Into Another Dimension: Developing a Simulation of 4D Space by Implementing 4D Rigid Body Dynamics and Ray Marching

Epifanio Torres

Advisor: Dr. Szymon Rusinkiewicz

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Arts
Department of Computer Science
Princeton University

May 2022

Abstract

In recent years, scholars and programmers have developed programs that simulate 4D spaces and allow users to interact with them [43, 45, 51]. Not only do these types of programs serve as great tools for introducing the concepts behind higher-dimensional spaces, they also allude to the many possibilities of gamifying these concepts and incorporating them into other forms of digital entertainment. Nevertheless, despite their benefits, these types of programs are very difficult to implement in practice due to the lack of resources and tools available to do so. In response to this lack of resources, this project outlines and explains the different steps and calculations necessary for implementing a simple 4D simulation that incorporates 4D rigid body dynamics and the ray marching algorithm. Additionally, this project provides a working code implementation of its 4D physics engine and rendering pipeline, both of which may serve as a potential starting point for future related projects.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Dr. Szymon Rusinkiewicz, for his kind words and helpful guidance throughout this project. I would also like to thank all of my friends and family for motivating me to push forward with this idea and supporting me throughout this difficult process. A special thanks to my friend Jalen Salmon for the dinner table conversation that we had couple of years ago that sparked the idea behind this entire project.

Contents

Chapter 1

Introduction

Given that we humans experience the world around us as a 3D space, notions regarding the possible existence of spaces with more than three dimensions have intrigued and baffled scholars and scientists alike for more than a century [40, 31]. Moreover, for individuals without formal training in theoretical mathematics and physics, the concepts behind possible higher-dimensional spaces remain largely inaccessible.

In order to help make these concepts more intuitive for a broader range of individuals, some scholars and programmers have developed programs that simulate 4D spaces and allow users to interact with them [43, 45, 51]. Not only do these types of programs serve as great tools for introducing the idea of higher-dimensional spaces, they also allude to the many possibilities of gamifying such spaces and incorporating them into other forms of digital entertainment.

Unfortunately, given that most game development engines only provide native support for simulating 2D and 3D spaces, developing programs that simulate 4D spaces often must be done from scratch. Additionally, since the focus of game development and physics engines has largely been placed on 2D and 3D simulations, very few resources exist that explain the computations behind 4D simulations and the steps required to implement them. Moreover, the resources that do exist usually describe their implementations at a very high level and do not provide any accompanying code, making them difficult to use as a starting point. In order to fill this gap, my project seeks to outline and explain the different steps and calculations necessary for implementing a simple 4D rigid body physics engine and rendering pipeline. In addition to providing detailed documentation regarding how to simulate a 4D space, this project also provides a working code implementation (built with the Unity game engine) that may serve as the starting point for future related projects.

1.1 Problem Description and Related Projects

In recent years, several programs have been developed with the express purpose of simulating and gamifying 4D spaces. Two such programs are *4D Toys* and *Miegakure* by Marc ten Bosch. Although both programs enable users to experiment with 4D spaces, *4D Toys* focuses on users' interactions with 4D objects while *Miegakure* encourages users to move through the 4D space to solve puzzles [43, 45]. In his paper titled "N-Dimensional Rigid Body Dynamics," Bosch describes some of the strategies and calculations that he used to create these two programs, providing a good overview of the math behind simulating higher-dimensional spaces; nevertheless, its explanations do not describe the project's specific implementation details or provide any accompanying code (or pseudocode).

Another project that encourages users to explore a 4D space is *4D Explorer* by Jelle Vermandere [51]. This project has an associated video that highlights several of the different strategies that it uses to define a 4D space (such as ray marching) [50]. Nevertheless, the video's description of the project's implementation is too general to serve as a good starting point for developing a similar project.

Rather than providing a general overview of simulating a 4D space (like the three projects above), this project instead provides a detailed description of the most essential components of a 4D simulation. Namely, these components are the following:

- the mathematical calculations necessary for representing a 4D space,

- the project's 4D objects (and their physical properties),
- the algorithms used to detect collisions between the 4D objects,
- the algorithms used to resolve collisions between the 4D objects,
- and the pipeline used to render the 4D space.

Given its emphasis on providing the implementation details of its 4D physics engine, this project resembles other projects that provide working implementations of other physics engines (like Erin Catto's *Box2D-lite* [14] and Randy Gaul's *qu3e* [25]).

1.2 Approach

Rather than using traditional vector algebra for its underlying calculations, this project follows Bosch's suggested approach of using geometric algebra (also known as Clifford algebra) instead. Since neither Bosch's nor Vermandere's projects describe the specific collision detection algorithms that they use, this project detects collisions between 4D objects by generalizing several collision detection algorithms that are commonly used in 3D physics engines. Although Bosch suggests using a generalized version of the impulse-based collision resolution algorithm described by Guendelman et al [11, 29], this project implements a 4D version of Catto's sequential impulse solver due to its good stacking behavior and realistic-looking collision resolution [16, 14].

Like both Bosch's and Vermandere's projects, this project visualizes 4D spaces by rendering 3D projections of them. In particular, this project implements the sphere tracing algorithm (similar to Vermandere's project) to render these 3D projections because sphere tracing can be used to define objects implicitly. Notably, defining objects implicitly with signed distance functions (as is the case with the sphere marching algorithm) is an arguably simpler and more mathematically motivated approach than Bosch's method of creating 4D meshes [50, 30, 44].

Chapter 2

Overview of Geometric Algebra

In order to represent 3D spaces, most game engines (such as Unity and Unreal Engine) rely on the use of scalars, vectors, and traditional vector algebra. In traditional vector algebra, each scalar is a real number that has a magnitude and a sign; similarly, each vector is a set of scalars that has a magnitude and a direction [17]. Geometrically, scalars correspond to 0-dimensional values, while vectors correspond 1-dimensional lines; as a result, scalars and vectors represent 0D and 1D subspaces of 3D space, respectively [17].

Given that it only contains definitions for scalars and vectors, traditional vector algebra does not contain any definitions for objects that represent subspaces with a dimension that is greater than one [17]. Thus, simulations of 3D space that rely on traditional vector algebra must represent planes (which are 2D subspaces of 3D space) with their normal vectors instead. Although this approach is sufficient for all of the necessary calculations corresponding to 3D space, this representation of planes falls apart in higher-dimensional spaces, where planes no longer have a single orthogonal vector [11]. Thus, direct representations of planes should instead be used when performing calculations in higher-dimensional spaces.

Since it contains definitions for objects with more than one dimension, geometric algebra is a powerful alternative to traditional vector algebra when representing higher-dimensional spaces [11]. Thus, this project uses geometric algebra as the foundation for its underlying mathematical computations. This chapter aims to provide an introduction to the key concepts and calculations from geometric algebra that are used throughout this project.

2.1 Summary of Geometric Algebra Operations

To begin its overview of geometric algebra, this chapter will briefly explain the properties of the different geometric algebra operations that are essential for this project. Although this chapter will not provide any examples of using these operations in practice, examples can be found in Appendix A.

2.1.1 The Outer Product

One essential operation of geometric algebra is known as the outer (or wedge) product and is denoted by the $\wedge$ symbol. For vectors a, b, c, and d, the outer product has the following important properties [17, 21]:

$$a \wedge b = -b \wedge a \tag{2.1}$$

$$a \wedge a = 0 \tag{2.2}$$

$$(a + b) \wedge (c + d) = a \wedge c + a \wedge d + b \wedge c + b \wedge d \tag{2.3}$$

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c \tag{2.4}$$

These properties mean that the outer product of two vectors is anti-symmetric, that the outer product of a vector with itself is 0, and that the outer product is distributive and associative.

Just as in traditional vector algebra, an arbitrary 3D vector a can be represented as the linear combination of orthonormal basis vectors (e_x, e_y, and e_z) in geometric algebra as follows: $a = a_x e_x + a_y e_y + a_z e_z$. With this in mind, the outer product of 3D vectors a and b can be calculated as:

$$\begin{aligned}
a \wedge b &= (a_x e_x + a_y e_y + a_z e_z) \wedge (b_x e_x + b_y e_y + b_z e_z) \\
&= a_x b_y e_x \wedge e_y + a_x b_z e_x \wedge e_z + a_y b_x e_y \wedge e_x + a_y b_z e_y \wedge e_z + a_z b_x e_z \wedge e_x + a_z b_y e_z \wedge e_y \\
&= a_x b_y e_x \wedge e_y + a_x b_z e_x \wedge e_z - a_y b_x e_x \wedge e_y + a_y b_z e_y \wedge e_z - a_z b_x e_x \wedge e_z - a_z b_y e_y \wedge e_z \\
&= (a_x b_y - a_y b_x) e_x \wedge e_y + (a_x b_z - a_z b_x) e_x \wedge e_z + (a_y b_z - a_z b_y) e_y \wedge e_z
\end{aligned}$$

In geometric algebra, the outer product of two non-parallel vectors $a \wedge b$ is called a *bivector* (or a 2-blade). Geometrically, each bivector represents a planar region with an area and an orientation (similar to how each vector represents a line with a length and direction) [17]. In the sum above, the bivectors $e_x \wedge e_y$, $e_x \wedge e_z$, and $e_y \wedge e_z$ represent the orthonormal basis bivectors of 3D space. Rather than writing each basis bivector as $e_u \wedge e_v$ (for basis vectors e_u and e_v), this project simplifies this notation as e_{uv}. With this simplified notation, the sum from above can be rewritten as:

$$a \wedge b = (a_x b_y - a_y b_x) e_{xy} + (a_x b_z - a_z b_x) e_{xz} + (a_y b_z - a_z b_y) e_{yz}$$

The outer product of arbitrary 4D vectors a and b can be calculated in a similar way:

$$\begin{aligned}
a \wedge b &= (a_x e_x + a_y e_y + a_z e_z + a_w e_w) \wedge (b_x e_x + b_y e_y + b_z e_z + b_w e_w) \\
&= a_x b_y e_{xy} + a_x b_z e_{xz} + a_x b_w e_{xw} + a_y b_x e_{yx} + a_y b_z e_{yz} + a_y b_w e_{yw} \\
&\quad + a_z b_x e_{zx} + a_z b_y e_{zy} + a_z b_w e_{zw} + a_w b_x e_{wx} + a_w b_y e_{wy} + a_w b_z e_{wz} \\
&= a_x b_y e_{xy} + a_x b_z e_{xz} + a_x b_w e_{xw} - a_y b_x e_{xy} + a_y b_z e_{yz} + a_y b_w e_{yw} \\
&\quad - a_z b_x e_{xz} - a_z b_y e_{yz} + a_z b_w e_{zw} - a_w b_x e_{xw} - a_w b_y e_{yw} - a_w b_z e_{zw} \\
&= (a_x b_y - a_y b_x) e_{xy} + (a_x b_z - a_z b_x) e_{xz} + (a_x b_w - a_w b_x) e_{xw} \\
&\quad + (a_y b_z - a_z b_y) e_{yz} + (a_y b_w - a_w b_y) e_{yw} + (a_z b_w - a_w b_z) e_{zw}
\end{aligned}$$

These two examples show that any bivector can be written as the linear combination of orthonormal basis bivectors, just as any vector can be written as the linear combination of orthonormal basis vectors.

Formalizing the Definition of Bivectors and Other Multivectors

Each object of geometric algebra has an associated grade that denotes the "dimension" of the geometric object that it describes [17]. For example, bivectors are assigned a grade of 2 because they describe 2D planar regions. Additionally, different objects (such as scalars, vectors, bivectors, and so on) can be added together to form *multivectors*. Although multivectors can contain objects of different grades, a multivector is said to be *homogeneous* if all of its objects have the same grade [21]. For example, whereas the multivector $a_1e_x + a_2e_y$ is homogeneous, the multivector $a_0 + a_1e_x + a_2e_{xy}$ is not. Homogeneous multivectors can be denoted with capital letters and a subscript corresponding to their grade (e.g., the notation A_r corresponds to a homogeneous multivector with a grade of r) [17].

Since 3D space is spanned by three orthogonal vectors, the only multivectors that can be defined in 3D space are scalars, bivectors, and trivectors [21]. Consequently, the geometric algebra corresponding to 3D space is spanned by one scalar, three orthonormal basis vectors (e_x, e_y, and e_z), three orthonormal basis bivectors (e_{xy}, e_{xz}, and e_{yz}), and one orthonormal basis trivector (e_{xyz}) [21]. In 3D space, all trivector terms in a multivector can be added together to form a single trivector, similar to how all scalar terms in a multivector can be combined to form a single scalar. As a result, in the geometric algebra corresponding to 3D space, the trivector e_{xyz} is often called the space's *pseudoscalar* [21].

Similarly, since 4D space is spanned by four orthogonal vectors, the only multivectors that can be defined in 4D space are scalars, bivectors, trivectors, and 4-vectors. Consequently, the geometric algebra corresponding to 4D space is spanned by one scalar, four orthonormal basis vectors (e_x, e_y, e_z, and e_w), six orthonormal basis bivectors (e_{xy}, e_{xz}, e_{xw}, e_{yz}, e_{yw}, and e_{zw}), four orthonormal basis trivectors (e_{xyz}, e_{xyw}, e_{xzw}, and e_{yzw}), and one orthonormal basis 4-vector (e_{xyzw}). In 4D space, the 4-vector acts as the space's pseudoscalar.

One of the most useful aspects of geometric algebra is that its products and operations can be calculated with multivectors of different grades [21], such as taking the outer product of a vector and a bivector (Appendix A.1). As a result, the following is a generalized version of the first property of the outer product that applies to multivectors A_r and B_s, with grades r and s respectively [17]:

$$A_r \wedge B_s = (-1)^{rs} B_s \wedge A_r \tag{2.5}$$

In particular, this property means that the outer product of two odd-grade multivectors is anti-symmetric. Additionally, it shows that both the outer product of two even-grade multivectors and the outer product of an even-grade multivector and an odd-grade multivector are symmetric.

2.1.2 The Inner Products

In addition to the outer product, geometric algebra defines a generalized version of the inner product (denoted by the $\cdot$ symbol) [21]. Similar to the inner product of two vectors in traditional vector algebra, the inner product of two multivectors A_r and B_r with the same grade r in geometric algebra produces a scalar value [21, 17].

Whereas traditional vector algebra only contains one definition of the inner product, geometric algebra contains two additional definitions that can be used to calculate the inner product of multivectors with different grades [17]. The first of these inner products is called the *left inner product* (or the left contraction) and is denoted by the $\rfloor$ symbol. For multivectors A_r and B_s with grades $r, s \geq 1$, some essential properties of the left inner product are [17]:

$$A_r \rfloor B_s = C_{(s-r)}, \text{ when } r \leq s \tag{2.6}$$

$$A_r \rfloor A_r = A_r \cdot A_r = \|A_r\|^2 \tag{2.7}$$

$$A_r \rfloor B_s = 0, \text{ when } r > s \text{ or if } A_r \text{ contains a vector that is orthogonal to } B_s \tag{2.8}$$

Similar to the inner product in traditional vector algebra, the left inner product is distributive, and the left inner product of two orthogonal vectors is 0.

The second additional inner product is the *right inner product* (or the right contraction) and is denoted with the $\llcorner$ symbol. For multivectors A_r and B_s (with grades of r and s respectively), the right inner product has the following relationship with the left inner product [17]:

$$B_s \llcorner A_r = (-1)^{r(s-1)} A_r \lrcorner B_s, \text{ when } r \leq s \tag{2.9}$$

This means that the right inner product shares similar properties with the left inner product:

$$A_r \llcorner A_r = A_r \cdot A_r = \|A_r\|^2 \tag{2.10}$$

$$B_s \llcorner A_r = 0, \text{ when } r > s \text{ or if } A_r \text{ contains a vector that is orthogonal to } B_s \tag{2.11}$$

Like the left inner product, the right inner product is distributive over addition, and the right inner product of two orthogonal vectors is 0.

2.1.3 The Geometric Product

Another important product of geometric algebra is the geometric product. The geometric product of multivectors A_r and B_s, where $r \leq s$ and $r \geq 1$, can be written in terms of the inner product and the outer product as follows [17, 21]:

$$A_r B_s = A_r \lrcorner B_s + A_r \wedge B_s \qquad\qquad B_s A_r = B_s \llcorner A_r + B_s \wedge A_r$$

As a result, the geometric product has the following important properties:

$$A_r B_s = A_r \wedge B_s, \text{ for orthogonal multivectors } A_r \text{ and } B_s \tag{2.12}$$

$$A_r A_r = \|A_r\|^2 \tag{2.13}$$

$$A_r(B_s + C_t) = A_r B_r + A_r C_t \tag{2.14}$$

$$A_r(B_s C_t) = (A_r B_s) C_t \tag{2.15}$$

Since the geometric product of two orthogonal multivectors is equal to their outer product, the geometric product of two orthonormal basis vectors can be written using the bivector notation described earlier. For example, the geometric product of orthonormal basis vectors e_x and e_y can be written as $e_x e_y = e_{xy}$. Additionally, since the outer product of two vectors is anti-symmetric, the geometric product of two orthonormal basis vectors is also anti-symmetric (such that $e_y e_x = e_{yx} = -e_{xy}$). Lastly, since the length of an orthonormal basis vector is equal to 1, the geometric product of an orthonormal basis vector with itself is also 1 (e.g., $e_x e_x = e_x \rfloor e_x + e_x \wedge e_x = 1^2 + 0 = 1$).

2.1.4 Other Geometric Algebra Operations

In addition to the outer, inner, and geometric products, geometric algebra defines several other operations that are useful for certain calculations. This chapter will briefly review a few of those other operations as they will be useful in later chapters.

