate a correct result regardless, as we are now sampling the source data at a sufficient frequency to avoid aliasing.

Finally, we need to transform the resulting image into sRGB color space, as the output frame buffer of our viewer application will do the same. We should now have a matching image that samples the texture with close to perfect accuracy, which should serve us well as a reference image to compare our other results to.



(a) World View



(b) Texture View

Figure 3.18. The ray tracing texture sampling approach using ray differentials. The additional rays are used to create the sample area on the texture.

Due to the size of the images, running the program would take over a minute to calculate the final results. To speed up the program, the ray calculations and pixel sampling was parallelized. Since the calculations for each pixel are independent from each other and only need read-only access to the source texture, this is done with relative ease compared to other problems. The calculations for each row of pixels was parallelized using the OpenMP "parallel for" instruction.

3.9 Viewing the Result

The viewer is a simple 3D graphics application utilizing the DirectX12 graphics API to display images to the VR headset. The code is based on the "hello-xr" sample from The Khronos Group (2023). Several features have been implemented: The app must be provided with a mesh file in glTF Binary (glb) format as well as a matching texture file and a specified projection. This will be either a spherical mesh for equirectangular images, or a cube mesh for cubemaps. In the case of a cubemap, there must be six different texture files with matching suffixes (.t, .b for top and bottom, and so on). The application will load the mesh and image and apply the texture as defined by the UV-coordinates of the mesh. In all cases 16x anisotropic filtering is used. When specified by the settings, our shader for adjusted anisotropic filtering gradients is also applied.

The rendered scene is displayed on any VR headset compatible with OpenXR. For the first frame, a fixed pose is loaded and the pose of the headset is ignored. The left-eye framebuffer is stored in an image file and can be used for comparisons of different display methods. The resolution of this image will depend on the display of the headset that is connected to the computer, as well as the supersampling settings that are configured for that headset. In our case a Valve Index headset was used, with a per-eye screen resolution of 1440×1600 pixels. With the default settings a $1.4\times$ oversampling was requested by the headset, resulting in our final images being 2016×2240 pixels in resolution.

After storing this image, the application enters the real-time mode, in which one can use the headset to look around the image for manual visual inspection. No further captures are taken, but this mode is useful to spot flickering behavior which might not be noticeable in single image captures.

3.10 Final Processing

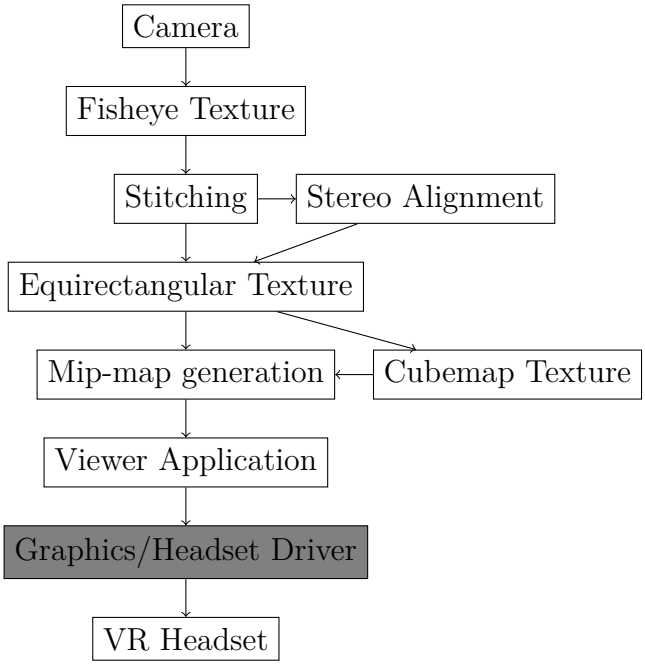


Figure 3.19. The scene has been rendered and is delivered to the XR framework for final processing.

The viewer application renders a perspective image using the pose information provided by the headset. However, this is not the final image that can be displayed on the headset. The specifics differ based on which headset and API is used, but in our case the image is given to the OpenXR runtime. Based on the headset (Valve Index in our case), an appropriate amount of barrel distortion is then applied to the perspective image. This corrects the distortion perceived by the user when looking at the display through the lenses of the headset. This is also why the runtime requests a higher render resolution than the headset display is capable of by default, since it can use the extra information to avoid quality loss from distorting the image. Further lens correction measures such as chromatic aberration might also be applied.

Even still, this might not be the last processing that is happening. If the application fails to reach a target frame rate (e.g. 90 frames per second), the runtime might use a previous frame and distort it based on the current headset position to synthesize a new image. This improves smoothness and avoids potential cases of motion sickness due to low frame rate, but might create visual artifacts.

Finally, the image is sent to the VR headset to be displayed. The user can look around and perceive the image as a continuous environment that is constantly being updated to match their perspective, hopefully unaware of all the processing that has been going on.

4 Evaluation

We have described the way to create our images with different methods. The three major options that can be changed are the choice of projection (cubemap vs. equirectangular), the resolution of the mesh (if applicable) and the use of our adjusted anisotropic filtering gradients. We have examined these options in the previous section and now need to compare them. For that purpose, we will take captured images and render a scene with them, creating separate results for each option we change. That allows us to compare the final images and evaluate which impact these options have on image quality.

