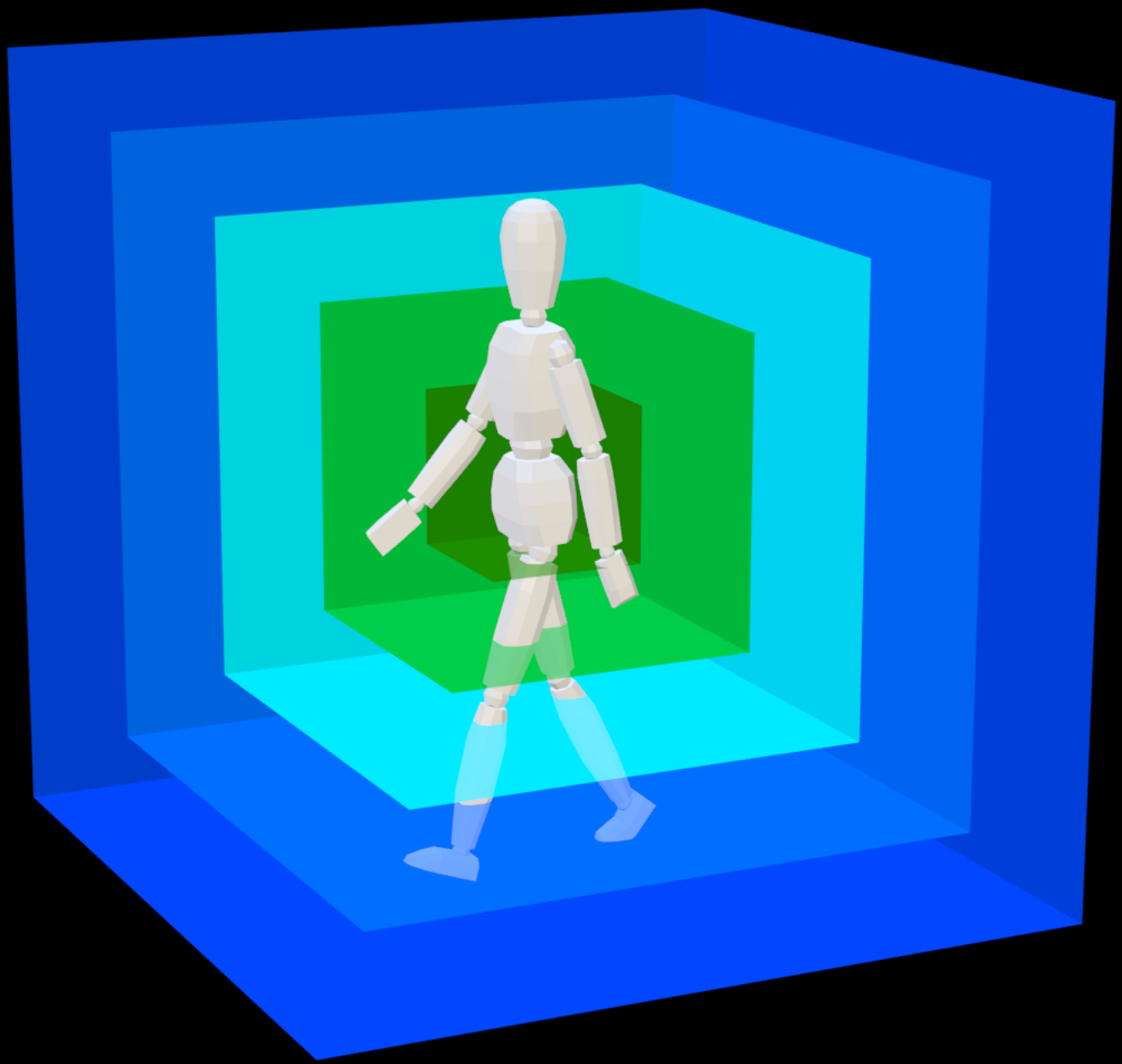
Cubemap-based Ambient Occlusion for Character Animation

Mirco Kroon



Delft University of Technology

TU Delft

Cubemap-based Ambient Occlusion for Character Animation

by

Mirco Kroon

Student Name	Student Number
Mirco Kroon	4577779

Supervisor:	Prof. Dr. Elmar Eisemann
2nd committee member:	Dr. Megha Khosla
3rd committee member:	Dr. Ricardo Marroquim
Defence date:	2023 - 06 - 16
Master programme:	MSc Computer Science (Software Technology track)
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, TU Delft

TU Delft

Preface

This report is a Master's Thesis written under the Computer Graphics and Visualization research group at the Computer Science faculty of Delft University of Technology. The original context of this project was to provide improvements to rendering techniques for sports visualisations, although the method presented is applicable more broadly. The project presents a new method of storing lighting data for animated characters in real-time applications, such as 3D games. Our method attempts to provide better performance and fewer trade-offs than existing real-time methods by pre-computing lighting data and storing it in commonly used data structures, combined with various encoding and compression techniques.

I would like to thank my supervisor Prof. Dr. Elmar Eisemann as well as Dr. Leonardo Scandolo for supervising the project and giving plenty of useful ideas and suggestions. I would also like to thank Heli Pajari and Bhavesh Nande for providing valuable feedback on the thesis report.

Mirco Kroon
Delft, May 2023

Contents

1

Introduction

Simulating accurate lighting is one of the most important parts of 3D rendering. This includes both direct illumination, where a light source is in direct line of sight of a surface, as well as indirect illumination, which is where light bounces off of one surface onto another. While direct illumination can be computed quite easily in real-time, indirect illumination essentially turns every part of the scene into a light source, and this cannot be simulated accurately in real-time. Because relying only on direct illumination would leave objects in shadow fully darkened, real-time light rendering techniques add an ambient light component that brightens all objects in the scene by the same amount, simulating primitive indirect lighting. However, indirect lighting should be blocked by nearby geometry as there are fewer directions for the light to reach the surface from, which the ambient component does not account for. This makes it difficult to identify the layout of a scene as this blocked lighting typically provides important cues for people to see depth and proximity between objects, as well as revealing the presence of geometry not directly visible in the image by allowing it to still block some ambient lighting. A few different methods exist to approximate this effect, the most common of which is called ambient occlusion.

Ambient occlusion works by darkening surfaces when there is geometry nearby that should block part of this lighting. While some of the darkening can be calculated ahead of time for static geometry such as walls, this cannot be done as easily for dynamic parts of the scene. As many scenes rely on characters that move around in and interact with the environment, this leaves important cues without an effective pre-computed method. Real-time methods exist to solve this problem, but for these to run efficiently many shortcuts are typically taken that may affect the final image. One example of such a shortcut is to use screenspace information, where only geometry that is directly visible is incorporated into the occlusion computation. This causes issues not only for characters that are just outside the boundaries of the screen but also when parts of the character are hidden behind another object. Examples of these scenarios are a character that is walking in or out of frame or an arm of the character hidden behind their torso. As such, screenspace methods can have several downsides.

We will research whether it is possible to rely on pre-computed compact data structures to encode ambient occlusion for animated characters. For this we will develop a method that aims to circumvent the issues present in screenspace methods by pre-calculating the occlusion each character casts on the environment around it at each stage in their animation cycles. This information will then be stored in a spherical encoding format that lets us specify the occlusion for each point in space around the character while keeping the memory usage independent of the size or complexity of the character model. To store this information we will make use of compression methods that can be efficiently queried at runtime. We will present two different methods that rely on different encoding methods, one based on polynomial curve fitting and one based on cosine basis functions using Blurhash[2]. The goal is to produce images that provide sufficient lighting cues to tell whether a character is in close proximity to static geometry in the scene by resembling the appearance of true indirect illumination. As our result is an approximation of realistic lighting, there will naturally be trade-offs between different aspects of the result. For example, one part of the image may be a bit darker than it should be, but brightening this will affect other parts of the image as well. Artists may prefer to manually tweak the result to adjust which of these trade-offs look more appealing for a specific scene, character, or context, for which we will provide parameters that can be adjusted either at runtime or pre-computation time. These parameters can also be tweaked

to achieve a particular art style for the image instead of a realistic result. Our method should minimize the effect on runtime performance and runtime memory usage, ideally performing similarly to or better than screenspace methods.

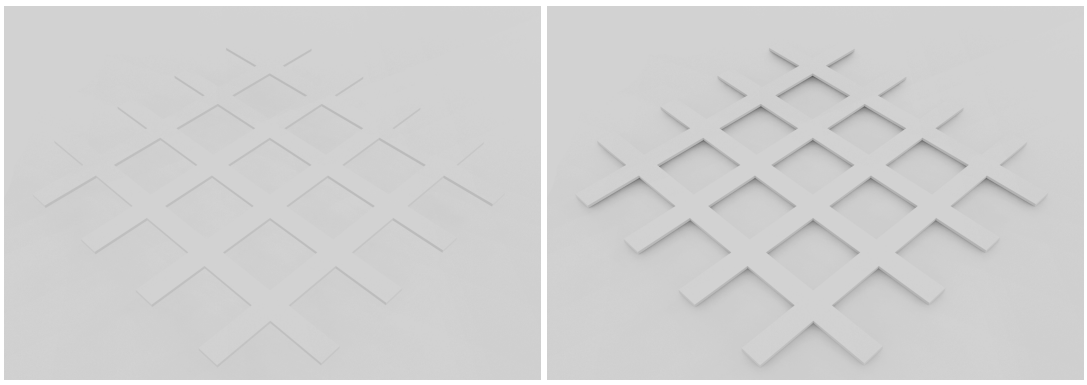
Possible use cases for the proposed technique include real-time rendering applications, such as games and mixed reality applications. Since the data is stored per animation, an ideal scenario is one where many similar characters are present in the scene, such as in a sports visualisation, where all players on the field share mostly the same animations and therefore lighting data.

2

Prior work

2.1. Ambient Occlusion

Ambient occlusion is the darkening caused by geometry blocking environmental illumination, which can help give cues about proximity and shape of objects in the scene, such as shown in Figure 2.1. The blocking of indirect illumination is difficult to compute correctly in real-time, hence faster methods are used to approximate it. One of the earliest implementations of real-time ambient occlusion was simply called Screen-Space Ambient Occlusion[14], which uses the depth map to compare nearby pixels. The idea behind this method forms the basis for most screenspace occlusion methods still in use today. As the method operates only on what is visible on screen, occlusion will often appear too bright around the edges of the screen. As there is no information on geometry outside of the camera view, the assumption is made that there is no geometry to block ambient lighting present there. Similarly, geometry that is behind other geometry from the perspective of the camera cannot be accounted for, leading to further artifacts of areas appearing brighter than they should.



(a) Ambient occlusion disabled **(b)** Ambient occlusion enabled

Figure 2.1: Basic scene showing the effects of ambient occlusion

The formulation used to compute ambient occlusion, similar to the one presented by Bavoil et al. [1], is presented in Equation 2.1.

$$ao(P) = 1 - \frac{1}{2\pi} \int \vec{N} \cdot \vec{D} \cdot V(\vec{D}) \cdot W(\vec{D}) d\vec{D} \tag{2.1}$$

Where:

- $\vec{N}$ Normal vector at point P
- $\vec{D}$ Vector from P to point of intersection with an occluding point
- V visibility function, returning 1 if an occluding point is found, otherwise 0
- W falloff function that reduces the occluding effect if an occluding point is farther away

Other screenspace methods exist that improve upon this idea. A newer method called AlchemyAO [13] uses an improved formulation over the earlier methods to reduce occlusion present on flat surfaces. As this method more closely resembles reference implementations of pre-calculated ambient occlusion, we will use this method as a reference for screenspace methods in general later on.

Real-time ambient occlusion can be done without relying on the depth buffer, which avoids the drawbacks of screenspace methods. This can be done by storing pre-computed ambient occlusion data at the vertices of a mesh or in a volume around it. Several such implementations exist but they have various limitations. One method for storing occlusion in a 3D grid is presented by Malmer et al. [12]. They use a similar concept to our method, although it does not support animation and has a lower shadow resolution. Using a full voxel grid also requires a larger amount of memory than our method.

To handle character animation Kontkanen et al. [9] uses a technique of per-vertex coefficients for self-occlusion of the character. While storing the occlusion per-vertex can work well for some cases, it becomes more expensive with increasing vertex counts in modern meshes and would require additional techniques to remain space efficient. One such technique is presented by Kirk et al. [8] which involves using clustering to group together vertices and reduce the cost of storing data for each vertex individually. Our technique works independently of the amount of detail in the mesh which avoids this problem entirely. It is also difficult to use per-vertex methods to cast occlusion on the surrounding environment since it would require finding the closest vertices to a point in space at runtime. As such these methods are more applicable to a character casting occlusion on itself than a character casting occlusion on the environment as our method does.

Newer methods exist to compute real-time ambient occlusion using raytracing [5]. Raytraced methods come with a significant performance cost and typically require specific hardware to run at acceptable performance levels, making it unsuitable for many environments. Similarly, voxel grids spanning the entire scene [4] can be used for ambient occlusion. This also comes with a significant performance cost, and would still need further adjustments to allow for animation.

2.2. Spherical encoding

To encode different values for each direction, we will need to use a spherical encoding system. Several different options exist for spherical encoding. Commonly used options for computer graphics are spherical harmonics and cubemaps. Spherical harmonics use coefficients to encode the surface of a sphere, where the number of coefficients grows the level of detail required. Spherical harmonics can be used to encode ambient occlusion. Typically they are used for the effect of occlusion from static objects on dynamic objects, such as from the walls of a scene onto a character model. Increasing the level of detail impacts the performance as more coefficient terms are included that need to be evaluated at runtime. This makes them unsuitable for higher resolution data, such as the occlusion caused by a complex character model.

Cubemaps can be used to encode a sphere by storing pixel values for each direction. Whereas the memory usage from cubemaps will increase as the resolution is increased, the time required to retrieve specific directional values remains constant, which is not the case for spherical harmonics. As such, when a high level of detail is required, cubemaps may be the preferred option.

Cubemaps are created by having a face for every side of a cube, as shown in Figure 2.2a. Each of these faces is a regular 2D texture. In our method we will place these cubemaps around the character as in Figure 2.2b. We will sometimes refer to the pixels in the textures of the cubemap as ”texels” to differentiate them from the pixels rendered to the screen. When working with cubemaps, reading from them is done by giving two coordinates indicating an angle in 3D space instead of referring to specific faces and texel coordinates. Since in our implementation we will also make use of the distance from the center of the cubemap, we will talk about cubemaps using a spherical coordinate system[20], consisting of the *radial distance* and two angles. We will refer to the two angles as the *direction*, and the radial distance as d or *distance*. Since the encoding method is spherical, we will also portray cubemaps as spheroids since this is a more accurate way to visualise our method.

(a) Cubemap faces unfolded **(b)** Cubemap surrounding a character

Figure 2.2: Examples of how cubemaps may be used

2.3. Compression methods

For compression of the occlusion data there are several papers that deal with a similar issue. Ritschel et al. [17] presents a technique of compressing depth maps by grouping together similar texures, typically those computed from directions close together to speed up visibility queries. For our method, we will have animation frames with relatively small differences between them, so it seems a similar technique could be used to compress our data as well. However, we expect this would create large jumps when switching between different grouped textures, making the animation no longer look smooth. Therefore we do not believe this compression scheme is directly applicable to our method.

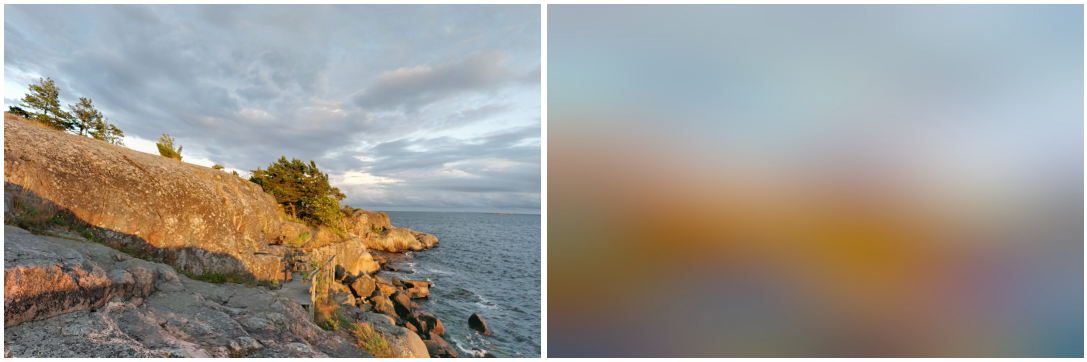
Other methods for compression more similar to our chosen method use basis functions to compress data where high-frequency detail is not too important. Jansen et al. [7] uses such a technique to compress opacity maps using Fourier transforms. Lokovic et al. [11] presents a technique to allow partial shadows to be cast by transparent objects by encoding the depth dimension using a visibility function. Our method will be based on a similar idea as these methods of encoding lighting data using

function encoding techniques, although we opted to use more commonly available encoding methods in combination with novel cubemap encoding methods.