Taking the Dual of a Multivector

Each homogeneous multivector in a geometric algebra has a corresponding dual multivector. Taking the dual of a homogeneous multivector can be done by calculating the geometric product of that multivector and the algebra's unit pseudoscalar [21]. For example, using the pseudoscalar of 4D space (e_{xyzw}), the duals of the orthonormal trivectors of 4D space can be calculated as:

$$e_{xyz} \cdot e_{xyzw} = -e_w \qquad e_{xyw} \cdot e_{xyzw} = e_z$$

$$e_{xzw} \cdot e_{xyzw} = -e_y \qquad e_{yzw} \cdot e_{xyzw} = e_x$$

Notably, the dual of a trivector (which corresponds to a 3D solid) in 4D space is a vector. This indicates that each 3D solid in 4D space has a single normal vector, similar to how each 2D plane in 3D space has a single normal vector.

Reversing a Multivector

Geometric algebra also defines an operation called *reversion* (denoted by the dagger symbol $a^\dagger$) that reverses the order of vectors in a product [21]. For example, the reverse of the wedge product of two basis vectors is $(e_x \wedge e_y)^\dagger = e_y \wedge e_x = -(e_x \wedge e_y)$. This example shows that the relationship between a multivector and its reverse will differ by at most a sign. In particular, the reverse of a multivector is equal to [17]:

$$A_r^\dagger = (-1)^{r(r-1)/2} A_r \tag{2.16}$$

This means that the reverse of a 4-vector is the same as the original 4-vector, but the reverse of a trivector differs from the original trivector by a sign.

Generalizing the Vector Triple Product

An important product for 3D physics calculations is known as the vector triple product and can be calculated with the cross product as follows: $a \times (b \times c)$, for vectors a, b, and c. Since the cross product doesn't exist in 4D space, the vector triple product can be rewritten with geometric algebra as [17]:

$$a \times (b \times c) \Rightarrow -a \lrcorner (b \wedge c) \tag{2.17}$$

$$(a \times b) \times c \Rightarrow -c \times (a \times b) = c \lrcorner (a \wedge b) \tag{2.18}$$

In higher-dimensional spaces where the cross product does not exist, the right-hand sides of the equations above can be used to calculate the vector triple product.

2.2 Rotations with Geometric Algebra

In 3D space, rotations are most often described and represented with vectors that correspond to an axis of rotation. Nevertheless, these rotations represent transformations that occur about the plane that is perpendicular to the axis vectors. For instance, rotating a point 2 radians about the x-axis corresponds to a rotation of

2 radians about the yz-plane because the point's y and z-values change due to the rotation (while the point's x-value remains the same).

Although describing rotations with vectors in 3D space tends to be appropriate in most contexts, rotations in higher-dimensional spaces cannot be described by axes of rotation because vectors in higher-dimensional space are orthogonal to multiple planes [11]. For example, the description of a rotation in 4D space as a "rotation of 2 radians about the x-axis" is insufficient because the x-axis is orthogonal to the yz, yw and zw-planes, meaning that such a description could be referring to a rotation about any (or all) of these orthogonal planes. This means that rotations in higher-dimensional space should instead describe the plane of rotation directly.

Fortunately, to represent rotations, geometric algebra defines a special object known as a rotor (R) that can be generalized to higher-dimensional spaces. In particular, the rotor corresponding to a rotation about the plane generated by vectors m and n can be calculated with the geometric product as follows [21]:

$$R = nm \tag{2.19}$$

With this concept of a rotor, the rotation of a vector a about the plane formed by vectors m and n can be expressed as [21]:

$$a' = RaR^{\dagger} \tag{2.20}$$

The resulting angle of rotation between a and a' is twice the angle between vectors m and n [10]. Notably, the equation above can be also be used to rotate multivectors, making rotors well-suited for describing rotations in higher-dimensional spaces.

Rotors can also be used to represent compositions of rotations. For example, rotating a vector a by a rotor R_1 and then rotating the result by a rotor R_2 results in the following: $R_2(R_1 a R_1^{\dagger})R_2^{\dagger}$ [21]. Since the geometric product is associative, this is the same as $(R_2 R_1)a(R_1^{\dagger} R_2^{\dagger})$. As a result, the composition of two rotations R_1 and R_2 is simply $R_2 R_1$. Because the geometric product of two vectors results in a homogeneous bivector (Appendix A.3), the rotor $R = mn$ only has bivector terms and can consequently be described as a *simple rotor*. Nevertheless, the composition of two simple rotors R_1 and R_2 does not necessarily produce a simple rotor. For example, in 4D space, the geometric product of two simple rotors can have a scalar term, six bivector terms, *and* a pseudoscalar term (Appendix A.5).

For the purposes of this project, rotors in 4D space are represented with these eight different terms, and simple rotors are represented as rotors whose scalar and pseudoscalar terms are 0. Lastly, this project converts each rotor R into its corresponding 4×4 rotation matrix by using R to rotate each of the column vectors of the 4×4 identity matrix [10].

2.2.1 Angular Velocities and Inertia with Geometric Algebra

Just as vectors are used in 3D space to describe rotations, they are also often used to describe objects' angular velocities. Nevertheless, attempting to represent an object's angular velocity with vectors in higher-dimensional space runs into the same issues as using vectors to represent rotations. As a result, angular velocities instead should directly describe how fast objects are rotating about a given plane of rotation, rather than how fast they are rotating around an axis of rotation. This can be done quite simply with bivectors [11, 21]. As a result, for the purposes of this project, each object's angular velocity is represented as a bivector ω.

Similarly, the inertia of higher-dimensional objects should be represented as the inertia about a plane of rotation rather than an axis of rotation. In 4D space, this means that the inertia $\mathcal{I}$ of 4D objects should be derived with respect to each possible plane of rotation. Additionally, the inertia tensor I of an object can be generalized to higher-dimensional space by treating it as a matrix that represents a linear mapping from bivectors to bivectors [11]. Notably, each inertia tensor in 4D space is a 6×6 matrix (because 6 orthogonal basis bivectors exist in 4D space). In order to multiply an inertia tensor and a bivector, the bivector can be rewritten as a column vector and then matrix multiplication can be performed as normal [11]. For example, multiplying a diagonal inertia tensor and a bivector $b = b_{xy}e_{xy} + b_{xz}e_{xz} + b_{xw}e_{xw} + b_{yz}e_{yz} + b_{yw}e_{yw} + b_{zw}e_{zw}$ can be done as follows:

$$Ib = \begin{bmatrix} i_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & i_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & i_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & i_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & i_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & i_6 \end{bmatrix} \begin{bmatrix} b_{xy} \\ b_{xz} \\ b_{xw} \\ b_{yz} \\ b_{yw} \\ b_{zw} \end{bmatrix} = \begin{bmatrix} i_1 b_{xy} \\ i_2 b_{xz} \\ i_3 b_{xw} \\ i_4 b_{yz} \\ i_5 b_{yw} \\ i_6 b_{zw} \end{bmatrix}$$

The resulting column vector can then rewritten in terms of the orthonormal basis bivectors as:

$$Ib = i_1 b_{xy}e_{xy} + i_2 b_{xz}e_{xz} + i_3 b_{xw}e_{xw} + i_4 b_{yz}e_{yz} + i_5 b_{yw}e_{yw} + i_6 b_{zw}e_{zw}$$

Chapter 3

4D Primitives

A few of the most important primitives (or objects) that are incorporated in 3D physics engines are spheres, capsules, boxes, and planes. Due to their simplicity, these primitives are often used by physics engines to approximate more intricate 3D objects, which reduces the overall complexity of the engines and improves their performance. Motivated by the importance of these primitives for 3D physics engines, this project incorporates their 4D counterparts: hyperspheres, hypercapsules, hyperboxes, and hyperplanes. Rather than referring to these 4D objects by their more common names, this project instead refers to them by adding the prefix "hyper-" to the names of their 3D analogs in order to more clearly express the geometric relationships between these 3D and 4D objects. This chapter is dedicated to explaining the different properties of hyperspheres, hypercapsules, hyperboxes, and hyperplanes in 4D space.

3.1 Hyperspheres

In 3D space, a sphere represents an object whose points are all a fixed distance away from some central point. In particular, this central point is the sphere's center, and the fixed distance is the sphere's radius. Notably, 3D spheres have both a *surface area* (which corresponds to the total area of the outside of the sphere [56]) and a *volume* (which corresponds to the amount of space within the sphere [57]).

Similar to a sphere in 3D space, a hypersphere in 4D space is an object whose points are all a fixed distance away from some central point, which means that each hypersphere can be described by its center and radius. Somewhat similar to 3D spheres, hyperspheres have a *hyper-surface area* and a *hypervolume*; specifically, the formulas for the hyper-surface area S_{hs} and hypervolume V_{hs} of a hypersphere with radius R are [54]:

$$S_{hs} = 2\pi^2 R^3 \tag{3.1}$$

$$V_{hs} = \frac{1}{2}\pi^2 R^4 \tag{3.2}$$

Consequently, the density [12] of a hypersphere with a uniformly distributed mass M can be calculated as:

$$\rho = \frac{M}{V} = \frac{M}{\frac{1}{2}\pi^2 R^4} = \frac{2M}{\pi^2 R^4} \tag{3.3}$$

These three formulas can be used to derive the moments of inertia of a hypersphere (with mass M and radius R) about its center of mass (Appendix C.1). In particular the moment of inertia of a hypersphere can be expressed in terms of the orthonormal basis bivectors of 4D space as follows:

$$\mathcal{I}_{xy} = \mathcal{I}_{xz} = \mathcal{I}_{xw} = \mathcal{I}_{yz} = \mathcal{I}_{yw} = \mathcal{I}_{zw} = \frac{1}{3}MR^2$$

As suggested by Bosch [11], the moment of inertia of 4D objects can be expressed as a 6×6 matrix that represents a linear mapping from bivectors to bivectors. Accordingly, the moment of inertia of a hypersphere (with respect to its center of mass) can be represented as a 6×6 diagonal matrix I with entries:

$$I_{11} = I_{22} = I_{33} = I_{44} = I_{55} = I_{66} = \frac{1}{3}MR^2$$

3.2 Hypercapsules

In 3D space, a vertical capsule can be generated by sweeping a sphere (with a radius R) along a line segment (with a height H) that is parallel to the y-axis [1]. Additionally, each capsule can be broken down into three simpler components: a cylinder and two half-spheres [52]. Breaking a capsule down into these three components is useful for simplifying the process of calculating its physical properties (such as its inertia) [34].

Similar to a vertical capsule in 3D space, a vertical hypercapsule in 4D space can be generated by sweeping a hypersphere (with a radius R) along a vertical line segment (with a height H). Additionally, a hypercapsule can also be broken down into three simpler components: a spherinder (which is the 4D analog of a cylinder) and two half-hyperspheres. Notably, the vertical component of the hyper-surface area of a vertical spherinder is $4\pi R^2 H$ [6], and the hyper-surface area of a hypersphere is $2\pi^2 R^3$. Thus, the hyper-surface area of a hypercapsule can be calculated as:

$$S_{hc} = 4\pi R^2 H + 2\pi^2 R^3 \tag{3.4}$$

Additionally, given that the hypervolume of a vertical spherinder is $\frac{4}{3}\pi R^3 H$ [6] and the hypervolume of a hypersphere is $\frac{1}{2}\pi^2 R^4$, the hypervolume of a hypercapsule is:

$$V_{hc} = \frac{4}{3}\pi R^3 H + \frac{1}{2}\pi^2 R^4 \tag{3.5}$$

These formulas can be used to derive the inertia of a hypercapsule (with mass M, radius R, and height H) about its center of mass (Appendix C.2), resulting in the following moments of inertia:

$$\begin{aligned}
\mathcal{I}_{xy} = \mathcal{I}_{yz} = \mathcal{I}_{yw} &= \frac{1}{3}M_s\left(\frac{2}{5}R^2 + \frac{H^2}{4}\right) + 2M_c * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right) \\
\mathcal{I}_{xz} = \mathcal{I}_{xw} = \mathcal{I}_{zw} &= \frac{2}{5}M_s R^2 + \frac{2}{3}M_c R^2
\end{aligned}$$

In the equations above, M_s represents the mass of the hypercapsule's spherinder component, and M_c represents the mass of one of the hypercapsule's half-hypersphere components. Assuming that the mass M of the hypercapsule is evenly distributed throughout its components, these values can be calculated as (Appendix C.2):

$$M_s = \frac{M}{1 + \frac{3\pi R^2}{8H}}$$
$$M_c = \frac{M}{\frac{8H}{3\pi R^2} + 1}$$

Thus, the inertia $\mathcal{I}$ of a hypercapsule (with respect to its center of mass) can be represented as the 6×6 diagonal matrix I with entries:

$$I_{11} = I_{44} = I_{55} = \frac{1}{3}M_s\left(\frac{2}{5}R^2 + \frac{H^2}{4}\right) + 2M_c * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right)$$
$$I_{22} = I_{33} = I_{66} = \frac{2}{5}M_sR^2 + \frac{2}{3}M_cR^2$$

3.3 Hyperboxes

One of the most common primitives in 3D space is a box (or cuboid). Although boxes have many different properties, some of their most important properties are the following [2]:

1. Boxes can be described with a position, orientation, and scale.
2. Each box has eight vertices, twelve edges, and six faces.
3. The scale of a box can be described by a vector $s = s_xe_x + s_ye_y + s_ze_z$. Notably, this scale vector denotes that edges parallel to the box's local x-axis have a length of s_x, edges parallel to the local y-axis have a length of s_y, and edges parallel to the local z-axis have a length of s_z.
4. A box's face shares an edge with its adjacent faces.
5. In 3D space, each face of a box has a single normal vector.

Notably, the 4D equivalent of a box is a hyperbox (also known as a tesseract) and shares many similar properties with boxes:

1. Hyperboxes can be described with a position, orientation, and scale.
2. Each hyperbox has sixteen vertices, thirty-two edges, twenty-four faces, and eight 3-cells [11, 53]. Similar to how the faces of a box are 2D rectangles, the

3-cells of a hyperbox are 3D boxes.

3. A hyperbox's scale can be described by the vector $s = s_x e_x + s_y e_y + s_z e_z + s_w e_w$. This vector denotes that edges parallel to the hyperbox's local x-axis have a length of s_x, edges parallel to the local y-axis have a length of s_y, edges parallel to the local z-axis have a length of s_z, and edges parallel to the local w-axis have a length of s_w.
4. A hyperbox's 3-cell shares a face with its adjacent 3-cells.
5. In 4D space, each 3-cell of a box has a single normal vector.

In addition to these properties, hyperboxes have a hyper-surface area and a hypervolume. The hyper-surface area and hypervolume of a hyperbox with 4D scale vector s are:

$$S_{hb} = 2 * (s_x s_y + s_x s_z + s_x s_w + s_y s_z + s_y s_w + s_z s_w) \tag{3.6}$$

$$V_{hb} = s_x s_y s_z s_w \tag{3.7}$$

These formulas can be used to derive the moments of inertia of a hyperbox (with mass M and scale vector s) about its center of mass (Appendix C.3). The resulting inertia of a hyperbox (with respect to its center of mass) can be represented as the 6×6 diagonal matrix I with entries:

$$I_{11} = \frac{1}{3}M(s_x^2 + s_y^2) \qquad I_{22} = \frac{1}{3}M(s_x^2 + s_z^2)$$
$$I_{33} = \frac{1}{3}M(s_x^2 + s_w^2) \qquad I_{44} = \frac{1}{3}M(s_y^2 + s_z^2)$$
$$I_{55} = \frac{1}{3}M(s_y^2 + s_w^2) \qquad I_{66} = \frac{1}{3}M(s_z^2 + s_w^2)$$

3.4 Hyperplanes

In 3D space, planes are 2D surfaces that extend infinitely in two directions and have an infinite area [55]. This makes planes very useful for representing the ground (and other flat surfaces) in 3D physics simulations. Given the normal vector of a plane n

and the plane's offset o along its normal vector, the distance of a point p from a plane can be calculated as [48]:

$$d = \frac{(p \cdot n) - o}{\|n\|^2} \tag{3.8}$$

Alternatively, given the normal vector n of a plane and a point x on the plane's surface, the distance of of a point p from a plane can be calculated as [48]:

$$d = \frac{(p \cdot n) - (n \cdot x)}{\|n\|^2} \tag{3.9}$$

Notably, a plane can be interpreted as the set of all points for which the equations above evaluate to zero [48].

This definition of a plane can be generalized to 4D space to represent an object called a hyperplane. Just like a plane in 3D space, each hyperplane in 4D space can be defined with a normal vector and a scalar offset (Equation 3.8) or a normal vector and a surface point (Equation 3.9). Notably, whereas the normal vector and surface points of a plane in 3D space are 3D vectors, the normal vector and surface points of a hyperplane are 4D vectors. Interestingly, since only solids have normal vectors in 4D space (as described in Chapter 2), hyperplanes in 4D space are solids that extend infinitely in three directions and have an infinite 3D volume.

Chapter 4

4D Collision Detection

In order to prevent two objects from intersecting each other, a physics engine must first be able to determine whether that pair of objects is colliding. That said, simply checking if the two objects are colliding is not sufficient for separating them. Instead, for each pair of colliding objects, a physics engine should also calculate and record the following values: the collision normal n (which is a vector pointing from one object's center to the other's), the position p of each contact point between the two colliding objects, and the amount of separation s between the objects at each contact point [28]. In the case of this project, once a collision between two objects has been detected, a ContactPoint4D object is initialized with these values for each contact point between the two colliding objects.

Given that this project incorporates a variety of 4D primitives, it implements different algorithms for detecting collisions between each possible type of 4D primitive. Due to the geometric relationship between these 4D primitives and their 3D counterparts, many of the collision detection algorithms used by this project are inspired by algorithms used in 3D physics engines. This chapter is dedicated to providing an explanation (and accompanying pseudocode) for each of the collision detection algorithms that this project implements.