So far, we have used some images taken with the Insta360 Evo camera. To get a greater variety of equirectangular images, a larger image set comprised of captures from multiple different cameras and locations was used. The image set contains 35 photos in equirectangular format, at resolutions ranging between 6080×3040 pixels and 10000×5000 pixels.

4.1 Image Metrics

We can visually inspect the resulting images from our renders to gather information about their quality and compare them to each other. However, since we have also created a high quality reference render with our ray tracing solution, we can also compare all other renders to it. This allows us to get a qualitative comparison that helps us rank each of the solutions based on more than just subjective reasoning.

We have created our ray traced image to be as close to optimal as possible, and will use it as a reference. To actually measure the quality difference to that image, we still need a metric however. Such comparative metrics are known as full-reference metrics, which there is a multitude of. We have chosen three specific metrics that take different approaches to measuring the image quality.

mean squared error (MSE) is the simplest of the metrics used in this thesis. It has often been described as inaccurate for measuring visual quality as perceived by humans (Teo & Heeger, 1994; Wang & Bovik, 2002), but is included due to its mathematical simplicity. It serves as a baseline comparison of how different the resulting images are, regardless of human perception.

The metric is calculated by simply taking the squared difference of each pixel in the two images and averaging the result. Or, as Horé and Ziou (2010) describes it, with f being our reference, g being the image that is compared and M, N being the image resolution:

$$\mathrm{MSE}(f,g) = \frac{1}{MN}\sum_{M}^{i=1}\sum_{N}^{j=1}(f_{ij} - g_{ij})^2$$

peak signal-to-noise ratio (PSNR) as a measurement is directly based on MSE, and in fact just re-states the result in a logarithmic scale. According to Wang et al. (2004) it still suffers from similar problems as MSE, being not always related to the image quality a human would judge the image at. It is still commonly used for image quality comparison however, for example in Hussain et al. (2019). While adapted versions of PSNR exist, Tran et al. (2017) have shown that it is still a valid comparison metric, at least compared to other more complicated variants of PSNR-based metrics.

The metric itself is defined as follows (Horé & Ziou, 2010), assuming a pixel has a color range from 0 to 255:

$$\mathrm{PSNR}(f,g) = 10log_{10}(255^2/\mathrm{MSE}(f,g))$$

structural similarity index measure (SSIM) is a metric that tries to rectify the issues relating to the human perception of image quality that were associated with the previous metrics. As described by Wang et al. (2004), it measures difference in *perceived changes in structural information* instead of *perceived errors* in the image. The intention is to create a measure that is closer to the perception of the human eye, instead of calculating the pure mathematical error that has been measured.

As the measure is significantly more complicated than MSE or PSNR, we won't be describing it in detail here. The actual implementation that was used for the evaluation of the images was provided by a python library.

4.2 Comparing Image Quality

We now have a way to compare the quality of our results against a reference. We cannot only use this approach to test the quality impact of our adjusted gradients method, but also measure other changes that might effect the final image quality. To perform these tests, a broad range of images was selected. We will at first use a number of synthetic images to test our adjusted gradients method, and evaluate the results individually. Additionally, a set of about 35 real-world photos captured with stereo cameras is used to create a broader dataset that also shows the effects of each approach in a more practical setting.

4.2.1 Adjusted Gradients

Our "adjusted gradients method" that was described in section 3.7.3 is meant to reduce aliasing artifacts when sampling an equirectangular texture, especially at the pole regions. Therefore, we have created our test pose in a way that captures the south pole of the sphere, containing the critical area.

For our first evaluation, we will use a synthetic checkerboard texture which will create high-frequency details. If such a texture is rendered without any filtering at all, it will create strong aliasing artifacts when shown at a distance. If the texture is flat, regular anisotropic filtering should be enough to alleviate this problem. However, for our case we will use this texture to project it on to a high-resolution sphere mesh.

We created two textures, one with black and white blocks that are two by two pixels in size, and one with blocks that are twice that size. The texture has a resolution of 8192×4096 pixels, which roughly matches the size of our later real-world captures. The scene was rendered at the resolution of the VR headset and the resulting image was captured and saved.

It is noteworthy that the captured image in figure 4.2 is not the final data sent to the display of the headset. The driver will still apply transformations to the image to correct for distortions caused by the lenses of the headset, as well as possible other techniques to generate intermediate frames in low-performance situations. Capturing this final result might be a more accurate measure of final image quality, but is

(a)

(b)

(c)

(d)

(e)

(f)