As we will be encoding two dimensions of data at the same time in our method, we looked into image compression algorithms that are geared towards high levels of compression rather than preserving image detail. In our implementation we decided to use Blurhash[2], an image encoding method developed by Wolt. The main purpose is to create a very small representation of an image that can be used as a placeholder until the real image has loaded in. Blurhash was chosen due to its relative simplicity and extensive open-source implementations available, allowing us to rapidly incorporate it into our method.

Blurhash works by using a discrete cosine transform with a compactly packed encoding. Since it encodes images in just a few bytes and has parameters that can be tweaked to adjust the number of output bytes, this is an easy way for us to encode 2D data to fit within a few bytes.

Figure 2.3 shows an example of the blurhash algorithm applied to an image.



(a) Original image **(b)** Computed blurhash (6x6 components)

Figure 2.3: Example of the blurhash result for a given image

3
Method

3.1. Concept

The goal of our method is to store the ambient occlusion at each point in the near vicinity of our character. This way, when world geometry is near our character we can quickly query the occlusion at that point in space with ideally minimal runtime cost. We also want to ensure our method supports character animation.

The easiest way to accomplish this is to construct a 3D grid around the character. This method is already used in some early implementations of stored ambient occlusion [12]. The issue with a grid is that the space requirement is n^3 for a chosen grid size. To reduce memory usage, alternative methods involve storing the occlusion per vertex in the model. However this means that increasing the vertex count of the model also increases the space required to store the lighting data[8]. This method also does not let us easily cast occlusion on nearby geometry.

Using cubemaps to store this information ensures that the occlusion remains independent from the model geometry, while keeping the space required to store it n^2 for a chosen cubemap resolution. The simplest solution would be to store the occlusion computed for each direction from the character in the texels of the cubemap, however, the occlusion changes depending on how close to the character a point is, for which we use the radial distance. Lastly we must also encode how the occlusion changes as we animate the model.

We will present two different methods to encode this information in the cubemaps. We will first go over some common steps for both methods, including how the cubemaps are positioned and how we compute the ambient occlusion. We will then go into more detail for both of our encoding methods and how they store the information.

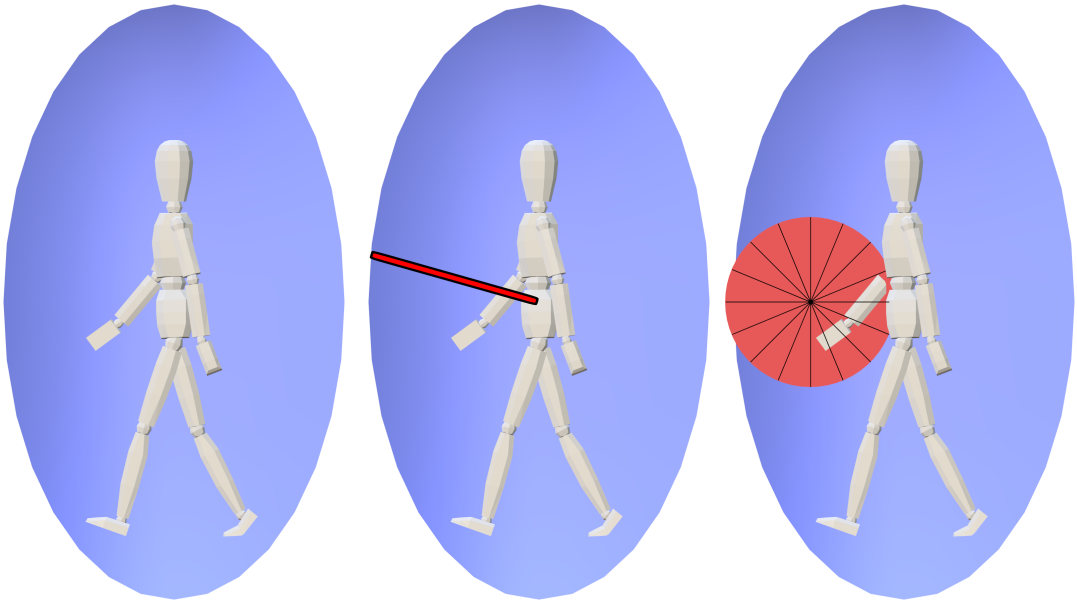
3.2. Generation

3.2.1. Cubemap positioning

The first step is to find the position and dimensions needed for our cubemap to fit nicely around our model. Many character models will have different dimensions over different axes, for example a human model will likely be more tall than wide so having a sphere would not as efficient as a spheroid adjusted to the size of the model. We first determine the cubemap center point by finding the center of the bounding box of the model. Assuming linear interpolation between keyframes, we need to iterate over all the keyframes in the model's animations. We take the minimum and maximum values encountered for all vertices. To find the radius, we take the distance to the center we found for each of the vertices. We find the maximum radius for each axis individually which lets us scale the bounding sphere to account for models that are elongated. Figure 3.1a shows how one such cubemap might look for a human character model. Note that the cubemap in this example extends well above the head of the character model, for example due to the character raising its hands during the animation cycle.

Note that the cubemap is large enough to include all character geometry at every stage of the animation.

Our cubemaps are made up of texels that each correspond to a particular 'direction' originating from the center point of our character mesh. For every direction, the number of which will depend on the resolution of our cubemaps, we will calculate and store occlusion values. The red line in Figure 3.1b shows for which points in space the occlusion values will be encoded in a particular texel.



(a) Cubemap surrounding character model **(b)** Sampling direction of a texel **(c)** Sampling rays for AO calculation

Figure 3.1: Diagrams of how cubemaps and character models are positioned.

3.2.2. Occlusion computation

For both of our methods we will have to calculate the occlusion at specific points in space relative to the character. We will call a point for which we are computing the occlusion point P. To calculate the occlusion we use uniform sphere sampling originating in P to search for character geometry inside the area through raycasting. Figure 3.1c shows how the sample rays might be distributed, although in practice this will be in three dimensions instead of two.

We compute the occlusion by applying formula 3.1, where a value of 1.0 means the point is fully occluded and a value of 0.0 means the point is in full brightness. When P is inside the mesh, we return a value of 1.0 as it is occluded from all light.

$$ao(P) = \sum^{S} \begin{cases} max(0, w \cdot N \cdot D \cdot (1 - len(D)/D_{max})^{\alpha} & \text{if collision found} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Where:

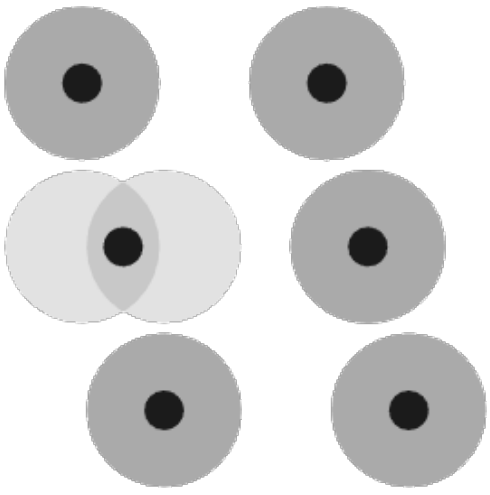
- S Number of sample points
- N Normal vector at point P
- D Vector from P to point of collision
- D_{max} Maximum sampling distance, refers to the size of the red circle in Figure 3.1c
- α Coefficient to control distance falloff
- w Coefficient to control strength of occlusion

The formula iterates over the number of sample points and casts a ray from point P into the scene for each. The formula uses the vector to the collision point as well as the normal vector of the original points to calculate the occlusion, the result of which will depend on the given coefficients. The final value is clamped between 0 and 1, as using a large weight may otherwise give values with a negative brightness. This formula is based on the occlusion formula used by Alchemy AO[13], although adapted for our use case.

3.2.3. Encoding methods

Our occlusion values will change depending on the direction, radial distance, and the time. Since the direction consists of two components this gives us a 4D space which we want to encode into cubemaps. Our cubemaps already take care of two of these by encoding the direction, however, this means that for each texel in the cubemap we need to encode the distance and time dimensions.

Having the time dimension in discrete steps creates a very unconvincing result when in motion. It looks like the occlusion is simply fading out in the old location and fading back in the new location, rather than moving with the animation. Figure 3.2 shows the occlusion as a grey circle as the black point moves across space. The ”fade-out-fade-in” effect in Figure 3.2a is what we want to avoid. As such, when finding a suitable method to encode this the time dimension should ideally not be separated into discrete steps.



(a) Discrete time steps **(b)** Continuous time

Figure 3.2: Comparison of how the occlusion follows a geometry point. As the black point moves from left to right, the grey circle shows how the occlusion follows.

Graphics cards allow us to use textures with up to four colour channels (red, green, blue and alpha), with each channel allowing up to 32 bits, or four byes, of data. This gives us a total of 16 bytes for each texel. This is still a very small amount of space, so we will have to come up with some simplifications to make storing our occlusion data feasible.

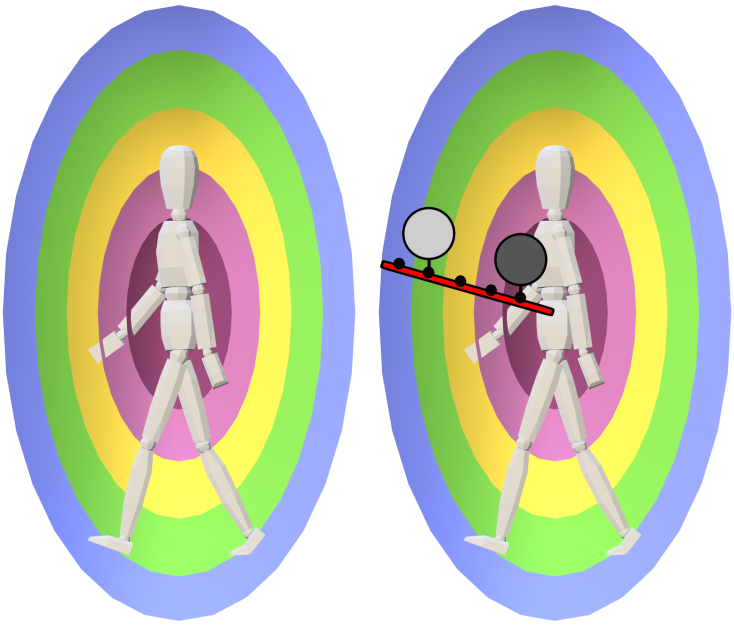
Our first method involves using a few different cubemaps nested inside each other to encode the radial distance information. This allows the full 16 bytes to be used to encode the time dimension. Since our distance is now in discrete steps, each corresponding to one cubemap, we will call this method ”discrete radial distance”. Our second method encodes both the distance and time dimension at once, therefore not creating discrete steps in the distance dimension. As such we will call this method ”continuous radial distance”.

In the next sections we will go over the steps to encode and render the result using both of these methods.

3.3. Discrete radial distance

3.3.1. Discrete radial distance - Ambient Occlusion computation

The discrete method relies on nesting a few different cubemaps with different scales, and computing the occlusion at each discrete distance separately. This means that for each direction around our character we will take multiple occlusion samples spaced out going from the outermost cubemaps to the center of the character. Figure 3.3a shows how these nested cubes might be laid out. The number of cubemaps used can be adjusted depending on the requirements of a specific character or scene. As we refer to the distance to the character as the radial distance d, the number of cubemaps used in the discrete method we will call the d-resolution, or S_d for the number of samples taken over the d domain. Figure 3.3 shows d-resolution of 5, in actual scenes we typically used a slightly higher value of 8 as this gave a good balance between space efficiency and rendering quality. The number of cubemaps needed for good looking results will vary between characters, for example, a mesh with a lot of straight edges will produce more noticeable artifacts due to the spherical encoding of the cubemaps, so using a higher resolution may be needed in such a case. Meshes with more natural curves, such as humans or other living characters, may be able to get away with a lower number as it is much harder to notice when the occlusion does not match exactly.

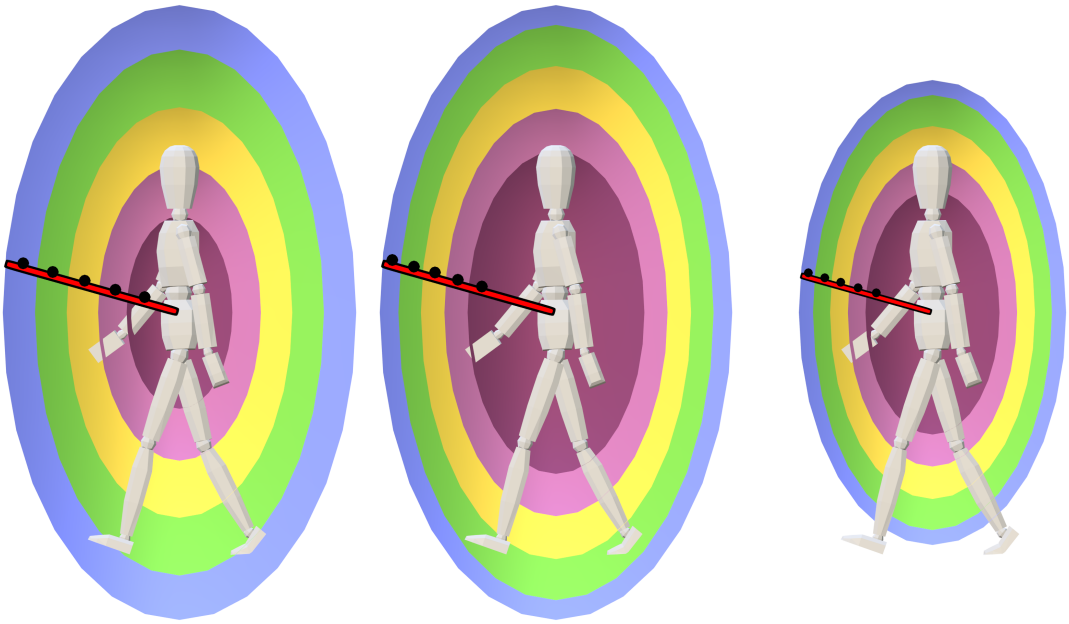


(a) Layout of nested cubemaps **(b)** Example occlusion points

Figure 3.3: Diagrams of how nested cubemaps are positioned around a character

It often happens that we might not need a lot of resolution close to the center of the mesh, especially if this would end up largely on the inside of the mesh. We want to avoid having the inner cubemaps go to waste. To avoid this, there are a few parameters we can tweak to adjust the spacing of these cubemaps to work better for specific models or uses cases. We can use an inner scaling to place the cubemaps closer together, and further from the center of the mesh. Figure 3.4b shows how the inner scaling moves the sample points away from the center mesh compared to the reference scaling in Figure 3.4a.

We can also scale the cubemaps themselves, called the outer scaling, this increases or reduces the space around the borders of the object. To avoid having geometry touching the border of the cubemaps, typically this value will be slightly above 1.0 to give a margin around the edges. Figure 3.4c shows the effect of lowering the outer scaling while keeping the inner scaling the same as Figure 3.4b, as shown, the feet of the model end up outside the bounding area, which means no occlusion would be rendered for these parts of the model.



(a) Reference scaling used previously **(b)** Adjustment to inner scaling **(c)** Adjustment to outer scaling

Figure 3.4: Example of how the cubemaps positioning and sample points change as we adjust the scaling parameters

Lastly, we need to account for the animation of the character. As the character moves around, we will see the occlusion changing over time as geometry of the character moves closer or further from our sample point. For this, we will use a number of sample points over the time domain, which we will call S_t. In the next section we will go over how this and other parameters affect the storage and compression of this data. To ensure we have sufficient resolution over the time domain to make our occlusion look smooth, we need to use a relatively high number of samples for this. We typically use a value of 255 for this so that we can store the indices in a byte without any loss of precision due to rounding.