4.1 Hypersphere-Hypersphere Collisions

The algorithm for detecting collisions between hyperspheres closely resembles the typical algorithm for detecting collisions between 3D spheres; essentially, it involves finding the distance between the two hyperspheres' centers and calculating the difference between this distance and the sum of the hyperspheres' radii [47]. If this difference is negative, then the spheres are colliding with each other and the collision information (such as the collision normal) should be calculated and recorded.

Algorithm 1 Detecting a Collision Between Hyperspheres HS_1 and HS_2

```
 1: Input
 2:     $c_1$    the center point of $HS_1$
 3:     $r_1$    the radius of the $HS_1$
 4:     $c_2$    the center point of $HS_2$
 5:     $r_2$    the radius of $HS_2$
 6: Output
 7:     $cp$     the contact point
 8: procedure SphereSphereCollision
 9:     $v := c_1 - c_2$
10:     $s := \|v\| - (r_1 + r_2)$          ▷ The separation between $HS_1$ and $HS_2$
11:     if $s > 0$ then
12:         return $none$
13:     end if
14:     $n := v \ / \ \|v\|$                 ▷ The collision normal
15:     $p_1 := c_1 - (r_1 * n)$            ▷ The nearest point on $HS_1$ to $HS_2$
16:     $p_2 := c_2 + (r_2 * n)$            ▷ The nearest point on $HS_2$ to $HS_1$
17:     $p := (p_1 + p_2) \ / \ 2$           ▷ The midpoint between $p_1$ and $p_2$
18:     initialize a new contact point $cp$
19:     set the $n$ field of $cp$ to be $n$
20:     set the $p$ field of $cp$ to be $p$
21:     set the $s$ field of $cp$ to be $s$
22:     return $cp$
23: end procedure
```

4.2 Hypersphere-Hypercapsule Collisions

As mentioned in Chapter 3, a hypercapsule can be interpreted as a hypersphere that has been swept along a line segment. Thus, one method for detecting collisions between a hypersphere and a hypercapsule involves finding the point on the hypercapsule's underlying line segment that is closest to the hypersphere's center [47].

To calculate the closest point on a line segment to another point, the first step is to calculate the direction vector v from the line segment's first endpoint l_a to the line segment's second endpoint l_b:

$$v = l_b - l_a \tag{4.1}$$

The next step is to calculate the vector u from the first endpoint to the point of interest p:

$$u = p - l_a \tag{4.2}$$

Taking the dot product of these two vectors and dividing by the length of v produces the scalar projection of u onto v:

$$s = \frac{(u \cdot v)}{\|v\|} \tag{4.3}$$

Then, the scalar projection s should be divided by the length of v in order to express its value in terms of the distance between the two endpoints of the line segment:

$$t = \frac{s}{\|v\|} \tag{4.4}$$

Notably, the value of t represents how much of the line segment's distance should be traveled from one of its endpoints to reach the closest point to p along the vector v. If t is less than 0 or greater than 1, then the closest point to p along the vector v is not actually on the line segment itself. As a result, the value of t should be clamped to be between 0 and 1 to ensure that it corresponds to the closest point *on* the line segment. Multiplying the clamped value of t by v and adding it to l_a results in the closest point c on the line segment to p:

$$c = l_a + t * v \tag{4.5}$$

With these steps, the closest point on the hypercapsule's line segment to the hypersphere's center can be calculated; then, the same algorithm for detecting collisions between hyperspheres (Algorithm 1) can be used by treating this closest point as the center of a second hypersphere [47].

Algorithm 2 Detecting Collisions Between Hypersphere HS and Hypercapsule HC

```
 1: Input
 2:     $c_S$      the center point of $HS$
 3:     $r_S$      the radius of $HS$
 4:     $l_{Ca}$   the first endpoint of the underlying line segment of $HC$
 5:     $l_{Cb}$   the second endpoint of the underlying line segment of $HC$
 6:     $r_C$      the radius of $HC$
 7: Output
 8:     $cp$       the contact point
 9: procedure SphereCapsuleCollision
10:     $c_C$ := ClosestPointOnLineSegment($c_S, l_{Ca}, l_{Cb}$)
11:     $cp$ := SphereSphereCollision($c_S, r_S, c_C, r_C$)
12:     return $cp$
13: end procedure
```

4.3 Hypersphere-Hyperbox Collisions

One simple method of detecting collisions between a hypersphere and a hyperbox involves finding the closest point on the hyperbox's surface to the hypersphere [27]. To calculate the closest point on a hyperbox's surface to a given point, the point should first be converted into the hyperbox's local object space (by subtracting the hyperbox's center position and then applying the reverse of the hyperbox's rotor). Afterwards, the scale of the hyperbox can be used to clamp the x, y, z, and w-values of the point. This results in the closest point on the hyperbox's surface (in the hyperbox's local space). To convert the closest point into world space, the hyperbox's rotor should be applied to it before adding the hyperbox's center position.

Algorithm 3 Closest Point on a Hyperbox's Surface

```
 1: Input
 2:     $p$       the point of interest
 3:     $c$       the center of the hyperbox
 4:     $s$       the scale vector of the hyperbox
 5:     $R$       the rotor of the hyperbox
 6: Output
 7:     $p_B$     the closest point on the hyperbox
 8: procedure ClosestPointOnHyperbox
 9:     $p_B := R^\dagger p R$        ▷ Converts the given point into the hyperbox's local space
10:     clamp $p_B.x$ to be between $-s.x/2$ and $s.x/2$
11:     clamp $p_B.y$ to be between $-s.y/2$ and $s.y/2$
12:     clamp $p_B.z$ to be between $-s.z/2$ and $s.z/2$
13:     clamp $p_B.w$ to be between $-s.w/2$ and $s.w/2$
14:     $p_B := (R\ p_B\ R^\dagger) + c$        ▷ Converts the closest point into world space
15:     return $p_B$
16: end procedure
```

Once the closest point on the hyperbox's surface has been found, the distance between this point and the hypersphere's center should be calculated [27]. If this distance is less than the hypersphere's radius, then the two objects are colliding and the collision information should be generated.

Algorithm 4 Detecting Collisions Between Hypersphere HS and Hyperbox HB

```
1: Input
2:     $c_S$    the center of $HS$
3:     $r_S$    the radius of $HS$
4:     $c_B$    the center of $HB$
5:     $s_B$    the scale vector of $HB$
6:     $R_B$    the rotor of $HB$
7: Output
8:     $cp$    the contact point
9: procedure SphereBoxCollision
10:     $p_B := \textsc{ClosestPointOnHyperbox}(c_S, c_B, s_B, R_B)$
11:     $v := c_S - p_B$
12:     $s := \|v\| - r_S$
13:     if $s > 0$ then
14:         return $none$
15:     end if
16:     $n := v \ / \ \|v\|$
17:     $p_S := c_S - (r_S * n)$                ▷ Nearest point on $HS$ to $HB$
18:     $p := (p_S + p_B) \ / \ 2$                ▷ Midpoint between $p_S$ and $p_B$
19:     initialize a new contact point $cp$
20:     set the $n$ field of $cp$ to be $n$
21:     set the $p$ field of $cp$ to be $p$
22:     set the $s$ field of $cp$ to be $s$
23:     return $cp$
24: end procedure
```

4.4 Hypersphere-Hyperplane Collisions

To detect collisions between a hypersphere and a hyperplane, the first step is to calculate the distance between the hypersphere's center and the hyperplane (using Equation 3.8 or 3.9) [19]. Subtracting this distance by the radius of the hypersphere results in the amount of separation between the hypersphere and the hyperplane. If the separation is negative, then the hypersphere and hyperplane are colliding and the collision information should be calculated.

Algorithm 5 Detecting a Collision Between Hypersphere HS and Hyperplane HP

```
1: Input
2:     $c_S$    the center of $HS$
3:     $r_S$    the radius of $HS$
4:     $n_P$    the normal of $HP$
5:     $o_P$    the offset of $HP$ along its normal
6: Output
7:     $cp$    the contact point
8: procedure SpherePlaneCollision
9:     $d := ((c_S \cdot n_P) - o_P) \ / \ \|n_P\|^2$        ▷ Dist of $c_S$ from $HP$
10:    $s := d - r_S$
11:    if $s > 0$ then
12:        return $none$
13:    end if
14:    $p := c_S - (r_S * n_P)$
15:    initialize a new contact point $cp$
16:    set the $n$ field of $cp$ to be $n_P$
17:    set the $p$ field of $cp$ to be $p$
18:    set the $s$ field of $cp$ to be $s$
19:    return $cp$
20: end procedure
```

4.5 Hypercapsule-Hypercapsule Collisions

To detect collisions between two hypercapsules, the closest point on the second hypercapsule's line segment to the first hypercapsule should be calculated [47]. Then, using this point, the closest point on the first hypercapsule's line segment should be calculated. These two closest points can then be treated as the centers of two hyperspheres. Then, the hypersphere-hypersphere collision detection algorithm can subsequently be used to check for a collision [47].

Algorithm 6 Detecting Collisions Between Hypercapsules HC_1 and HC_2

```
 1: Input
 2:     $l_{1a}$    the first endpoint of the line segment of $HC_1$
 3:     $l_{1b}$    the second endpoint of the line segment of $HC_1$
 4:     $r_1$     the radius of $HC_1$
 5:     $l_{2a}$    the first endpoint of the line segment of $HC_2$
 6:     $l_{2b}$    the second endpoint of the line segment of $HC_2$
 7:     $r_2$     the radius of $HC_2$
 8: Output
 9:     $cp$     the contact point
10: procedure CapsuleCapsuleCollision
11:     $d_1 := \|l_{2a} - l_{1a}\|$
12:     $d_2 := \|l_{2b} - l_{1a}\|$
13:     $d_3 := \|l_{2a} - l_{1b}\|$
14:     $d_4 := \|l_{2b} - l_{1b}\|$
15:     if $d_3 < d_1$ or $d_3 < d_2$ or $d_4 < d_1$ or $d_4 < d_2$ then
16:         $c_1 := l_{1b}$                ▷ $l_{1b}$ is the closest endpoint of $HC_1$ to $HC_2$
17:     else
18:         $c_1 := l_{1a}$                ▷ $l_{1a}$ is the closest endpoint of $HC_1$ to $HC_2$
19:     end if
20:     $c_2 :=$ ClosestPointOnLineSegment($c_1, l_{2a}, l_{2b}$)
21:     $c_1 :=$ ClosestPointOnLineSegment($c_2, l_{1a}, l_{1b}$)
22:     $cp :=$ SphereSphereCollision($c_1, r_1, c_2, r_2$)
23:     return $cp$
24: end procedure
```

4.6 Hypercapsule-Hyperbox Collisions

To detect collisions between a hypercapsule and a hyperbox, the first step is to calculate the closest point on the hypercapsule's line segment to the center of the hyperbox. This point can then be treated as the center of a hypersphere, and the algorithm for detecting hypersphere-hyperbox collisions can be performed:

Algorithm 7 Detecting a Collision Between Hypercapsule HC and Hyperbox B

```
1:  Input
2:      $l_{Ca}$    the first endpoint of the line segment of $HC$
3:      $l_{Cb}$    the second endpoint of the line segment of $HC$
4:      $r_C$     the radius of $HC$
5:      $c_B$     the center of $HB$
6:      $s_B$     the scale vector of $HB$
7:      $R_B$     the rotor of $HB$
8:  Output
9:      $cp$      the contact point
10: procedure CapsuleBoxCollision
11:     $c_C := \textsc{ClosestPointOnLineSegment}(c_B, l_{Ca}, l_{Cb})$
12:     $cp := \textsc{SphereBoxCollision}(c_C, r_C, c_B, s_B, R_B)$
13:     return $cp$
14: end procedure
```

4.7 Hypercapsule-Hyperplane Collisions

One simple approach to detecting collisions between a hypercapsule and a hyperplane involves treating the endpoints of the hypercapsule's line segment as the centers of two different hyperspheres and performing the hypersphere-hyperplane collision detection algorithm twice. Notably, since the hypercapsule's end points can be simultaneously colliding with the hyperplane, each endpoint can have its own corresponding contact point. As a result, rather than generating a single contact point for collisions between hypercapsules and hyperplanes, the collision detection algorithm should instead return a list containing the contact point(s) between a hypercapsule and a hyperplane.

Algorithm 8 Detecting a Collision Between Hypercapsule HC and Hyperplane HP

```
Input
    $l_{Ca}$    the first endpoint of the line segment of $HC$
    $l_{Cb}$    the second endpoint of the line segment of $HC$
    $r_C$       the radius of $HC$
    $n_P$       the normal of $HP$
    $o_P$       the offset of $HP$ along its normal
Output
    $CP$        a list of contact points
procedure CapsulePlaneCollision
    initialize $CP$ as an empty list of contact points
    $cp_1$ := SpherePlaneCollision($l_{Ca}, r_C, n_P, o_P$)
    if $cp_1$ is not none then
        append $cp_1$ to $CP$
    end if
    $cp_2$ := SpherePlaneCollision($l_{Cb}, r_C, n_P, o_P$)
    if $cp_2$ is not none then
        append $cp_2$ to $CP$
    end if
    return $CP$
end procedure
```

4.8 Hyperbox-Hyperbox Collisions

In order to detect collisions between hyperboxes, one possible approach makes use of the Separating Axis Theorem, which posits that two convex objects cannot be intersecting each other if they can be separated along at least one axis vector [22, 18]. Thus, based on the Separating Axis Theorem, detecting collisions between two convex objects involves verifying that the two objects *cannot* be separated along any of their potentially separating axis vectors.

Notably, the potentially separating axis vectors are not arbitrary, but rather are based on the geometry of the two convex objects themselves. For example, in 3D space, the potentially separating axes of two boxes correspond to the boxes' edge direction vectors and the cross product of their edge direction vectors [11, 22]. Geometrically, each potentially separating axis represents the set of planes that are perpendicular to the axis. As a result, in 3D space, checking if two convex objects are separated along an axis essentially corresponds to checking if the two objects can be separated by some plane that is perpendicular to the axis of separation.

The Separating Axis Theorem can be easily extended to 4D space by checking if two 4D objects can be separated along some potentially separating axis. Geometrically, the 4D version of the Separating Axis Theorem corresponds to checking if the two 4D objects can be separated by some hyperplane. Whereas the potentially separating axes of 3D boxes can be calculated with the cross product, the potentially separating axes of 4D hyperboxes should be calculated with the outer product instead. In particular, the potentially separating axes of two hyperboxes in 4D space correspond to the hyperboxes' edge direction vectors and the outer product of their edge direction vectors and their face bivectors [11].

Table 4.1: The Potentially Separating Axes of Two Hyperboxes

A_1	A_2	A_3	A_4
B_1	B_2	B_3	B_4
$A_1 \wedge (B_1 \wedge B_2)$	$A_2 \wedge (B_1 \wedge B_2)$	$A_3 \wedge (B_1 \wedge B_2)$	$A_4 \wedge (B_1 \wedge B_2)$
$A_1 \wedge (B_1 \wedge B_3)$	$A_2 \wedge (B_1 \wedge B_3)$	$A_3 \wedge (B_1 \wedge B_3)$	$A_4 \wedge (B_1 \wedge B_3)$
$A_1 \wedge (B_1 \wedge B_4)$	$A_2 \wedge (B_1 \wedge B_4)$	$A_3 \wedge (B_1 \wedge B_4)$	$A_4 \wedge (B_1 \wedge B_4)$
$A_1 \wedge (B_2 \wedge B_3)$	$A_2 \wedge (B_2 \wedge B_3)$	$A_3 \wedge (B_2 \wedge B_3)$	$A_4 \wedge (B_2 \wedge B_3)$
$A_1 \wedge (B_2 \wedge B_4)$	$A_2 \wedge (B_2 \wedge B_4)$	$A_3 \wedge (B_2 \wedge B_4)$	$A_4 \wedge (B_2 \wedge B_4)$
$A_1 \wedge (B_3 \wedge B_4)$	$A_2 \wedge (B_3 \wedge B_4)$	$A_3 \wedge (B_3 \wedge B_4)$	$A_4 \wedge (B_3 \wedge B_4)$

Just as the column vectors of a 3D box's rotation matrix correspond to its edge direction vectors [22], the column vectors of a hyperbox's rotation matrix also correspond to its edge direction vectors. As a result, the potentially separating axes of two 4D hyperboxes can be calculated in terms of the column vectors of the first hyperbox's rotation matrix (which are A_1, A_2, A_3, and A_4) and the column vectors of the second hyperbox's rotation matrix (which are B_1, B_2, B_3, and B_4) (Table 4.1).

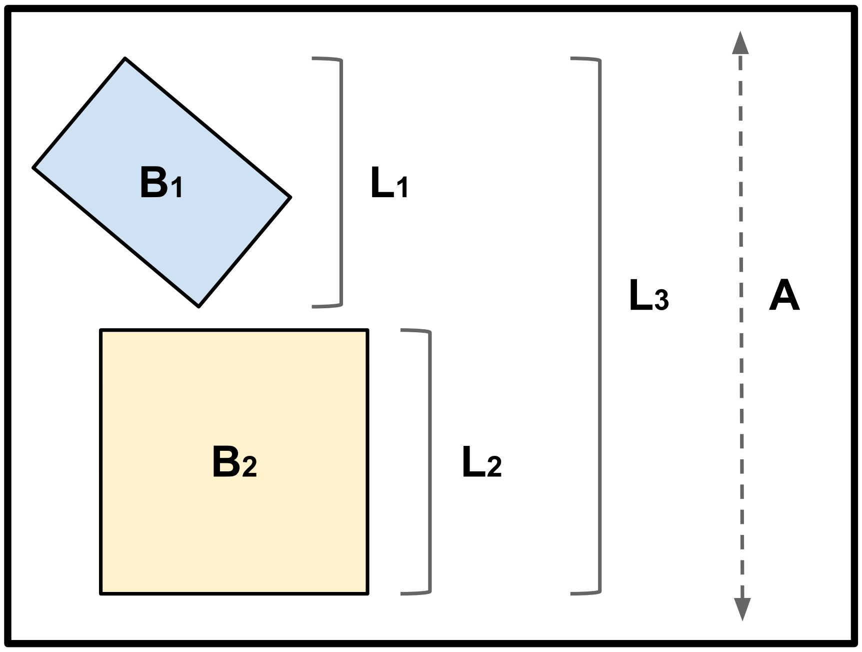

Figure 4.1: The lengths of B_1 and B_2 along the separating axis A are L_1 and L_2 respectively. The length L_3 corresponds to the length between the outermost vertices of both B_1 and B_2 along A.