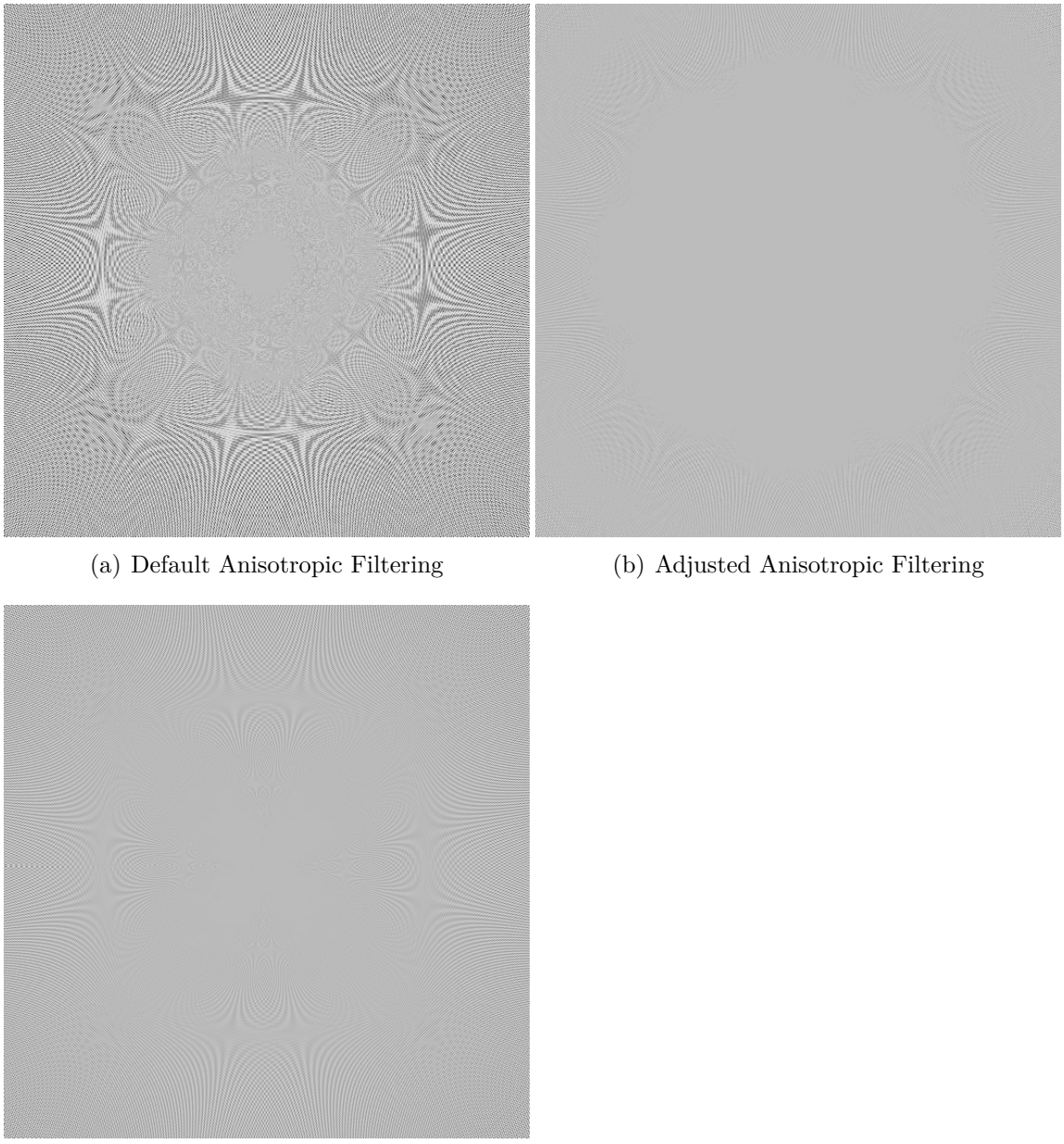
Figure 4.1. Selected images from the spherical image dataset. The following cameras were used to capture the images: Insta360 EVO for (a), unknown for (b), Insta360 OneRS for (c),(d),(e), GoPro Hero3+ for (f).

dependant on factors like performance and specific settings that vary between different headsets. Therefore, we opted to capture the frame before such modifications take place.

Due to the properties of the equirectangular projection, the checkerboard blocks will get progressively smaller as they get closer to that pole. This creates a transition to higher frequency image data, which is close to a "worst case" scenario for texture sampling and nicely visualizes the effects aliasing artifacts can create.

A section of these resulting captures can be seen in figure 4.2. 4.2(a) shows the regular 16× anisotropic filtering without any adjustment of the gradients. Aliasing patterns are clearly visible and are most prominent in the shown area around the pole of the sphere. When viewed in motion through the headset, strong flickering is visible in this area.

In figure 4.2(b), the shader from figure 3.17 that applies the adjusted gradients for anisotropic filtering is used. The aliasing artifacts are almost completely removed, although small sections of the texture still show them. It is not entirely clear why these effects are still visible in some areas, however, some contributing factors might be that the anisotropic filtering algorithm itself might not produce perfect results in



(a) Default Anisotropic Filtering

(b) Adjusted Anisotropic Filtering

(c) Ray Traced Reference

Figure 4.2. Two generated captures of the south pole of a sphere. A 8192×4096 pixel checkerboard texture with four pixel wide blocks is used as a source texture for the equirectangular projection. Both images use anisotropic filtering, but only (b) uses our adjusted gradients.

Table 4.1. Image quality metrics for the 2px checkerboard image.

Settings	MSE	PSNR	SSIM
Cubemap	6328.8	10.117	0.1149
Equirectangular, high-res, adjusted	4656.4	11.450	0.1359
Equirectangular, high-res, no adjustment	5401.7	10.805	0.1128
Equirectangular, low-res, adjusted	5496.6	10.729	0.0275
Equirectangular, low-res, no adjustment	6250.0	10.171	0.0018

Values are comparison metrics to the ray traced reference image.

Table 4.2. Image quality metrics for the 4px checkerboard image.