3.3.2. Discrete radial distance - Storage & Compression

We will first calculate the space needed to store this data without any further compression. Recall we have our parameters S_d and S_t for the samples over the distance and time domains, as well as the resolution of the cubemap which we will call $S_x \cdot S_y$. Since our cubemaps have six faces, this gives us Equation 3.2. If we assume all values are stored in bytes, using parameters that provide a decent looking result requires 200 megabytes of space.

$$N = 6 \cdot S_d \cdot S_t \cdot S_x \cdot S_y \tag{3.2}$$
$$N = 6 \cdot 8 \cdot 255 \cdot 128 \cdot 128 \tag{3.3}$$
$$N \approx 200 \text{ megabytes} \tag{3.4}$$

As 200 megabytes of space for a single animation is quite a lot, we need to find a way to store this data more efficiently. Recall we want to encode this within the only 16 bytes available per texel when using one cubemap. A naïve solution would be to simply take 16 ”keyframes” across our time domain and storing each of these individually. While this method produces an acceptable result when the animation aligns with one of these keyframes, it does not look convincing in motion due to the fading effect described in section 3.2.3. The solution that we have come up with for this problem is to use polynomial curve fitting.

3.3.3. Discrete radial distance - Curve fitting

Before we can implement our curve fitting solution we first need to consider what a sample of our data looks like. In Figure 3.5 you can see an example of occlusion data for two different directions of an animation. Especially in Figure 3.5b there are some large discontinuities in the data, which makes it difficult to encode in a single curve.

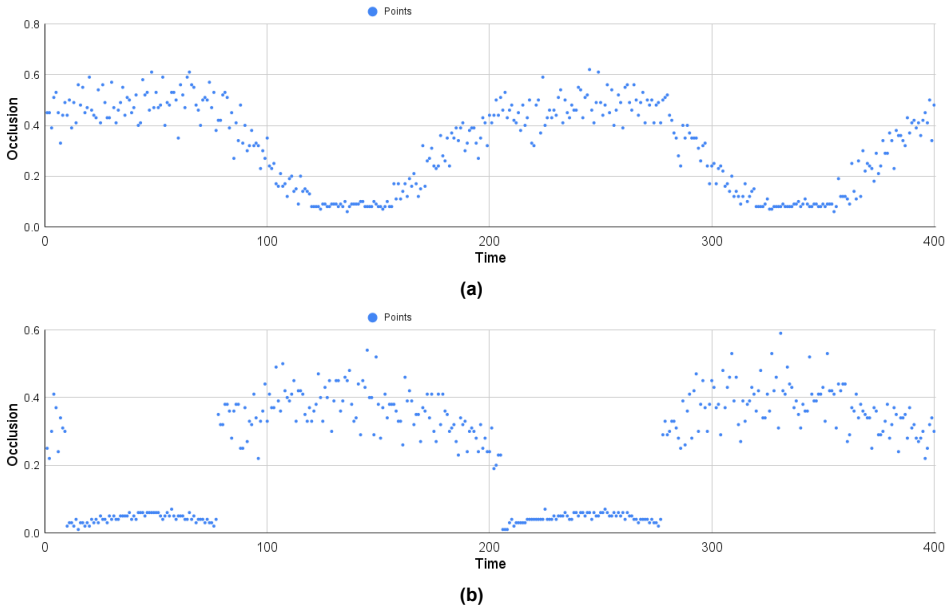


Figure 3.5: Examples of occlusion values over an animation cycle

As such our plan is to split the dataset into multiple different curves, each of them corresponding to one of the sections in the chart. For the animations we looked at, we found that splitting the data into four sections should produce a good result. This also works well with our target space of 16 bytes, as it divides nicely to give us four available bytes per curve section. If we are splitting the dataset we will also need to encode the starting position of each curve, leaving us with three bytes per curve. As we previously chose a number of samples over the time domain of 255, we can easily encode the starting point into a single byte without any loss of precision.

Since we are planning to encode the coefficients in very limited number of bits, we will not have a large amount of precision available to work with. Repeated multiplication will lead to compounding error values if the coefficient is encoded with limited accuracy. Higher degree polynomials lead to more multiplications for some of the coefficients. As such it would be wise to limit the degree of the polynomial to avoid having multiplications with one coefficient happening too many times in a row. We settled on second degree polynomials, which require three coefficients, meaning we can fit exactly one coefficient into each byte.

The four curve segments will be separated at specific points in the graph, found by looking for the biggest jumps or discontinuities in the graph. The algorithm for this works by going over every point in the dataset and averaging a window of a few datapoints to either side to compute the difference between them. We then find the four largest jumps in the data to use as the breakpoints for our curves. To preserve efficient usage of our curves and ensure we have enough datapoints to perform curve fitting on each section, we do not want sections of just a couple of datapoints. As such we require a minimum spacing between breakpoints to ensure that each curve segment will be of some minimum length.

We perform curve fitting on each of these individual segments. Our model function reflects a degree two polynomial, with constraints set to limit the coeffient domain to $[-4, 4]$ to ensure a minimum amount of precision during the encoding. Encoding curves further away from the origin often leads to larger coefficients being found, which would mean less accurate results for curve sections later on in the animation. To avoid loss of accuracy from this effect we move the datapoints so that they start at the origin no matter the actual location of the segment.

If we use this method to compute curves for the datapoints shown in Figure 3.5 we get Figure 3.6. One issue that's present in chart 3.6b is that the very start of the graph should actually be part of the segment at the end, as our animation is cyclic. The method used for these curves does not allow a segment to "wrap around" leading to the first curve not being fit very effectively. To solve this, our implementation finds not three but the four largest discontinuities in the graph, thus the last segment wraps around and produces a fit slightly better than shown in Figure 3.6.

The chosen charts represent the most complex sections of this particular animation. As every animation has different looking curves, some animations may require encoding in a larger number of sections, which we did not include within the scope of our implementation. In chapter 5 we go over how our algorithm performs with some other animations.

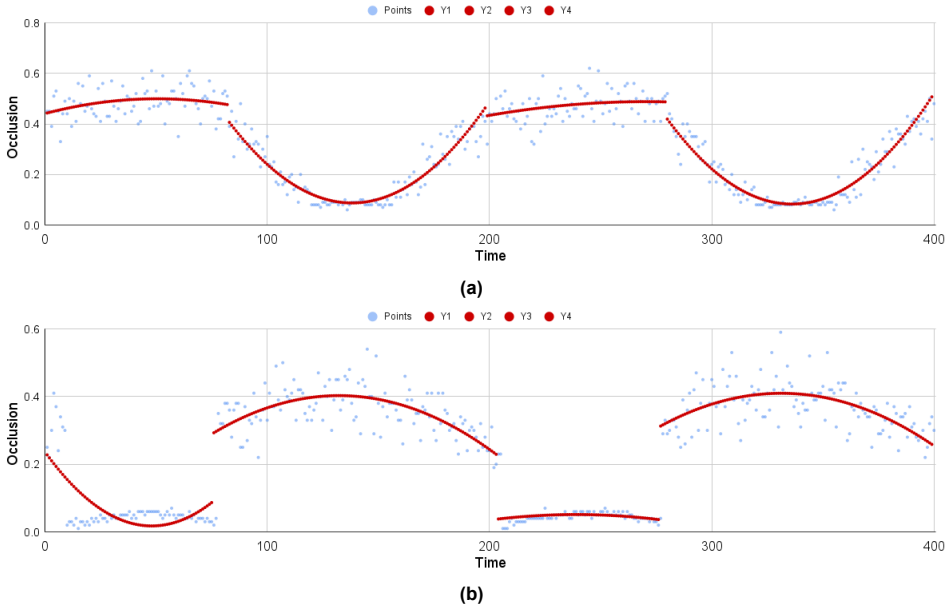


Figure 3.6: Examples of curves fit to occlusion values over an animation cycle

3.3.4. Discrete radial distance - Encoding

Since we want to encode our curves within a texel of our cubemap, we need to fit them into a space of 16 bytes. We have four curves, each of which has three coefficients and a starting offset. We can easily encode the offset into a byte as the number of time samples we are using is 255. For our coefficients we simply store these as a value indicating their position in the domain of $[-4, 4]$. This means that we have an accuracy of $8/255 \approx 0.0314$ for our coefficients.

3.3.5. Discrete radial distance - Rendering

To render our ambient occlusion in the scene we pass our generated cubemaps to a shader, along with the radius parameters for each axis of our spheroid. We also include the position and rotation of the corresponding model and a time value t relating to the current point in our animation cycle. To sample our cubemap we first need a direction vector, which we find by subtracting the character position from the sampling point. The magnitude of this vector represents our radial distance. We use tri-linear interpolation to smoothen the result. To interpolate between two cubemaps, we multiply our magnitude by the number of cubemaps and round the value to find the two adjacent cubemaps. If we are outside the range of the outermost cubemap we leave the occlusion at zero. For both of the cubemaps we then sample with the direction vector, and perform some bitwise operations to extract the encoded coefficients. We apply the functions to our t value to get the occlusion required, and interpolate between the two cubemaps we sampled. Lastly we multiply the result by our strength value which lets us increase the effect of occlusion at runtime.

3.4. Continuous radial distance

3.4.1. Continuous radial distance - Encoding

To further reduce the space required to store the result we will look into another method of compressing the occlusion information. Ideally, we want to reduce the number of cubemaps needed, which is difficult to do in the discrete method without causing more noticeable artefacts on the boundaries of the cubemaps, leading to a loss in visual quality. Therefore we will be encoding the distance dimension not as discrete steps but as a continuous function. Since we no longer store them as one cubemap per discrete distance step, We can use a much larger distance resolution.

For each direction, we want to store a 2D field of values that correspond to the time and distance dimensions. Figure 3.7 shows an example of such an input field. Now we want to find a method to effectively encode this field in as little space as possible. As the discontinuities still exist within this encoding method, we first considered using a method similar to before where the field is split into sections that are each encoded individually. Finding suitable points to split the field is difficult, as the jumps will not be at the same time value for every distance value, making a simple grid separation infeasible.

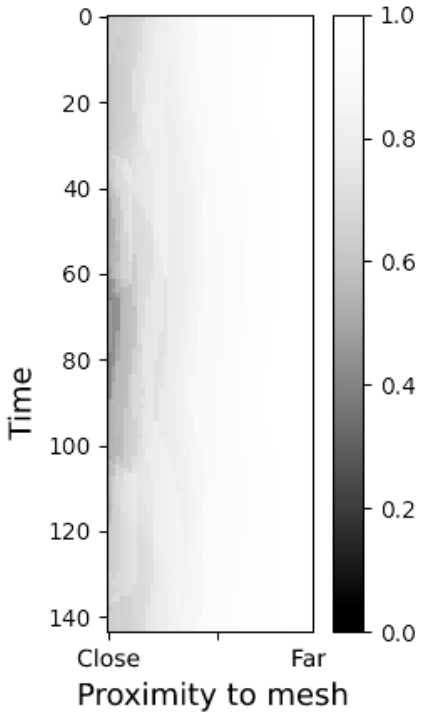
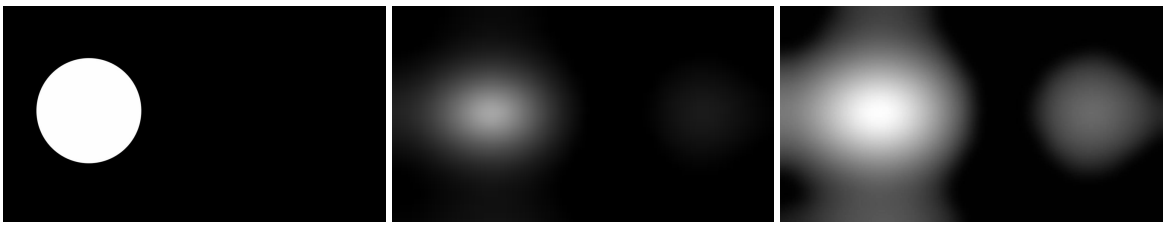


Figure 3.7: Example of the input data we want to encode

Due to the complexity of this problem we decided to look towards existing methods to encode 2D fields. Since we can treat our dataset effectively as an image, we can make use of the large number of options available to encode images. Image encoding algorithms are typically focused on preserving detail rather than achieving a particular output size. For our use case, we are looking for a consistent encoding size regardless of the input data. Small details are also not generally noticeable when rendering occlusion, so for our use case a highly lossy image compression algorithm would be ideal. Most lossy image compression algorithms work using basis functions, however methods such as JPEG compression split the image into blocks at fixed points rather than adjusting to the underlying data[6], so this may create additional artifacts. JPEG encoded images are also still too large for our use case. A method that is specifically designed to create highly compacted image representations using basis functions is Blurhash[2], which uses a discrete cosine transform combined with a compact encoding of the parameters.

Unfortunately this method alone is not sufficient to create a satisfactory result as the method can create some dark areas where they should not be, resulting in noticeable artifacts during rendering. Figure 3.8 shows an example of how such an error might appear. Many of the basis functions used will have a period smaller than the size of the image, combined with the limited number of available basis functions this means that encoding one bright spot in one area of the image may have unintended effects on the other side.



(a) Input for Blurhash algorithm **(b)** Resulting encoded image with true colours **(c)** Result image with exaggerated colours to highlight the error

Figure 3.8: Example of error patterns in blurhash method.

Fortunately, there are predictable error patterns for our case that we can correct. For our inputs to the algorithm we will often find dark areas on the side of the image that corresponds to a low distance to the model, and less dark areas far away from the model. To improve the result further we compute the difference between the intended results, and the result acquired from evaluating the blurhash encoded image. We then average all the errors over the distance axis of the residual data, and fit a four degree polynomial to this residual. After subtracting this polynomial we do the same with the time axis. The end result is that we want to store our blurhash encoding, as well as two polynomials to correct for the approximation.

To fit these polymials we first calculate the residuals by subtracting the computed blurhash results from the input. During this step we can choose to apply weights to the data to influence what sort of errors are mitigated the most. Since it is much more noticable for our method to have areas that show up as too dark rather than too bright, we give error values that are too bright a weight of only 60% compared to dark areas. As it is quite subjective where the balance is between having fewer excess dark spots compared to excess bright spots, this value is a parameter that can be tweaked by artists based on the desired output.

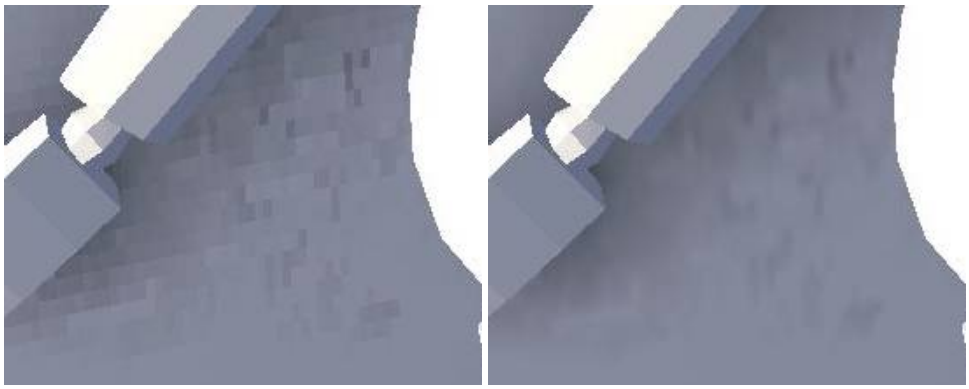
The number of bytes required by the blurhash encoding depends on the chosen parameters, which are the number of basis functions along each axis. For a typical encoding using three to four on each axis is sufficient. We decided to go for a slightly higher amount to allow for more resolution in our result. We need 8 bytes of data to store two polynomials, and using blurhash parameters of 5x5 results in a 54 byte encoding. This adds up to 62 bytes, which fits nicely within four cubemaps with a little bit of space left.

