Ruprecht-Karls-Universität Heidelberg
Fakultät für Mathematik und Informatik
Institut für Informatik

Bachelor Thesis

Ambient Occlusion in 4D

Name:	Maria Regina Lily Djami
Matriculation Number:	3347645
Instructor:	Prof. Dr. Filip Sadlo
Date of Submission:	June 11, 2021

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe und die Grundsätze und Empfehlungen “Verantwortung in der Wissenschaft” der Universität Heidelberg beachtet wurden.

[illegible]

Date of submission: June 11, 2021

Zusammenfassung

Datenvisualisierung ist hilfreich für die Analyse abstrakter Daten. Nützliche Erkenntnisse und ein besseres Verständnis können durch Datenvisualisierung gewonnen werden. Forschungsdaten sind oft komplex und hochdimensional. Die Visualisierung von hochdimensionalen Daten erfordert die Reduktion der Dimension der Daten, damit die Daten auf ein 2D- oder 3D-Bild projiziert werden können. Dies führt natürlich zu einem gewissen Verlust an Informationen. Durch die Projektion der Daten auf ein 4D-Bild bleiben mehr Informationen erhalten. Allerdings sind 4D-Strukturen nicht intuitiv zu verstehen. Außerdem sind die Visualisierungsmethoden im 3D-Raum weitaus besser als im 4D-Raum. Die verfügbaren 3D-Grafik-Algorithmen sind in der Lage, realistische und qualitativ hochwertige Bilder zu erzeugen.

Eine Methode in der 3D-Grafik ist die Ambient Occlusion (Umgebungsverdeckung). Das Ziel dieser Arbeit ist Ambient Occlusion in den 4-dimensionalen Raum zu erweitern. Es wurde ein direkter mathematischer Ansatz gewählt, um die punkt-basierte ambient occlusion mit Raytracing im 3D-Raum auf den 4D-Raum zu erweitern. Das Ergebnis ist eine 4D-Ambient-Occlusion, die in der Lage ist, volumetrische Oberflächen von 4D-Strukturen zu berücksichtigen.

Abstract

Data visualization is helpful for analyzing abstract data. Useful insights and better understanding can be gained from data visualization. Research data are often complex and high-dimensional. Visualization of high-dimensional data requires the reduction of the data's dimension so that the data can be projected to a 2D or 3D image. This of course leads to some loss of information. Projecting the data to a 4D image allows more information to be retained, however 4D structures are not as intuitive to understand. Moreover, visualization methods in 3D space is far superior than in 4D space. Available 3D graphics algorithms are able to produce realistic, high-quality images.

One method in 3D graphics is ambient occlusion. This thesis aims to extend ambient occlusion into 4-dimensional space. A direct mathematical approach was taken to extend vertex-based ray-traced ambient occlusion in 3D space to 4D space. This results in 4D ambient occlusion that is able to take volumetric surfaces of 4D structures into consideration.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Structure . . . 2

2 Fundamentals **3**
- 2.1 Ray Tracing . . . 3
  - 2.1.1 Ray Generation . . . 4
  - 2.1.2 Ray-Object Intersection . . . 6
- 2.2 Creating Triangle Mesh . . . 8
- 2.3 Ambient Occlusion . . . 9

3 Related Work **12**
- 3.1 Ambient Occlusion . . . 12
- 3.2 Visualization of High-Dimensional Data . . . 13

4 Method **15**
- 4.1 4D Space . . . 15
- 4.2 Ray Generation to a 3D Image Plane . . . 16
- 4.3 Ray-Object Intersection in 4D . . . 17
- 4.4 Ambient Occlusion in 4D . . . 18
- 4.5 Evaluation Methods . . . 21
- 4.6 Implementation Details . . . 21
  - 4.6.1 Structures . . . 22
  - 4.6.2 Rendering a 4D Scene . . . 23
  - 4.6.3 Computing Ambient Occlusion Value . . . 23
  - 4.6.4 Exporting 4D Renders to ParaView . . . 24

5 Results **25**
- 5.1 Experimental Results . . . 25
- 5.2 Theoretical Results . . . 27
- 5.3 Discussion . . . 30

6 Conclusion **32**

Bibliography **33**

1 Introduction

1.1 Motivation

Visualization of abstract data is helpful for analyzing and understanding the abstract data. By representing the data visually, complex relationships can be portrayed in a simple and efficient manner. However, research data are often complex and high-dimensional. To project such data into a 2D or 3D image, the data's dimensionality needs to be reduced. This leads to a loss in information. Reducing the data and visualizing it not as a 3D image, but as a 4D image, will allow more information to be retained in the visualization. This can lead to new discovery of insights and understandings in the data. Example of this can be found in the medical field, where 4D visualization of medical data is becoming more popular as a tool for diagnosis and surgical planning.

However, the 4D space is not as intuitively understood as 3D space. As our world exists in a 3D space, it is difficult to visualize spaces in higher dimensions. However, scenes in 4D space can be projected to the 3D space, which allows for visualization of 4D structures. Despite the availability of 4D visualization methods, the methods of visualization and rendering in 3D are still vastly more advanced than the methods available in 4D. Over the years, many algorithms have been developed to produce photorealistic results Ganovelli et al., 2014.

These algorithms are also being optimized, so that good results can be produced without compromising computation cost. Recursive global illumination using ray tracing can produce shadows, reflections, and indirect lights, but it is very expensive and is not suitable for real-time applications. Algorithms such as ambient occlusion approximates the results of global illumination, but is less expensive. Ambient occlusion gives the rendered image more depth and clearer separation between objects, making the image more realistic ARVIlab, 2018.

Using the rendering and visualization methods available in 3D to 4D may improve the quality of 4D visualization, making it more up to par with 3D visualization. Rendering the 4D space with shadows and reflection can also give a better understanding of the 4D space.

1.2 Objectives

This bachelor thesis builds on preexisting algorithms for 3D rendering and extend these algorithms to the 4D space. More specifically, this thesis will focus on ambient occlusion. The ambient occlusion method used in this thesis uses ray tracing to calculate the occlusion values of each vertex in a scene, in other words vertex-based ray traced ambient occlusion. To view the result of the ambient occlusion, the ambient occlusion value will be used to determine the color of the geometries.

To extend the preexisting algorithms into 4D space, an direct analytical approach was taken. The mathematical concepts behind the algorithms are discussed in detail. Then, the same concepts are then applied in 4D space.

1.3 Structure

In Chapter 2 we discuss the classic ray tracing and ambient occlusion methods in 3D. Chapter 3 introduces other related works regarding ambient occlusion and 4D visualization. In Chapter 4, methods used to extend ambient occlusion into 4D are presented. Implementation details are discussed in this chapter as well. Experimental results of ambient occlusion in 4D are then presented and discussed in Chapter 5. Finally, Chapter 6 gives a conclusion of the thesis.

2 Fundamentals

This chapter discusses the basic rendering algorithm ray tracing in detail. Important concepts in computer graphics, such as ray generation and ray-object intersection are likewise presented. This chapter also discusses the classic ambient occlusion algorithm in 3D.

2.1 Ray Tracing

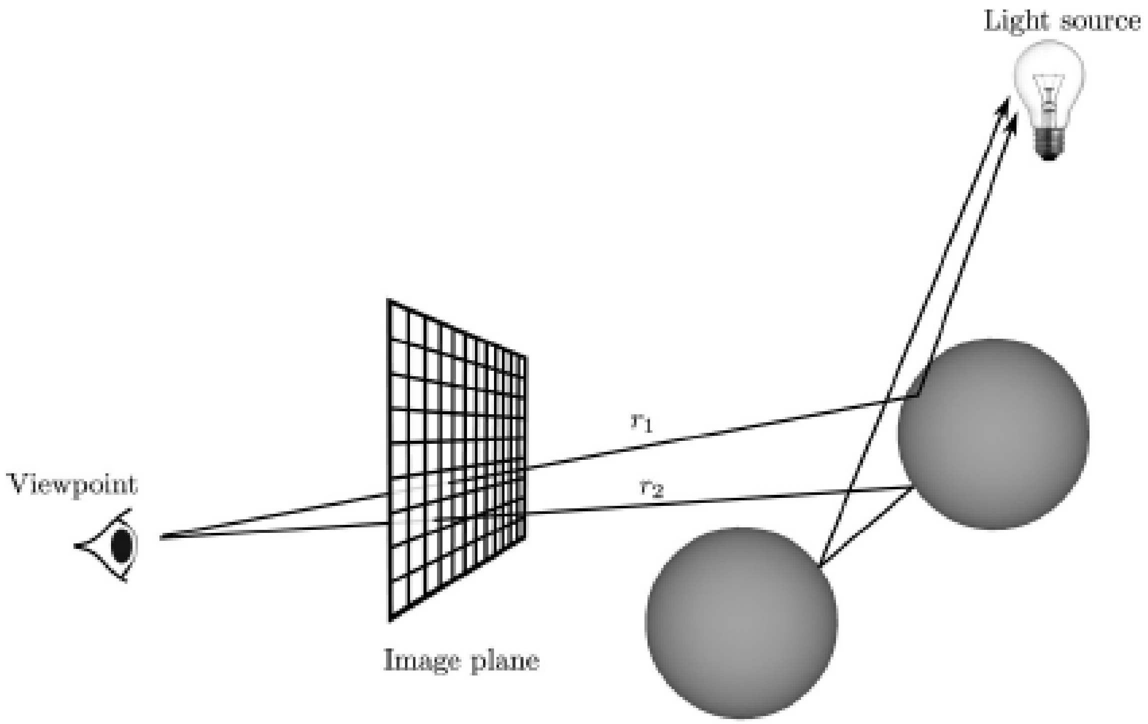


Figure 2.1: An illustration of ray tracing (Ganovelli et al., 2014)

To display a 3D scene, a rendering algorithm such as *ray tracing* is needed. Ray tracing simulates the mechanics of how our eyes see. Our eyes can see the world around us because light rays from light sources travel across space into our eyes.