With this in mind, the first step of detecting collisions between two hyperboxes is to calculate their potentially separating axes. Then, for each potentially separating axis, the algorithm should calculate the length of the first hyperbox and the length of the second hyperbox along the axis (e.g., L_1 and L_2 in Figure 4.1) [8]. Then, the length between the outermost vertices of both hyperboxes should be calculated (e.g., L_3 in Figure 4.1). If the length between the hyperboxes' outermost vertices is greater than the sum of length of the first hyperbox and the length of the second hyperbox along the separating axis, then the two hyperboxes can be separated along the axis vector, which means they are not colliding. Otherwise, the hyperboxes cannot be separated along the axis.

The following pseudocode calculates the separation between two hyperboxes along an axis [8]:

Algorithm 9 Separation of Hyperbox HB_1 and HB_2 Along an Axis

```
Input
    $a$       the axis of interest
    $V_1$     the list of vertices of $HB_1$
    $V_2$     the list of vertices of $HB_2$
Output
    $s$       the separation between the $HB_1$ and $HB_2$ along $a$
procedure SeparationAlongAnAxis
    $min_1 := \infty$
    $max_1 := -\infty$
    for each $v$ in $V_1$ do
        $d_1 := v \cdot a$
        $min_1 = \textsc{Min}(min_1, d_1)$
        $max_1 = \textsc{Max}(max_1, d_1)$
    end for
    $min_2 := \infty$
    $max_2 := -\infty$
    for each $u$ in $V_2$ do
        $d_2 := u \cdot a$
        $min_2 = \textsc{Min}(min_2, d_2)$
        $max_2 = \textsc{Max}(max_2, d_2)$
    end for
    $l_1 := max_1 - min_1$                                  ▷ Length of $HB_1$ along $a$
    $l_2 := max_2 - min_2$                                  ▷ Length of $HB_2$ along $a$
    $l_3 := \textsc{Max}(max_1, max_2) - \textsc{Min}(min_1, min_2)$  ▷ Dist between the farthest verts
    $s := (l_1 + l_2) - l_3$
    return $s$
end procedure
```

Notably, even if the hyperboxes cannot be separated along one axis, they still may be able to be separated along another axis. As a result, each potentially separating axis should be checked. If the hyperboxes cannot be separated along any of the potentially separating axis vectors, then they must be colliding.

Although detecting collisions with the Separating Axis Theorem is straightforward, generating the collision information with the Separating Axis Theorem isn't. One approach for generating the collision information involves keeping track of the axis along which the two hyperboxes are intersecting the least. This axis is often called the "minimum translation vector" and can be treated as the collision normal for collision resolution [33]. One thing to note is that the axis of least penetration may be pointing in the wrong direction (since the separation between the two hyperboxes along the axis is the same regardless of the axis' direction). As a result, the sign of the minimum translation vector should be flipped if the dot product of the axis and the vector pointing from the second hyperbox to the first hyperbox is negative [46].

With the collision normal, the contact manifold can be generated by performing a generalized version of 3D Sutherland-Hodgman clipping algorithm [20, 26]. The first step of this generalized algorithm is to find the 3-cell of a hyperbox that is most parallel to the collision normal; this 3-cell is known as the reference 3-cell. The second step involves finding the 3-cell (of the other hyperbox) whose normal is most anti-parallel to the collision normal; this 3-cell is known as the incident 3-cell. The next step is to clip the vertices of the incident 3-cell with the reference 3-cell (and its neighboring 3-cells). This results in the list of vertices of the incident 3-cell that are inside/on the reference 3-cell. These vertices represent the contact manifold of the collision. A more detailed description of the generalized Sutherland-Hodgman algorithm can be found in Appendix D.

Combining all of these different steps produces the following algorithm for detecting collisions between hyperboxes:

Algorithm 10 Detecting Collisions Between Hyperboxes HB_1 and HB_2

```
1: Input
2:     $A$       the list of potentially separating axes
3:     $p_1$     the center position of $HB_1$
4:     $V_1$     the list of vertices of $HB_1$
5:     $C_1$     the list of 3-cells of $HB_1$
6:     $p_2$     the center position of $HB_2$
7:     $V_2$     the list of vertices of $HB_2$
8:     $C_2$     the list of 3-cells of $HB_2$
9: Output
10:    $CP$      the list of contact points
11: procedure BoxBoxCollision
12:    $a_{min} := A[0]$                    ▷ Initializes the minimum translation vector
13:    $s_{min} := \infty$                  ▷ Initializes the least separation along an axis
14:    for each $a$ in $A$ do
15:        $s :=$ SeparationAlongAnAxis($a, V_1, V_2$)
16:        if $s > 0$ then                  ▷ The current axis separates the hyperboxes
17:            return $none$
18:        else if $s < s_{min}$ then
19:            $a_{min} := a$
20:            $s_{min} := s$
21:        end if
22:    end for
23:    $n := a_{min}$     ▷ Sets the collision normal to be the minimum translation vector
24:    if $(p_1 - p_2) \cdot n < 0$ then
25:        $n := -n$                        ▷ Corrects the direction of the normal
26:    end if
27:    $CP :=$ ClipHyperboxes($n, C_1, C_2$)
28:    return $CP$
29: end procedure
```

4.9 Hyperbox-Hyperplane Collisions

For hyperbox-hyperplane collisions, the only potentially separating axis is the hyperplane's normal. Since all points on a hyperplane are located some fixed offset along its normal vector, a collision has occurred if the hyperplane's offset is between the hyperbox's outermost vertices (with respect to the hyperplane's normal). In such cases, all of the vertices in a hyperbox that have a negative distance from the hyperplane represent the collision's contact manifold. Putting this together (and some algorithms from Appendix D) yields the following collision detection algorithm:

Algorithm 11 Detecting Collisions Between Hyperbox HB and Hyperplane HP

```
1: Input
2:     $V_B$     the list of vertices of $HB$
3:     $C_B$     the list of 3-cells of $HB$
4:     $n_P$     the normal of $HP$
5:     $o_P$     the offset of $HP$ along its normal
6: Output
7:     $CP$      the list of contact points
8: procedure BoxPlaneCollision
9:     $min := \infty$
10:    $max := -\infty$
11:    for each $v$ in $V_B$ do
12:        $d := v \cdot n_P$
13:        $min :=$ Min$(min, d)$
14:        $max :=$ Max$(max, d)$
15:    end for
16:    if $o_P < min$ or $o_P > max$ then
17:        return $none$
18:    end if
19:    $c_I :=$ GetMostAntiParallel$(n, C_A)$
20:    $CP :=$ RemoveOutsideVertices$(n, c_I)$
21:    return $CP$
22: end procedure
```

Chapter 5

4D Collision Resolution

Once a collision between two objects has been detected and the collision information has been calculated and recorded, a physics engine should then resolve the collision by separating the two objects. One good approach to separating two colliding objects involves modifying their velocities by applying an *impulse* to each object in opposite directions [29]. Notably, this impulse-based approach to resolving collisions often leads to more realistic behavior and can be even used to implement friction [29, 15]. As a result, this project implements an impulse-based collision resolution algorithm (inspired by Erin Catto's sequential impulse solver [15, 14]). This chapter aims to provide an overview of the calculations behind impulse-based collision resolution algorithms, while also describing the implementation of this project's specific approach.

5.1 Calculating Normal Impulses

To separate two colliding objects A and B at a contact point p, an equal and opposite impulse along the collision normal n can be applied to each object. This type of impulse is commonly called a *normal impulse* [15].

The first step of calculating a normal impulse j_n at a contact point p is to calculate the relative velocity Δv_p at that point. In 3D, for contact point p on object A with a velocity vector v_A and an angular velocity vector ω_A, the instantaneous velocity v_p at p can be calculated with the following equation: $v_p = v_A + r_A \times \omega_A$, where r_A points from A's center to p [49]. Notably, in this equation, the cross product is used to linearize the object's angular velocity at p.

Since the cross product doesn't exist in 4D space, the object's angular velocity (which is a bivector) should instead be linearized by taking the left inner product of r and ω. This produces the following equation for the instantaneous velocity of a point p on an object A [11]:

$$v_p = v_A + r_A \,\lrcorner\, \omega_A \tag{5.1}$$

Based on this equation, the relative velocity of a contact point p, with respect to colliding objects A and B, can be calculated as:

$$\Delta v_p = (v_A + r_A \,\lrcorner\, \omega_A) - (v_B + r_B \,\lrcorner\, \omega_B) \tag{5.2}$$

Additionally, the relative normal velocity of a contact point p can be calculated as:

$$\Delta v_{n,p} = \Delta v_p \cdot n \tag{5.3}$$

In 3D space, given the collision normal n and the relative normal velocity $\Delta v_{n,p}$ at a contact point p, the normal impulse j_n at p can be calculated as [49]:

$$j_n = \frac{-\Delta v_{n,p}}{\frac{1}{m_A} + [I_A^{-1}(r_A \times n) \times r_A] \cdot n + \frac{1}{m_B} + [I_B^{-1}(r_B \times n) \times r_B] \cdot n}$$

In the equation above, m_A and I_A correspond to the mass and inertia tensor of object A, respectively. Likewise, m_B and I_B correspond to the mass and inertia tensor of object B, respectively. In order to use this equation in 4D space, the vector triple product $(r \times n) \times r$ can be rewritten with geometric algebra (Equation 2.18), which results in the following equation for calculating a normal impulse:

$$j_n = \frac{-\Delta v_{n,p}}{\frac{1}{m_A} + [I_A^{-1}(r_A \,\lrcorner\, (r_A \wedge n))] \cdot n + \frac{1}{m_B} + [I_B^{-1}(r_B \,\lrcorner\, (r_B \wedge n))] \cdot n} \tag{5.4}$$

After the normal impulse j_n at contact point p has been calculated, the following equations can be used to apply j_n to the objects to separate them [11]:

$$v_{A+} = v_{A-} + \frac{j_n * n}{m_A} \tag{5.5}$$

$$v_{B+} = v_{B-} - \frac{j_n * n}{m_B} \tag{5.6}$$

$$\omega_{A+} = \omega_{A-} + I_A^{-1}(r_A \wedge (j_n * n)) \tag{5.7}$$

$$\omega_{B+} = \omega_{B-} - I_B^{-1}(r_B \wedge (j_n * n)) \tag{5.8}$$

In the equations above, v_- refers to the velocity of an object prior to applying the normal impulse, and v_+ refers to the velocity of an object after applying the impulse; likewise, ω_- refers to an object's angular velocity before applying the impulse, and ω_+ refers to an object's angular velocity after applying the impulse.

If one of the colliding objects has a fixed position, its velocity in Equation 5.2 and its $\frac{1}{m}$ term in equation 5.4 can be treated as 0; additionally, its velocity should not be modified by the normal impulse. Likewise, if a colliding object has a fixed rotation, both its angular velocity in Equation 5.2 and its $[I^{-1}(r \lrcorner (r \wedge n))] \cdot n$ term in Equation 5.4 can be treated as 0, and its angular velocity should not be updated.

5.2 Calculating Tangent Impulses

To apply friction at the contact point, an impulse should be applied to each object in the direction that is opposite to the point's relative tangential velocity. This type of impulse is commonly referred to as a *tangent impulse* [15].

To calculate a tangent impulse j_t, the relative velocity Δv_p of contact point p should first be calculated with Equation 5.2. Afterwards, the relative tangential velocity $\Delta v_{t,p}$ can be calculated with the following equation:

$$\Delta v_{t,p} = \Delta v_p - (\Delta v_p \cdot n) * n \tag{5.9}$$

Additionally, the tangent vector t can be calculated by normalizing $\Delta v_{t,p}$:

$$t = \frac{\Delta v_{t,p}}{\|\Delta v_{t,p}\|} \tag{5.10}$$

Given the coefficient of friction μ, the relative tangential velocity $\Delta v_{t,p}$, and the tangent vector t, the tangent impulse j_t can be calculated as [49]:

$$j_t = \frac{-\mu * \Delta v_{t,p}}{\frac{1}{m_A} + [I_A^{-1}(r_A \lrcorner (r_A \wedge t))] \cdot t + \frac{1}{m_B} + [I_B^{-1}(r_B \lrcorner (r_B \wedge t))] \cdot t} \tag{5.11}$$

The tangent impulse j_t can be then applied to objects A and B as follows:

$$v_{A+} = v_{A-} + \frac{j_t * t}{m_A} \tag{5.12}$$

$$v_{B+} = v_{B-} - \frac{j_t * t}{m_B} \tag{5.13}$$

$$\omega_{A+} = \omega_{A-} + I_A^{-1}(r_A \wedge (j_t * t)) \tag{5.14}$$

$$\omega_{B+} = \omega_{B-} - I_B^{-1}(r_B \wedge (j_t * t)) \tag{5.15}$$

If a colliding object has a fixed position, its corresponding linear terms in Equations 5.9 and 5.11 can be treated as 0, and its velocity should not be updated with the tangent impulse. Likewise, if an object has a fixed rotation, its corresponding angular terms in equations 5.9 and 5.11 can be treated as 0, and its angular velocity should not be updated.

5.3 Implementing a Sequential Impulse Solver

For each pair of colliding objects at a given time-step, the equations described in Sections 5.1 and 5.2 can be used to calculate and apply the necessary impulses for separating the two colliding objects and applying friction. A naive impulse-based approach for resolving collisions only applies impulses to the colliding objects once per time-step. Although this is a reasonable approach for simple physics engines, it often suffers from poor stacking behavior and can lead to jitter while objects are at rest [15]. In order to tackle these issues, Erin Catto proposes an iterative approach to impulse-based collision resolution known as a *sequential impulse solver* [16, 15, 14]. Unlike a naive impulse-based approach, a sequential impulse solver applies impulses to the colliding objects multiples times during each time-step [15].

Sequential impulse solvers work by accumulating the impulses that are being repeatedly calculated and applied to each contact point during the current time-step [15]. Notably, these accumulated impulses are expected to converge to the true impulses between the colliding objects (i.e., the impulses that fully separate them) after enough iterations [15]. While applying and accumulating the impulses, Catto's sequential impulse solver approach employs impulse caching, impulse clamping, and warm starting [15, 24]. Impulse caching refers to storing and reusing the accumulated impulse for each contact point; impulse clamping refers to ensuring that the accumulated impulse is between acceptable values; and warm starting refers to beginning each time-step with the accumulated impulses from the previous time-step [24]. Despite not implementing warm starting for the sake of simplicity, this project incorporates both impulse caching and clamping.

Broad Phase Collision Detection

The very first step of the sequential impulse solver is to iterate through all unique pairs of objects in the space in order to detect any collisions between them [15]. For each unique pair, the appropriate collision detection algorithm (from Chapter 4) should be used to determine if the two objects are indeed colliding. Whenever a collision is detected, the resulting list of contact points should be recorded.

As described in Chapter 4, this project represents each contact point as an object of class ContactPoint4D, which contains fields that are used to store the values that are necessary for resolving the collision at the contact point. Notably, since this project's sequential impulse solver employs impulse caching, the ContactPoint4D class contains fields for storing the accumulated normal and tangent impulses. Additionally, the ContactPoint4D class contains fields for precomputing and storing the values that remain constant at each contact point during a given time-step.

Integrating Forces

After the broad phase collision step, the next step of a sequential impulse solver is to integrate the forces acting on all of the objects in the space [15]. Notably, for the purposes of this project, the only force acting on the objects is the force due to the Earth's gravity. Notably, gravity results in constant acceleration of about 10 m/s^2 towards the Earth (regardless of an object's mass) [13]. Thus, in order to simulate the force of gravity, this project simply applies a downward acceleration of 10 m/s^2 along the y-axis to each object.

Precomputing Collision Constants

Since the sequential impulse solver approach is an iterative method, it requires recomputing many values while repeatedly applying normal and tangent impulses to the colliding objects. Nevertheless, a few values that are used to calculate the normal and tangent impulses at a contact point do not change while iterating during a given time-step. Before iteratively resolving the collisions in the current time-step, the collision solver can precompute and store the following values for each contact point between every pair of colliding objects A and B:

- r_A: the vector that points from the center of mass of A to the contact point
- r_B: the vector that points from the center of mass of B to the contact point
- t: the tangent velocity vector
- m_n: the mass at the contact point along the collision normal
- m_t: the mass at the contact point along the tangent vector
- b: the Baumgarte stabilization factor

Based on Equation 5.4, the relative mass m_n along the collision normal can be calculated as:

$$m_n = \frac{1}{\frac{1}{m_A} + [I_A^{-1}(r_A \lrcorner (r_A \wedge n))] \cdot n + \frac{1}{m_B} + [I_B^{-1}(r_B \lrcorner (r_B \wedge n))] \cdot n}$$

Based on Equation 5.11, the relative mass m_t along the tangent can be calculated as:

$$m_t = \frac{1}{\frac{1}{m_A} + [I_A^{-1}(r_A \lrcorner (r_A \wedge t))] \cdot t + \frac{1}{m_B} + [I_B^{-1}(r_B \lrcorner (r_B \wedge t))] \cdot t}$$

Lastly, the Baumgarte stabilization factor b can be calculated as [24, 14]:

$$b = -f * \frac{1}{\Delta T} * \textsc{Min}(0,\ s + a)$$

In the equation above, f corresponds to the bias factor (which should be between 0.1 and 0.3); ΔT corresponds to delta time (which is the amount of time between each time-step); s corresponds to amount the separation of objects A and B at the contact point; and a corresponds the amount of penetration between objects that is allowed.