3.4.2. Continuous radial distance - Rendering

The rendering for using the blurhash encoding method is fairly straightforward, as we just extract the blurhash encoding from our cubemaps and use it to compute the value for our given time and distance position. We also compute the polynomial results and subtract these from the blurhash value. In some cases, the errors produced by the blurhash algorithm may lead to artifacts that are more noticeable depending on the position or time during the animation at which they occur. As such, artists may want to tweak how much weight is given to the blurhash component compared to the two polynomial components to allow the method to look appealing for a larger range of character models and animations. While coefficient values of 1.0 represents the minimal error compared to the input, sometimes a larger overall error may still produce a result that looks better. Larger coefficients slightly reduce excess darkness at the cost of removing more of the occlusion, while smaller coefficients bring us closer to the unaltered blurhash output. Similar to the discrete method, our occlusion may be multiplied by a strength parameter to increase the darkening. The blurhash decoding method also includes a parameter that can be used to adjust the contrast of the result. Similarly to the strength parameter this can be adjusted at runtime to change the final appearance.

3.4.3. Interpolation

As the resolution of our cubemaps is relatively low, using this method on its own may give a pixelated appearance to the environment. Therefore we want to find a way to interpolate between texels of our cubemap. Our encoding method does not allow for us to easily interpolate between coefficient values, since two similar functions may not necessarily have similar coefficients. Therefore we have to compute the values for a few different points and interpolate between those afterwards. Figure 3.9 shows the effect of enabling this interpolation on the rendering result.



(a) Interpolation disabled **(b)** Interpolation enabled

Figure 3.9: Cubemap pixel interpolation

4

Implementation

This chapter will cover how we implemented our method by going more in depth about some of the technical choices we made. We will go over both the rendering and pre-computation stages for both the discrete and continuous methods. Our implementation is written in C++ using OpenGL.

4.1. Pre-computation

Since pre-computation can take a relatively long time, we want to make sure our implementation can handle unexpected crashes or interruptions without having to re-compute everything from scratch. To achieve this we perform the fitting for one cube face at a time, then write the results to the disk before moving on to the next one. Pre-computation works by writing all the data for each face to a large data array, then performing one of the two fitting methods depending or whether it is the discrete or continuous methods.

In both methods we use raycasting using uniformly distributed sampling points around a sphere and Formula 3.1 to compute the occlusion at each point in space.

4.1.1. Discrete radial distance

Recall our discrete method works by splitting the radial distance into a number of distinct cubemaps. This means we can easily compute the results for one distance step and then save the results, and we only have to worry about a single dimension when encoding our data into the cubemap. We first need to find the discontinuities in our dataset so that we can split it according to these and perform curve fitting per segment. Algorithm 1 shows how these breakpoints are determined by computing differences between the points immediately left and right of every point in time, using a window of a few points to account for noise. We then select the largest differences while keeping a minimum space between selected indices.

Algorithm 1 Pseudocode for finding discontinuities in our dataset

```
w ← 5                                   ▷ Window size, using a value of 5 in our implementation
s ← 32                                  ▷ Minimum spacing between discontinuities
V ←input occlusion values
D ← (0, ...0)                           ▷ Differences array, same size as input array
for i ← 0 to len(V) do
    l ← 0
    r ← 0
    for k ← 0 to w do
        l ← l + V_{i−k}                 ▷ For values below 0, loop around to end of array
        r ← r + V_{i+k}                 ▷ For values above len(V), loop around to start of array
    end for
    D_i ← (i, |l − r|)                  ▷ Tuple consisting of index and difference
end for
D ← D sorted by difference
B ← ∅
while len(B)< 4 do
    (i, x) ← head of D                  ▷ Every iteration, remove first (index, difference) tuple from list
    a ← true                            ▷ Check if point is too close to any previously found ones
    for b in B do
        if dist(i, b) < m then          ▷ Dist function takes into account length of array
            a ← false
        end if
    end for
    if a then
        B ← B + i                       ▷ Add to set of breakpoints
    end if
end while
```

After determining breakpoints we can perform the curve fitting on each section individually. For this we use SciPy [19], which we can call from our C++ code using pybind11 [16]. SciPy includes a curve_fit function [18] which can be used to fit a polynomial curve to given data. While there are numerous implementations and methods available to fit polynomial curves, the advantage of the SciPy implementation is that we can specify bounds on the computed coefficient values. If we do not limit the range of coefficients, we will not be able to encode them in a byte with sufficient accuracy to produce good looking rendering results.

The curve fitting method returns an array of coefficients which we can then store. The bounds we set for our coefficients are [-4, 4], which allow them to be encoded into one byte by adding four to map our range to be between zero and eight. We then multiply the result by $\frac{255}{8}$ to get a byte value, which is stored in our cubemap and passed to our shader.

Rendering

To render our results a GLSL shader receives the position of the character model, as well as the cubemaps. We do not want to interpolate between values when reading out the cubemaps, as two adjacent directions may be similar but have very different coefficients. We do want to interpolate the resulting values to avoid seeing resolution artefacts for smaller cubemap sizes, as shown in Figure 3.9. To solve

this we manually sample adjacent pixels and interpolate between them after calculating the occlusion values.

Sampling adjacent pixels in a cubemap is not trivial to do, since the sampling input is an angle rather than pixel coordinates. Therefore we need a method to convert our angles to pixel coordinates. To do this, we pass in an additional small cubemap with coordinate pixel values encoded in it instead of colour data. Our reference cubemap has the internal format of `GL_RGBA32F`. The red and green colour channels simply range from zero in the top-left corner to one in the bottom-right corner. The blue channel is a value ranging from zero to five indicating the current face of the cubemap. Sampling from this reference cubemap in our shader using the direction gives us the face and pixel coordinates we need to sample our actual data, and allows us to easily sample adjacent pixels without needing to consider the angles as we would if we were sampling using a direction only. As interpolation does not happen across the boundaries of faces of the cubemap, seams may be around the edges of each of the faces. Although this effect is generally difficult to notice, it could be resolved by repeating the data on the edges of the cubemap over multiple faces. We did not include this in our implementation as it did not seem necessary.

The actual encoded data is not passed in as a cubemap but a 2D texture with 6 layers. The blue channel of our reference cubemap indicates which layer we are on, while the red and green channels indicate which pixel. We can now sample from the four closest pixels and interpolate between them using the float values acquired from the reference cubemap.

4.1.2. Continuous radial distance

For our continuous method we need to encode two dimensions of data, which we do using the blurhash algorithm[2]. A C implementation for the algorithm is available [3], which we can use as a basis for our implementation in C++. The blurhash method encodes red, green and blue colour channels, whereas we only have a single channel of occlusion which we need to encode. In order to put our implementation together quickly we decided to work around this limitation of the algorithm rather than changing the implementation to circumvent this issue. Fortunately these colour channels are encoded independently, meaning we can re-purpose the three colour channels to better fit our needs. Therefore, we split the time domain into three parts that are encoded independently. This also gives us an advantage when rendering the result, as our shader only needs to consider one third of the basis functions to get the occlusion value at one point. One downside of this is there may be some small discontinuities at the points where the dataset is split. The blurhash algorithm usually encodes the result in an ASCII string using a list of available characters for base83 encoding. In our case we do not need an ASCII string so we instead store the base83 values directly. This does mean one of the bits of each byte remains unused, this could be further optimised to use less space in memory. We chose not to do this as it would not be enough to remove an entire cubemap, meaning that optimising the memory usage using this would add a significant amount of complexity to our implementation.

As discussed, our blurhash method also includes two polynomials fit over the residuals over both the time and distance domain. We compute the output values acquired by decoding the blurhash coefficients back to occlusion values, identical to what would happen on the GPU when computing an occlusion value from a given blurhash input. We can then take the difference between the input values and the output values and once again use SciPy to fit a curve to this data. We first compute a curve against the distance axis, then add the values of this axis to the output occlusion values. We then fit a second curve to the error on the time axis. Both of these polynomials use 4 coefficients.

As discussed our chosen blurhash parameters add up to 54 bytes, plus 8 bytes for the polynomials. The number of bytes needed to encode, as well as the processing time required to decode the image on the GPU, both increase as the number of basis functions goes up. We settled on using five basis functions on both axes. These 54 bytes also include the number of basis functions, however, these are shared between all directions and therefore do not have to be included. Overall this means there is room to encode more information using this method which will be left as a potential future improvement.

Rendering

Typically the blurhash decoding step works by first iterating over all the basis functions, and then iterating over every possible pixel in the image to compute their colour values. Since each instance of the shader is responsible for rendering a single pixel only, we do not need the second iteration of the algorithm in our shader implementation which substantially reduces the runtime of the decoding algo-

rithm. Unfortunately we do still have to iterate over all the basis functions, but the number of these is generally small depending on the chosen parameters. As the blurhash algorithm is not designed to run on the GPU, there may be room for optimisations in the implementation. Similar to the previous method we can sample multiple adjacent pixels of our cubemaps to interpolate between them for a smoother rendering result.

4.2. Reference implementation

We wanted to have a fully raytraced ambient occlusion method to compare against for reference. This method takes the current camera angle to render a single ambient occlusion frame to a texture which can then be used in rendering. As it runs fully on the CPU it's not suitable for real-time rendering, but can be used to get single frames as reference material.

The rendering works by casting a ray from the camera for every pixel on the screen, finding the collision point in the scene and casting sampling rays from the collision point. Initially the sampling method we used generated a given number of random samples in a semi-sphere around the model, based on the normal vector at the impact point. However, the computation method for the cubemap based ambient occlusion uses full sphere sampling, meaning that our reference implementation had a different calculation method for occlusion than our actual implementation. To make the results more comparable this was later replaced by sampling from uniformly distributed points, based on the vertices of an icosphere.

The ambient occlusion is then calculated through Formula 3.1.

5

Results & discussion

To evaluate the results of our methods, we will look at the performance when rendering various models. We use a human model with a walking animation found online [10], a godzilla model with a basic walking animation [15] also found online, and a cube with an animation of rotating 360 degrees. We will evaluate our method in terms of the accuracy with which it reproduces the computed reference occlusion, as well as the appearance of the final rendered result and the performance. We will also be comparing our results to a screenspace method based on AlchemyAO [13].

5.1. Runtime performance

We measured the runtime performance by looking at the framerate achieved when rendering a scene using each of the methods. The system used for these performance evaluations has an AMD 5600X six-core CPU and an Nvidia 3060ti GPU. As the performance results for all the methods are largely independent of the complexity of the scene and the model, we chose to use only the human model in the scene shown in the rendering results later on.

Figure 5.1 shows the average framerate over a couple of minutes for each of the methods. AlchemyAO requires using post-processing, such as Gaussian blur, to reduce the noise in the result. In our implementation this reduces the framerate from quite close to the blurhash method to quite a bit worse. The discrete method is significantly more efficient as it works with only four functions instead of the 25 basis functions used in the blurhash method. It may be possible to further improve the performance of the blurhash method by using GPU-specific optimisations, since our implementation mostly re-implements the algorithm as it was designed to work on a CPU environment. Both the blurhash and alchemy methods have parameters that can be altered to change the runtime. For blurhash, reducing the number of basis functions will improve performance at the cost of less accurate occlusion. For AlchemyAO, reducing the sampling rate will increase performance but create a noisier and less consistent result. Overall we can see our discrete method provides considerably lower runtime cost than the screenspace method. The blurhash method is quite similar in performance to the screenspace AlchemyAO method, and tweaking their parameters could allow for either of the methods to perform better.

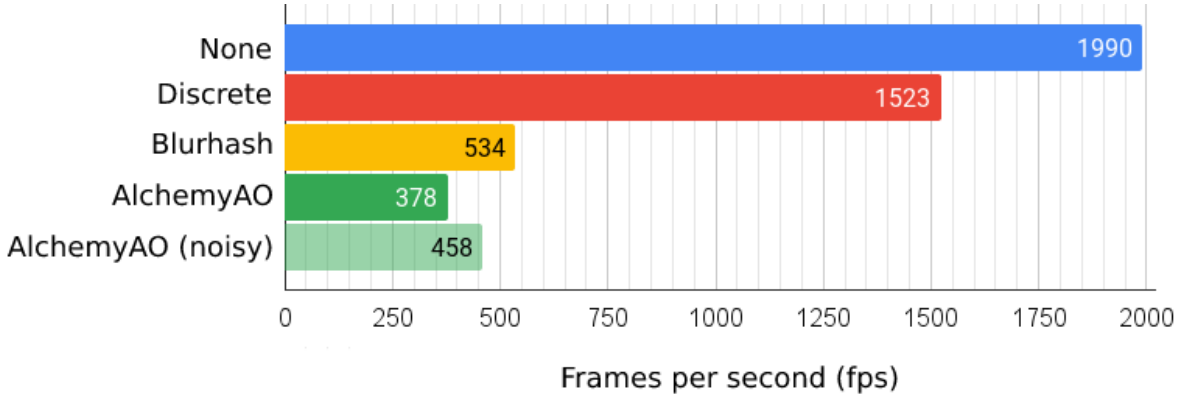


Figure 5.1: Runtime performance for each of the methods

5.2. Pre-computation performance

As the cubemaps can easily be pre-computed and stored, the performance of the pre-computation is not as important as runtime performance. However, artists may want to tweak pre-computation parameters and iterate over various options quickly, as such we will cover this part briefly as well.

Figure 5.2 shows the time it took to compute each of the results we evaluated, with orange highlighting the portion of the runtime dedicated to computing the input occlusion values using raytracing. The variance in runtime between different models within the same method is caused by the increased model complexity increasing the raytracing runtime, whereas the variance between the two methods is caused by the time taken to perform the data fitting. Due to an inefficient implementation using Python, the performance of the discrete method is quite poor. The blurhash method is considerably faster although the runtime is also limited by the raytracing performance. Fortunately there is a lot of room for improvement in the pre-computation, as we did not put much focus on trying to optimise this part of the method. For example, multi-threading or raytracing using the GPU could be used to speed up that portion significantly.

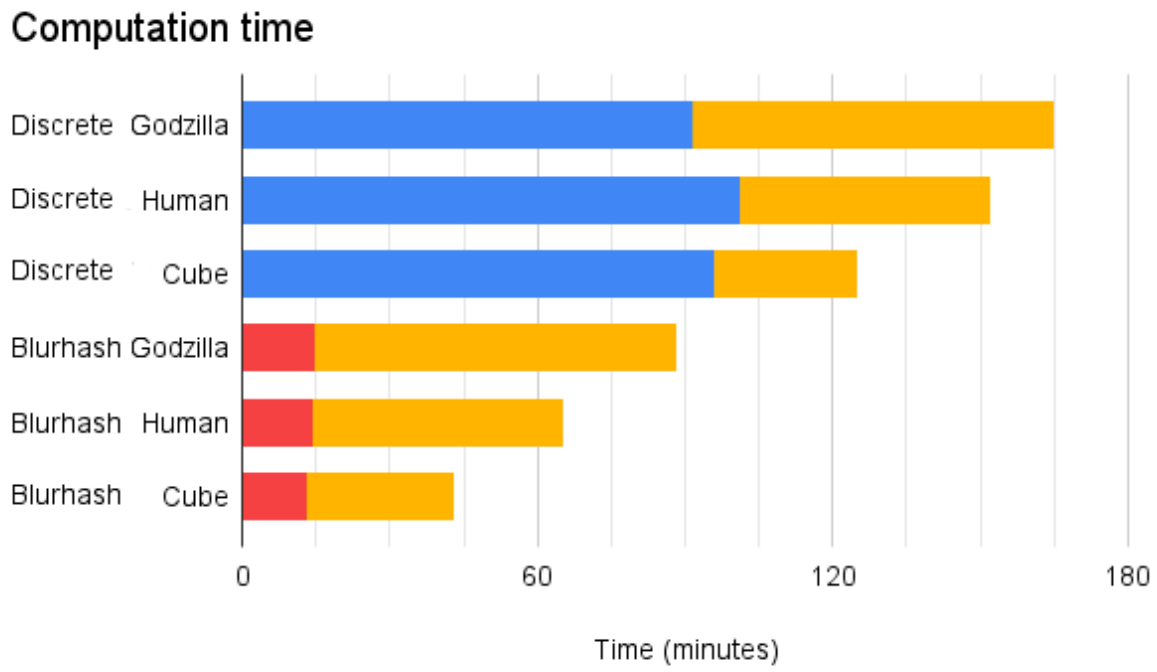


Figure 5.2: Pre-comptation time for each of the methods

5.3. Memory usage

Each individual cubemap is made up of 6 faces with a variable resolution, for which we used 128x128 pixels in all of our results. As discussed before, each texel in our cubemaps uses 16 bytes. This results in a total memory usage of 1.5 MiB per cubemap. Our discrete method uses eight cubemaps for a total of 12 MiB, while our blurhash method uses four cubemaps for a total of 6 MiB. The blurhash method does not make full use of all of this space due to the encoding method used.