However, instead of starting at the light source and following the path of the light rays into the eye, ray tracing shoot rays from the eye (viewpoint, or camera) through the image plane and into the scene. We then trace the path of shot rays through the 3D scene. If a ray reaches a light source, we can then compute the color of this ray based on its path through the 3D scene (Shirley et al., 2018).

Algorithm 2.1 A pseudocode of ray tracing (Shirley et al., 2018)

```
for each pixel in image:
  create a ray from viewpoint through the pixel image plane
  find closest ray-object intersection
  compute color of pixel according to the path of the ray
  set pixel color to computed value
```

A *ray* $R = (O, Dir)$ consists of a point of origin O and a directional vector Dir. A ray from the viewpoint through the pixel image plane s has its origin O at the location of the viewpoint. Its directional vector is defined as $Dir = s - e$. To find a point P on a ray R, we use the the parametric line equation in 3D, which is given by:

$$P(t) = O + tDir \tag{2.1}$$

where t is the distance from the origin O. Using this equation, we can find any point p on a ray given a distance t from the viewpoint O. It is important to note that $P = e$ at $t = 0$, and $P = s$ at $t = 1$. More generally, for $0 < t_1 < t_2$, $P(t_1)$ is closer to the viewpoint than $P(t_2)$. For $t < 0$, $P(t)$ is located behind the viewport. These characteristics of t are important when calculating the closest object that intersects the ray that is not behind the viewport (Shirley et al., 2018).

2.1.1 Ray Generation

To render an image, we must first create an image plane. The image plane represents the computer screen on which the image is to be rendered. The image plane is divided into a grid, as illustrated in Figure 2.1. One section of the grid represents one pixel on the computer screen. As described in Algorithm 2.1, we need to generate rays which start at the viewpoint and go through each pixel. In other words, these rays go through each section of the grid on the image plane.

As we previously defined, a ray consists of a point of origin O and a directional vector Dir. The origin O of this ray is user-defined as the viewpoint. The directional vector Dir of this ray needs to be calculated. We have defined the directional vector Dir as the difference between the viewpoint e and a point s on the image plane $Dir = s - e$. Thus, the problem of generating rays that go through the image plane is a problem of finding points s on the image plane (Shirley et al., 2018).

Since the image plane is divided into grids, we can think of this as a coordinate system. This coordinate system is also known as the *pixel coordinates*. It has a range of $[0, w]$ along the x-axis, and $[0, h]$ along the y-axis, with w being the width of the image, and h the height of the image. We can easily identify a point $s_P = (x, y)$ on the image plane, which is also the center of each pixel on this coordinate system (Akeley et al., 2013).

In order to generate a ray to point s_P, we need to transform point s_P into the same space as the viewpoint, which is in the *camera coordinate space*. This takes several steps. First, we need to normalize point s so that it is in the *normalized device space*. The normalized device coordinate has a range of $[0, 1]$ on both the x-axis and the y-axis. We can compute the normalized point $s_N = (x_N, y_N)$ by dividing its x-component with the width, and its y-component with its height. Because each point is on the center of each pixel, it is offset by 0.5 to the left, so we need to subtract 0.5 from both x and y after normalization (Akeley et al., 2013).

$$x_N = x/w - 0.5$$
$$y_N = y/h - 0.5$$

Now s_p needs to be transformed into the *screen space*, which has a range of [-1, 1] along its y-axis, and [-image aspect ratio, image aspect ratio] along its x-axis. Notice that y_N needs to be negated because the origin (point 0,0) in the raster space and the normal device coordinate space is in the top left corner, whereas the origin of the screen space is in the middle of the space. To map the y-coordinate correctly, we need to negate the y-coordinate. We also need to scale the coordinate space with the aspect ratio of the image. To do this we need to multiply the x-component by the image's aspect ratio $s_x = s_x \cdot (w/h)$. The resulting screen space coordinate of point s is then given as

$$s_{screen} = (s_x, -s_y, 0).$$

Finally the screen space coordinates of point s need to be transformed into the camera coordinate space. For this, the viewing angle α of the viewpoint need to be considered. To scale the screen coordinates with the viewing angle of the viewpoint, the screen coordinates need to be multiplied with $\tan(\frac{\alpha}{2})$. The camera coordinates of point s is then given as $s_{camera} = (s_x \cdot \tan(\frac{\alpha}{2}), s_y \cdot \tan(\frac{\alpha}{2}))$ (Akeley et al., 2013).

2.1.2 Ray-Object Intersection

A crucial step in ray tracing is finding the point at which a ray $R(O, Dir)$ intersects with any object in front of the viewpoint. We first solve this problem for triangles, and then use the intersection with triangles to solve the ray intersection with more complex geometries. To compute the ray-triangle intersections, we use the barycentric coordinates of the triangle.

The *barycentric coordinate system* describes the location of a point relative to a simplex, which in the case of a plane is a triangle (Koecher et al., 2007). Given a point $P \in \mathbb{R}^3$ and a triangle with vertices $A, B, C \in \mathbb{R}^3$, point p can be represented as

$$P = \lambda_1 A + \lambda_2 B + \lambda_3 C \tag{2.2}$$

$$\text{with} \quad \lambda_1 + \lambda_2 + \lambda_3 = 1 \tag{2.3}$$

$(\lambda_1, \lambda_2, \lambda_3)$ are the *barycentric coordinates* of point P. According to (Koecher et al., 2007), the barycentric coordinates have the following properties:

1. If the barycentric coordinates are all positive, point P lies inside the triangle;
2. If any of the barycentric coordinates is 0, point P lies on one of the edges of the triangle;
3. If any of the barycentric coordinates is negative, point P lies outside the triangle.

Using these properties, we can check for an intersection between a point p and a triangle. Calculating for the barycentric coordinates requires solving a linear system. Combining Equation 2.2 and Equation 2.3 we get:

$$\begin{aligned}
\lambda_1 + \lambda_2 + \lambda_3 = 1 \Rightarrow \lambda_1 &= 1 - \lambda_2 - \lambda_3 \\
P &= \lambda_1 A + \lambda_2 B + \lambda_3 C \\
&= (1 - \lambda_2 - \lambda_3)A + \lambda_2 B + \lambda_3 C \\
&= A\lambda_2(B - A) + \lambda_3(C - A).
\end{aligned} \tag{2.4}$$

Using the definition of p from Equation 2.1, we get the following equation:

$$O + tDir = A + \lambda_2(B - A) + \lambda_3(C - A) \tag{2.5}$$

$$O - A = -tDir + \lambda_2(B - A) + \lambda_3(C - A). \tag{2.6}$$

Expanding the Equation 2.4 into three equations for each coordinates, we get

$$\begin{aligned} tx_{Dir} + \lambda_2(x_B - x_A) + \lambda_3(x_C - x_A) =& x_O - x_A \\ ty_{Dir} + \lambda_2(y_B - y_A) + \lambda_3(y_C - y_A) =& y_O - x_A \\ tz_{Dir} + \lambda_2(z_B - z_A) + \lambda_3(z_C - z_A) =& z_O - z_A, \end{aligned}$$

which can also be expressed as a linear system

$$\begin{bmatrix} x_{Dir} & (x_B - x_A) & (x_C - x_A) \\ y_{Dir} & (y_B - y_A) & (y_C - y_A) \\ z_{Dir} & (z_B - z_A) & (z_C - z_A) \end{bmatrix} \begin{bmatrix} t \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} x_O - x_A \\ y_O - x_A \\ z_O - z_A \end{bmatrix}. \tag{2.7}$$

Solving this linear system will give us 2 of the 3 barycentric coordinates and the distance t from the viewpoint, at which the ray intersects with the triangle. Finding only 2 of the 3 barycentric coordinates is sufficient because the sum of the barycentric coordinates are equal to 1, as shown in Equation 2.3. Using this property, we can find the third barycentric coordinate given that 2 are known. To solve this linear system efficiently, we can use *Cramer's rule.*

Cramer's Rule (Cramer, 1750): Given a system of n linear equations for n unknowns represented as a matrix multiplication $Ax = b$, the solution $x = (x_1, ..., x_n)^T$is given by

$$x_i \frac{\det(A_i)}{\det(A)} \qquad i\text{=1,...,n},$$

where A_i is the matrix A with its i-th column replaced by the vector b (Cramer, 1750). Using Cramer's rule, the solution to our linear system from Equation 2.7 is given as follows:

$$\begin{aligned} t =& \frac{\det((0-A) \quad (B-A) \quad (C-A))}{\det(Dir \quad (B-A) \quad (C-A))} \\ \lambda_2 =& \frac{\det(Dir \quad (0-A) \quad (C-A))}{\det(Dir \quad (B-A) \quad (C-A))} \\ \lambda_3 =& \frac{\det(Dir \quad (B-A) \quad (0-A))}{\det(Dir \quad (B-A) \quad (C-A))} \end{aligned}$$

Calculating the determinant of a 3×3 matrix can also be done by calculating the cross and dot product of the column vectors.