Resolving Collisions

After the precomputation step, the next step of the sequential impulse solver approach is to iteratively accumulate impulses and apply them to each of the contact points. To do this, the solver should repeatedly loop through the list of contact points for a set number of times. Each time the solver reaches a contact point while repeatedly looping through the list, the solver should calculate the normal impulse at the contact point, accumulate the impulse, and then apply it.

The normal impulse at a contact point can be applied/accumulated as follows [14]:

Algorithm 12 Applying the Accumulated Normal Impulse to Objects A and B

```
 1: Input
 2:     $\Delta v$    the relative velocity at the contact point
 3:     $cp$    a contact point object
 4: procedure ApplyAccumulatedNormalImpulse
 5:     $\Delta v_n := \Delta v \cdot cp.n$
 6:     $j_n := (cp.b - \Delta v_n) * cp.m_n$
 7:     $j_{prev} := cp.j_n$
 8:     $cp.j_n := \textsc{Max}(j_{prev} + j_n, 0)$
 9:     $j_n := cp.j_n - j_{prev}$
10:     use $j_n$ to update the velocity of $A$ with Equation 5.5
11:     use $j_n$ to update the velocity of $B$ with Equation 5.6
12:     use $j_n$ to update the angular velocity of $A$ with Equation 5.7
13:     use $j_n$ to update the angular velocity of $B$ with Equation 5.8
14: end procedure
```

Afterwards, the solver can apply/accumulate the tangent impulse as follows [14]:

Algorithm 13 Applying the Accumulated Tangent Impulse to Objects A and B

```
 1: Input
 2:     $\Delta v$    the relative velocity at the contact point
 3:     $\mu$    the coefficient of friction for this collision
 4:     $cp$    a contact point object
 5: procedure ApplyAccumulatedTangentImpulse
 6:     $\Delta v_t := \Delta v \cdot cp.t$
 7:     $j_t := -\Delta v_t * cp.m_t$
 8:     $j_{prev} := cp.j_t$
 9:     $cp.j_t := \textsc{Clamp}(j_{prev} + j_t,\ -\mu * cp.j_n,\ \mu * cp.j_n)$
10:     $j_t := cp.j_t - j_{prev}$
11:     use $j_t$ to update the velocity of $A$ with Equation 5.12
12:     use $j_t$ to update the velocity of $B$ with Equation 5.13
13:     use $j_t$ to update the angular velocity of $A$ with Equation 5.14
14:     use $j_t$ to update the angular velocity of $B$ with Equation 5.15
15: end procedure
```

After looping through the list of contact points for a set number of times, the solver should stop and proceed to the final step.

Integrating Velocities

The last step of the sequential impulse solver approach is to integrate the velocity and angular velocity of each object in the space. The equation for integrating the velocity of the object is [11]:

$$p_{T+1} = p_T + v_T * \Delta T \tag{5.16}$$

In this equation, the object's velocity at the current time-step T is multiplied by delta time ΔT, and the resulting vector is added to the current position of the object. The equation for integrating the angular velocity of the object is [11]:

$$R_{T+1} = R_T - (\frac{1}{2} * \omega * \Delta T) R_T \tag{5.17}$$

In this equation, the angular velocity (represented as a bivector) is first divided by 2 and multiplied by delta time. Then, then the geometric product of the resulting bivector and the object's orientation (represented as a rotor) is calculated. The resulting rotor is then subtracted from the object's orientation.

Chapter 6

4D Rendering Pipeline

In order to render a 4D space, this project's approach resembles Bosch's approach by rendering 3D projections of the space. That said, whereas Bosch's projects represent 4D objects with meshes and use a slicing algorithm to render them [11, 9], this project instead represents 4D objects implicitly with signed distance functions and uses the ray marching algorithm to render them, similar to Vermandere's project [51].

Although different ray marching methods exist, this project's ray marching approach is based on *sphere tracing*, an iterative algorithm that uses spheres to march a point along a ray's direction vector until it gets close enough to an object or reaches some max distance [30, 36]. Notably, to implement the sphere tracing algorithm, objects in the space must be defined implicitly with a signed distance function (Appendix E). As the name suggests, these functions can be used to calculate the signed distance that a given point is from a primitive: points that have a positive distance are outside the primitive, points that have a negative distance are inside the primitive, and points that have a distance of 0 lie on the surface of the primitive [30]. One major advantage of this approach is that the implicit representations of 4D objects are defined with the same geometric properties as their 3D counterparts. For example, similar to how the signed distance function of a sphere outputs a value of 0 for all 3D points that are a fixed distance away from the sphere's 3D center point, the signed distance function of a hypersphere should output a value of 0 for all 4D points that are a fixed distance away from the hypersphere's 4D center point. For this reason, this project represents objects implicitly and implements the sphere tracing algorithm to render 3D projections of 4D space. This chapter provides an explanation of this project's camera model and its implementation of the sphere tracing algorithm.

6.1 The 4D Camera Model

In most 3D rendering pipelines, perspective cameras have three important properties: a position, orientation, and view frustum. A camera's position defines the location of its view in the world space; a camera's orientation defines the rotation of its view about the different planes of rotation; and a camera's view frustum defines the region of space that the camera will render [32, 41, 42]. The camera model for this project closely resembles a typical 3D perspective camera model, except that the camera has a 4D position and orientation. Like a 3D perspective camera, the view frustum of this project's camera model is also a 3D frustum. This means that this project's camera will only render 3D regions (or 3D projections) of the 4D space.

For traditional 3D perspective cameras, the first step to cast a ray from a pixel is to convert the pixel's coordinates into the camera's view space (by applying the camera's screen-to-view-space matrix). The next step is to calculate the direction vector from the camera's origin in view space (which is a 3D vector of all zeros) to the pixel's view space coordinates; notably, this direction vector represents the ray's direction in view space coordinates [5]. In order to express this direction vector in world space coordinates, it can be multiplied by the camera's rotation matrix. Additionally, to express the ray's origin in world space coordinates it can simply be set to the camera's position in world space [5].

For this project, the process of casting a ray from a 3D perspective camera with a 4D position and orientation is quite similar. Just as before, the pixel can be converted into the camera's view space by multiplying it by the camera's screen-to-view-space matrix (which is the same matrix as in 3D space). Then, the 3D direction vector from the pixel's view space coordinates to the camera's origin in view space (which is a 3D vector of all zeros) can be calculated [5]. This direction vector can then be converted into a 4D vector by setting its w-axis component to be 0. Then, the 4D direction vector should be multiplied by the camera's 4×4 rotation matrix, resulting in the ray's direction in 4D world space. Lastly, the camera's 4D position can be treated as the ray's origin.

To implement its rendering pipeline, this project uses an HLSL shader in the Unity Game Engine. This shader was built upon the implementation described by Peter Olthof in his YouTube series about implementing ray marching in Unity [36].

6.2 Overview of the Sphere Tracing Algorithm

For each pixel of the camera's image, the first step of the sphere tracing algorithm is to calculate the origin and direction of the pixel's ray (Section 6.1).

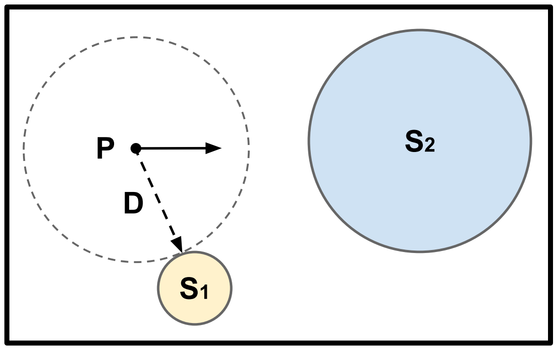

Figure 6.1: The ray has an origin P and a direction indicated by the solid arrow. The closest object to P is D units away.

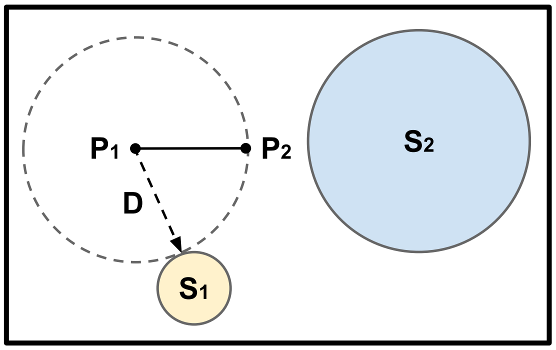

Figure 6.2: The ray's origin P_1 is translated D units along its direction vector, making its new origin P_2.

Once the pixel's ray has been defined, the sphere tracing algorithm can be performed. The algorithm begins by using the signed distance functions of the primitives in the space to calculate the distance D of the closest object to the ray's origin (Figure

6.1) [30, 36]. Since D corresponds to the shortest distance of any object to the ray's origin, no other objects can be within D units of the origin, which means that the spherical space (with a radius of D) around the ray's origin is empty. This means that the origin of the ray can be marched D units along its direction vector without intersecting any object (Figure 6.2).

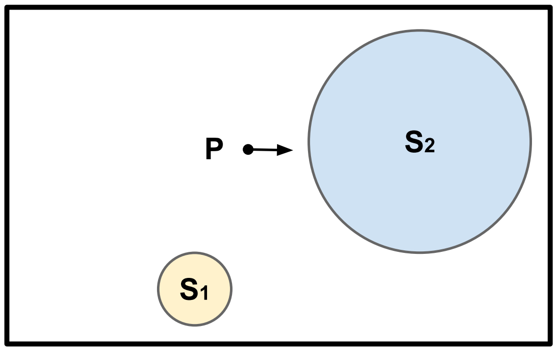

Figure 6.3: Since the ray's new origin is not on the surface of an object or inside an object, the algorithm repeats.

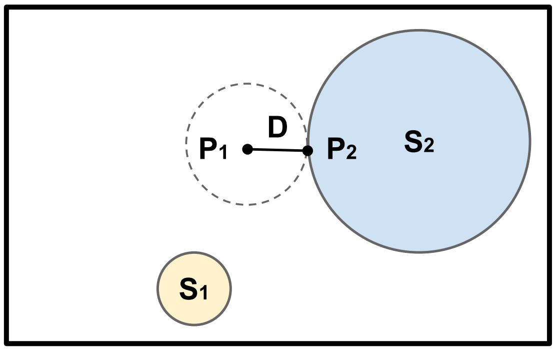

Figure 6.4: The ray's origin is translated along its direction. Since P_2 is on the surface of S_2, the algorithm terminates.

From this new origin, the algorithm repeats and calculates the distance from its new origin to the closest object (Figure 6.3). If this distance is greater than 0, then the new origin must lie outside of an object, which means that the algorithm can continue [30, 36]. If this distance is roughly 0 (or less than 0), then the new origin must lie on the object (or within it). This means that the algorithm can terminate and can return the new origin as the intersection point. The algorithm should also return the color of the object with which it intersected (e.g., for Figure 6.4, the algorithm would return the light blue color of S_2) [36].

In order to prevent the algorithm from repeating infinitely, a maximum number of iterations and a maximum amount of distance that can be marched should be set [36]. Notably, the image resolution of the algorithm's output improves whenever the maximum number of iterations is increased; however, the computational cost of the algorithm also increases with the maximum number of iterations. This means that the maximum number of iterations should be set at a reasonable value that balances both image resolution and performance. In pseudocode, the sphere tracing algorithm can be represented as:

Algorithm 14 Sphere Tracing

```
1: Input
2:     $o$         the origin of the ray
3:     $v$         the direction of the ray
4:     $d_{max}$   the maximum distance of objects that can be rendered
5:     $i_{max}$   the maximum number of ray marching iterations
6: Output
7:     $h$         a boolean that is true if an object has been hit and false otherwise
8:     $p$         the nearest intersection point of the ray and an object
9:     $c$         the color of the nearest object that was hit
10: procedure SphereTracing
11:     $h := false$
12:     $d_{total} := 0$                              ▷ The total distance marched so far
13:     $i := 0$
14:     repeat
15:         $p := o + d_{total} * v$
16:         $d, c :=$ DistToNearestObject($p$)
17:         if $d \lesssim 0$ then
18:             $h := true$
19:         end if
20:         $d_{total} := d_{total} + d$
21:     until $i == i_{max}$ or $d_{total} > d_{max}$ or $h$ is $true$
22:     return $h, p, c$
23: end procedure
```

For the purposes of this project, the maximum number of iterations is 200, and the maximum distance that can be marched is 200 units.

6.2.1 Implementing a Lambertian Shading Model

Once the sphere tracer gets the color of the object with which it intersected, the color should be adjusted based on some lighting model in order to simulate the effect of light hitting objects. To do this, the first value that should be calculated is the normal at the intersection point. Since the gradient of the signed distance field at a given point is equal to the point's normal vector [38, 36], the algorithm should calculate the derivative of the signed distance field at the point along each of the orthogonal axis vectors (i.e., the x, y, z, and w-axes). This can be done by calculating the distances of the signed distance field from a slight positive and negative offset from the intersection point [38, 36]. Taking the difference of these distances results in the normal's corresponding value for that axis. The following pseudocode demonstrates how to do this:

Algorithm 15 Calculating the Normal at a Point on the SDF

```
 1: Input
 2:     $p$        the intersection point
 3: Output
 4:     $n$        the normal of the signed distance field at the intersection point
 5: procedure CalculatePointNormal
 6:     initialize a new vector $n$
 7:     $o := 0.0001$                          ▷ Sets the offset to be some small value
 8:     for each orthogonal axis vector $v$ do
 9:         $p_o := p$                         ▷ Creates a copy of the intersection point
10:         set the $v$-axis value of $p_o$ to be $p.v + o$
11:         $d^+ :=$ DistToNearestObject($p_o$)
12:         set the $v$-axis value of $p_o$ to be $p.v - o$
13:         $d^- :=$ DistToNearestObject($p_o$)
14:         set the $v$-axis value of $n$ to be $(d^+ - d^-)$
15:     end for
16:     return $n$
17: end procedure
```

With the normal vector at the contact point, the lighting at the contact point can be calculated. Similar to Olthof's ray marching implementation [36], this project uses the Lambertian (or Diffuse) Shading Model due its simplicity. Given the normal at the point (n), the normalized direction of the light (l), the color of the light (c), and the intensity of the light (i), the Lambertian lighting at a point can be calculated as:

$$s = (l \cdot n) * c * i \tag{6.1}$$

This value of s can then be multiplied by the color of the nearest object that was found with the sphere tracing algorithm.

6.2.2 Casting Shadows

The sphere tracing algorithm can be easily extended to implement shadows as well. Given the point of intersection and the light's direction, the sphere tracing algorithm can be performed again to determine if an object lies between the intersection point and the light source [36]. If the sphere tracing algorithm detects another object between the intersection point and the light source, then the original intersection point should have a shadow on it. To ensure that the algorithm terminates, the sphere tracing algorithm should stop after a certain max distance has been reached.

Rather than simply rendering the color of an intersection point whenever the line between the point and the light source is unobstructed, the algorithm should instead calculate the partial light obstruction of any objects that are close to the intersection point [39, 36]. In particular, objects that are closer to the intersection point should cast darker partial shadows [39]. Additionally, objects that are closer to the ray from the intersection point to the light source should also cast darker partial shadows [39].

Chapter 7

Discussion

7.1 Project Summary

This project began by describing the ways in which geometric algebra can be used to generalize different calculations from traditional vector algebra to higher-dimensional spaces. After explaining the key concepts of geometric algebra, this project then described how to represent the 4D counterparts of the most essential 3D primitives. Next, this project explained the different collision detection algorithms that can be used to determine whether a pair of 4D objects were colliding. In order to resolve collisions between 4D primitives, this project suggested using a generalized version of Catto's sequential impulse solver approach (which involves repeatedly applying and accumulating impulses during each time-step). Lastly, this project explained how to use the sphere tracing algorithm to render 3D projections of 4D spaces.

7.2 Limitations and Future Work

Since Catto's sequential impulse solver approach assumes that all collisions are inelastic, one limitation of this project's physics engine is that it cannot be used to simulate elastic (or partially elastic) collisions. Thus, future work on this project could be done to modify its sequential impulse solver to incorporate elastic collisions as well [3].

Another limitation of this project relates to the ray marching algorithm itself. Although sphere tracing is an intuitive method for rendering 3D projections of 4D primitives, one major downside of the algorithm is that it suffers from poor performance whenever the signed distance field of scene is very complex. That said, recent scholarship has proposed different optimizations to the sphere tracing algorithm (such as the segment tracing algorithm [23]), which means that future work on this project could involve optimizing its ray marching approach further.

Despite including a rudimentary player controller, a major limitation of this project has to do with its lack of robust player movement or interaction with the space. In the project's current implementation, players can only move with respect to their local z-axis direction (by pressing the up and down arrows) and their local w-axis direction (by pressing the right and left arrows). A more robust implementation of player movement would allow players to move along all 4 possible axes simultaneously, giving them the widest range of movement possibilities in the space.

7.3 Conclusion

Given the scarcity of resources available that outline all of the steps necessary for simulating 4D space, this project and its associated code are meant to serve as a working example of a simple simulation of 4D space. Although other approaches may also exist for implementing a 4D physics engine and rendering pipeline, the approaches outlined in this paper can serve as a good starting point for future algorithms and projects.