Settings	MSE	PSNR	SSIM
Cubemap	5704.5	10.568	0.2602
Equirectangular, high-res, adjusted	5430.0	10.782	0.2437
Equirectangular, high-res, no adjustment	6847.8	9.7752	0.2341
Equirectangular, low-res, adjusted	6735.3	9.8472	0.1344
Equirectangular, low-res, no adjustment	8252.2	8.9650	0.1192

Values are comparison metrics to the ray traced reference image.

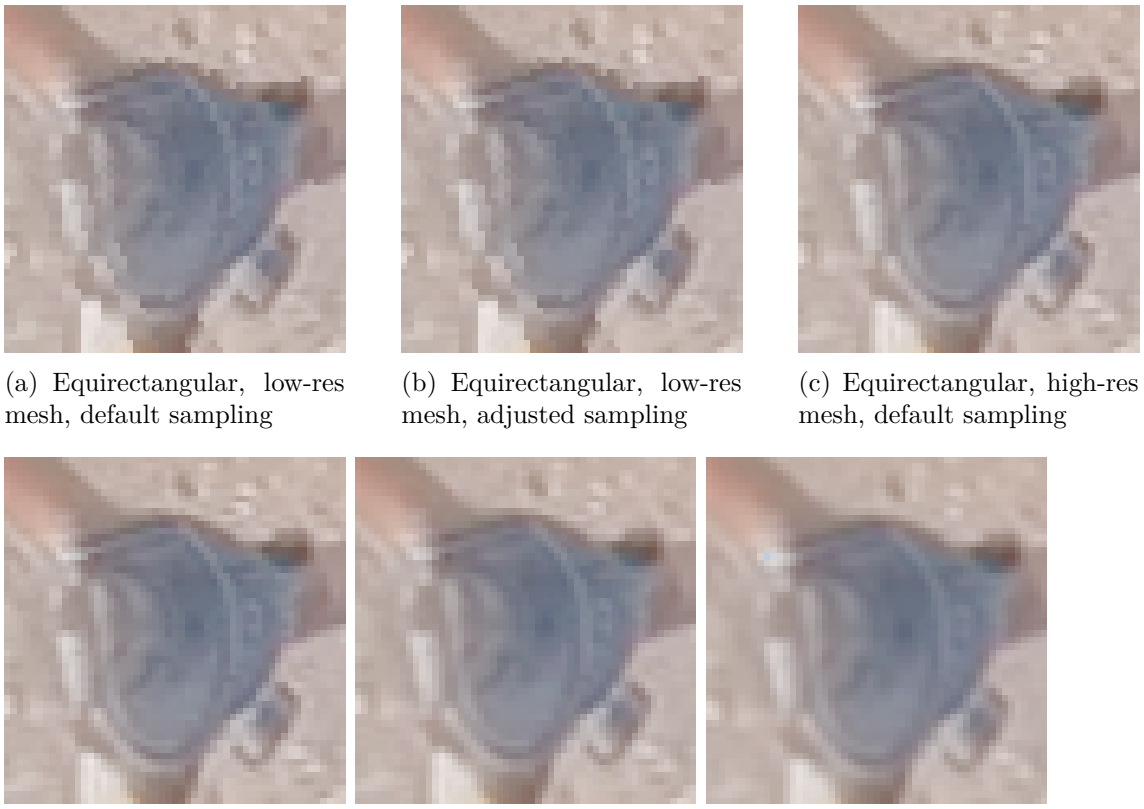
all cases. From a subjective point of view the flickering visible when looking around with the headset was no longer perceived.

The subtle moiré patterns equally appear in figure 4.2(c), which was created with our ray tracing approach, further leading to the conclusion that these artifacts come from something other than the anisotropic filtering issues we have discussed so far.

The scores for tables 4.1 and 4.2 were created with the previously described metrics, and compare the real-time sphere renders from 4.2(a) and 4.2(b) to the ray traced image in 4.2(c). Additionally, two low-resolution sphere meshes are measured as well. However, the mesh resolution will be closer examined in a later section.

If we compare the metrics from tables 4.1 and 4.2, we can see that for both MSE and PSNR the same pattern emerges: The captures using the adjusted gradient method are scoring significantly better than their counterparts. Using a higher resolution sphere improves the scores significantly as well, but the low-resolution mesh adjusted gradients scores better than the unadjusted high-resolution mesh. SSIM shows slightly different results. Adjusting the gradients still improves the score, but using a higher sphere resolution has a much bigger impact.

For each photo, six captures were generated. The first one is a ray traced view



(a) Equirectangular, low-res mesh, default sampling

(b) Equirectangular, low-res mesh, adjusted sampling

(c) Equirectangular, high-res mesh, default sampling

(d) Equirectangular, high-res mesh, adjusted sampling

(e) Cubemap

(f) ray traced reference

Figure 4.3. Comparison of different sampling methods and selected models. Point of view is looking down, cropped around the pole of the sphere. The image used for this picture is seen in figure 4.1(e).

that will be used for comparing the rest of the image set to. The other captures were rendered and saved by the viewer application with the following settings: One cubemap render that uses the textures generated from the equirectangular image, which each face at a quarter of the equirectangular resolution. Two renders of a low-resolution resolution sphere of about 4000 triangles, with and without the shader that uses the modified anisotropic filtering gradients. And another two renders of a high-resolution sphere of 16000 triangles, with and without the shader.