A voxel grid to store the occlusion would result in as much as 32 MiB if we assume the same resolution of 128 on every axis and the same number of bytes per unit to encode the time domain. The screenspace method does not use any additional memory except for a buffer to store the occlusion result, which is necessary to compute the Gaussian blur used to denoise the result. This buffer is a small constant amount of memory, whereas the memory used by our method scales with the number of animations present.

Overall our method uses 38% to 19% of the memory a grid based method would, with some room for further improvement possible for the blurhash method.

5.4. Encoding

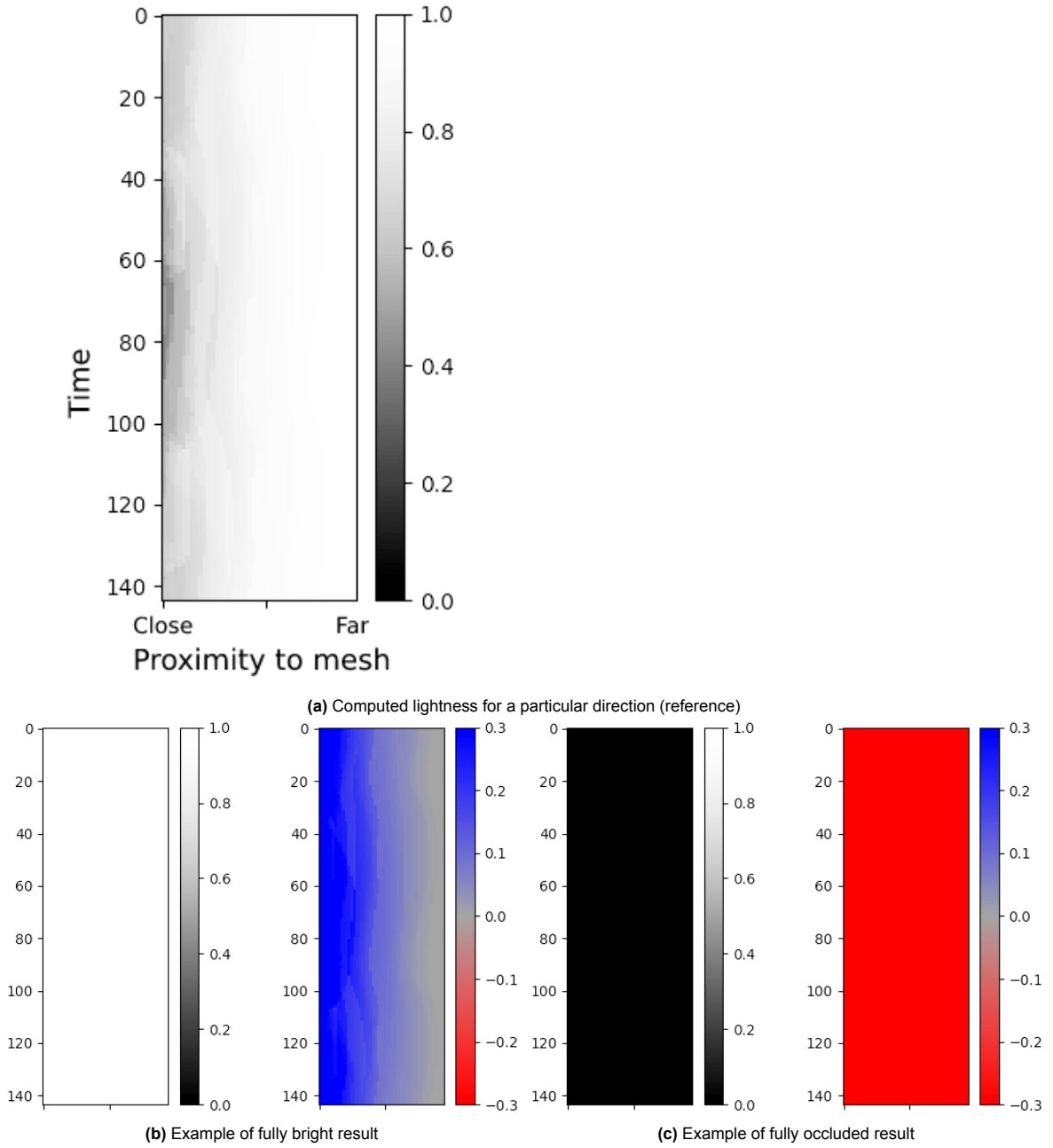
To evaluate our encoding method we will compare the original ambient occlusion to the resulting values from our encoding methods. For this we consider some values over the distance and time domains at a few arbitrary directions from our model. Since each such direction is stored as a simple texture comparing these is straightforward. We will compare the results from our discrete method to the results from the continuous method.

For ambient occlusion, incorrectly showing an area as too bright tends to be much less noticeable than incorrectly showing an area as too dark. Because these different sorts of errors show up very differently in a render, we decided to compute the errors for them separately.

5.4.1. Individual directions

Looking at a particular direction, a single cubemap 'pixel', we can generate a chart to show the time and proximity domain for it. Figure 5.3a shows an example of such a graph, where the brightness of each pixel in the graph is the brightness of the point in space corresponding to it. A dark pixel means a lot of occlusion, whereas a light pixel means very little occlusion.

To evaluate the results of each of our algorithms we will show side-by-side graphs of the computed lightness value next to a difference graph. The difference graph shows in red when a point is too dark, and in blue when a point is too bright. Generally the errors shown in red are more noticeable in the scene than errors shown in blue. Examples of how this looks for having no occlusion at all, or no light at all, can be seen in 5.3b and 5.3c respectively.



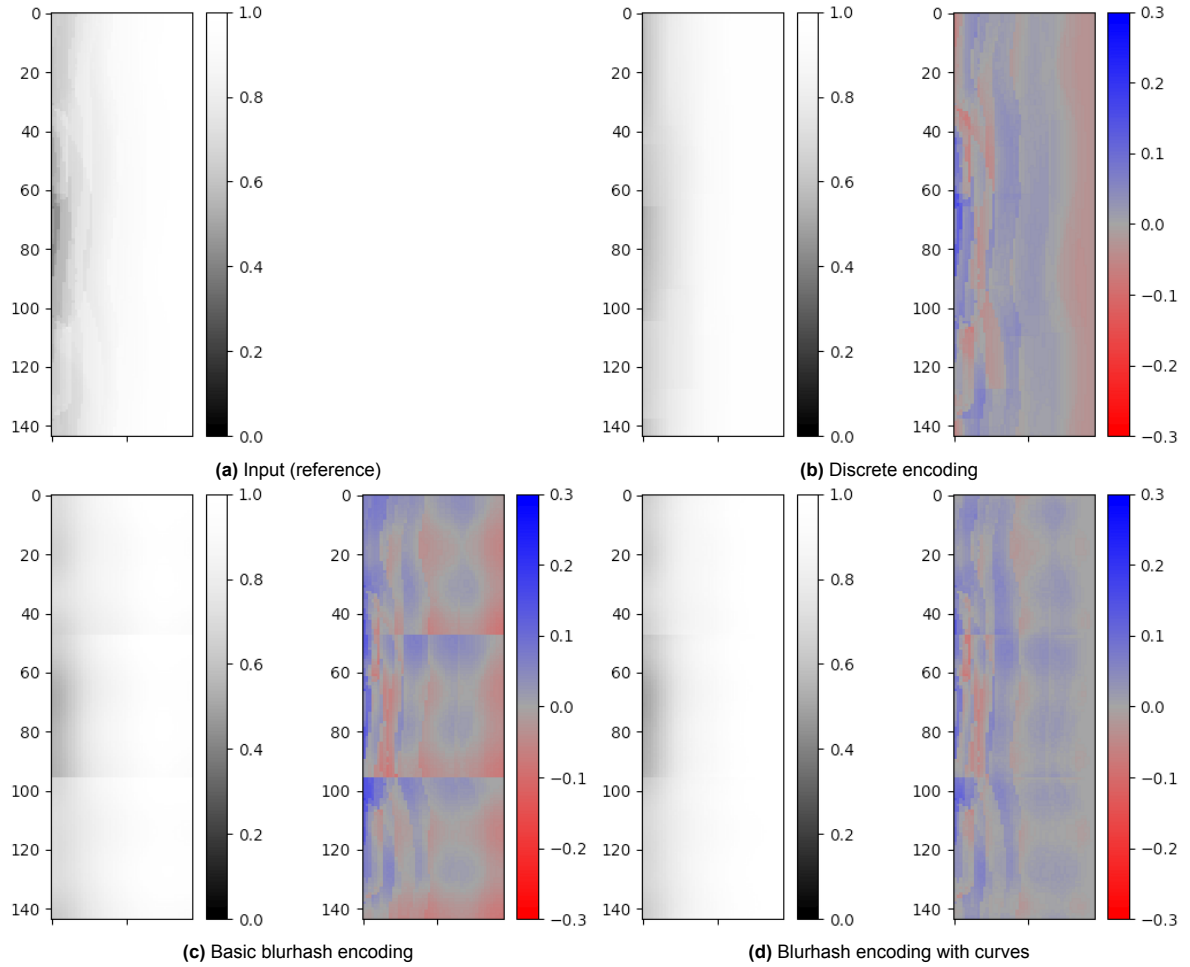
(a) Computed lightness for a particular direction (reference)

(b) Example of fully bright result

(c) Example of fully occluded result

Figure 5.3: Graphs of lightness as encoded in a particular cubemap direction. Y-axis shows the time domain, X-axis shows the proximity to the mesh. For each pair of graphs the left graph shows the computed result, and the right graph shows the error compared to the reference.

Figure 5.4 shows the results for an arbitrary direction. If we add the square of all the errors in such a graph together we can get a single error value, as well as individual error values for the positive and negative errors. These error values are shown in Table 5.1. For this sample we can see due to the relatively simple input data that both the discrete and blurhash with curve encoding do a good job representing the data. The basic blurhash encoding shows very noticeable discontinuities over the time domain in the form of two clearly visible horizontal lines. By looking at the error graph we also see that the right side of the graph shows an area in red for both discrete and basic blurhash encoding, which shows up in the rendering as a dark ”ring” at a fixed distance from the mesh. The error curves used in the second blurhash encoding method resolves this, showing almost no red areas in the error graph. Looking at the error values in the table confirms that the negative error is significantly reduced in the BlurhashCurve encoding method, while keeping the positive error relatively low as well.



(a) Input (reference)

(b) Discrete encoding

(c) Basic blurhash encoding

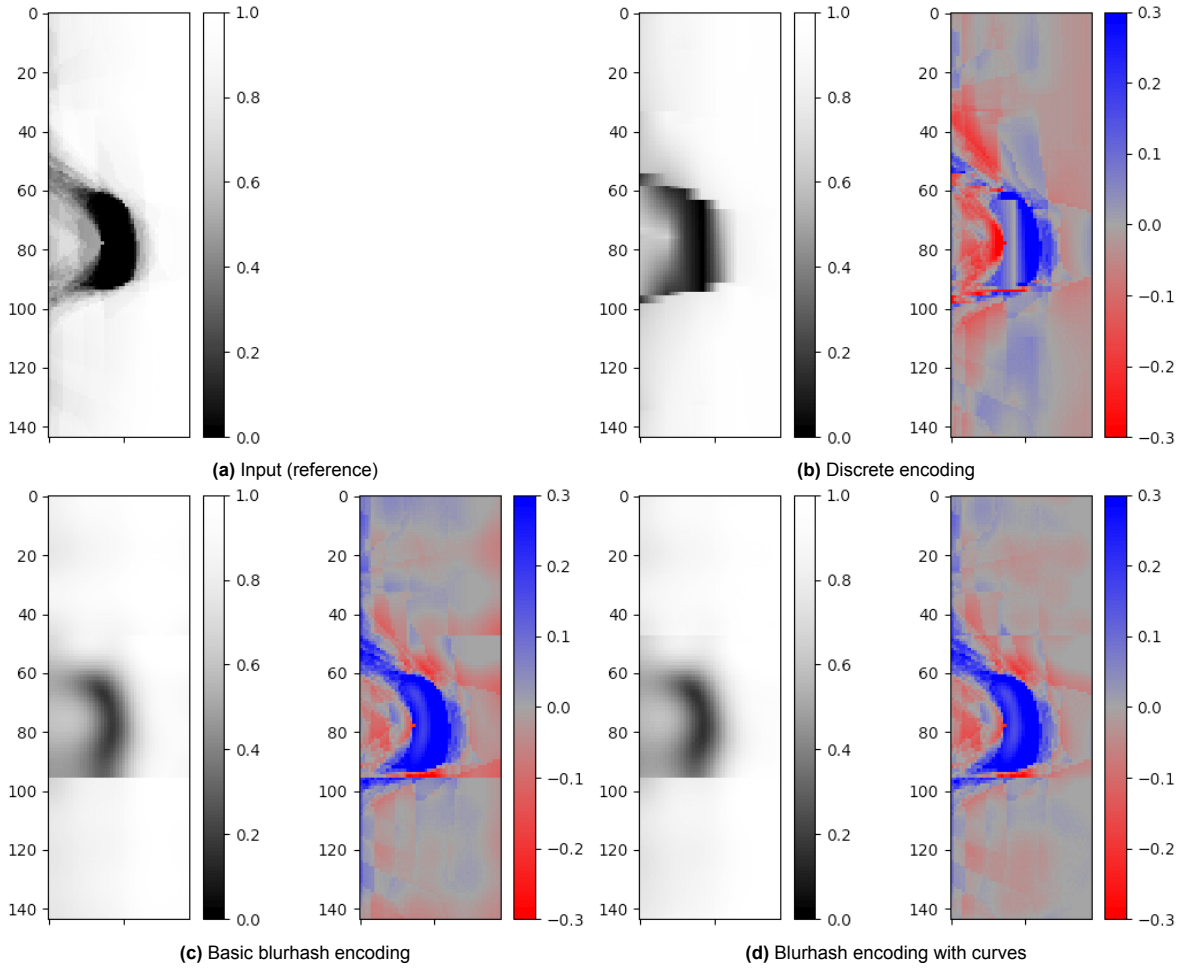
(d) Blurhash encoding with curves

Figure 5.4: Lightness values for sample A

Sample A	Positive	Negative	Total
Discrete	0.98	0.96	1.94
Blurhash	0.87	2.43	3.3
BlurhashCurve	1.11	0.08	1.19

Table 5.1: Error values for sample A

The sample out of the dataset showing the worst total error is represented by sample B, in Figure 5.5 and 5.2. In this case we see a large dark area in the input, indicating these sample points were inside the mesh. It is not very important for our rendering results to correctly represent these dark areas, as the inside of the mesh will not be visible anyway. For this reason the area showing up in bright blue in the error graphs, indicating the result was much brighter than the input, is not a major issue. Unfortunately the area around this dark spot is also affected, as the result near this area is much too dark. This shows some limitations in our method as the used encoding methods allow for dark areas ”bleeding” over not only the space domain, but also over the time domain. In the rendering results this may show up as dark areas in space where a part of the mesh will be later on in the animation cycle.