Appendix A

Examples of 4D Geometric Algebra

A.1 The Outer Product of a Vector and a Bivector

The outer product of a 4D vector a and bivector b can be calculated as follows:

$$\begin{aligned}
a \wedge b &= (a_x e_x + a_y e_y + a_z e_z + a_w e_w) \wedge (b_{xy} e_{xy} + b_{xz} e_{xz} + b_{xw} e_{xw} + b_{yz} e_{yz} \\
&\quad + b_{yw} e_{yw} + b_{zw} e_{zw}) \\
&= a_x b_{xy} e_{xxy} + a_x b_{xz} e_{xxz} + a_x b_{xw} e_{xxw} + a_x b_{yz} e_{xyz} + a_x b_{yw} e_{xyw} + a_x b_{zw} e_{xzw} \\
&\quad + a_y b_{xy} e_{yxy} + a_y b_{xz} e_{yxz} + a_y b_{xw} e_{yxw} + a_y b_{yz} e_{yyz} + a_y b_{yw} e_{yyw} + a_y b_{zw} e_{yzw} \\
&\quad + a_z b_{xy} e_{zxy} + a_z b_{xz} e_{zxz} + a_z b_{xw} e_{zxw} + a_z b_{yz} e_{zyz} + a_z b_{yw} e_{zyw} + a_z b_{zw} e_{zzw} \\
&\quad + a_w b_{xy} e_{wxy} + a_w b_{xz} e_{wxz} + a_w b_{xw} e_{wxw} + a_w b_{yz} e_{wyz} + a_w b_{yw} e_{wyw} + a_w b_{zw} e_{wzw}
\end{aligned}$$

The outer product of a vector with itself is equal to 0, and the outer product of two vectors is anti-symmetric. As a result, this can be further simplified to be:

$$\begin{aligned}
a \wedge b &= a_x b_{yz} e_{xyz} + a_x b_{yw} e_{xyw} + a_x b_{zw} e_{xzw} - a_y b_{xz} e_{xyz} - a_y b_{xw} e_{xyw} + a_y b_{zw} e_{yzw} \\
&\quad + a_z b_{xy} e_{xyz} - a_z b_{xw} e_{xzw} - a_z b_{yw} e_{yzw} + a_w b_{xy} e_{xyw} + a_w b_{xz} e_{xzw} + a_w b_{yz} e_{yzw}
\end{aligned}$$

Taking the dual of all trivector terms yields the following:

$$\begin{aligned}a \wedge b = &-a_x b_{yz} e_w + a_x b_{yw} e_z - a_x b_{zw} e_y + a_y b_{xz} e_w - a_y b_{xw} e_z + a_y b_{zw} e_x \\ &-a_z b_{xy} e_w + a_z b_{xw} e_y - a_z b_{yw} e_x + a_w b_{xy} e_z - a_w b_{xz} e_y + a_w b_{yz} e_x \\ = &(a_y b_{zw} - a_z b_{yw} + a_w b_{yz}) e_x + (-a_x b_{zw} + a_z b_{xw} - a_w b_{xz}) e_y \\ &+(a_x b_{yw} - a_y b_{xw} + a_w b_{xy}) e_z + (-a_x b_{yz} + a_y b_{xz} - a_z b_{xy}) e_w\end{aligned}$$

This can be rewritten with the following traditional vector notation:

$$a \wedge b = \begin{bmatrix} a_y b_{zw} - a_z b_{yw} + a_w b_{yz} \\ -a_x b_{zw} + a_z b_{xw} - a_w b_{xz} \\ a_x b_{yw} - a_y b_{xw} + a_w b_{xy} \\ -a_x b_{yz} + a_y b_{xz} - a_z b_{xy} \end{bmatrix}$$

Notably, the outer product of a vector and a bivector is symmetric, so $a \wedge b = b \wedge a$.

A.2 The Left Inner Product of a Vector and a Bivector

The left inner product of a 4D vector a and a bivector b can be calculated as:

$$\begin{aligned}
a \rfloor b &= (a_x e_x + a_y e_y + a_z e_z + a_w e_w) \rfloor (b_{xy} e_{xy} + b_{xz} e_{xz} + b_{xw} e_{xw} + b_{yz} e_{yz} \\
&\quad + b_{yw} e_{yw} + b_{zw} e_{zw}) \\
&= a_x b_{xy} e_x \rfloor e_{xy} + a_x b_{xz} e_x \rfloor e_{xz} + a_x b_{xw} e_x \rfloor e_{xw} + a_x b_{yz} e_x \rfloor e_{yz} + a_x b_{yw} e_x \rfloor e_{yw} \\
&\quad + a_x b_{zw} e_x \rfloor e_{zw} + a_y b_{xy} e_y \rfloor e_{xy} + a_y b_{xz} e_y \rfloor e_{xz} + a_y b_{xw} e_y \rfloor e_{xw} + a_y b_{yz} e_y \rfloor e_{yz} \\
&\quad + a_y b_{yw} e_y \rfloor e_{yw} + a_y b_{zw} e_y \rfloor e_{zw} + a_z b_{xy} e_z \rfloor e_{xy} + a_z b_{xz} e_z \rfloor e_{xz} + a_z b_{xw} e_z \rfloor e_{xw} \\
&\quad + a_z b_{yz} e_z \rfloor e_{yz} + a_z b_{yw} e_z \rfloor e_{yw} + a_z b_{zw} e_z \rfloor e_{zw} + a_w b_{xy} e_w \rfloor e_{xy} + a_w b_{xz} e_w \rfloor e_{xz} \\
&\quad + a_w b_{xw} e_w \rfloor e_{xw} + a_w b_{yz} e_w \rfloor e_{yz} + a_w b_{yw} e_w \rfloor e_{yw} + a_w b_{zw} e_w \rfloor e_{zw}
\end{aligned}$$

The left inner product between a vector and an orthogonal bivector is equal to 0. Additionally, since the left inner product of a vector and itself is equal to the vector's square magnitude, the left inner product of a unit vector with itself is simply 1. Thus, the sum above can be further simplified to be:

$$\begin{aligned}
a \rfloor b &= a_x b_{xy} e_x \rfloor e_{xy} + a_x b_{xz} e_x \rfloor e_{xz} + a_x b_{xw} e_x \rfloor e_{xw} + a_y b_{xy} e_y \rfloor e_{xy} \\
&\quad + a_y b_{yz} e_y \rfloor e_{yz} + a_y b_{yw} e_y \rfloor e_{yw} + a_z b_{xz} e_z \rfloor e_{xz} + a_z b_{yz} e_z \rfloor e_{yz} \\
&\quad + a_z b_{zw} e_z \rfloor e_{zw} + a_w b_{xw} e_w \rfloor e_{xw} + a_w b_{yw} e_w \rfloor e_{yw} + a_w b_{zw} e_w \rfloor e_{zw} \\
&= a_x b_{xy} e_y + a_x b_{xz} e_z + a_x b_{xw} e_w - a_y b_{xy} e_x + a_y b_{yz} e_z + a_y b_{yw} e_w \\
&\quad - a_z b_{xz} e_x - a_z b_{yz} e_y + a_z b_{zw} e_w - a_w b_{xw} e_x - a_w b_{yw} e_y - a_w b_{zw} e_z \\
&= (-a_y b_{xy} - a_z b_{xz} e_x - a_w b_{xw}) e_x + (a_x b_{xy} - a_z b_{yz} - a_w b_{yw}) e_y \\
&\quad + (a_x b_{xz} + a_y b_{yz} - a_w b_{zw}) e_z + (a_x b_{xw} + a_y b_{yw} + a_z b_{zw}) e_w
\end{aligned}$$

This can be rewritten with traditional vector notation as:

$$a \lrcorner b = \begin{bmatrix} -a_y b_{xy} - a_z b_{xz} e_x - a_w b_{xw} \\ a_x b_{xy} - a_z b_{yz} - a_w b_{yw} \\ a_x b_{xz} + a_y b_{yz} - a_w b_{zw} \\ a_x b_{xw} + a_y b_{yw} + a_z b_w \end{bmatrix}$$

Since a vector a has a grade of 1 and a bivector b has a grade of 2, the right inner product of b and a is equal to $(-1)^{1(2-1)} a \lrcorner b = -a \lrcorner b$.

A.3 The Geometric Product of a Vector and a Bivector

The geometric product of a 4D vector a and a bivector b can be calculated as follows:

$$\begin{aligned}
ab &= (a_x e_x + a_y e_y + a_z e_z + a_w e_w)(b_{xy} e_{xy} + b_{xz} e_{xz} + b_{xw} e_{xw} + b_{yz} e_{yz} + b_{yw} e_{yw} + b_{zw} e_{zw}) \\
&= a_x b_{xy} e_{xxy} + a_x b_{xz} e_{xxz} + a_x b_{xw} e_{xxw} + a_x b_{yz} e_{xyz} + a_x b_{yw} e_{xyw} + a_x b_{zw} e_{xzw} \\
&\quad + a_y b_{xy} e_{yxy} + a_y b_{xz} e_{yxz} + a_y b_{xw} e_{yxw} + a_y b_{yz} e_{yyz} + a_y b_{yw} e_{yyw} + a_y b_{zw} e_{yzw} \\
&\quad + a_z b_{xy} e_{zxy} + a_z b_{xz} e_{zxz} + a_z b_{xw} e_{zxw} + a_z b_{yz} e_{zyz} + a_z b_{yw} e_{zyw} + a_z b_{zw} e_{zzw} \\
&\quad + a_w b_{xy} e_{wxy} + a_w b_{xz} e_{wxz} + a_w b_{xw} e_{wxw} + a_w b_{yz} e_{wyz} + a_w b_{yw} e_{wyw} + a_w b_{zw} e_{wzw}
\end{aligned}$$

The geometric product of a unit vector with itself is equal to 1, and the geometric product of two orthogonal vectors is anti-symmetric. Thus, this can be simplified as:

$$\begin{aligned}
ab &= a_x b_{xy} e_{xxy} + a_x b_{xz} e_{xxz} + a_x b_{xw} e_{xxw} + a_x b_{yz} e_{xyz} + a_x b_{yw} e_{xyw} + a_x b_{zw} e_{xzw} \\
&\quad - a_y b_{xy} e_{xyy} - a_y b_{xz} e_{xyz} - a_y b_{xw} e_{xyw} + a_y b_{yz} e_{yyz} + a_y b_{yw} e_{yyw} + a_y b_{zw} e_{yzw} \\
&\quad + a_z b_{xy} e_{xyz} - a_z b_{xz} e_{xzz} - a_z b_{xw} e_{xzw} - a_z b_{yz} e_{yzz} - a_z b_{yw} e_{yzw} + a_z b_{zw} e_{zzw} \\
&\quad + a_w b_{xy} e_{xyw} + a_w b_{xz} e_{xzw} - a_w b_{xw} e_{xww} + a_w b_{yz} e_{yzw} - a_w b_{yw} e_{yww} - a_w b_{zw} e_{zww} \\
&= a_x b_{xy} e_y + a_x b_{xz} e_z + a_x b_{xw} e_w + a_x b_{yz} e_{xyz} + a_x b_{yw} e_{xyw} + a_x b_{zw} e_{xzw} \\
&\quad - a_y b_{xy} e_x - a_y b_{xz} e_{xyz} - a_y b_{xw} e_{xyw} + a_y b_{yz} e_z + a_y b_{yw} e_w + a_y b_{zw} e_{yzw} \\
&\quad + a_z b_{xy} e_{xyz} - a_z b_{xz} e_x - a_z b_{xw} e_{xzw} - a_z b_{yz} e_y - a_z b_{yw} e_{yzw} + a_z b_{zw} e_w \\
&\quad + a_w b_{xy} e_{xyw} + a_w b_{xz} e_{xzw} - a_w b_{xw} e_x + a_w b_{yz} e_{yzw} - a_w b_{yw} e_y - a_w b_{zw} e_z
\end{aligned}$$

This can be further simplified by taking the duals of the trivector terms:

$$
\begin{aligned}
ab = {} & a_x b_{xy} e_y + a_x b_{xz} e_z + a_x b_{xw} e_w - a_x b_{yz} e_w + a_x b_{yw} e_z - a_x b_{zw} e_y \\
& - a_y b_{xy} e_x + a_y b_{xz} e_w - a_y b_{xw} e_z + a_y b_{yz} e_z + a_y b_{yw} e_w + a_y b_{zw} e_x \\
& - a_z b_{xy} e_w - a_z b_{xz} e_x + a_z b_{xw} e_y - a_z b_{yz} e_y - a_z b_{yw} e_x + a_z b_{zw} e_w \\
& + a_w b_{xy} e_z - a_w b_{xz} e_y - a_w b_{xw} e_x + a_w b_{yz} e_x - a_w b_{yw} e_y - a_w b_{zw} e_z \\
= {} & (-a_y b_{xy} + a_y b_{zw} - a_z b_{xz} - a_z b_{yw} - a_w b_{xw} + a_w b_{yz}) e_x \\
& + (a_x b_{xy} - a_x b_{zw} + a_z b_{xw} - a_z b_{yz} - a_w b_{xz} - a_w b_{yw}) e_y \\
& + (a_x b_{xz} + a_x b_{yw} - a_y b_{xw} + a_y b_{yz} + a_w b_{xy} - a_w b_{zw}) e_z \\
& + (a_x b_{xw} - a_x b_{yz} + a_y b_{xz} + a_y b_{yw} - a_z b_{xy} + a_z b_{zw}) e_w
\end{aligned}
$$

This sum can now be represented as the following 4D vector:

$$
ab = \begin{bmatrix}
-a_y b_{xy} + a_y b_{zw} - a_z b_{xz} - a_z b_{yw} - a_w b_{xw} + a_w b_{yz} \\
a_x b_{xy} - a_x b_{zw} + a_z b_{xw} - a_z b_{yz} - a_w b_{xz} - a_w b_{yw} \\
a_x b_{xz} + a_x b_{yw} - a_y b_{xw} + a_y b_{yz} + a_w b_{xy} - a_w b_{zw} \\
a_x b_{xw} - a_x b_{yz} + a_y b_{xz} + a_y b_{yw} - a_z b_{xy} + a_z b_{zw}
\end{bmatrix}
$$

Notably, the geometric product of a vector and a bivector is not symmetric (i.e, $ab \neq ba$ for vector a and bivector b). For the sake of completion, the geometric product ba is equal to the following vector:

$$
ba = \begin{bmatrix}
a_y b_{xy} + a_y b_{zw} + a_z b_{xz} - a_z b_{yw} + a_w b_{xw} + a_w b_{yz} \\
-a_x b_{xy} - a_x b_{zw} + a_z b_{xw} + a_z b_{yz} - a_w b_{xz} + a_w b_{yw} \\
-a_x b_{xz} + a_x b_{yw} - a_y b_{xw} - a_y b_{yz} + a_w b_{xy} + a_w b_{zw} \\
-a_x b_{xw} - a_x b_{yz} + a_y b_{xz} - a_y b_{yw} - a_z b_{xy} - a_z b_{zw}
\end{bmatrix}
$$

A.4 The Geometric Product of a Vector and a Rotor

The geometric product of a 4D vector a and a rotor r can be calculated as:

$$
\begin{aligned}
ar &= (a_xe_x + a_ye_y + a_ze_z + a_we_w)(r_s + r_{xy}e_{xy} + r_{xz}e_{xz} + r_{xw}e_{xw} + r_{yz}e_{yz} + r_{yw}e_{yw} \\
&\quad + r_{zw}e_{zw} + r_{xyzw}e_{xyzw}) \\
&= a_xr_se_x + a_xr_{xy}e_{xxy} + a_xr_{xz}e_{xxz} + a_xr_{xw}e_{xxw} + a_xr_{yz}e_{xyz} + a_xr_{yw}e_{xyw} + a_xr_{zw}e_{xzw} \\
&\quad + a_xr_{xyzw}e_{xxyzw} + a_yr_se_y + a_yr_{xy}e_{yxy} + a_yr_{xz}e_{yxz} + a_yr_{xw}e_{yxw} + a_yr_{yz}e_{yyz} \\
&\quad + a_yr_{yw}e_{yyw} + a_yr_{zw}e_{yzw} + a_yr_{xyzw}e_{yxyzw} + a_zr_se_z + a_zr_{xy}e_{zxy} + a_zr_{xz}e_{zxz} \\
&\quad + a_zr_{xw}e_{zxw} + a_zr_{yz}e_{zyz} + a_zr_{yw}e_{zyw} + a_zr_{zw}e_{zzw} + a_zr_{xyzw}e_{zxyzw} + a_wr_se_w \\
&\quad + a_wr_{xy}e_{wxy} + a_wr_{xz}e_{wxz} + a_wr_{xw}e_{wxw} + a_wr_{yz}e_{wyz} + a_wr_{yw}e_{wyw} \\
&\quad + a_wr_{zw}e_{wzw} + a_wr_{xyzw}e_{wxyzw} \\
&= a_xr_se_x + a_xr_{xy}e_y + a_xr_{xz}e_z + a_xr_{xw}e_w + a_xr_{yz}e_z + a_xr_{yw}e_z - a_xr_{zw}e_y \\
&\quad + a_xr_{xyzw}e_x + a_yr_se_y - a_yr_{xy}e_x + a_yr_{xz}e_w - a_yr_{xw}e_z + a_yr_{yz}e_z + a_yr_{yw}e_w \\
&\quad + a_yr_{zw}e_x + a_yr_{xyzw}e_y + a_zr_se_z - a_zr_{xy}e_w - a_zr_{xz}e_x + a_zr_{xw}e_y - a_zr_{yz}e_y \\
&\quad - a_zr_{yw}e_x + a_zr_{zw}e_w + a_zr_{xyzw}e_z + a_wr_se_w + a_wr_{xy}e_z - a_wr_{xz}e_y - a_wr_{xw}e_x \\
&\quad + a_wr_{yz}e_x - a_wr_{yw}e_y - a_wr_{zw}e_z + a_wr_{xyzw}e_w \\
&= (a_xr_s - a_yr_{xy} + a_yr_{zw} - a_zr_{xz} - a_zr_{yw} - a_wr_{xw} + a_wr_{yz} + a_xr_{xyzw})e_x \\
&\quad + (a_yr_s + a_xr_{xy} - a_xr_{zw} + a_zr_{xw} - a_zr_{yz} - a_wr_{xz} - a_wr_{yw} + a_yr_{xyzw})e_y \\
&\quad + (a_zr_s + a_xr_{xz} + a_xr_{yw} - a_yr_{xw} + a_yr_{yz} + a_wr_{xy} - a_wr_{zw} + a_zr_{xyzw})e_z \\
&\quad + (a_wr_s + a_xr_{xw} - a_xr_{yz} + a_yr_{xz} + a_yr_{yw} - a_zr_{xy} + a_zr_{zw} + a_wr_{xyzw})e_w
\end{aligned}
$$