We will now evaluate the results for our larger dataset, a subset of which can be seen in figure 4.1. These photos have a big variety in content and should provide results that show the effects of our method in a real-world setting. Most of these images feature a mostly homogeneous blue sky, except for 4.1(a). This is relevant since the sky is at the top of the image, and therefore will be around the north pole of the sphere mesh. If the image is distorted there, it will have a less noticeable

impact if the pixels in the area are very similar. Some of the images are captured at ground level, while (b) and (f) are drone shots. Especially (f) features a very noisy forest at the bottom part of the sphere.

We will now look at an example capture from our dataset, generated with the image in 4.1(e). This should show the effects of our method in an non-synthetic image example. Figure 4.3 shows the most distorted area on a sphere mesh, the pole. The artifacts introduced by the low-resolution spheres are mostly blocky edges, at the points where the triangle edges are discontinuous across texture space. The presence of these is very noticeable, and hides any changes that could be visible between the default and adjusted filtering methods. When using the high-resolution mesh, these artifacts are strongly diminished. However, they are not entirely gone when compared to the cubemap or ray traced image.

The cubemap view, compared to all other images, does not look at a pole or corner of the mesh. Since the view is facing straight down at the bottom face of the cube, there is also close to no anisotropic filtering needed. This should result in an image that is close to optimal, at least in this area. However, comparing it to the ray tracing image it is noticably less blurry. This might be due to the difference in filtering approaches, as the cubemap uses mip-maps that were generated with the box filter and then sampled by whatever algorithm the graphics card implements. In comparison, our ray tracing solution calculates the area on the source texture directly and uses a varying amount of sample points.

Compared to the checkerboard image, the changes in score of the photo dataset are less pronounced between the categories. This supports the idea that the checkerboard image is a particularly bad case for creating artifacts in the image.

Figure 4.4 shows a different view of the same texture as in figure 4.3. This look at the equator shows that the artifacts at that latitude of the sphere are much less intense. The only visually noticeable artifact of the low-resolution images are the slight distortions of the bars of the railing, which should run in an almost perfect straight line (although the sphere shape should cause some slight curvature at this angle). The differences between the default and adjusted sampling can not be seen by eye, as the filtering at the equator is close to isotropic and the distortion of the equirectangular texture is very small.

(a) Equirectangular, low-res mesh, default sampling

(b) Equirectangular, low-res mesh, adjusted sampling

(c) Equirectangular, high-res mesh, default sampling

(d) Equirectangular, high-res mesh, adjusted sampling

(e) Cubemap

(f) Ray traced reference

Figure 4.4. Comparison of different sampling methods and selected models. Point of view is looking back in close to a horizontal direction near the equator. At this angle, there is only a small difference between all of the pictures. The photo used is seen in figure 4.1(e).

The captures are stored as separate image files and were used to generate the metrics in the following steps. The python library "skimage.metrics" was used to apply standard implementations of three common image quality metrics: MSE, PSNR and SSIM. Each of the generated captures was compared to the ray traced reference image generated from the same photo. The exact values were stored in JSON format for further processing.

Tables 4.3, 4.4 and 4.5 show the averaged results of these measures. When we look at these averages, we find that for all three metrics the scores with adjusted gradients are higher than for their matching unadjusted images. Similarly, the results for the high-resolution spheres score better than the low-resolution spheres for every metric. However, for MSE and PSNR the difference between low-res adjusted and high-res unadjusted is relatively low.

Table 4.3. Combined *Mean Squared Error* measurements for the photo image set.

Image Settings	MSE Mean	MSE Variance	MSE StdDev
Cubemap	130.9	8302	91.12
Equirectangular, high-res, adjusted	127.8	6843	82.73
Equirectangular, high-res, no adjustment	143.8	8756	93.58
Equirectangular, low-res, adjusted	174.8	10943	104.6
Equirectangular, low-res, no adjustment	191.7	13353	115.6

Table 4.4. Combined *Peak Signal-to-Noise Ratio* measurements for the photo image set.

Image Settings	PSNR Mean	PSNR Variance	PSNR StdDev
Cubemap	28.12	11.32	3.364
Equirectangular, high-res, adjusted	28.13	10.59	3.254
Equirectangular, high-res, no adjustment	27.66	11.19	3.345
Equirectangular, low-res, adjusted	26.69	10.19	3.192
Equirectangular, low-res, no adjustment	26.32	10.63	3.261

Table 4.5. Combined *Structural Similarity Index Measure* for the photo image set.

Image Settings	SSIM Mean	SSIM Variance	SSIM StdDev
Cubemap	0.778	0.007	0.082
Equirectangular, high-res, adjusted	0.780	0.007	0.081
Equirectangular, high-res, no adjustment	0.762	0.008	0.087
Equirectangular, low-res, adjusted	0.714	0.010	0.099
Equirectangular, low-res, no adjustment	0.696	0.011	0.104

(a) 960 Triangles (b) 3968 Triangles (c) 16128 Triangles (d) 32040 Triangles

Figure 4.5. Cropped captures of sphere meshes with different resolutions, all rendered with adjusted anisotropic filtering gradients active.

4.2.2 Sphere Resolution

To get a better understanding of how the resolution and topology of the sphere impacts the quality, we can look at a more varied set of examples. Four spheres composed of 960, 3968, 16128 and 32040 triangles are compared in this section. The triangle count is limited by our viewer application, which does not support meshes with more than 32767 vertices.