(a) Input (reference) **(b)** Discrete encoding

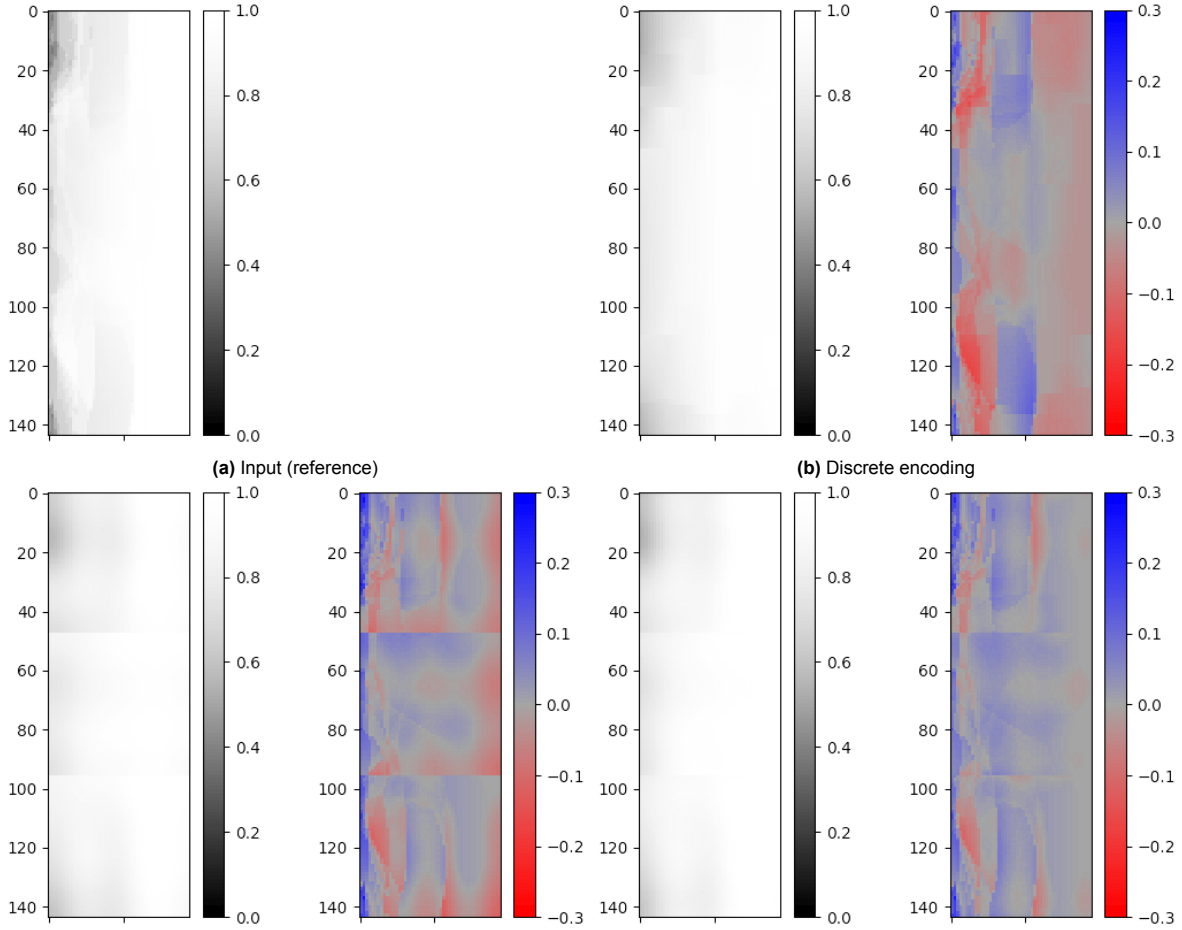
(c) Basic blurhash encoding **(d)** Blurhash encoding with curves

Figure 5.5: Lightness values for sample B (worst total error)

Sample B	Positive	Negative	Total
Discrete	32.85	5.61	38.46
Blurhash	48.2	6.21	54.41
BlurhashCurve	44.14	4.61	48.75

Table 5.2: Error values for sample B

Sample C, shown in Figure 5.6 and Table 5.3 shows a direction where the error is very close to the median error of our entire dataset. Similar to sample A, we see how the BlurhashCurve encoding method does well at removing excess dark areas from the data, at the cost of increasing the excess brightness when compared to the basic Blurhash encoding.



(a) Input (reference)

(b) Discrete encoding

(c) Basic blurhash encoding

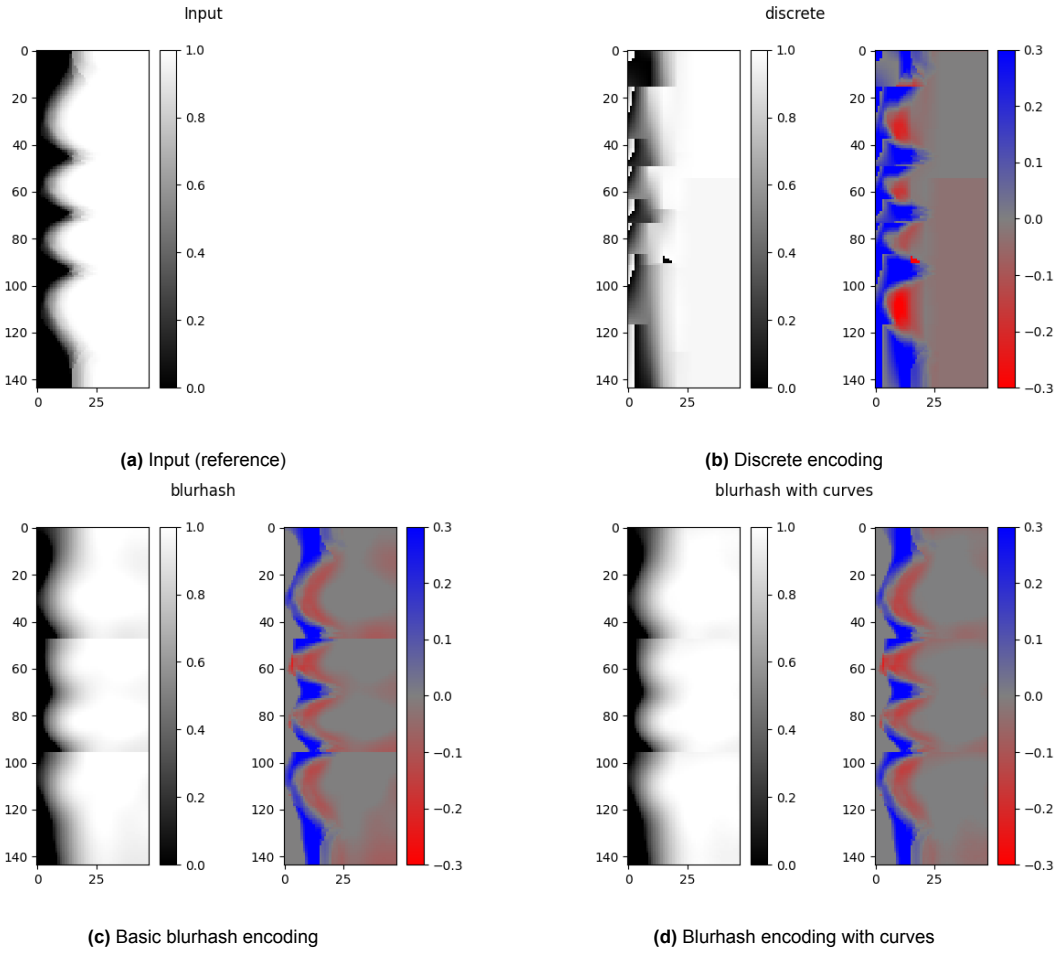
(d) Blurhash encoding with curves

Figure 5.6: Lightness values for sample C (approximately median error)

Sample C	Positive	Negative	Total
Discrete	2.87	2.69	5.56
Blurhash	0.88	3.42	4.3
BlurhashCurve	1.7	0.46	2.16

Table 5.3: Error values for sample C

To see how the method performs with different models, we have another scene where a large cube does a simple animation of spinning around its axis. Figure 5.7 shows the time-proximity graphs for a direction that starts near one of the corners of the box. Since the box spins in a circle, we see the occlusion reduce to a minimum where the box is at a 45 degree angle from the chosen direction, and reach a high point as each of the corners passes by. This animation results in fairly high frequency input data, which is difficult to encode efficiently using our methods. We can see the discrete method do especially poorly, producing a messy result that does not resemble the input data much. This is due to the discrete method splitting the time domain into four parts along discontinuities, which are not clearly visible in this animation. Each segment also contains too much data to be efficiently encoded, likely we would need at least 5 or more segments for this animation to work well. The blurhash methods do slightly better, showing much stronger resemblance to the input data as well as much lower total error values.



(a) Input (reference)

(b) Discrete encoding

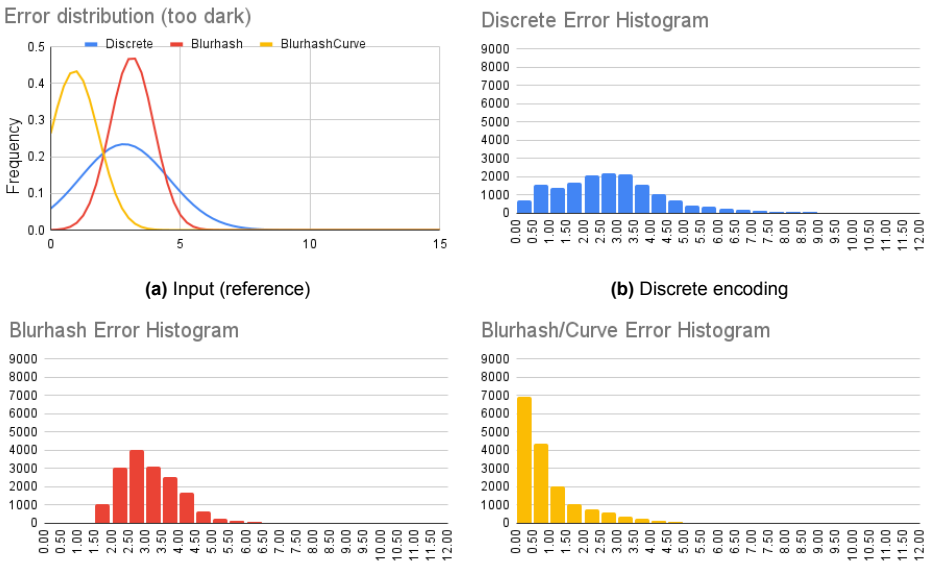
(c) Basic blurhash encoding

(d) Blurhash encoding with curves

Figure 5.7: Lightness values for box animation

5.4.2. Overall results

In Figure 5.8 and 5.9 we have plotted how the errors are distributed over all the samples on the human animation. The first chart shows only the excessive darkness, which is generally more noticeable. We see both the blurhash methods have tighter spreads than the discrete method, although the method without curves also has generally higher error values. The clearest result from these graphs is how the amount of excessive darkness errors are significantly reduced when comparing the blurhash and blurhash with curves methods. Despite significantly lowering this error, the amount of areas that are too bright is only marginally worse.



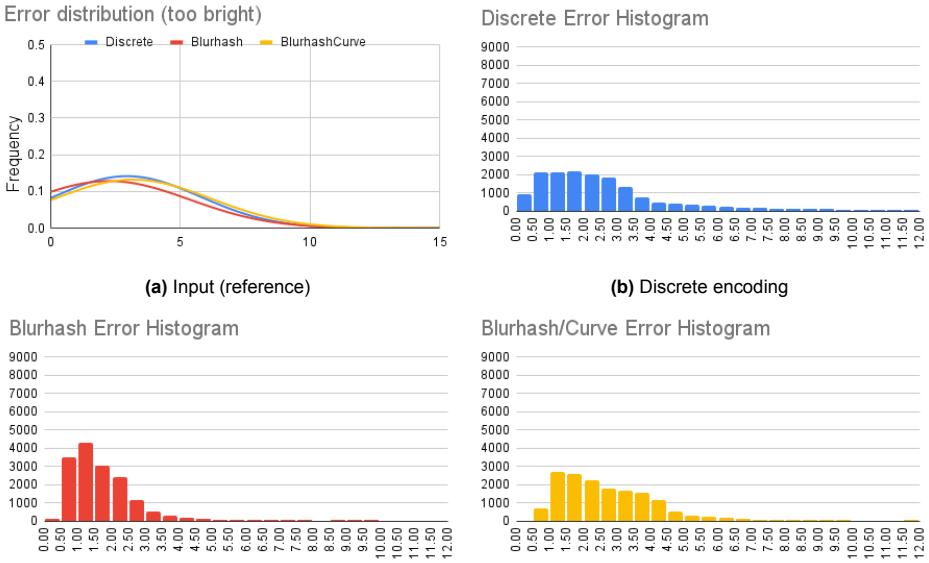
(a) Input (reference)

(b) Discrete encoding

(c) Basic blurhash encoding

(d) Blurhash encoding with curves

Figure 5.8: Error distributions on human model, showing only negative errors (areas that are too dark)



(a) Input (reference)

(b) Discrete encoding

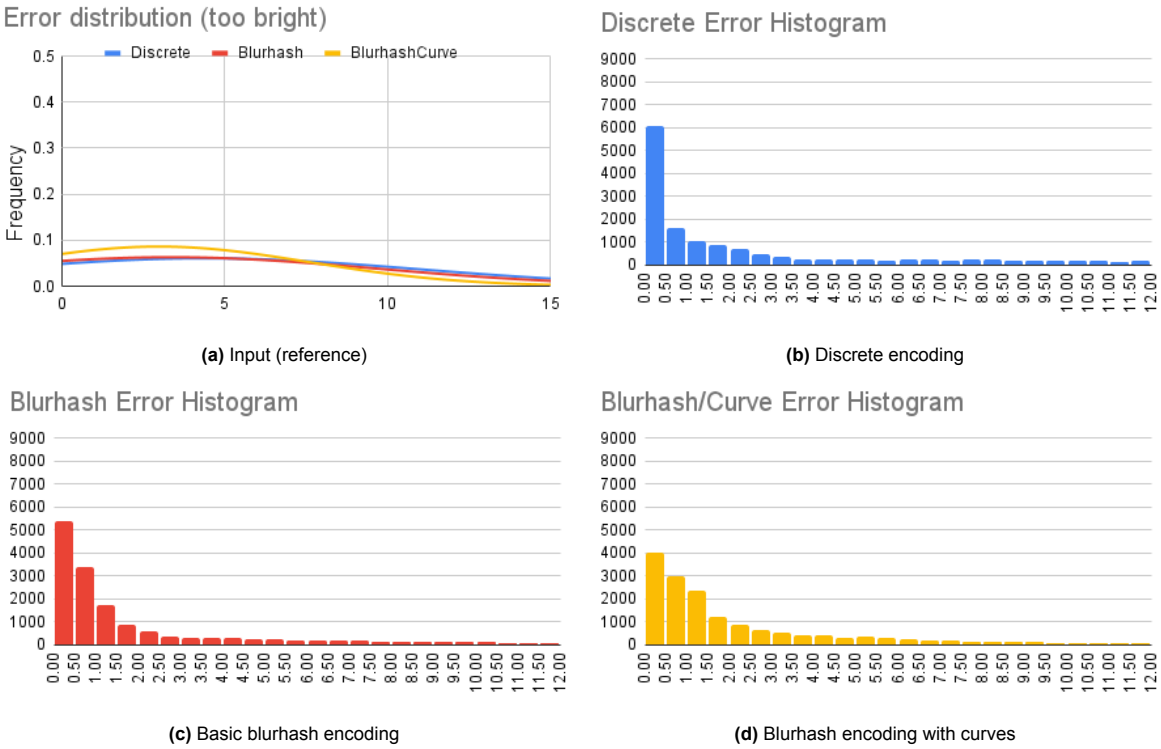
(c) Basic blurhash encoding

(d) Blurhash encoding with curves

Figure 5.9: Error distributions on human model, showing only positive errors (areas that are too bright)

We also plotted the brightness error for the godzilla model in Figure 5.10. In this case we see relatively small differences between the methods, although the discrete method seems to have a slightly lower error than the other methods.

The error plots for the excessive darkness on godzilla or the box model have been left out as they are mostly uniformly distributed.