Using this algorithm, we can find whether a ray intersects a triangle, as well as the distance t between the viewpoint and the ray-intersection. Finding the distance t is important, so that we know which object is closer to the viewpoint. This algorithm can also be used to also check for ray-object intersection of other geometries, as long as these geometries are constructed out of triangles. This representation of 3-dimensional structures using triangles is also known as a *triangle mesh*.

2.2 Creating Triangle Mesh

Creating a triangle mesh out of a complex geometrical structure is non-trivial. The *marching cubes* algorithm introduced by (Doi et al., 1991), which solves the problem of finding surface of discrete 3-dimensional volumes. This algorithm divides an input volume into a set of cubes, each cube defined by its 8 vertices. Each cube is then processed, and checked to see if it contains a section of the input volume. Then, an appropriate triangular mesh is generated, which approximates the input volume of the section contained in the cube. The triangular mesh are generated via look-up table, which consists of all possible configurations for a triangle mesh in a cube. The original marching cubes algorithm by (Doi et al., 1991) gives 15 different possible configurations.

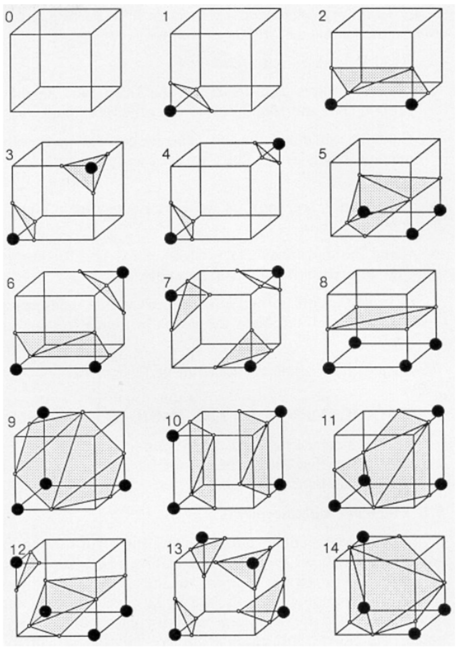


Figure 2.2: 15 configurations of triangle mesh in a cube (Doi et al., 1991).

2.3 Ambient Occlusion

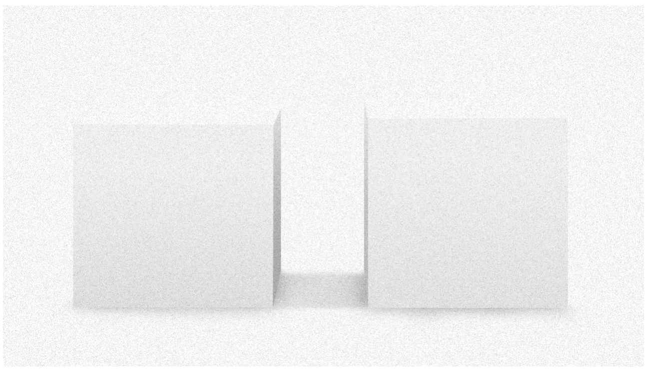


Figure 2.3: An rendering of ambient occlusion in a scene with 2 cubes

Ambient occlusion is a shading technique that simulates shading that exists on surfaces, even if the entire scene is well lit. This shading technique does not take light sources into consideration, but only other geometries in the scene. Ambient occlusion is an approximation of global illumination. It is much cheaper to compute the ambient occlusion values than to recursively calculate the global illumination using ray tracing. Ambient occlusion is usually rendered separately before being combined with the normal ray-traced render of a scene. Shading using ambient occlusion darkens the edges of different geometries. This gives a clearer separation between different geometries in the scene.

The occlusion value is obtained by integrating the visibility function $V(p, w)$. This function returns 1 should ray (p, w) not intersect with any object in the scene, and 0 otherwise. Practically this intergral is computed using Monte-Carlo approximation. This is done by *hemispheric sampling*. For each point p on the scene, a hemisphere Ω is constructed. Rays are then shot from point p to points w in the hemisphere to check for intersection with objects in the scene. This can also be written as the following function

$$A(p, n) = \frac{1}{\pi} \int\limits_{w \in \Omega} V(p, w) |w \cdot n| dw \tag{2.8}$$

where $V(p, w)$ is the visibility function. The integral value is practically implemented using Monte-Carlo integration.

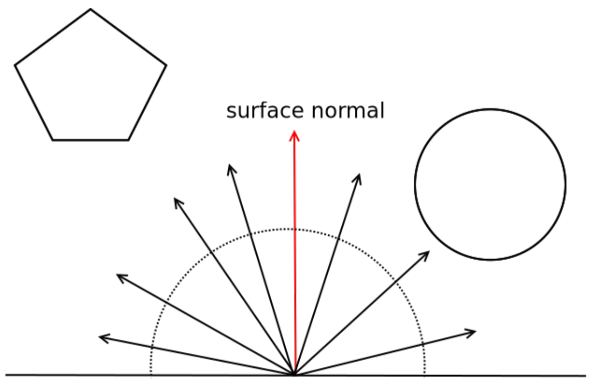


Figure 2.4: An illustration of ambient occlusion calculation on a point P on a surface. Notice that the sampled rays are facing towards the same direction as the surface normal. None are pointing into the surface.

To sample points on a hemisphere Ω, we can use the spherical coordinates, which describe a point $P \in \mathbb{R}^3$ using two angles $\phi_1 \in [0, 2\pi]$ and $\phi_2 \in [0, \pi]$ with regard to a sphere Ω with a center point c and radius r as follows:

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} c + r\cos(\phi_1) \\ c + r\sin(\phi_1)\cos(\phi_2) \\ c + r\sin(\phi_1)\sin(\phi_2) \end{bmatrix} \tag{2.9}$$

Using this spherical coordinates, we can sample points on a hemisphere by selecting 2 random values for $\phi_1 \in [0, 2\pi]$ and $\phi_2 \in [0, \pi]$. This will give us any random values on a sphere centered on our selected point P. However, this may result in many sampled rays pointing into the geometrical structure, and not out of its surface and away from the structure. To avoid this, the sampled rays need to be pointing at the same direction as the surface normal. The dot product of two vectors is positive, if the angle formed between them is less than $90°$. Using this property, the direction of the sampled ray can be verified by calculating the dot product of the sampled ray and the surface normal, and checking that this dot product is positive (Bär, 2018).

3 Related Work

There have been many research done in ambient occlusion, as well as visualization of high-dimensional data. This chapter presents related works regarding visualization of high-dimensional data, specifically 4D visualization, as well as works in ambient occlusion. Section 3.1 discusses various methods for calculating or approximating ambient occlusion. Section 3.2 discusses related works in visualization of high-dimensional data.

3.1 Ambient Occlusion

Screen Space Ambient Occlusion (SSAO) was developed by Crytek and was used in the video game engine *CryEngine 2* in 2007, which was used in the video game *Crysis* (Mittring, 2007). This variant of ambient occlusion is the first method that allows ambient occlusion computation in real time. Using SSAO in *Crysis* made its graphics noticably more realistic tha n other video games of its time (Hayden, 2019). SSAO implements ambient occlusion as a pixel shader, and uses the scene's depth buffer (z-buffer) to approximate the occlusion value on every pixel on the screen. This requires heavy sampling to create good results, which in this case requires reading many textures for each pixel. This is very costly and needs further optimization to perform well. As SSAO computes the ambient occlusion values based only on pixels of the final image, SSAO does not depend on scene complexity (ARVIlab, 2018). This makes SSAO efficient with dynamic scenes. Moreover, it can fully be implemented in the GPU to further accelerate the calculation. However, SSAO only takes the current visible scene into consideration. Occlusion caused by objects that are currently not in the same frame are not generated.