The clearly visible artifacts in figure 4.5a clearly show how the image quality is impacted by low mesh resolution. The texture is cut into discontinuous pieces around the pole of the mesh, which come from the gaps in the UV-map as seen in figure 3.14b and d. In an optimal setup, the triangles would be so close to the top of the texture, that the gaps would only appear in an area of the texture that is extremely horizontally stretched. This would effectively making them invisible. With such a low mesh resolution the triangles are not close enough, making them visible in the output.

With increasing resolution, the triangles become smaller and so become the gaps. In figure 4.5b the triangle count has roughly quadrupled, resulting in less noticeable discontinuities. However, they are still clearly visible. Images 4.5c and d look very similar and no longer have obvious artifacts. Only by directly switching between them in an image editor can the differences be seen.

4.2.3 Cubemap Supersampling

In chapter 3 we generated cubemaps from equirectangular images. This is necessary because we generally can not directly produce cubemaps from fisheye projections,

Table 4.6. Image quality metrics for the sphere mesh resolution comparison.

Settings	MSE	PSNR	SSIM
960 Triangles	216.80	24.770	0.5348
3968 Triangles	79.913	29.105	0.7705
16128 Triangles	50.735	31.078	0.8380
32040 Triangles	47.758	31.340	0.8437

Sphere meshes are compared to a "perfect" ray traced sphere.

Figure 4.6. Base image used for comparison in figure 4.7.

as stitching algorithms generally have been designed with an equirectangular result in mind. We claimed that by setting the cubemap resolution to a quarter of the equirectangular image, we would match the pixel count and only require 2× supersampling to avoid any aliasing artifacts and maintain acceptable quality. To confirm this assumption, we have created a comparison of cubemaps generated from equirectangular images, each with different supersampling settings.

We started with a base fisheye projected image captured directly by the camera (figure 4.6). We then skip any stitching or other processing and apply the different projection variations. Finally we use the result to render an image with our viewer for comparison.

For reference purposes an extra image that uses the usual stitching and equirectangular conversion process has been created, which was rendered in the ray tracing mode of our viewer. The result is shown in figure 4.7g. The crop used for all images

Table 4.7. Image quality metrics for the cubemap comparison from figure 4.7.

Settings	MSE	PSNR	SSIM
Fisheye → Cubemap (no SS)	9538	8.336	0.225
Fisheye → Equirect (no SS)	7299	9.498	0.241
Fisheye → Equirect → Cubemap (no SS)	7876	9.168	0.324
Fisheye → Cubemap (2× SS)	5646	10.613	0.510
Fisheye → Equirect (2× SS)	5643	10.616	0.515
Fisheye → Equirect → Cubemap (2× SS)	5649	10.611	0.518

Values are comparison metrics to the ray traced reference image 4.7(g).

has been specifically chosen to not lie in the area that would be affected by stitching algorithms.

For all other captures from figure 4.7, we directly project the image without any stitching or other processing taking place. Only the left half of the image was used for this. The view for the comparisons points in a direction so that only this half is visible, and no stitching is needed. We compare three different conversion paths, with and without supersampling applied: First, direct conversion to a cubemap, which would be optimal but is not possible if stitching is required. Second, a conversion to equirectangular format, which is used to show the state of the intermediate step. And finally, the way we created the cubemaps for our previous comparisons, by taking the equirectangular texture and converting it again.

The different effects of supersampling on the projections are clearly visible. The top row of images (4.7a to 4.7c) does not have supersampling applied in any of the processing steps. While the fisheye to cubemap conversions seems relatively unaffected, the conversions to equirectangular seems to create significant artifacts, easily visible at the high contrast edge of the bricks. The extra step of converting the equirectangular image to a cubemap in figure 4.7c further worsens the result.

The second row of images (4.7d to 4.7f) looks a lot more similar. The direct conversion to a cubemap is very similar to its counterpart 4.7a, while the two other images are significantly compared to their versions with no supersampling applied.

(a) Fisheye → Cubemap (no Supersampling)

(b) Fisheye → Equirect (no Supersampling)

(c) Fisheye → Equirect → Cubemap (no Supersampling)

(d) Fisheye → Cubemap (2× Supersampling)

(e) Fisheye → Equirect (2× Supersampling)

(f) Fisheye → Equirect → Cubemap (2× Supersampling each step)

(g) Fisheye → Equirect (stitched) → ray traced

Figure 4.7. Cropped captures from the viewer application using different approaches for projection and conversion. All images are based on the same fisheye photo, but have undergone different conversions. (a), (b) and (c) do not use supersampling to convert between the projections. (g) is generated as a reference with the proprietary equirectangular converter application of the camera.

5 Discussion

The goal of this thesis has been to explore the process of taking spherical images and displaying them in virtual reality, while providing the best possible image quality throughout. This started with the discussed methods of capturing an image to a fisheye texture or similar. From that point on, the texture has to be carefully processed.

When working with stereo captures, we have discussed reasons why the stereo images should be vertically aligned. Applications like stmani3 by Fernandez Galaz (2021) align perspective stereo images by extrapolating camera poses from matching feature points. Using a hypothetical application with an algorithm adapted to spherical stereo images such as shown by Abraham and Förstner (2005) or Ohashi et al. (2017) would likely allow for a better viewing experience in virtual reality.