(a) Input (reference)

(b) Discrete encoding

(c) Basic blurhash encoding

(d) Blurhash encoding with curves

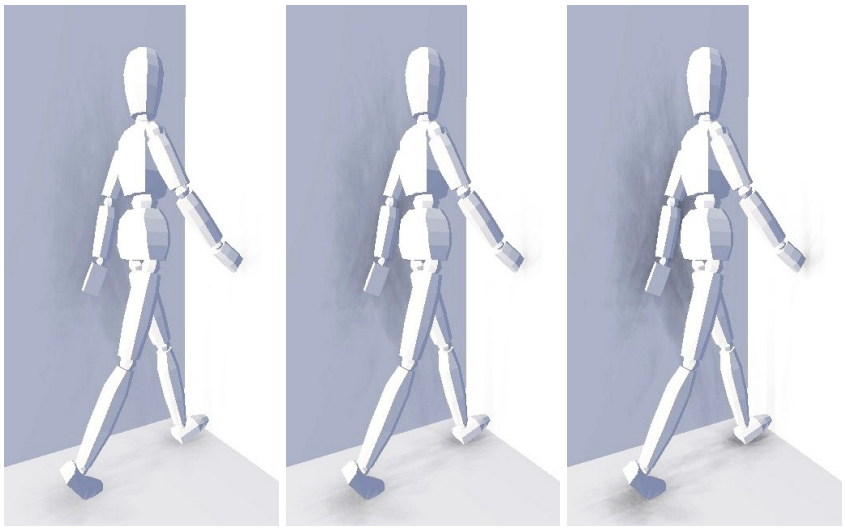
Figure 5.10: Error distributions on godzilla model, showing only positive errors (areas that are too bright)

5.5. Rendering

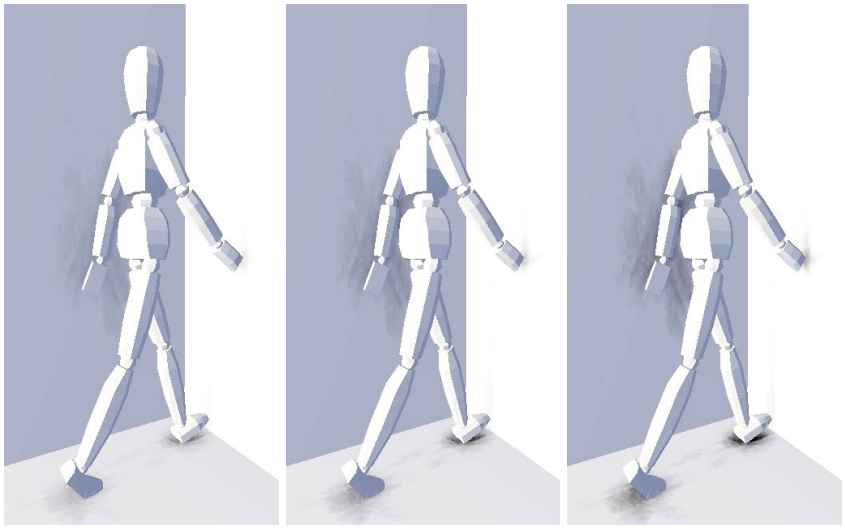
To compare the visual quality of our implementation we will look at a few sample scenes encoded using our methods. However, since our blurhash and discrete method have a few different parameters, we will first go over the choices we made with respect to the chosen parameters for our comparisons.

Figure 5.11 shows how altering the parameters used in the shader of our blurhash method affects the produced image. While increasing both strength and contrast makes the occlusion more noticeable, it also introduces additional artefacts. For example, 5.11d has more visible occlusion around the feet, but makes some of the occlusion around the shoulder area harder to see. Increasing the strength such as in 5.11c creates a large dark area on the wall that seems much darker than it should be. For our comparisons we have chosen to keep the effect more subdued while avoiding artefacts, since we found it is generally more distracting to have occlusion where there should not be any than to have occlusion being slightly less visible than it should be.

The reference raytraced implementation evaluates the occlusion on a point on a surface, meaning one half of the sampling sphere is always fully occluded as this falls behind the surface. This is different from our pre-computed methods, since they evaluate the occlusion at a point in space that is not necessarily close to a surface. As such our pre-computed method samples from half a sphere instead of a full sphere, reducing the strength to 0.5 will bring the results more in line. Therefore all the comparisons will use a strength of 0.5 and a contrast of 1.1.



(a) Strength 0.50, contrast 1.1 **(b)** Strength 0.75, contrast 1.1 **(c)** Strength 1.00, contrast 1.1

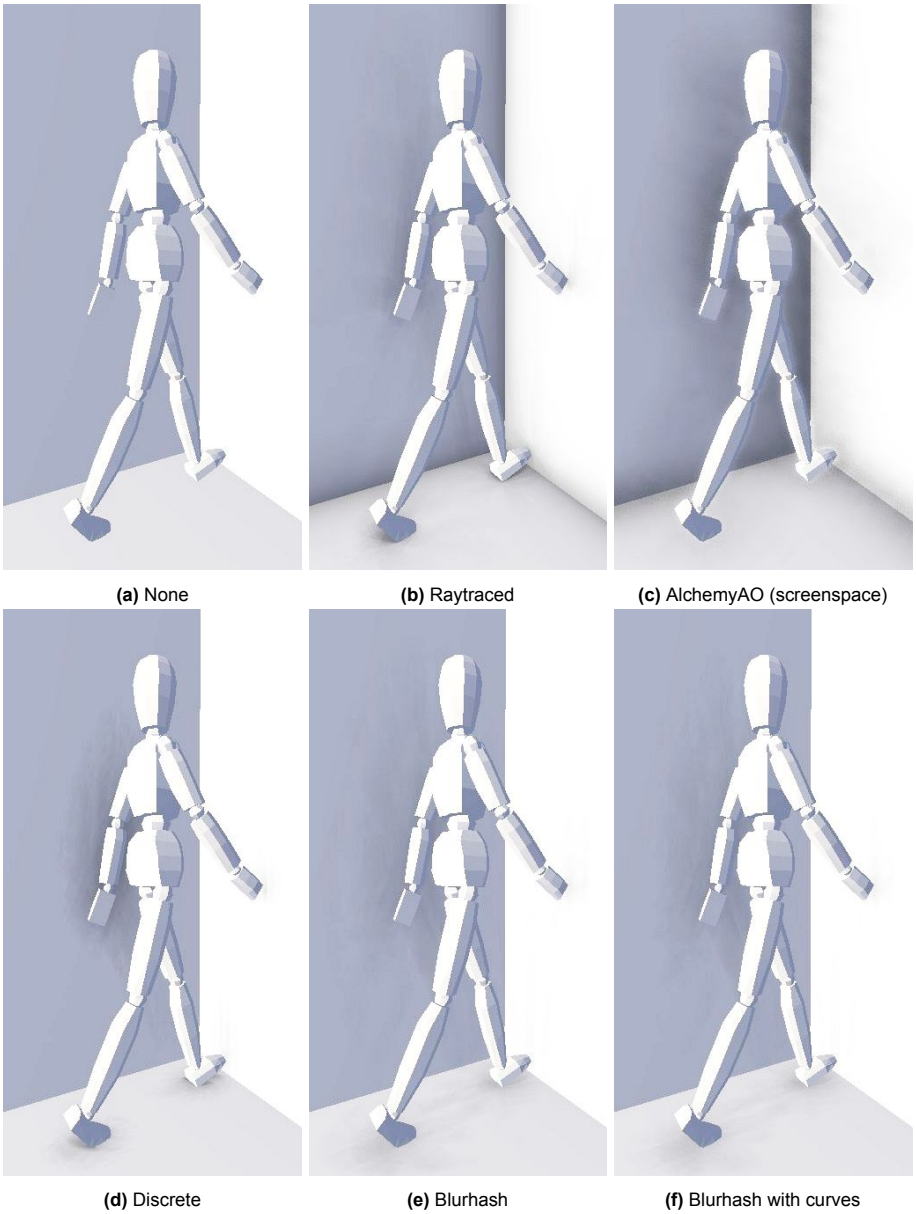


(d) Strength 0.50, contrast 2.0 **(e)** Strength 0.75, contrast 2.0 **(f)** Strength 1.00, contrast 2.0

Figure 5.11: Comparison of various blurhash rendering parameters

We will now compare the rendering results across different methods, including our own methods as well as a fully raytraced reference implementation and a screenspace method. The results here are somewhat subjective as each method has different parameters than can be tweaked to get slightly different results, and some of the artifacts created may be less noticeable than others. Note that the raytraced and screenspace methods produce occlusion also for the walls in the scene, whereas our cubemap method produces occlusion only cast by the model itself. We will only focus on the occlusion cast by the model on the environment.

Figure 5.12 shows the results at the start of the animation cycle. Overall the results produced by the discrete and blurhash methods seem relatively close to those of the raytraced scene. The blurhash result is a little bit more subtle than the discrete one due to the chosen parameters. In this particular example, the blurhash and blurhash with curves method are indistinguishable. The reason for this is that the curves method mostly deals with removing artifacts that show up at higher time values, as described in section 3.4.1.

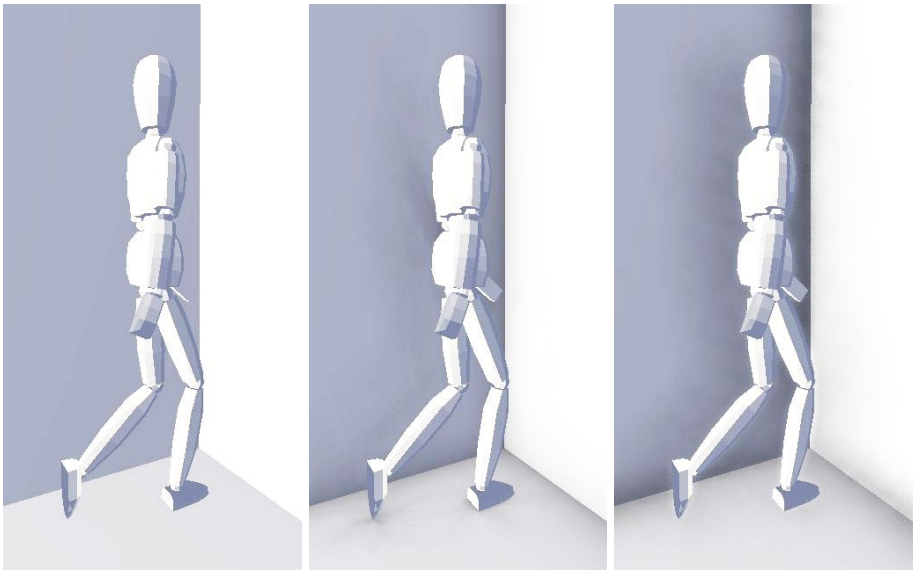


(a) None **(b)** Raytraced **(c)** AlchemyAO (screenspace)

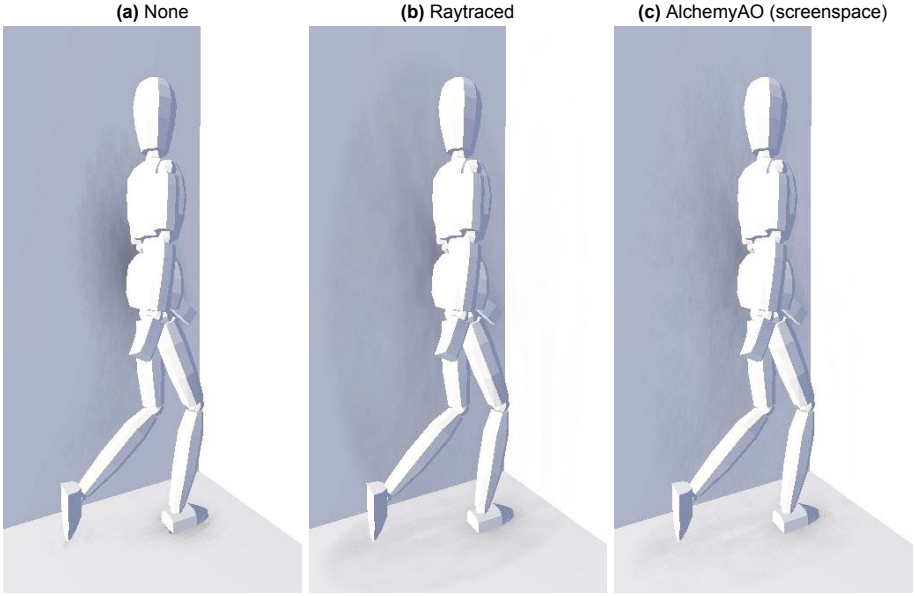
(d) Discrete **(e)** Blurhash **(f)** Blurhash with curves

Figure 5.12: Comparison of various occlusion methods at start of animation cycle using model of a human

When looking at a point further in the animation cycle Figure 5.13 the differences become more apparent. The blurhash method now shows a large dark halo around the model, which we could also see clearly in Figure 5.6, corresponding to the red areas around 50 and 95 time points. The result is significantly improved when adding in the curves, as the halo is now largely gone. The discrete method seems to be missing some occlusion around the feet, as the back foot has hardly any darkening around it at this point in time. In this case, the blurhash with curves result seems to be the best out of our methods, although it shows some excessive darkening in some areas.



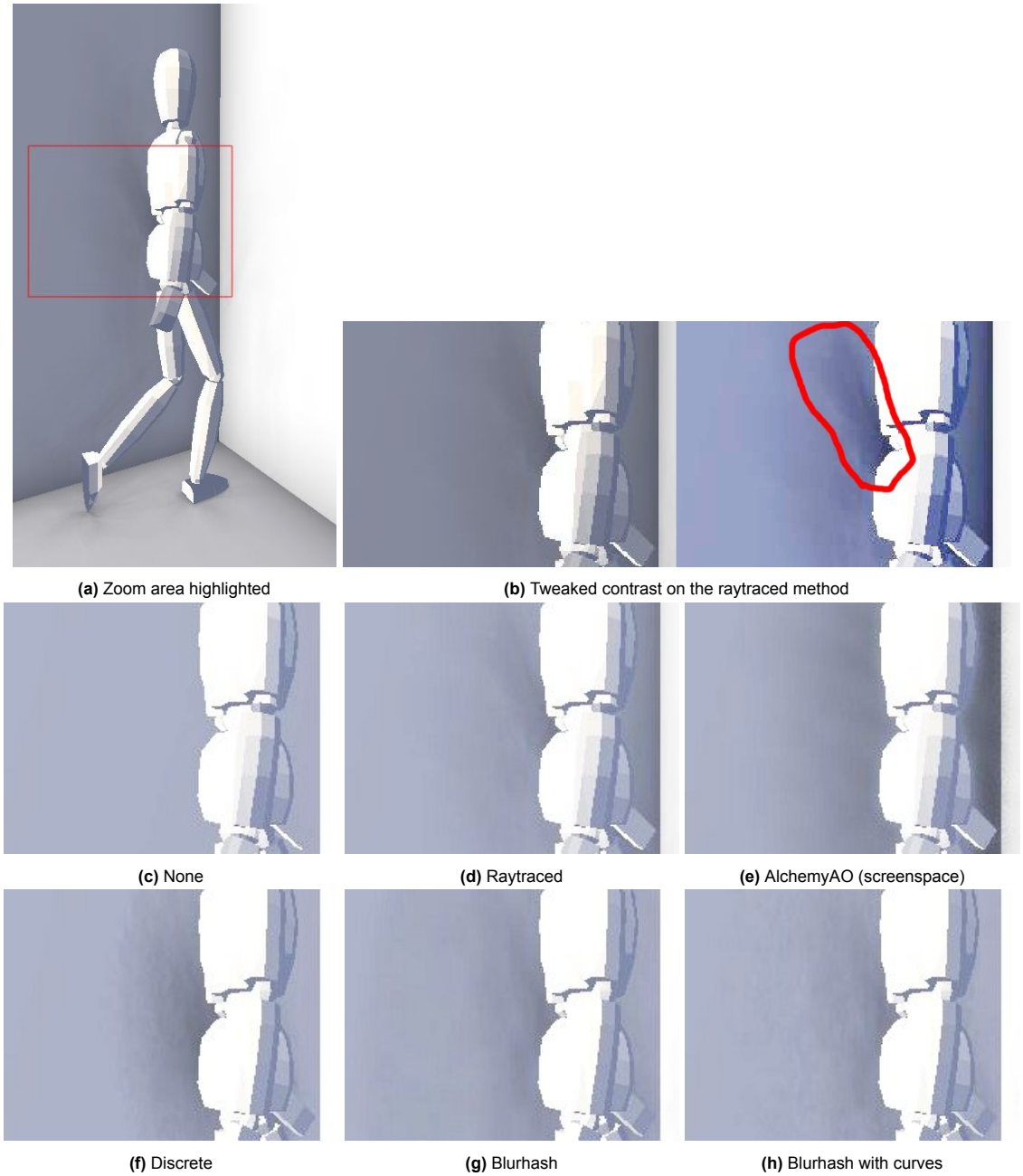
(a) None **(b)** Raytraced **(c)** AlchemyAO (screenspace)



(d) Discrete **(e)** Blurhash **(f)** Blurhash with curves

Figure 5.13: Comparison of various occlusion methods 66% into the animation cycle using model of a human

We will now zoom in on the area behind the model to look more closely at the differences between the methods, visible in Figure 5.14. When looking at the raytraced method, we can see an area behind the torso is a little bit darker due to the arm being closer to the wall than the rest of the torso. Figure 5.14b shows a circled area of the raytraced method where this is particularly visible, with tweaked contrast settings. When we compare all the methods used, we see that the screenspace method produces no visible darkening in this area compared to the rest of the torso. This makes sense as the arm is not visible on the screen here, thus this information is not used in rendering the occlusion. While subtle, all three of our own methods do show a noticeable darker area here, especially visible in the discrete method. This result shows one of the advantages of our methods over screenspace methods.