This sum can now be represented as the following 4D vector:

$$ar = \begin{bmatrix} a_x r_s - a_y r_{xy} + a_y r_{zw} - a_z r_{xz} - a_z r_{yw} - a_w r_{xw} + a_w r_{yz} + a_x r_{xyzw} \\ a_y r_s + a_x r_{xy} - a_x r_{zw} + a_z r_{xw} - a_z r_{yz} - a_w r_{xz} - a_w r_{yw} + a_y r_{xyzw} \\ a_z r_s + a_x r_{xz} + a_x r_{yw} - a_y r_{xw} + a_y r_{yz} + a_w r_{xy} - a_w r_{zw} + a_z r_{xyzw} \\ a_w r_s + a_x r_{xw} - a_x r_{yz} + a_y r_{xz} + a_y r_{yw} - a_z r_{xy} + a_z r_{zw} + a_w r_{xyzw} \end{bmatrix}$$

The geometric product of a vector and a rotor is not symmetric, so for the sake of completion, the geometric product ra is equal to the following vector:

$$ra = \begin{bmatrix} a_x r_s + a_y r_{xy} + a_y r_{zw} + a_z r_{xz} - a_z r_{yw} + a_w r_{xw} + a_w r_{yz} - a_x r_{xyzw} \\ a_y r_s - a_x r_{xy} - a_x r_{zw} + a_z r_{xw} + a_z r_{yz} - a_w r_{xz} + a_w r_{yw} - a_y r_{xyzw} \\ a_z r_s - a_x r_{xz} + a_x r_{yw} - a_y r_{xw} - a_y r_{yz} + a_w r_{xy} + a_w r_{zw} - a_z r_{xyzw} \\ a_w r_s - a_x r_{xw} - a_x r_{yz} + a_y r_{xz} - a_y r_{yw} - a_z r_{xy} - a_z r_{zw} - a_w r_{xyzw} \end{bmatrix}$$

A.5 The Geometric Product of Two Bivectors

The geometric product of a bivectors a and b can be calculated as:

$$\begin{aligned}
ab &= (a_{xy}e_{xy} + a_{xz}e_{xz} + a_{xw}e_{xw} + a_{yz}e_{yz} + a_{yw}e_{yw} + a_{zw}e_{zw})(b_{xy}e_{xy} + b_{xz}e_{xz} + b_{xw}e_{xw} \\
&\quad + b_{yz}e_{yz} + b_{yw}e_{yw} + b_{zw}e_{zw}) \\
&= a_{xy}b_{xy} - a_{xy}b_{xz}e_{yz} - a_{xy}b_{xw}e_{yw} + a_{xy}b_{yz}e_{xz} + a_{xy}b_{yw}e_{xw} \\
&\quad + a_{xy}a_{zw}e_{xyzw} + a_{xz}b_{xy}e_{yz} - a_{xz}b_{xz} - a_{xz}b_{xw}e_{zw} \\
&\quad - a_{xz}b_{yz}e_{xy} - a_{xz}b_{yw}e_{xyzw} + a_{xz}b_{zw}e_{xw} + a_{xw}b_{xy}e_{yw} \\
&\quad + a_{xw}b_{xz}e_{zw} - a_{xw}b_{xw} + a_{xw}b_{yz}e_{xyzw} - a_{xw}b_{yw}e_{xy} - a_{xw}b_{zw}e_{xz} \\
&\quad - a_{yz}b_{xy}e_{xz} + a_{yz}b_{xz}e_{xy} + a_{yz}b_{xw}e_{xyzw} - a_{yz}b_{yz} - a_{yz}b_{yw}e_{zw} \\
&\quad + a_{yz}b_{zw}e_{yw}e_{xw} - a_{yw}b_{xy}e_{xw} - a_{yw}b_{xz}e_{xyzw} + a_{yw}b_{xw}e_{xy} \\
&\quad + a_{yw}b_{yz}e_{zw} - a_{yw}b_{yw} - a_{yw}b_{zw}e_{yz} + a_{zw}b_{xy}e_{xyzw} \\
&\quad - a_{zw}b_{xz}e_{xw} + a_{zw}b_{xw}e_{xz} - a_{zw}b_{yz}e_{yw} + a_{zw}b_{yw}e_{yz} - a_{zw}b_{zw} \\
&= (a_{xy}b_{xy} - a_{xz}b_{xz} - a_{xw}b_{xw} - a_{yz}b_{yz} - a_{yw}b_{yw} - a_{zw}b_{zw}) \\
&\quad + (-a_{xz}b_{yz} - a_{xw}b_{yw} + a_{yz}b_{xz} + a_{yw}b_{xw})e_{xy} \\
&\quad + (a_{xy}b_{yz} - a_{xw}b_{zw} - a_{yz}b_{xy} + a_{zw}b_{xw})e_{xz} \\
&\quad + (a_{xy}b_{yw} + a_{xz}b_{zw} - a_{yw}b_{xy} - a_{zw}b_{xz})e_{xw} \\
&\quad + (-a_{xy}b_{xz} + a_{xz}b_{xy} - a_{yw}b_{zw} + a_{zw}b_{yw})e_{yz} \\
&\quad + (-a_{xy}b_{xw} + a_{xw}b_{xy} + a_{yz}b_{zw} - a_{zw}b_{yz})e_{yw} \\
&\quad + (-a_{xz}b_{xw} + a_{xw}b_{xz} - a_{yz}b_{yw} + a_{yw}b_{yz})e_{zw} \\
&\quad + (a_{xy}b_{zw} - a_{xz}b_{yw} + a_{xw}b_{yz} + a_{yz}b_{xw} - a_{yw}b_{xz} + a_{zw}b_{xy})e_{xyzw}
\end{aligned}$$

Notably, this demonstrates that the geometric product of two simple rotors R_1 and R_2 (which are bivectors) in 4D space produces a complex rotor with a scalar term, six bivector terms, and one pseudoscalar term.

A.6 The Geometric Product of a Bivector and a Rotor

The geometric product of a bivector b and a rotor r can be calculated as:

$$\begin{aligned}
br &= (b_{xy}e_{xy} + b_{xz}e_{xz} + b_{xw}e_{xw} + b_{yz}e_{yz} + b_{yw}e_{yw} + b_{zw}e_{zw})(r_s + r_{xy}e_{xy} + r_{xz}e_{xz} + r_{xw}e_{xw} \\
&\quad + r_{yz}e_{yz} + r_{yw}e_{yw} + r_{zw}e_{zw} + r_{xyzw}e_{xyzw}) \\
&= b_{xy}r_se_{xy} - b_{xy}r_{xy} - b_{xy}r_{xz}e_{yz} - b_{xy}r_{xw}e_{yw} + b_{xy}r_{yz}e_{xz} + b_{xy}r_{yw}e_{xw} \\
&\quad + b_{xy}r_{zw}e_{xyzw} - b_{xy}r_{xyzw}e_{zw} + b_{xz}r_se_{xz} + b_{xz}r_{xy}e_{yz} - b_{xz}r_{xz} - b_{xz}r_{xw}e_{zw} \\
&\quad - b_{xz}r_{yz}e_{xy} - b_{xz}r_{yw}e_{xyzw} + b_{xz}r_{zw}e_{xw} + b_{xz}r_{xyzw}e_{yw} + b_{xw}r_se_{xw} + b_{xw}r_{xy}e_{yw} \\
&\quad + b_{xw}r_{xz}e_{zw} - b_{xw}r_{xw} + b_{xw}r_{yz}e_{xyzw} - b_{xw}r_{yw}e_{xy} - b_{xw}r_{zw}e_{xz} - b_{xw}r_{xyzw}e_{yz} \\
&\quad + b_{yz}r_se_{yz} - b_{yz}r_{xy}e_{xz} + b_{yz}r_{xz}e_{xy} + b_{yz}r_{xw}e_{xyzw} - b_{yz}r_{yz} - b_{yz}r_{yw}e_{zw} \\
&\quad + b_{yz}r_{zw}e_{yw} - b_{yz}r_{xyzw}e_{xw} + b_{yw}r_se_{yw} - b_{yw}r_{xy}e_{xw} - b_{yw}r_{xz}e_{xyzw} + b_{yw}r_{xw}e_{xy} \\
&\quad + b_{yw}r_{yz}e_{zw} - b_{yw}r_{yw} - b_{yw}r_{zw}e_{yz} + b_{yw}r_{xyzw}e_{xz} + b_{zw}r_se_{zw} + b_{zw}r_{xy}e_{xyzw} \\
&\quad - b_{zw}r_{xz}e_{xw} + b_{zw}r_{xw}e_{xz} - b_{zw}r_{yz}e_{yw} + b_{zw}r_{yw}e_{yz} - b_{zw}r_{zw} - b_{zw}r_{xyzw}e_{xy} \\
&= (b_{xy}r_{xy} - b_{xz}r_{xz} - b_{xw}r_{xw} - b_{yz}r_{yz} - b_{yw}r_{yw} - b_{zw}r_{zw}) \\
&\quad + (b_{xy}r_s - b_{xz}r_{yz} - b_{xw}r_{yw} + b_{yz}r_{xz} + b_{yw}r_{xw} - b_{zw}r_{xyzw})e_{xy} \\
&\quad + (b_{xy}r_{yz} + b_{xz}r_s - b_{xw}r_{zw} - b_{yz}r_{xy} + b_{yw}r_{xyzw} + b_{zw}r_{xw})e_{xz} \\
&\quad + (b_{xy}r_{yw} + b_{xz}r_{zw} + b_{xw}r_s - b_{yz}r_{xyzw} - b_{yw}r_{xy} - b_{zw}r_{xz})e_{xw} \\
&\quad + (-b_{xy}r_{xz} + b_{xz}r_{xy} - b_{xw}r_{xyzw} + b_{yz}r_s - b_{yw}r_{zw} + b_{zw}r_{yw})e_{yz} \\
&\quad + (-b_{xy}r_{xw} + b_{xz}r_{xyzw} + b_{xw}r_{xy} + b_{yz}r_{zw} + b_{yw}r_s - b_{zw}r_{yz})e_{yw} \\
&\quad + (-b_{xy}r_{xyzw} - b_{xz}r_{xw} + b_{xw}r_{xz} - b_{yz}r_{yw} + b_{yw}r_{yz} + b_{zw}r_s)e_{zw} \\
&\quad + (b_{xy}r_{zw} - b_{xz}r_{yw} + b_{xw}r_{yz} + b_{yz}r_{xw} - b_{yw}r_{xz} + b_{zw}r_{xy})e_{xyzw}
\end{aligned}$$

The geometric product of a bivector and a rotor is symmetric, which means that $br = rb$.

A.7 The Geometric Product of Two Rotors

The geometric product of two rotors is:

$$
\begin{aligned}
pq &= (p_s + p_{xy}e_{xy} + p_{xz}e_{xz} + p_{xw}e_{xw} + p_{yz}e_{yz} + p_{yw}e_{yw} + p_{zw}e_{zw} + p_{xyzw}e_{xyzw})\\
&\quad (q_s + q_{xy}e_{xy} + q_{xz}e_{xz} + q_{xw}e_{xw} + q_{yz}e_{yz} + q_{yw}e_{yw} + q_{zw}e_{zw} + q_{xyzw}e_{xyzw})\\
&= p_sq_s + p_sq_{xy}e_{xy} + p_sq_{xz}e_{xz} + p_sq_{xw}e_{xw} + p_sq_{yz}e_{yz} + p_sq_{yw}e_{yw} + p_sq_{zw}e_{zw}\\
&\quad + p_sq_{xyzw}e_{xyzw} + p_{xy}q_se_{xy} - p_{xy}q_{xy} - p_{xy}q_{xz}e_{yz} - p_{xy}q_{xw}e_{yw} + p_{xy}q_{yz}e_{xz}\\
&\quad + p_{xy}q_{yw}e_{xw} + p_{xy}q_{zw}e_{xyzw} - p_{xy}q_{xyzw}e_{zw} + p_{xz}q_se_{xz} + p_{xz}q_{xy}e_{yz} - p_{xz}q_{xz}\\
&\quad - p_{xz}q_{xw}e_{zw} - p_{xz}q_{yz}e_{xy} - p_{xz}q_{yw}e_{xyzw} + p_{xz}q_{zw}e_{xw} + p_{xz}q_{xyzw}e_{yw} + p_{xw}q_se_{xw}\\
&\quad + p_{xw}q_{xy}e_{yw} + p_{xw}q_{xz}e_{zw} - p_{xw}q_{xw} + p_{xw}q_{yz}e_{xyzw} - p_{xw}q_{yw}e_{xy} - p_{xw}q_{zw}e_{xz}\\
&\quad - p_{xw}q_{xyzw}e_{yz} + p_{yz}q_se_{yz} - p_{yz}q_{xy}e_{xz} + p_{yz}q_{xz}e_{xy} + p_{yz}q_{xw}e_{xyzw} - p_{yz}q_{yz}\\
&\quad - p_{yz}q_{yw}e_{zw} + p_{yz}q_{zw}e_{yw} - p_{yz}q_{xyzw}e_{xw} + p_{yw}q_se_{yw} - p_{yw}q_{xy}e_{xw} - p_{yw}q_{xz}e_{xyzw}\\
&\quad + p_{yw}q_{xw}e_{xy} + p_{yw}q_{yz}e_{zw} - p_{yw}q_{yw} - p_{yw}q_{zw}e_{yz} + p_{yw}q_{xyzw}e_{xz} + p_{zw}q_se_{zw}\\
&\quad + p_{zw}q_{xy}e_{xyzw} - p_{zw}q_{xz}e_{xw} + p_{zw}q_{xw}e_{xz} - p_{zw}q_{yz}e_{yw} + p_{zw}q_{yw}e_{yz} - p_{zw}q_{zw}\\
&\quad - p_{zw}q_{xyzw}e_{xy} + p_{xyzw}q_se_{xyzw} - p_{xyzw}q_{xy}e_{zw} + p_{xyzw}q_{xz}e_{yw} - p_{xyzw}q_{xw}e_{yz}\\
&\quad - p_{xyzw}q_{yz}e_{xw} + p_{xyzw}q_{yw}e_{xz} - p_{xyzw}q_{zw}e_{xy} + p_{xyzw}q_{xyzw}\\
&= (p_sq_s - p_{xy}q_{xy} - p_{xz}q_{xz} - p_{xw}q_{xw} - p_{yz}q_{yz} - p_{yw}q_{yw} - p_{zw}q_{zw} + p_{xyzw}q_{xyzw})\\
&\quad + (p_sq_{xy} + p_{xy}q_s - p_{xz}q_{yz} - p_{xw}q_{yw} + p_{yz}q_{xz} + p_{yw}q_{xw} - p_{zw}q_{xyzw} - p_{xyzw}q_{zw})e_{xy}\\
&\quad + (p_sq_{xz} + p_{xy}q_{yz} + p_{xz}q_s - p_{xw}q_{zw} - p_{yz}q_{xy} + p_{yw}q_{xyzw} + p_{zw}q_{xw} + p_{xyzw}q_{yw})e_{xz}\\
&\quad + (p_sq_{xw} + p_{xy}q_{yw} + p_{xz}q_{zw} + p_{xw}q_s - p_{yz}q_{xyzw} - p_{yw}q_{xy} - p_{zw}q_{xz} - p_{xyzw}q_{yz})e_{xw}\\
&\quad + (p_sq_{yz} - p_{xy}q_{xz} + p_{xz}q_{xy} - p_{xw}q_{xyzw} + p_{yz}q_s - p_{yw}q_{zw} + p_{zw}q_{yw} - p_{xyzw}q_{xw})e_{yz}\\
&\quad + (p_sq_{yw} - p_{xy}q_{xw} + p_{xz}q_{xyzw} + p_{xw}q_{xy} + p_{yz}q_{zw} + p_{yw}q_s - p_{zw}q_{yz} + p_{xyzw}q_{xz})e_{yw}\\
&\quad + (p_sq_{zw} - p_{xy}q_{xyzw} - p_{xz}q_{xw} + p_{xw}q_{xz} - p_{yz}q_{yw} + p_{yw}q_{yz} + p_{zw}q_s - p_{xyzw}q_{xy})e_{zw}\\
&\quad + (p_sq_{xyzw} + p_{xy}q_{zw} - p_{xz}q_{yw} + p_{xw}q_{yz} + p_{yz}q_{xw} - p_{yw}q_{xz} + p_{zw}q_{xy} + p_{xyzw}q_s)e_{xyzw}
\end{aligned}
$$

The geometric product of two rotors is symmetric, so $pq = qp$.