Moving the image further along the process, we will need to convert and stitch a fisheye image into an equirectangular or cubemap projection. While stitching has not been discussed in depth in this thesis, the conversion process between projections is a potential way to lose image quality.

We have shown that for each processing step that transforms the texture, one has to be aware about how the texture is sampled. Our assumption that supersampling is needed to maintain better quality over direct sampling was supported by the example in figure 4.7. However, this is just a single example and only compares 2× supersampling to no extra sampling at all. While we have made a suggestion that with the correct choice of resolution the sample rate should be enough, we haven't experimentally confirmed that higher supersampling values won't create even better results.

Looking at the next step, the custom mip-mapping solution appeared to have barely made any difference in the generated textures. The error created by the incorrect sampling area seems to be too small to create major differences. Additionally, in the case of a spherical mesh, the image is mostly looked staight on, making the

transition between mip-map levels very gradual. This would make it unlikely to show any perceptible changes in the final image. Therefore we did not choose to implement a system that loads these modified mip-maps.

The biggest impact on quality throughout experimentation on this thesis was the resolution of the mesh used to estimate a sphere for equirectangular images. A low resolution would create artifacts across most of the image, distorting and bending straight lines. Additionally, due to the UV-layout of the mesh as seen in figure 3.14, the triangles at the poles would create large jarring distortions. To get a good quality result for the sample image in figure 4.5, we found the sphere resolution needed to be around 10000 triangles or more. Going significantly below that would create visible artifacts as seen in the first picture. One can always go higher, but this of course has an impact on rendering time. Especially on standalone VR devices with limited rendering budgets and battery life, the trade-off is unlikely to be worth it.

Finally, we described and examined our method of adjusting the texture space gradients for anisotropic filtering. The synthetic test image in figure 4.2 showed promising results in reducing image artifacts around the pole region. However, when applied to a real-world example in figure 4.3, the difference to the unadjusted images is very hard to see by visual inspection. While this is a somewhat disappointing result, as the main purpose of increasing the image quality is to further the visual experience of the viewer, at lest the comparison metrics (table 4.5) show that there is a measurable improvement.

These metrics are however also only a comparison to our assumed “optimal” ray traced image. The metric itself is not a direct measure of quality, and is only used as a comparison between the individual methods.

Looking at the sampling that is going on, we can assume that the artifacts will get worse the higher the ratio of texture resolution to headset resolution becomes. This is because one pixel of the headset will map to more texture pixels, requiring stronger anisotropic filtering. Therefore, these artifacts may appear more strongly with lower resolution headsets or higher resolution photos, and our method could become more useful in that case. Additionally, techniques like foveated rendering that are used in modern VR headsets will only render high resolution information where it matters, but might still want the lower resolution sections to be free of flickering artifacts.

Bibliography

Abraham, S., & Förstner, W. (2005). Fish-eye-stereo calibration and epipolar rectification. *ISPRS Journal of Photogrammetry and Remote Sensing*, *59*(5), 278–288 (cit. on pp. 7, 52).

Blender Documentation Team. (2023). *Blender 3.6 manual*. Retrieved August 13, 2023, from https://docs.blender.org/manual/en/latest/editors/3dview/toolbar/add_uvsphere.html. (Cit. on p. 24)

Bourke, P. (2023). *Sphere2persp*. Retrieved August 15, 2023, from http://paulbourke.net/panorama/sphere2persp/. (Cit. on p. 20)

Chang, E., Kim, H. T., & Yoo, B. (2020). Virtual reality sickness: A review of causes and measurements. *International Journal of Human–Computer Interaction*, *36*(17), 1658–1682 (cit. on p. 6).

Fernandez Galaz, A. (2021). Alignment of stereo digital images (cit. on pp. 7, 15, 52).

Greene, N. (1986). Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, *6*(11), 21–29 (cit. on p. 18).

Greene, N., & Heckbert, P. S. (1986). Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications*, *6*(6), 21–27 (cit. on pp. 8, 27, 31).

Ho, T., & Budagavi, M. (2017). Dual-fisheye lens stitching for 360-degree imaging. *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2172–2176 (cit. on p. 15).

Horé, A., & Ziou, D. (2010). Image quality metrics: Psnr vs. ssim. *2010 20th International Conference on Pattern Recognition*, 2366–2369 (cit. on p. 37).

Hosseini, M., & Swaminathan, V. (2016). Adaptive 360 vr video streaming: Divide and conquer. *2016 IEEE International Symposium on Multimedia (ISM)*, 107–110 (cit. on pp. 3, 8).

Hussain, I., Kwon, O.-J., & Choi, S. (2019). An evaluation of three representative 360-degree image projection formats in terms of jpeg2000 coding efficiency. *Proceedings of the 2019 4th International Conference on Intelligent Information Technology*, 26–31 (cit. on p. 37).

Hwang, A. D., & Peli, E. (2014). Instability of the perceived world while watching 3d stereoscopic imagery: A likely source of motion sickness symptoms [PMID: 26034562]. *i-Perception*, *5*(6), 515–535 (cit. on p. 6).

Igehy, H. (1999). Tracing ray differentials. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 179–186 (cit. on p. 31).

Jin, E. W., Miller, M. E., & Bolin, M. R. (2006). Tolerance of misalignment in stereoscopic systems. *Proc. ICIS*, 370–373 (cit. on p. 6).