Screen space directional ambient occlusion (SSDO) introduced in (Ritschel et al., 2009) extends the approximation introduced in SSAO and uses it not only to calculate an occlusion value, but also to calculate directional shadows and indirect color bleeding. This is done by including one indirect bounce of ambient light into the occlusion calculation. The resulting image is more vibrant, as it includes ambient lightning in the shadows. An example of SSDO can be seen in Figure 3.1.

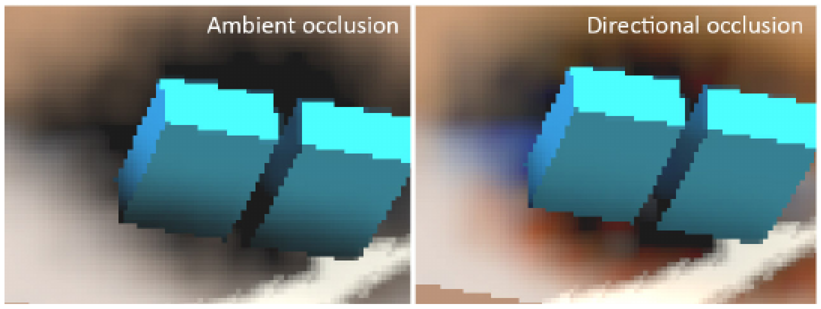


Figure 3.1: Results from SSAO (left) and SSDO (right). Note the color in the shadows from SSDO, giving it a more realistic result Ritschel et al., 2009.

Real-time ambient occlusion using ray tracing (also known as *ray-traced ambient occlusion*, RTAO) is only available since 2018, and was developed for use with Nvidia's RTX platform, which allows for real-time ray tracing (Nvidia Game Works, 2018). Using ray tracing to create ambient occlusion addresses the limitations in SSAO, as ray tracing is not limited to only visible objects, but all objects in a scene. The results of RTAO is more correct than SSAO. RTAO also produces higher quality images.

3.2 Visualization of High-Dimensional Data

As our world is 3-dimensional, we cannot directly observe structures outside of the 3-dimensional space. This makes it difficult to visualize 4-dimensional structures. Studies of the difficulties of 2-dimensional beings to understand the 3-dimensional structures gave us insights on how to understand and visualize 4-dimensional structures.

The problem of viewing 4D structures was first explored in (Noll, 1967). Noll's solution to rendering 4D structures is by using wireframes with perspective projection. This visualization does not give a sufficient understanding to the viewer. (Hollasch, 1991) extends the wireframe method to include more visual cues, as well as introduces ray tracing methods to visualize 4D structures.

Another method of visualizing 4D space are by rendering hypersphere "peels". This resulted in good quality images of rotation in 4D space (Banchoff et al., 1999). 4D structures can also be visualized by producing 3-dimensional slices of the structure. This is based on the idea of how 2-dimensional beings would see 3-dimensional structures. This is presented in (Banchoff et al., 1999).

An approach which is also used in this thesis is to render 4D objects as a 3D image. (Steiner et al., 1987) used scanplane conversion to project 4D objects to 3D voxel fields. The resulting voxels are then rendered as semi-transparent so that the projected 4D object is visible.

4 Method

In this chapter we build on top of the rendering and shading algorithms explained in chapter 2, extending them to work in a 4-dimensional space. Section 4.1 gives an explanation of 4-dimensional spaces. Section 4.2 discusses the projection of 4-dimensional scenes to a 3D image plane, and how to generate camera rays. Section 4.3 extends the ray-triangle intersection to 4D. Finally, Section 4.4 discusses the challenges of calculating ambient occlusion in a 4-dimensional space, and the solution for each of these problems

4.1 4D Space

The 4-dimensional space has 4 basis vectors which are orthogonal of one another. There is a one more rotational axis in a 4-dimensional space. This makes the four-dimensional structures much more complex than their 3-dimensional counterparts. In the 4-dimensional space, each point p consists of 4 coordinates $p = (x, y, z, w)$. The fourth variable w is often interpreted to be time, but this is not necessarily the case.

As previously discussed in Section 2.1.2, 3-dimensional structures are modelled using triangle mesh to solve the problem of ray-object intersection. This mesh only take the surface of a 3D structure into consideration. Extending this to a 4-dimensional space, we can model 4-dimensional structures using a mesh that only takes the 4D structure's surface into consideration. Surfaces of 3-dimensional structures are 2-dimensional planes, and extending this idea to a 4-dimensional structure, the surfaces of a 4-dimensional structure are 3-dimensional volumes.

4.2 Ray Generation to a 3D Image Plane

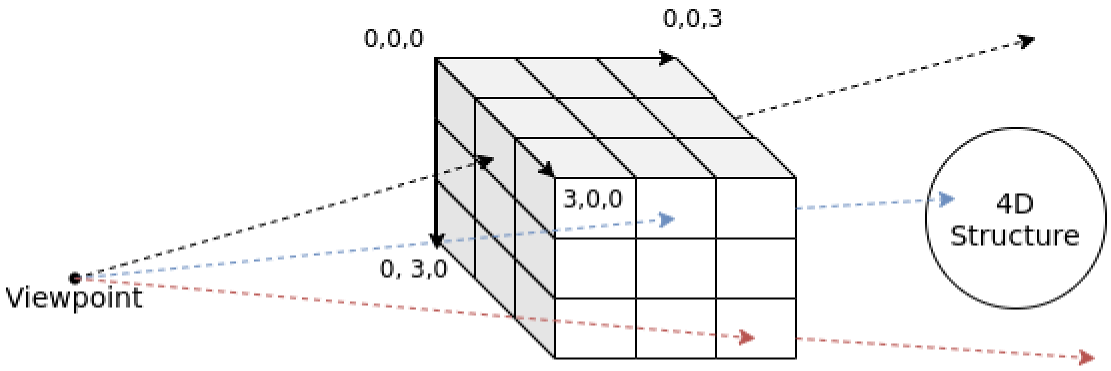


Figure 4.1: A $3 \times 3 \times 3$ 3D image plane. Notice that the blue ray is shooting through the center of voxel $(2, 0, 1)$, and the red ray is shooting through the center of voxel $(2, 2, 2)$.

To render an image in 4D we need to project the image to an image plane. Whereas in 3D space, the image plane is a 2D plane, in the 4D case, the image plane is a 3D image plane. This means that the image plane is not flat, but has depth. Analogous to the 3D case, we can create a pixel grid and project it on this 3D image plane. Instead of squares, each pixel is now mapped to a voxel in the 3D image plane. This means that each pixel value now includes a volumetric information.

In the 3D case, we generate a ray by finding a point s on the image plane. We do this by moving along the x and y axes of the pixel coordinates and transforming the pixel coordinates into camera coordinates. In 4D, we need to also move along the z axes while we keep the w-coordinate constant. The steps to finding a point s is otherwise similar to the 3D case: we first normalize all coordinates, and offset them, as is explained in Section 2.1.1. In 4D space, however, we need to do this also for the z-coordinate. Afterwards, we need to transform the point coordinates into screen space coordinates. Just like in the 3D case, the x-coordinate is scaled to image's aspect ratio. However, instead of negating the y coordinate like in the 3-dimensional case, it is the z-coordinate that needs to be negated.

The screen space coordinates of the ray directions also need to be transformed into camera coordinates. For the scope of this thesis, we only look at the case of a static camera position. Because the camera position is static, its position can be determined so that the screen space coordinates of the ray directions are the same as their camera coordinates. This can be done by placing the camera at point (0, 0, 0, 1).

4.3 Ray-Object Intersection in 4D

Similarly to the 3-dimensional case, we also solve the problem of ray-object intersection in 4-dimensional space by breaking the object into simpler geometries. In the 3-dimensional case, triangles are used, which is a 2-simplex. Extending this idea to the 4-dimensional case, a 3-simplex, which is a tetrahedron, is used instead. As a tetrahedron is also simplex, we can calculate the barycentric coordinates of a point with regards to a tetrahedron. The representation of a point $P \in \mathbb{R}^4$ in a barycentric coordinate system based on a tetrahedron with vertices $A, B, C, D \in \mathbb{R}^4$ is defined as:

$$P = \lambda_1 A + \lambda_2 B + \lambda_3 C + \lambda_4 D \tag{4.1}$$

$$\text{with} \quad \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1. \tag{4.2}$$

Analogous to the steps we toook in Section 2.1.2, we can combine the definition of p from Equation 2.1 with Equation 4.1:

$$\begin{aligned} O + tDir &= \lambda_1 A + \lambda_2 B + \lambda_3 C + \lambda_4 D \\ &= A(1 - \lambda_2 - \lambda_3 - \lambda_4) + \lambda_2 B + \lambda_3 C + \lambda_4 D \\ &= A + \lambda_2(B - A) + \lambda_3(C - A) + \lambda_4(D - A) \\ O - A &= -tDir + \lambda_2(B - A) + \lambda_3(C - A) + \lambda_4(D - A) \end{aligned}$$