(a) Zoom area highlighted

(b) Tweaked contrast on the raytraced method

(c) None

(d) Raytraced

(e) AlchemyAO (screenspace)

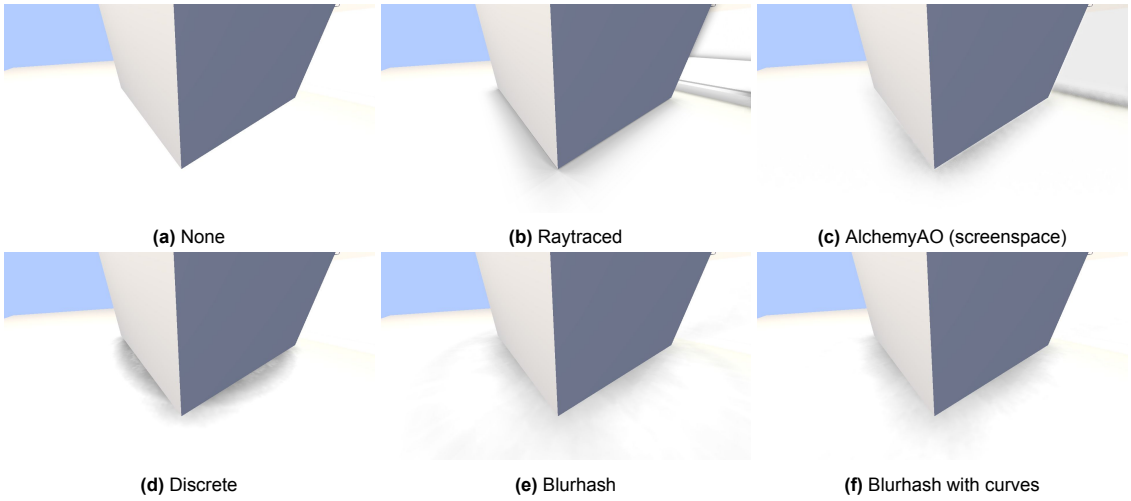
(f) Discrete

(g) Blurhash

(h) Blurhash with curves

Figure 5.14: Comparison of various occlusion methods zoomed in on the arm of the model

Figure 5.15 shows how the rendering performs using a box model, floating a few centimetres off the ground. The screenspace method shows less darkening around the back of the cube compared to the other methods, further highlighting the limitations of the screenspace solution. This comparison shows very clearly the downsides of the discrete method, as we see a clear circular area at the base of the box, as well as some banding around the corner caused by boundaries between the different nested cubemaps. Generally, the discrete method seems to work best for 'messy' models where it may not be easy for a person to tell whether the result is correct. In a case such as this where people have a stronger idea of what a correct result should look like, the downsides are more apparent. We also see the regular blurhash method producing halo effects once again, making the blurhash with curves method a clear front-runner in this scene.

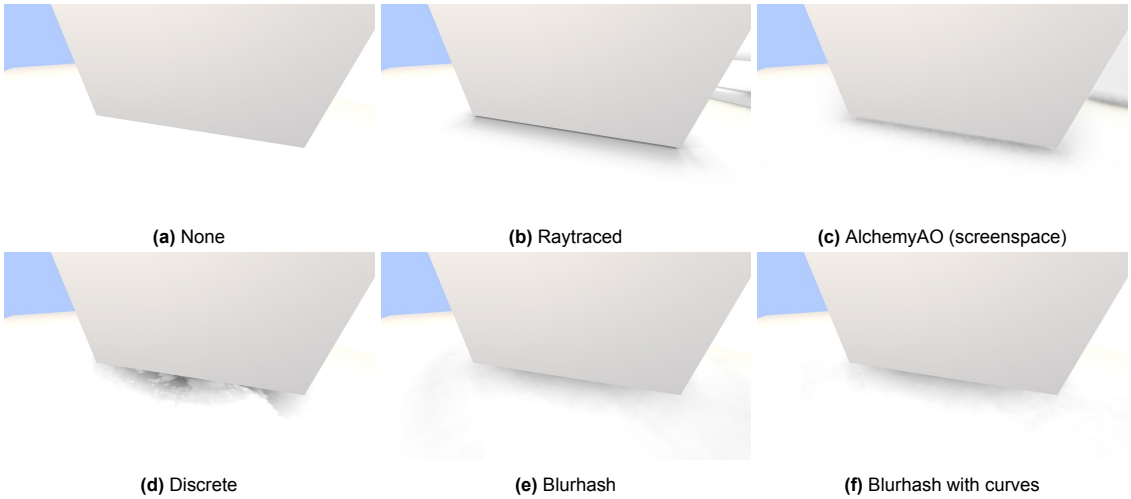


(a) None **(b)** Raytraced **(c)** AlchemyAO (screenspace)

(d) Discrete **(e)** Blurhash **(f)** Blurhash with curves

Figure 5.15: Comparison of various occlusion methods at start of animation cycle of the box model

Figure 5.16 shows the same scene with the box further ahead in the animation cycle, which involves it doing a full 360 degree rotation. The result here shows even more clearly the problems with the discrete method, as we now see a patchy pattern showing up. Likely, this is due to the fact that there are more than four points along which it makes sense to split the dataset, as shown in 5.7, and as such it will pick slightly different ones for adjacent points leading to this patchy appearance. This could be solved by adding further weights to the discontinuity selection method to prefer picking curves similar to adjacent points, or by increasing the number of curves used to encode such animation cycles. Luckily most real-world animation cycles involve less high-frequency movement. We can also see a stronger contrast between the area directly below the cube and just beside it in our raytraced method, while all our methods show more of a smooth transition. The lack of a clear line is due to the compression methods used not allowing strong contrast between adjacent points to remain. The screenspace method does a bit better showing a clearer distinction, although it also uses blurring to remove noise making the boundary less distinct. Once again it seems like the blurhash with curves method comes out on top.

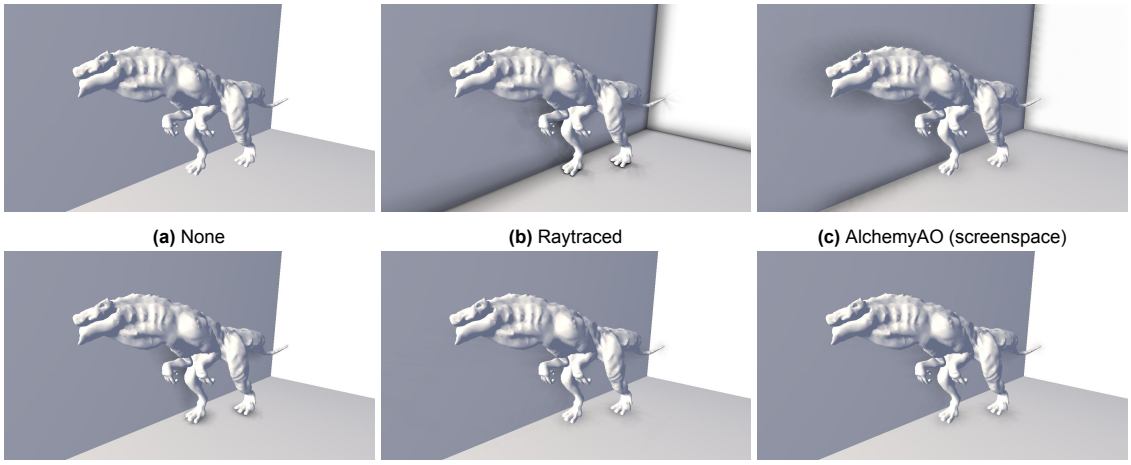


(a) None **(b)** Raytraced **(c)** AlchemyAO (screenspace)

(d) Discrete **(e)** Blurhash **(f)** Blurhash with curves

Figure 5.16: Comparison of various occlusion methods 20% into the animation cycle of the box model

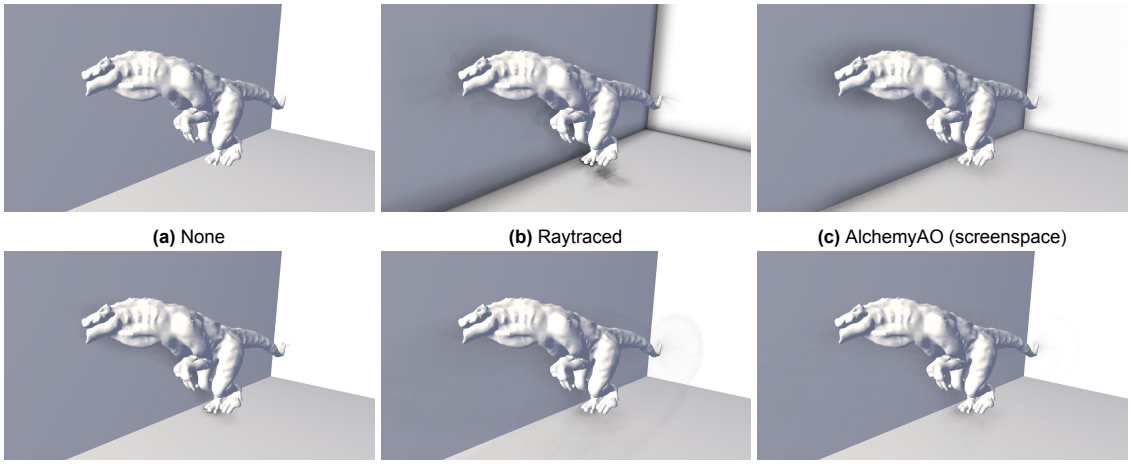
Figure 5.17 and 5.18 show comparisons for another model, this time we use godzilla to show how models with dimensions very different from a human are still encoded well. Mostly the results are similar to the human model, with the discrete and blurhash with curves methods both producing decent results.



(a) None **(b)** Raytraced **(c)** AlchemyAO (screenspace)

(d) Discrete **(e)** Blurhash **(f)** Blurhash with curves

Figure 5.17: Comparison of various occlusion methods at start of animation cycle using a godzilla model



(a) None **(b)** Raytraced **(c)** AlchemyAO (screenspace)

(d) Discrete **(e)** Blurhash **(f)** Blurhash with curves

Figure 5.18: Comparison of various occlusion methods 32% into the animation cycle using a godzilla model

6

Future work

For our implementation, we focused on relatively short looping animations, so there may be limitations when this method is used for more complex scenarios. Blending between different animations may also provide additional complications.

The blurhash [2] algorithm is not ideal for use on the GPU, as it does not benefit from hardware encoding. A future improvement on this would be to find an encoding scheme that does. The blurhash method is designed for a specific use case of providing temporary low-fidelity representations of images, and as such was created with very different design considerations than what we are using it for in our method. A similar method designed with the goal of occlusion in mind may produce better results. There's also room to improve the space efficiency of the encoding, as the encoding scheme is chosen to allow for encoding in text rather than binary form.

Furthermore there are some issues with the way the blurhash encoding is currently handled which creates small jumps over the time domain. Improving this should be fairly straightforward.

There are issues with the darkness from inside of the mesh bleeding out both over the time and space domains, causing excess darkening to occur. The encoding schemes should give minimal weight to the areas in the meshes as these are not supposed to be visible during rendering. Currently, they are given the same weight as visible areas. This limitation in the implementation is responsible for several undesired artifacts.

The pre-computation performance of the algorithms is quite poor and there is a lot of room for improvement in this area.

7

Conclusion

The method presented gives satisfactory results at low runtime cost, while keeping memory requirements quite low as well. Our method overcomes some of the limitations of screenspace methods of missing hidden and off-screen geometry. A major drawback of our method is the long pre-computation stage, largely because optimising this was not a main focus of our research.

We found that the accuracy of the results of our method depends on the type of animation that is used, where especially our discrete method does not do well for high-frequency animations.

There are a lot of parameters in the method that may be tweaked depending on the use case to strike a balance between better rendering performance or lower memory requirements. Unfortunately there are a few artifacts present in the results that may need to be further reduced before this method could be reliably used in a production environment.

Overall we believe our method proves that it is possible to base ambient occlusion on pre-computed methods stored in simple data structures, and that this may offer better rendering results and better performance than relying purely on screenspace methods.

References

[1] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. “Image-space horizon-based ambient occlusion”. In: *ACM SIGGRAPH 2008 talks*. ACM, Aug. 2008. DOI: 10.1145/1401032.1401061. URL: https://doi.org/10.1145/1401032.1401061.

[2] *Blurhash*. 2017. URL: https://github.com/woltapp/blurhash.

[3] *Blurhash C implementation*. 2020. URL: https://github.com/woltapp/blurhash/tree/master/C.

[4] Cyril Crassin et al. “Interactive Indirect Illumination Using Voxel Cone Tracing”. In: *Computer Graphics Forum* 30.7 (Sept. 2011), pp. 1921–1930. DOI: 10.1111/j.1467-8659.2011.02063.x. URL: https://doi.org/10.1111/j.1467-8659.2011.02063.x.

[5] Pascal Gautron. “Real-time ray-traced ambient occlusion of complex scenes using spatial hashing”. In: Cited by: 5. 2020. DOI: 10.1145/3388767.3407375. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85091991176&doi=10.1145%2f3388767.3407375&partnerID=40&md5=ee945d978e86f57fb8b7ae6cea15a1ca.

[6] ISO/IEC 10918-1:1994. *Information technology – Digital compression and coding of continuous-tone still images – Requirements and guidelines*. International Organization for Standardization, Geneva, Switzerland. 1994.

[7] Jon Jansen and Louis Bavoil. “Fourier Opacity Mapping”. In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’10. Washington, D.C.: Association for Computing Machinery, 2010, pp. 165–172. ISBN: 9781605589398. DOI: 10.1145/1730804.1730831. URL: https://doi.org/10.1145/1730804.1730831.

[8] Adam G. Kirk and Okan Arikan. “Real-Time Ambient Occlusion for Dynamic Character Skins”. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D ’07. Seattle, Washington: Association for Computing Machinery, 2007, pp. 47–52. ISBN: 9781595936288. DOI: 10.1145/1230100.1230108. URL: https://doi.org/10.1145/1230100.1230108.

[9] Janne Kontkanen and Timo Aila. “Ambient Occlusion for Animated Characters”. In: EGSR ’06. Nicosia, Cyprus: Eurographics Association, 2006, pp. 343–348. ISBN: 3905673355.

[10] Sebastian Lague. *Human model*. 2017. URL: https://youtu.be/gFf5eGCjUUg.

[11] Tom Lokovic and Eric Veach. “Deep Shadow Maps”. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 385–392. ISBN: 1581132085. DOI: 10.1145/344779.344958. URL: https://doi.org/10.1145/344779.344958.

[12] Mattias Malmer et al. *Fast Precomputed Ambient Occlusion for Proximity Shadows*. Research Report RR-5779. INRIA, Dec. 2005, p. 19. URL: https://hal.inria.fr/inria-00070242.

[13] Morgan McGuire et al. “The Alchemy Screen-Space Ambient Obscurance Algorithm”. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG ’11. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, pp. 25–32. ISBN: 9781450308960. DOI: 10.1145/2018323.2018327. URL: https://doi.org/10.1145/2018323.2018327.

[14] Martin Mittring. “Finding next gen”. In: *ACM SIGGRAPH 2007 courses*. ACM, Aug. 2007. DOI: 10.1145/1281500.1281671. URL: https://doi.org/10.1145/1281500.1281671.

[15] NaelB. *Godzilla model*. 2021. URL: https://sketchfab.com/3d-models/zilla-b9811ff2659447afbbef55c8b38f121d.

[16] *pybind11*. 2021. URL: https://github.com/pybind/pybind11.

[17] Tobias Ritschel et al. “Interactive Illumination with Coherent Shadow Maps”. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR’07. Grenoble, France: Eurographics Association, 2007, pp. 61–72. ISBN: 9783905673524.

[18] *SciPy curve_fit*. 2021. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html.

[19] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[20] Wikipedia. *Spherical coordinate system — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Spherical%20coordinate%20system&oldid=1142703172. [Online; accessed 09-March-2023]. 2023.