Jonathan Blow. (2001). *Mipmapping, part 1*. Retrieved September 9, 2023, from http://number-none.com/product/Mipmapping,%20Part%201/index.html. (Cit. on p. 7)

Landau, H. (1967). Sampling, data transmission, and the nyquist rate. *Proceedings of the IEEE*, *55*(10), 1701–1706 (cit. on p. 21).

Lo, I.-C., Shih, K.-T., & Chen, H. H. (2018). Image stitching for dual fisheye cameras. *2018 25th IEEE International Conference on Image Processing (ICIP)*, 3164–3168 (cit. on p. 15).

Microsoft. (2020). *Programming guide for dds*. Retrieved September 9, 2023, from https://learn.microsoft.com/en-us/windows/win32/direct3ddds/dx-graphics-dds-pguide. (Cit. on p. 7)

Ohashi, A., Yamano, F., Masuyama, G., Umeda, K., Fukuda, D., Irie, K., Kaneko, S., Murayama, J., & Uchida, Y. (2017). Stereo rectification for equirectangular images. *2017 IEEE/SICE International Symposium on System Integration (SII)*, 535–540 (cit. on pp. 7, 15, 52).

Papadimitriou, D., & Dennis, T. (1996). Epipolar line estimation and rectification for stereo image pairs. *IEEE Transactions on Image Processing*, *5*(4), 672–676 (cit. on p. 6).

Pharr, M., Jakob, W., & Humphreys, G. (2016). Physically based rendering: From theory to implementation. Morgan Kaufmann Publishers Inc. (Cit. on p. 22).

Pintore, G., Ganovelli, F., Pintus, R., Scopigno, R., & Gobbetti, E. (2018). Recovering 3d indoor floor plans by exploiting low-cost spherical photography. *PG (Short Papers and Posters)*, 45–48 (cit. on p. 1).

Ribas Costa, V. A., Durand, M., Robson, T. M., Porcar-Castell, A., Korpela, I., & Atherton, J. (2022). Uncrewed aircraft system spherical photography for the

vertical characterization of canopy structural traits. *New Phytologist*, *234*(2), 735–747 (cit. on p. 1).

Schwalbe, E. (2005). Geometric modelling and calibration of fisheye lens camera systems. *Proc. 2nd Panoramic Photogrammetry Workshop, Int. Archives of Photogrammetry and Remote Sensing*, *36*(Part 5), W8 (cit. on p. 14).

Snyder, J. P. (1987). *Map projections–a working manual* (Vol. 1395). US Government Printing Office. (Cit. on pp. 9, 17, 28).

Teo, P., & Heeger, D. (1994). Perceptual image distortion. *Proceedings of 1st International Conference on Image Processing*, *2*, 982–986 vol.2 (cit. on p. 37).

The Khronos Group. (2023). *Openxr-sdk-source*. Retrieved August 7, 2023, from https://github.com/KhronosGroup/OpenXR-SDK-Source/tree/main/src/tests/hello_xr. (Cit. on pp. 2, 33)

Tran, H. T. T., Ngoc, N. P., Bui, C. M., Pham, M. H., & Thang, T. C. (2017). An evaluation of quality metrics for 360 videos. *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, 7–11 (cit. on p. 37).

Walmsley, A. P., & Kersten, T. P. (2020). The imperial cathedral in königslutter (germany) as an immersive experience in virtual reality with integrated 360° panoramic photography. *Applied Sciences*, *10*(4) (cit. on pp. 2, 5).

Wan, L., Wong, T.-T., & Leung, C.-S. (2007). Isocube: Exploiting the cubemap hardware. *IEEE Transactions on Visualization and Computer Graphics*, *13*(4), 720–731 (cit. on p. 19).

Wang, Z., & Bovik, A. (2002). A universal image quality index. *IEEE Signal Processing Letters*, *9*(3), 81–84 (cit. on p. 37).

Wang, Z., Bovik, A., Sheikh, H., & Simoncelli, E. (2004). Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, *13*(4), 600–612 (cit. on p. 37).

Wikipedia contributors. (2023). *Mipmap aliasing comparison*. https://commons.wikimedia.org/wiki/File:Mipmap_Aliasing_Comparison.png. (Cit. on p. 8)

Williams, L. (1998). Pyramidal parametrics. In *Seminal graphics: Pioneering efforts that shaped the field* (pp. 65–75). Association for Computing Machinery. (Cit. on pp. 3, 7, 31).

Ying, X., Hu, Z., & Zha, H. (2006). Fisheye lenses calibration using straight-line spherical perspective projection constraint. In P. J. Narayanan, S. K. Nayar,

& H.-Y. Shum (Eds.), *Computer vision – accv 2006* (pp. 61–70). Springer Berlin Heidelberg. (Cit. on p. 14).

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Würzburg, September 12, 2023

Till Wübbers

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Versicherung zur Leistungserbringung

Titel der Abschlussarbeit:

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):

Eingereicht durch (Vorname, Nachname, Matrikel):

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich keiner anderer als der in den beigefügten Verzeichnissen angegebenen Hilfsmittel bedient habe. Alle Textstellen, die wörtlich oder sinngemäß aus Veröffentlichungen Dritter entnommen wurden, sind als solche kenntlich gemacht. Alle Quellen, die dem World Wide Web entnommen oder in einer digitalen Form verwendet wurden, sind der Arbeit beigefügt.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Ort, Datum, Unterschrift