This gives us a similar linear system like in the 3-dimensional case, which can also be solved using the Cramer's rule. Since O, Dir, A, B, C, D are now points in a 4-dimensional case, consisting of 4 components (x, y, z, w). This yields us 4×4 matrices in our linear system

$$\begin{bmatrix} x_{Dir} & (x_B - x_A) & (x_C - x_A) & (x_D - x_A) \\ y_{Dir} & (y_B - y_A) & (y_C - y_A) & (y_D - x_A) \\ z_{Dir} & (z_B - z_A) & (z_C - z_A) & (z_D - x_A) \\ w_{Dir} & (w_B - w_A) & (w_C - w_A) & (w_D - x_A) \end{bmatrix} \begin{bmatrix} t \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{bmatrix} = \begin{bmatrix} x_O - x_A \\ y_O - x_A \\ z_O - z_A \\ w_0 - w_A \end{bmatrix}. \tag{4.3}$$

The solution of this linear system is given as follows:

$$\begin{aligned}
t =& \frac{\det((O-A) \quad (B-A) \quad (C-A))}{\det(Dir \quad (B-A) \quad (C-A) \quad (D-A))} \\
\lambda_2 =& \frac{\det(Dir \quad (O-A) \quad (C-A))}{\det(Dir \quad (B-A) \quad (C-A) \quad (D-A))} \\
\lambda_3 =& \frac{\det(Dir \quad (B-A) \quad (O-A))}{\det(Dir \quad (B-A) \quad (C-A) \quad (D-A))} \\
\lambda_4 =& \frac{\det(Dir \quad (B-A) \quad (C-A) \quad (O-A))}{\det(Dir \quad (B-A) \quad (C-A) \quad (D-A))}
\end{aligned}$$

.

Because we are now calculating the determinant of 4×4 matrices, we can no longer use the cross and dot products of the column vectors to calculate the determinant. However, since we only need to calculate the determinant of matrices with a set number of columns and rows, the simplest solution is to use the Laplace extension. This reduces the calculation of a determinant into simple float multiplications, additions, and subtractions.

Just like in the 3D case, we can use this algorithm to also calculate the ray-object intersection of other, more complicated structures in 4D. The prerequisite of this is to represent other structures in 4D as constructed out of tetrahedra, thus creating a *tetrahedral mesh*.

4.4 Ambient Occlusion in 4D

To perform ambient occlusion in 4D, we need to solve hemispheric sampling in 4D. Furthermore, ambient occlusion in 4D differs from the 3D case in that the ambient occlusion values are calculated for volumetic surfaces instead of planar surfaces. This section will present the approaches taken to solve these problems.

The 4D equivalent of a sphere in 3D space is a 3-sphere, also known as a hypersphere. We can use the spherical coordinates as defined in Equation 2.3 and extend this to 4D. The generalized spherical coordinates for n-sphere is given as follows:

$$p = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} r\cos(\phi_1) \\ r\sin(\phi_1)\cos(\phi_2) \\ r\sin(\phi_1)\sin(\phi_2)\cos(\phi_3) \\ \vdots \\ r\sin(\phi_1)\dots\sin(\phi_{n-2})\cos(\phi_{n-1}) \\ r\sin(\phi_1)\dots\sin(\phi_{n-2})\sin(\phi_{n-1}) \end{bmatrix}. \tag{4.4}$$

We can then specifically define for the case $n = 4$ and a centerpoint c:

$$p = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} c + r\cos(\phi_1) \\ c + r\sin(\phi_1)\cos(\phi_2) \\ c + r\sin(\phi_1)\sin(\phi_2)\cos(\phi_3) \\ c + r\sin(\phi_1)\sin(\phi_2)\sin(\phi_3) \end{bmatrix}. \tag{4.5}$$

In compared to the 3D case, we now need one extra angle. Similarly to the 3D case we need to make sure that the sampled points are in the same direction as the the surface's normal vector.

As the surfaces in 4D are made out of tetrahedron, the calculation for the normal vector needs to be extended to 4D. In the 3D case, the normal vector of a plane can be calculated by the cross product of two vectors on said plane. The cross product is not defined for 4D, however the cross product can also be formulated as the formal determinant.

$$a \times b = \begin{vmatrix} x & y & z \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

Using cofactor expansion on the first row, we will get the components of the resulting vector, which is the same as the conventional methods to calculate the cross product of two 3-dimensional vectors (Bär, 2018).

We can extend this determinant into 4D space. However, this is not a cross product in the conventional sense. This is because this "product" is a product of 3 vectors instead of just 2. It is not possible to find an orthogonal vector to only 2 vectors in 4-dimensional space as this will result in an unsolvable linear system.

$$a \times b \times c = \begin{vmatrix} x & y & z & w \\ a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \end{vmatrix}$$

Solving this determinant will result in a vector that is orthogonal to the vectors a, b, c.

Returning to the problem of finding a normal vector that is orthogonal to the surface of a 4-dimensional structure, we need to find an orthogonal vector to a volumetric surface. To do this, we need to find 3 orthogonal vectors within this volume. In the case of a tetrahedron $T = (v0, v1, v2, v3)$, we can choose an anchor vector $v0$ and select the edges from $v0$ to the other 3 vectors $v1, v2, v3$.

$$\begin{aligned} a &= (v1 - v0) \\ b &= (v2 - v0) \\ c &= (v3 - v0) \end{aligned}$$

This is the direct extension of the 3D case. In the 3D case, 2 edges on a triangle, which makes up the surfaces of 3D structures in computer graphics, are used to find the surface normal.

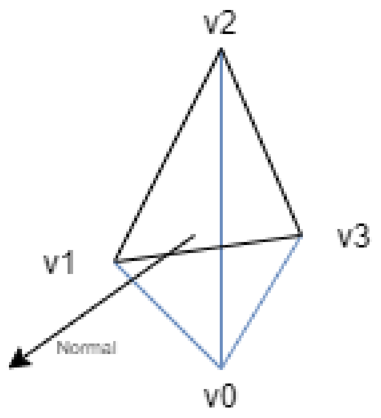


Figure 4.2: 3 edges needed for the calculations of a tetrahedron's normal.

The variant of ambient occlusion implemented in this thesis is ray-traced vertex-based ambient occlusion. This method uses ray-tracing to calculate the occlusion values, and thus takes the entire scene into consideration. The occlusion values are calculated for each vertex in the scene. The occlusion values are then averaged over a tetrahedron during rendering.

4.5 Evaluation Methods

To evaluate the results of ambient occlusion in 4D, several comparisons are made. To provide ground truths for comparisons, the classical ambient occlusion in 3D was rendered. The variant ambient occlusion used for the 3D case is analogous to the one used in 4D: a vertex-basesd, ray-traced ambient occlusion. Because of hardware limitations, only simple scenes consisting of two geometries are rendered. So that the results of ambient occlusion in 3D are comparable to the results of ambient occlusion in 4D, the simple scenes used in 3D are then extended into 4D. This is also done because it is otherwise very difficult to create 4D meshes.

Moreover, simple 4D scenes are rendered with and without ambient occlusion to show the impact of ambient occlusion algorithms on 4D scenes. Simple scenes in this case means that the scenes consist of structures created by putting several tetrahedra together. These structures do not represent any particular 3D structures. The effects of ambient occlusion in 4D is observed by variating the sample values. The effects of the image plane's resolution, more specifically increasing image plane's depth, is also observed.

4.6 Implementation Details

The implementation of ambient occlusion in 4-dimensional space is done in C++ using Visual Studio as the integraged development environment of choice. As rendering 4-dimensional space is more computationally exepensive, the need to accelerate the computation time rose. To accelerate the computation time, OpenCL (Khronos Group, 2020) is used to provide a framework for parallel computing. The OpenCL kernel is used to compute the ambient occlusion values for each vertex in both the 3-dimensional case, as well as the 4-dimensional case.

Even though the implementation is mainly written in C++, the OpenCL kernel language is based on C, and is not compatible with structures from C++ , such as classes and vectors. Thus, objects are instead implemented as C structs using native OpenCL vector datatypes cl_float3 and cl_float4.

The method used to render 4-dimensional space results in a 3D volume data. This volume data need further processing so that it can be visualized. The generated 3D volume data in this thesis is visualized using ParaView, an open source scientific visualization tool (Sandia National Laboratories, 2021).

4.6.1 Structures

In this section, details are given for different structures used in the implementation.

Ray represents a ray in 3-dimensional space.

- cl_float3 origin: the origin of the ray
- cl_float3 dir: the direction of the ray

Ray4 represents a ray in 4-dimensional space. Ray4 has the same attributes as Ray, but consists of 4-dimensional vectors instead of 3-dimensional vectors.

- cl_float4 origin: the origin of the ray
- cl_float4 dir: the direction of the ray

cl_float3 and cl_float4 are vector structures from OpenCL. These are used so that the structures are compatible with OpeCL, and computations can be parallelized. This vector structure is also used for the vertices of the triangular and tetrahedral mesh.

TriangleMesh contains information of all the geometries in a scene.

- cl_float3 *vertices: an array that contains all vertices in a scene in a 3-dimensional space
- int face: the number of faces in the scene
- int *vertIndex: a list indices that corresponds to the vertices array. Each 3 indices belong to a triangle

TetraMesh contains information of all the geometries in a scene.

- cl_float4 *vertices: an array that contains all vertices in a scene in a 4-dimensional space
- int vol: the number of volumes in a scene
- int *vertIndex: a list indices that corresponds to the vertices array. Each 4 indices belong to a tetrahedron.

4.6.2 Rendering a 4D Scene

To render the 4-dimensional scene to a 3-dimensional image plane, we need to create a way to traverse all the voxels in the 3-dimensional image plane. The approach taken in this thesis is to save the values of this 3-dimensional image plane in a simple 1-dimensional array the size of the image width × image height × image depth. We can go through this array, generate a camera ray, and assign each array element with the appropriate RGB values according to the camera ray-object intersection. To generate a camera ray, we need to convert the index of a 1-dimensional array to its corresponding pixel coordinate. This can be done using simple module, division, and subtraction operations.

Algorithm 4.1 Converting index of a 1-dimensional array to pixel coordinates

```
 for (int i = 0; i < width * height * depth; i++){
         int x = i % width;
         int z = i / (width * height);
         int y = (i - (z * width * height)) / width;

         //calculate camera ray
         //calculate RGB values
}
```

This approach avoids using multiple nested for-loops, one for each dimension. When using nested loops, the indices do not need to be converted into pixel coordinates, as they are already in pixel coordinates. However, the indices need to be converted into a 1-dimensional array index afterwards. Using a single loop and converting to pixel coordinates avoids complex nested loops, and allows for simpler access to each voxel element. This is beneficial, as rendering 4D scenes is already computationally heavy.

4.6.3 Computing Ambient Occlusion Value

As this thesis aims to produce only static scenes as opposed to calculating ambient occlusion in real time, ambient occlusion values can be pre-computed before rendering the scene and saved as a scalar value for each vertices in the scene. The ambient occlusion values can be stored in a simple array similar to how vertices are saved in a mesh. This array can be easily saved as a binary file. Measures are taken to ensure that the index of the vertex and its corresponding ambient occlusion value are kept the same, so that the right ambient occlusion value is assigned to the right vertex when reading the ambient occlusion values.

The ambient occlusion value is a scalar value saved as a float. To render this, the ambient occlusion value is multiplied by the triple $(1, 1, 1)$ to convert this scalar value into pixel RGB values. This creates an RGB value with equal value for all components. This produces a gray color, which is appropriate for shadows (occlusion) that are generated by ambient occlusion.

The ambient occlusion values are saved for each vertex in the scene; however, rendering does not render each vertex of a scene, but the surfaces at ray intersection points. To evaluate the ambient occlusion value of this surface, the ambient occlusion value of its vertices are averaged. This means that for each tetrahedron, the ambient occlusion value is the average ambient occlusion value of its vertices.

4.6.4 Exporting 4D Renders to ParaView

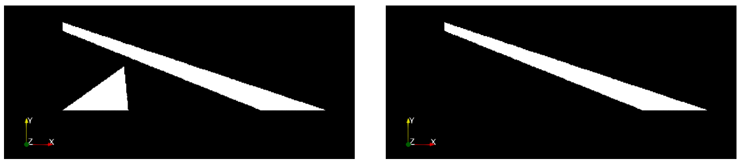
The resulting 3-dimensional image plane is saved as an array of 3-dimensional float vector representing the RGB values of each voxel. This array is then exported into a binary .RAW file, which can be read into ParaView using the ImageReader. To render the images correctly, the data scalar type should be set to float with data byte order LittleEndian. As the color of each voxel is saved as RGB triples, the scalar components should be set to 3. The image is then converted into volume data, so that the resulting 3D image can be throughly inspected. For clarity. the images are rendered against a white background. Because the background is not part of the scene, there should not be any ambient occlusion caused by the background and the test structures. There should only be occlusion caused by the test structures themselves. Any shadows created against the background should thus be attributed to ParaView's rendering engine, and not as the result of ambient occlusion calculations. As this thesis is only concerned with shadows and occlusion, the images are rendered in black and white, based on their occlusion values, white representing an area with no occlusion, and black representing an area with high occlusion value. This allows for an intuitive understanding of the resulting images.

5 Results

This chapter presents the implementation details, as well as experimental results. Section 4.1 details the different frameworks, structures, and functions used in the implementation. Section 4.2 then presents the results of ambient occlusion in the 4-dimensional space, and provide a comparison with ambient occlusion in the 3-dimensional space.

5.1 Experimental Results

To verify the correctness of the ambient occlusion implementation in 4D, a simple 4D scene with 2 tetrahedra was rendered. This scene was projected to a $50 \times 50 \times 2$ 3D image plane. A low resolution was used for this initial test so that fixes can quickly be made in the implementation. The result can be seen in Figure 5.1. As this image plane had only a depth of 2, the image is almost flat. The result of ambient occlusion can be seen in the smaller tetrahedron that is occluded by ambient occlusion. It is worthy of note that the tetrahedron on this projection is entirely occluded - this is not the expected result of ambient occlusion.



(a) without ambient occlusion (b) with ambient occlusion

Figure 5.1: A simple scene in 4D with 2 tetrahedra, projected to a $300 \times 300 \times 2$ 3D image plane

The next step is to increase the resolution of the image plane. The same image is once again rendered, but this time to a $50 \times 50 \times 50$ image plane. In a higher resolution with regards to the image depth, we can get a better understanding of the volume of the two tetrahedra. Here we can see that the smaller tetrahedron is also entirely occluded, like in the lower resolution result from Figure 5.1. The gray values visible in the resulting images are created by ParaView's rendering engine. This does not reflect the calculated ambient occlusion values. This includes the sections of the structures that have become transparent after ambient occlusion is applied. In these cases the ambient occlusion value is 1, the maximum possible value. This means that this section has been entirely occluded. Taking a slice of the scene with ambient occlusion, we can see that the smaller tetrahedron is entirely occluded despite it being rendered as semi-transparent in ParaView.

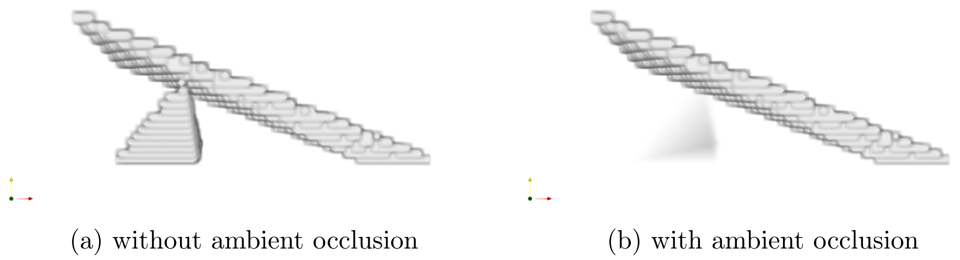

(a) without ambient occlusion (b) with ambient occlusion

Figure 5.2: A simple scene in 4D with 2 tetrahedra, projected to a $50 \times 50 \times 50$ 3D image plane

Ambient occlusion is applied to another scene. In this 4-dimensional scene, two separate geometries, each consisting of two tetrahedra, are rendered with and without ambient occlusion. Again, we see that the left object is entirely occluded, becoming more like a shadow, whereas the right is not. This 4-dimensional structure has a convex side and a concave side.

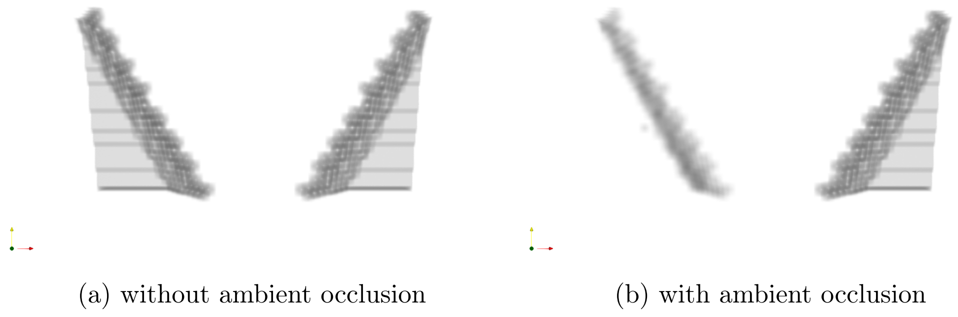

(a) without ambient occlusion (b) with ambient occlusion

Figure 5.3: A simple scene in 4D with 2 tetrahedra, projected to a $50 \times 50 \times 50$ 3D image plane

Another test scene is also rendered with two cubical geometries with a cut that extends along the fourth axis, the w-axis. Like the previous structures, these structures also have concave and convex sides when rendered to a 3D image plane. As with the other two test scenes rendered, ambient occlusion seems to affect the structure on the left more than the structure on the right. Similar to the first test scene of two tetrahedra (Figure 5.1), the left structure have also been entirely occluded, which causes ParaView to render it with transparency. However, one tetrahedron of this structure remained opaque. This includes a convex geometry, which have the expected high ambient occlusioin value, as it is surrounded by other surfaces.

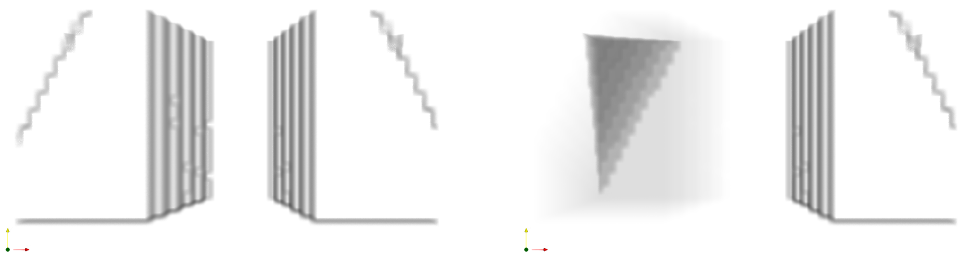

(a) without ambient occlusion (b) with ambient occlusion

Figure 5.4: A simple scene in 4D with 2 tetrahedra, projected to a $50 \times 50 \times 50$ 3D image plane

This is not the expected outcome of ambient occlusion in a 4-dimensional space. Ambient occlusion should be consistently affecting all geometries in a scene. Thus, if one object becomes more transparent, other objects in the scene should also become more transparent. Moreover, ambient occlusion should not occlude an entire structure.

5.2 Theoretical Results

In this section, the experimental results of ambient occlusion in 4D are presented. To validate the methods of ambient occlusion in 4D, a comparison with the classical ambient occlusion in 3D is given. Equivalent geometries are rendered in 3D and 4D so that the results are comparable.

The expected results of ambient occlusion should more resemble the ambient occlusion in 3D space. If we take a simple example of ambient occlusion in 3D space of two cubes, we see that only the edges of the cube are occluded, as pictured in Figure 5.5. In this example, a floor was also rendered as part of the scene. This creates a shadow between the two cubes. We also see that the front sides of the cubes are a little bit occluded because of the floor, creating a gradient on the cubes' faces. This shows us that ambient occlusion should result in a larger variance of values. The experimental results of ambient occlusion in 4D space do not have this characteristic gray values and gradients. Surfaces are either entirely occluded, or not occluded.

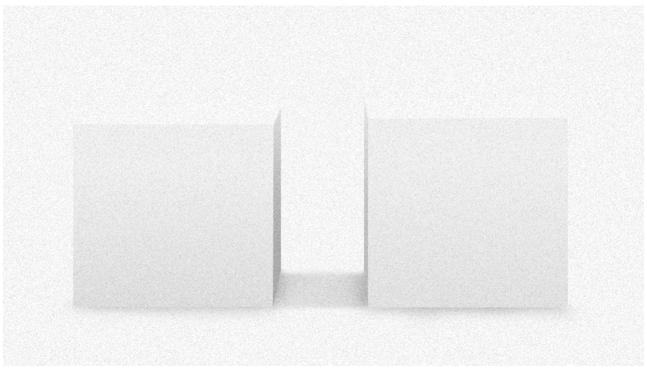


Figure 5.5: Ambient occlusion of 2 cubes in 3D

To look at what possible ambient occlusion results in 4D, we can take a look at each examples and theoretically analyze them. Starting with the simples example of 2 tetrahedra in a scene, Figure 5.6 shows the construction of the scene. One tetrahedron is marked blue, and the other red. We see here, that the red tetrahedron extends beyond the image plane. Intuitively, we can conclude thata vertices 1, 2, and 3 should have the highest ambient occlusion values. Vertices 4,7, and 5 should have the highest ambient occlusion values. The ambient occlusion values of vertices 0 and 6 should be somewhere in the middle.

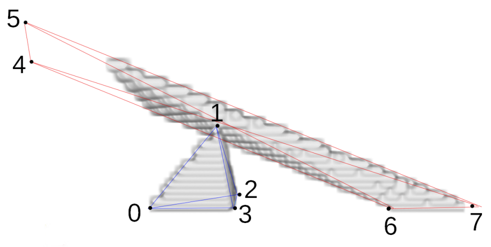


Figure 5.6: Construction of the scene from Figure 5.1

This means, the average ambient occlusion value of the blue tetrahedron should be high, resulting it being more occluded than the red tetrahedron. Because of the ambient occlusion value of vertex 6, the red tetrahedron should also be slilghtly occluded. The experimental results fullfills the expectation that the blue tetrahedron should be more occluded than the red tetrahedron, however, it should not be entirely occluded like in the experimental result, as vertex 0 does not have a high occlusion value. Moreover, the red tetrahedron should actually also be occluded, unlike in the experimental result, which does not occlude the red tetrahedron at all.

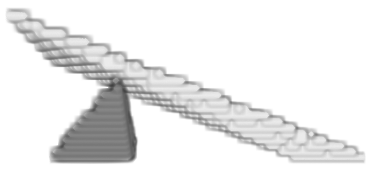


Figure 5.7: Expected results of 4D ambient occlusion on caluclated manually

We can apply this analysis to the other examples as well. This is the scene with two structures created out of two tetrahedra. The construction of this scene is pictured in Figure 5.8. Here we see the two different tetrahedra that makes up the two structures: a red tetrahedron and a blue tetrahedron. We see here that the red tetrahedron extends beyond the image plane and thus cut off.

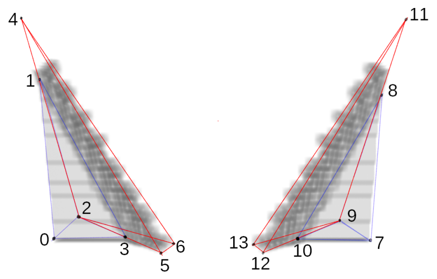


Figure 5.8: Construction of the scene from Figure

Intuitively, we see that the vertices 1, 3, 8, and 10 should have the highest ambient occlusion values. Because the ambient occlusion value of a tetrahedron is the average of the value of its vertices, we can determine the ambient occlusion value of each tetrahedron. As the vertices 1, 3, 8, and 10 belong to the blue tetrahedron, they should have a higher ambient occlusion value, making them darker. In contrast, the red tetrahedra each only contains one vertex with high ambient occlusion value. This should make the red tetrahedron have a higher average ambient occlusion value, making them lilghter. This theoretical result can be seen in Figure 5.9.

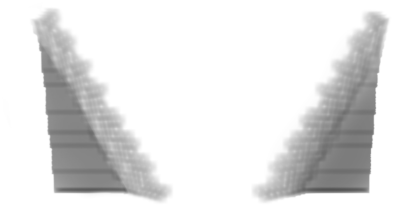


Figure 5.9: Expected results of 4D ambient occlusion caluclated manually

As we can see, this theoretical analysis gives the opposite result when compared to the experimental result. None of the tetrahedron is entirely occluded, and both tetrahedra are equally affected by ambient occlusion. In compared to 3D ambient occlusion, we can see that ambient occlusion in 4D takes volume surfaces into consideration, not planar surfaces as in the case in 3D space. This means, occlusion occurs in volumes. A section of a volume can be occluded in 4D.

5.3 Discussion

It is possible to extend ambient occlusion algorithm into 4D, however, the experimental results are different from the expected results when compared with the results of ambient occlusion in 3D. In the experimetal results, tetrahedra are either entirely occluded, or not at all, which is uncharacteristic of ambient occlusion. Thus, a theoretical result is also presented, which shows the possible results of ambient occlusion in 4D.

However, even the theoretical result does not fully show the results of ambient occlusion in 4D. The examples given in this thesis show the results of ambient occlusion of possible tetrahedral mesh. However, the tetrahedral mesh does not represent the surfaces of any 4D structures, as it does not enclose a 4D structure. As such, characteristics of ambient occlusion in 4D space cannot be fully observed.

To fully understand the effects of ambient occlusion in 4D structures, a tetrahedral mesh that represents the surface of 4D structures is needed. Finding this tetrahedral mesh is non-trivial, as 4D structures are already complex and difficult to visualize. In the 3D case, *marching cubes* can be used to find the surface of discrete volumes, as discussed in Section 2.2.

An extension of the marching cubes algorithm in higher dimensions was introduced in (Bhaniramka et al., 2000). The extended marching cubes algorithm by (Bhaniramka et al., 2000) generates a d-simplex mesh for dimensions $d \geq 3$. Instead of using cubes, this algorithm uses hypercubes. The size of the look-up table is defined as 2^{2^d}, with d being the desired dimensions. In 4D, this implementation takes 4D functions which represents 4D structures, and generates the tetrahedral mesh which represents the surface of the input function. As the proposed algorithm aims to work not with a specific dimension d, but for all $d \geq 3$, the look-up table was not provided. Using this algorithm to generate tetrahedral meshes would require that the look-up table is first calculated, which is also non-trivial. (Bhaniramka et al., 2000) developed a convex-hull-based algorithm to generate the look-up table using hypercubes in higher dimensions. The implementation of this algorithm in C++ is provided on this website.

Since this method uses a look-up table the size of 2^{2^d}, the look-up table is considerable larger than what is used in the original marching cubes algorithm, and scales exponentially with the dimension. Even for the 3D case $d = 3$, which would already require a table the size of 2^8. In the case of 4D, the required size of the look-up table is then 2^{16}.

Moreover, it is not easy to integrate the extended marching cubes implementation with the ambient occlusion implementation in C++ and OpenCL. As previously stated, OpenCL is based on C, and not C++. Class structures which are available in C++ are not compatible with OpenCL. The low-level implementation of OpenCL is also hardware dependant. This means, the way OpenCL behave may change depending on the hardware. On the hardware used in this thesis, OpenCL is very limited. Writing comments in the OpenCL kernel code resulted in compilation error. This made reading and debugging the OpenCL code more difficult.

Because the complexities of marching cubes is beyond the scope of this thesis, it was not implemented. However, using the marching cubes to obtain good test data would be very beneficial in verifying the proposed algorithm of ambient occlusion in 4D, as well as in analyzing the characteristics of ambient occlusion in 4D. A tetrahedral mesh generated using the marching cubes algorithm represents a 4D structure. A generated mesh is also larger, giving more context to occluded tetrahedra.

Another approach to analyzing the characteristics of ambient occlusion in 4D would be to use different projection methods. In this thesis, only the perspective projection method was used. However, other methods of projection exists in computer graphics, such as the parallel projection. In this projection, rendering rays are not generated from one viewpoint, but from a viewing plane. Straight rays are generated from the viewing plane to the image plane. Parallel projection eliminates the distortioin created by perspective projection, which scales and skews objects based on its distance to the viewpoint. Using parallel projection could a provide a result that can be easier to understand intuitively.

6 Conclusion

The main goal of this thesis is to extend preexisting ambient occlusion algorithm into 4D space so that ambient occlusion can also be used in the visualization of 4D data. The approach taken is to directly extend the algorithm mathematically into 4D space. The resulting algorithm is vertex based ambient occlusion in 4D space that calculates the occlusion values using ray tracing.

An implementation of this algorithm in C++ and OpenCL is introduced. This implementation produces raw binary 3D images which can be rendered using ParaView for further analysis.

This implementation is then tested using simple tetrahedral structures. The results of the implementation do not comply to the expected results of ambient occlusion in 4D space, so theoretical results are also given to provide an understanding of how ambient occlusion behaves in 4D space. The theoretical results show that ambient occlusion in 4D space gives a similar result in compared to ambient occlusion in 3D space. In contrast to 3D space, which has 2D surfaces, 4D space has 3D surfaces. This results in ambient occlusion in 4D space occluding entire volumes, not just planar surfaces.

As previously discussed, the experimental tests in this thesis are still insufficient to fully observe and understand the characteristics of ambient occlusion in 4D space. To fully understand the effects of ambient occlusion in 4D space, more testing using tetrahedral meshes of 4D structures is necessary.

Another possible extension to this thesis is to optimize the hemispheric sampling process. The selection of angles can narrowed down further, so that the generated rays are are more efficiently selected, improving the calculation time.

Further works can also explore other possibilities of improving visualization of 4D data using other methods availabe in 3D space. This thesis focuses only in shading methods, however, adding a specular element to a rendered image also improves the quality of the rendered image and gives a better understanding of the surface texture.

Bibliography

AKELEY, Kurt; FOLEY, James D.; SKLAR, David F.; MCGUIRE, Morgan, 2013. *Computer Graphics*. Addison-Wesley Professional.

ARVILAB, 2018. Ambient Occlusion: An Extensive Guide on its Algorithms and Use in VR. *arvivr* [online] [visited on 2021-04-11]. Available from: `https://vr.arvilab.com/blog/ambient-occlusion`.

BANCHOFF, Thomas F; CERVONE, Davide P, 1999. An Interactive Gallery on the Internet:" Surfaces Beyond the Third Dimension". *International Journal of Shape Modeling*. Vol. 5, no. 01, pp. 7–22.

BÄR, Christian, 2018. *Lineare Algebra und analytische Geometrie*. Springer.

BHANIRAMKA, Praveen; WENGER, Rephael; CRAWFIS, Roger, 2000. Isosurfacing in higher dimensions. In: *Proceedings Visualization 2000. VIS 2000 (Cat. No. 00CH37145)*, pp. 267–273.

CRAMER, Gabriel, 1750. *Introduction à l'Analyse des lignes courbes algébriques*.

DOI, Akio; KOIDE, Akio, 1991. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE TRANSACTIONS on Information and Systems*. Vol. 74, no. 1, pp. 214–224.

GANOVELLI, Fabio; CORSINI, Massimiliano; PATTANAIK, Sumanta; DI BENEDETTO, Marco, 2014. *Introduction to Computer Graphics*. Chapman and Hall/CRC.

HAYDEN, Lisa, 2019. What is Ambient Occlusion? Does it Matter in Games? *The Wired Shopper* [online] [visited on 2021-04-20]. Available from: `https://thewiredshopper.com/ambient-occlusion/?nonitro=1`.

HOLLASCH, Steven R., 1991. *Four-Space Visualization of 4D Objects*. MA thesis. Arizona State University.

KHRONOS GROUP, 2020. *OpenCL*. Version 3.0. Available also from: `https://www.khronos.org/opencl/`.

KOECHER, Max; KRIEG, Aloys, 2007. *Ebene Geometrie*. Springer.

MITTRING, Martin, 2007. Finding next Gen: CryEngine 2. In: *ACM SIGGRAPH 2007 Courses*. San Diego, California: Association for Computing Machinery. SIGGRAPH 07. Available from DOI: `10.1145/1281500.1281671`.

NOLL, A Michael, 1967. A computer technique for displaying n-dimensional hyperobjects. *Communications of the ACM.*

NVIDIA GAME WORKS, 2018. *Ray Traced Ambient Occlusion* [`https://www.youtube.com/watch?v=yag6e2Npw4M`]. Accessed 15/04/21.

RITSCHEL, Tobias; GROSCH, Thorsten; SEIDEL, Hans-Peter, 2009. Approximating Dynamic Global Illumination in Screen Space. In: *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.*

SANDIA NATIONAL LABORATORIES Kitware Inc, Los Alamos National Laboratory, 2021. *ParaView.* Version 5.9.0. Available also from: `https://www.paraview.org`.

SHIRLEY, Peter; MARSCHNER, Steve, 2018. *Fundamentals of Computer Graphics, 4th Edition.* A K Peters/CRC Press.

STEINER, K Victor; BURTON, Robert P, 1987. HIDDEN VOLUMES-THE 4TH-DIMENSION. *Computer Graphics World.* Vol. 10, no. 2, pp. 71–74.

List of Figures

2.1 An illustration of ray tracing (Ganovelli et al., 2014) 3
2.2 15 configurations of triangle mesh in a cube (Doi et al., 1991). . . . 9
2.3 An rendering of ambient occlusion in a scene with 2 cubes 9
2.4 An illustration of ambient occlusion calculation on a point P on a surface. Notice that the sampled rays are facing towards the same direction as the surface normal. None are pointing into the surface. 10

3.1 Results from SSAO (left) and SSDO (right). Note the color in the shadows from SSDO, giving it a more realistic result Ritschel et al., 2009. 13

4.1 A $3 \times 3 \times 3$ 3D image plane. Notice that the blue ray is shooting through the center of voxel $(2, 0, 1)$, and the red ray is shooting through the center of voxel $(2, 2, 2)$. 16
4.2 3 edges needed for the calculations of a tetrahedron's normal. . . . 20

5.1 A simple scene in 4D with 2 tetrahedra, projected to a $300 \times 300 \times 2$ 3D image plane 25
5.2 A simple scene in 4D with 2 tetrahedra, projected to a $50 \times 50 \times 50$ 3D image plane 26
5.3 A simple scene in 4D with 2 tetrahedra, projected to a $50 \times 50 \times 50$ 3D image plane 26
5.4 A simple scene in 4D with 2 tetrahedra, projected to a $50 \times 50 \times 50$ 3D image plane 27
5.5 Ambient occlusion of 2 cubes in 3D 28
5.6 Construction of the scene from Figure 5.1 28
5.7 Expected results of 4D ambient occlusion on caluclated manually . 29
5.8 Construction of the scene from Figure 29
5.9 Expected results of 4D ambient occlusion caluclated manually . . . 30