Appendix B

The Discrete Features of a Hyperbox

Table B.1: The Vertices of a Hyperbox

$\mathbf{v}_1 = c + A\begin{bmatrix}-s_x & -s_y & -s_z & -s_w\end{bmatrix}^T$	$\mathbf{v}_9 = c + A\begin{bmatrix}-s_x & s_y & s_z & -s_w\end{bmatrix}^T$
$\mathbf{v}_2 = c + A\begin{bmatrix}s_x & -s_y & -s_z & -s_w\end{bmatrix}^T$	$\mathbf{v}_{10} = c + A\begin{bmatrix}-s_x & s_y & -s_z & s_w\end{bmatrix}^T$
$\mathbf{v}_3 = c + A\begin{bmatrix}-s_x & s_y & -s_z & -s_w\end{bmatrix}^T$	$\mathbf{v}_{11} = c + A\begin{bmatrix}-s_x & -s_y & s_z & s_w\end{bmatrix}^T$
$\mathbf{v}_4 = c + A\begin{bmatrix}-s_x & -s_y & s_z & -s_w\end{bmatrix}^T$	$\mathbf{v}_{12} = c + A\begin{bmatrix}s_x & s_y & s_z & -s_w\end{bmatrix}^T$
$\mathbf{v}_5 = c + A\begin{bmatrix}-s_x & -s_y & -s_z & s_w\end{bmatrix}^T$	$\mathbf{v}_{13} = c + A\begin{bmatrix}s_x & s_y & -s_z & s_w\end{bmatrix}^T$
$\mathbf{v}_6 = c + A\begin{bmatrix}s_x & s_y & -s_z & -s_w\end{bmatrix}^T$	$\mathbf{v}_{14} = c + A\begin{bmatrix}s_x & -s_y & s_z & s_w\end{bmatrix}^T$
$\mathbf{v}_7 = c + A\begin{bmatrix}s_x & -s_y & s_z & -s_w\end{bmatrix}^T$	$\mathbf{v}_{15} = c + A\begin{bmatrix}-s_x & s_y & s_z & s_w\end{bmatrix}^T$
$\mathbf{v}_8 = c + A\begin{bmatrix}s_x & -s_y & -s_z & s_w\end{bmatrix}^T$	$\mathbf{v}_{16} = c + A\begin{bmatrix}s_x & s_y & s_z & s_w\end{bmatrix}^T$

The set of vertices $\mathcal{V}$ of a hyperbox can be calculated by using its center position c, rotation matrix A, and scale s (Table B.1). To represent a hyperbox's vertices, this project defines a class named VERTEX that which contains fields for a vertex's point and unique ID number.

Table B.2: The Edges of a Hyperbox

$\mathbf{e}_1 = \{\mathbf{v}_1, \mathbf{v}_2\}$	$\mathbf{e}_9 = \{\mathbf{v}_3, \mathbf{v}_9\}$	$\mathbf{e}_{17} = \{\mathbf{v}_6, \mathbf{v}_{12}\}$	$\mathbf{e}_{25} = \{\mathbf{v}_{10}, \mathbf{v}_{13}\}$
$\mathbf{e}_2 = \{\mathbf{v}_1, \mathbf{v}_3\}$	$\mathbf{e}_{10} = \{\mathbf{v}_3, \mathbf{v}_{10}\}$	$\mathbf{e}_{18} = \{\mathbf{v}_6, \mathbf{v}_{13}\}$	$\mathbf{e}_{26} = \{\mathbf{v}_{10}, \mathbf{v}_{15}\}$
$\mathbf{e}_3 = \{\mathbf{v}_1, \mathbf{v}_4\}$	$\mathbf{e}_{11} = \{\mathbf{v}_4, \mathbf{v}_7\}$	$\mathbf{e}_{19} = \{\mathbf{v}_7, \mathbf{v}_{12}\}$	$\mathbf{e}_{27} = \{\mathbf{v}_{11}, \mathbf{v}_{14}\}$
$\mathbf{e}_4 = \{\mathbf{v}_1, \mathbf{v}_5\}$	$\mathbf{e}_{12} = \{\mathbf{v}_4, \mathbf{v}_9\}$	$\mathbf{e}_{20} = \{\mathbf{v}_7, \mathbf{v}_{14}\}$	$\mathbf{e}_{28} = \{\mathbf{v}_{11}, \mathbf{v}_{15}\}$
$\mathbf{e}_5 = \{\mathbf{v}_2, \mathbf{v}_6\}$	$\mathbf{e}_{13} = \{\mathbf{v}_4, \mathbf{v}_{11}\}$	$\mathbf{e}_{21} = \{\mathbf{v}_8, \mathbf{v}_{13}\}$	$\mathbf{e}_{29} = \{\mathbf{v}_{12}, \mathbf{v}_{16}\}$
$\mathbf{e}_6 = \{\mathbf{v}_2, \mathbf{v}_7\}$	$\mathbf{e}_{14} = \{\mathbf{v}_5, \mathbf{v}_8\}$	$\mathbf{e}_{22} = \{\mathbf{v}_8, \mathbf{v}_{14}\}$	$\mathbf{e}_{30} = \{\mathbf{v}_{13}, \mathbf{v}_{16}\}$
$\mathbf{e}_7 = \{\mathbf{v}_2, \mathbf{v}_8\}$	$\mathbf{e}_{15} = \{\mathbf{v}_5, \mathbf{v}_{10}\}$	$\mathbf{e}_{23} = \{\mathbf{v}_9, \mathbf{v}_{12}\}$	$\mathbf{e}_{31} = \{\mathbf{v}_{14}, \mathbf{v}_{16}\}$
$\mathbf{e}_8 = \{\mathbf{v}_3, \mathbf{v}_6\}$	$\mathbf{e}_{16} = \{\mathbf{v}_5, \mathbf{v}_{11}\}$	$\mathbf{e}_{24} = \{\mathbf{v}_9, \mathbf{v}_{15}\}$	$\mathbf{e}_{32} = \{\mathbf{v}_{15}, \mathbf{v}_{16}\}$

The edges of a hyperbox can be represented as the unique pairs of vectors that differ by a value along the same axis (Table B.2). Instead of explicitly storing a list of edges for each hyperbox, this project instead represents the edges of a hyperbox implicitly by storing the vertices of a face as a clockwise-ordered set.

Table B.3: The Clockwise-Ordered Vertices of a Hyperbox's Faces

$\mathbf{f}_1 = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_6, \mathbf{v}_3)$	$\mathbf{f}_9 = (\mathbf{v}_2, \mathbf{v}_7, \mathbf{v}_{14}, \mathbf{v}_8)$	$\mathbf{f}_{17} = (\mathbf{v}_5, \mathbf{v}_8, \mathbf{v}_{14}, \mathbf{v}_{11})$
$\mathbf{f}_2 = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_7, \mathbf{v}_4)$	$\mathbf{f}_{10} = (\mathbf{v}_3, \mathbf{v}_6, \mathbf{v}_{12}, \mathbf{v}_9)$	$\mathbf{f}_{18} = (\mathbf{v}_5, \mathbf{v}_{10}, \mathbf{v}_{15}, \mathbf{v}_{11})$
$\mathbf{f}_3 = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_8, \mathbf{v}_5)$	$\mathbf{f}_{11} = (\mathbf{v}_3, \mathbf{v}_6, \mathbf{v}_{13}, \mathbf{v}_{10})$	$\mathbf{f}_{19} = (\mathbf{v}_6, \mathbf{v}_{12}, \mathbf{v}_{16}, \mathbf{v}_{13})$
$\mathbf{f}_4 = (\mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_9, \mathbf{v}_4)$	$\mathbf{f}_{12} = (\mathbf{v}_3, \mathbf{v}_9, \mathbf{v}_{15}, \mathbf{v}_{10})$	$\mathbf{f}_{20} = (\mathbf{v}_7, \mathbf{v}_{12}, \mathbf{v}_{16}, \mathbf{v}_{14})$
$\mathbf{f}_5 = (\mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_{10}, \mathbf{v}_5)$	$\mathbf{f}_{13} = (\mathbf{v}_4, \mathbf{v}_7, \mathbf{v}_{12}, \mathbf{v}_9)$	$\mathbf{f}_{21} = (\mathbf{v}_8, \mathbf{v}_{13}, \mathbf{v}_{16}, \mathbf{v}_{14})$
$\mathbf{f}_6 = (\mathbf{v}_1, \mathbf{v}_4, \mathbf{v}_{11}, \mathbf{v}_5)$	$\mathbf{f}_{14} = (\mathbf{v}_4, \mathbf{v}_7, \mathbf{v}_{14}, \mathbf{v}_{11})$	$\mathbf{f}_{22} = (\mathbf{v}_9, \mathbf{v}_{12}, \mathbf{v}_{16}, \mathbf{v}_{15})$
$\mathbf{f}_7 = (\mathbf{v}_2, \mathbf{v}_6, \mathbf{v}_{12}, \mathbf{v}_7)$	$\mathbf{f}_{15} = (\mathbf{v}_4, \mathbf{v}_9, \mathbf{v}_{15}, \mathbf{v}_{11})$	$\mathbf{f}_{23} = (\mathbf{v}_{10}, \mathbf{v}_{13}, \mathbf{v}_{16}, \mathbf{v}_{15})$
$\mathbf{f}_8 = (\mathbf{v}_2, \mathbf{v}_6, \mathbf{v}_{13}, \mathbf{v}_8)$	$\mathbf{f}_{16} = (\mathbf{v}_5, \mathbf{v}_8, \mathbf{v}_{13}, \mathbf{v}_{10})$	$\mathbf{f}_{24} = (\mathbf{v}_{11}, \mathbf{v}_{14}, \mathbf{v}_{16}, \mathbf{v}_{15})$

Each face of a hyperbox can be represented as a clockwise-ordered subset of vertices (Table B.3). This clockwise-ordering can be used to implicitly define each face's edges. For example, the ordering of the vertices of $\mathbf{f}_1$ is $\mathbf{v}_1 \rightarrow \mathbf{v}_2 \rightarrow \mathbf{v}_6 \rightarrow \mathbf{v}_3 \rightarrow \mathbf{v}_1$ (Table 3.2), which means that the edges of $\mathbf{f}_1$ are $\{\mathbf{v}_1, \mathbf{v}_2\}$, $\{\mathbf{v}_2, \mathbf{v}_6\}$, $\{\mathbf{v}_6, \mathbf{v}_3\}$, and $\{\mathbf{v}_3, \mathbf{v}_1\}$. Since none of the algorithms used in this project explicitly require references to a hyperbox's edges, this implicit representation of edges is sufficient for the purposes of this project. To represent a hyperbox's faces, this project defines a class named FACE that which contains fields for the face's clockwise-ordered list of vertices and unique ID number.

Table B.4: The 3-Cells of a Hyperbox

$\mathbf{c}_1 = \{\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_4, \mathbf{f}_7, \mathbf{f}_{10}, \mathbf{f}_{13}\}$	$\mathbf{c}_5 = \{\mathbf{f}_2, \mathbf{f}_3, \mathbf{f}_6, \mathbf{f}_9, \mathbf{f}_{14}, \mathbf{f}_{17}\}$
$\mathbf{c}_2 = \{\mathbf{f}_{16}, \mathbf{f}_{17}, \mathbf{f}_{18}, \mathbf{f}_{21}, \mathbf{f}_{23}, \mathbf{f}_{24}\}$	$\mathbf{c}_6 = \{\mathbf{f}_{10}, \mathbf{f}_{11}, \mathbf{f}_{12}, \mathbf{f}_{19}, \mathbf{f}_{22}, \mathbf{f}_{23}\}$
$\mathbf{c}_3 = \{\mathbf{f}_1, \mathbf{f}_3, \mathbf{f}_5, \mathbf{f}_8, \mathbf{f}_{11}, \mathbf{f}_{16}\}$	$\mathbf{c}_7 = \{\mathbf{f}_4, \mathbf{f}_5, \mathbf{f}_6, \mathbf{f}_{12}, \mathbf{f}_{15}, \mathbf{f}_{18}\}$
$\mathbf{c}_4 = \{\mathbf{f}_{13}, \mathbf{f}_{14}, \mathbf{f}_{15}, \mathbf{f}_{20}, \mathbf{f}_{22}, \mathbf{f}_{24}\}$	$\mathbf{c}_8 = \{\mathbf{f}_7, \mathbf{f}_8, \mathbf{f}_9, \mathbf{f}_{19}, \mathbf{f}_{20}, \mathbf{f}_{21}\}$

Each 3-cell of a hyperbox can be represented as a subset of the hyperbox's faces (Table B.4). Since solids have single normal vector in 4D space (Chapter 2), each 3-cell has a single normal vector. One way to calculate the normal of a 3-cell is to first calculate the outer product of three orthogonal edges and then take the dual of the resulting trivector. In particular, the normals of a hyperbox's 3-cells (as shown in Table B.4) can be expressed in terms of its rotation matrix A with column vectors A_1, A_2, A_3, and A_4:

Normal of $\mathbf{c}_1 = -A_4$	Normal of $\mathbf{c}_5 = -A_2$
Normal of $\mathbf{c}_2 = A_4$	Normal of $\mathbf{c}_6 = A_2$
Normal of $\mathbf{c}_3 = -A_3$	Normal of $\mathbf{c}_7 = -A_1$
Normal of $\mathbf{c}_4 = A_3$	Normal of $\mathbf{c}_8 = A_1$

In order to represent 3-cells in its code, this project defines a class named Cell, which contains fields for a 3-cell's normal vector, list of faces, unique ID number, and list of IDs for its adjacent 3-cells.

Appendix C

The Moments of Inertia of 4D Primitives

C.1 The Moment of Inertia of a Hypersphere

Due to the geometric relationship between spheres and hyperspheres, the moment of inertia of a hypersphere can be calculated similarly to the moment of inertia of a sphere. To begin deriving a hypersphere's moment of inertia, the hypersphere's differential mass (with respect to its radius) should first be calculated. Since a hypersphere's hypervolume is $V = \frac{1}{2}\pi^2 R^4$ and hyper-surface area is $2\pi^2 R^3$ [54], the density of a hypersphere (with uniformly distributed mass M and radius R) is:

$$\rho = \frac{M}{\frac{1}{2}\pi^2 R^4} = \frac{2M}{\pi^2 R^4}$$

Additionally, the hypersphere's differential hypervolume (with respect to the distance r from its center) can be calculated as:

$$dV = 2\pi^2 r^3 dr$$

Thus, the differential mass of a hypersphere can be calculated as:

$$dM = \rho * dV = \frac{2m}{\pi^2 R^4} * 2\pi^2 r^3 dr = \frac{4Mr^3 dr}{R^4}$$

Since hyperspheres are symmetric, the inertia of a hypersphere (with a uniform density) about its center of mass is the same for all planes of rotation, such that $\mathcal{I}_{cm,xy} = \mathcal{I}_{cm,xz} = \mathcal{I}_{cm,xw} = \mathcal{I}_{cm,yz} = \mathcal{I}_{cm,yw} = \mathcal{I}_{cm,zw}$. Thus, by integrating over the hypervolume of a hypersphere, the moment of inertia of a hypersphere, with respect to its center of mass, can be calculated as:

$$\begin{aligned}
6\mathcal{I}_{cm} &= \mathcal{I}_{cm,xy} + \mathcal{I}_{cm,xz} + \mathcal{I}_{cm,xw} + \mathcal{I}_{cm,yz} + \mathcal{I}_{cm,yw} + \mathcal{I}_{cm,zw} \\
&= \int (x^2 + y^2) dM + \int (x^2 + z^2) dM + \int (x^2 + w^2) dM \\
&\quad + \int (y^2 + z^2) dM + \int (y^2 + w^2) dM + \int (z^2 + w^2) dM \\
&= \int (3x^2 + 3y^2 + 3z^2 + 3w^2) dM \\
&= 3 \int (x^2 + y^2 + z^2 + w^2) dM
\end{aligned}$$

Since the equation of a hypersphere (with a radius R) is $R^2 = x^2 + y^2 + z^2 + w^2$, the integral above becomes:

$$6\mathcal{I}_{cm} = 3 \int r^2 * dM = 3 \int r^2 * \frac{4Mr^3 dr}{R^4} = 3 \int \frac{4m * r^5 dr}{R^4}$$

Integrating this from $r = 0$ to $r = R$ produces:

$$6\mathcal{I}_{cm} = 3 \int_0^R \frac{4M * r^5 dr}{R^4} = \frac{12M}{R^4} \int_0^R r^5 dr = \frac{12M}{R^4} * \left| \frac{1}{6} r^6 \right|_0^R = \frac{12M}{R^4} * \frac{R^6}{6}$$

This can be further simplified to get:

$$\mathcal{I}_{cm} = \frac{1}{3} M R^2$$

Thus, the moments of inertia of a hypersphere (with mass M and radius R) about each plane are:

$$\begin{aligned}
\mathcal{I}_{xy} &= \frac{1}{3} M R^2 & \mathcal{I}_{xz} &= \frac{1}{3} M R^2 & \mathcal{I}_{xw} &= \frac{1}{3} M R^2 \\
\mathcal{I}_{yz} &= \frac{1}{3} M R^2 & \mathcal{I}_{yw} &= \frac{1}{3} M R^2 & \mathcal{I}_{zw} &= \frac{1}{3} M R^2
\end{aligned}$$

C.2 The Moment of Inertia of a Hypercapsule

In 3D, the moment of inertia of a capsule can be calculated by combining the inertia of its cylinder and hemisphere components [34]. Likewise, the moment of inertia of a hypercapsule can be calculated by combining the inertia of its spherinder component ($\mathcal{I}_s$) and its half-hypersphere caps ($\mathcal{I}_c$) (both of which are derived later in this subsection). This results in the following inertia for a hypercapsule:

$$\begin{aligned}
\mathcal{I}_{xy} &= \mathcal{I}_{s,xy} + 2(\mathcal{I}_{c,xy}) = \frac{1}{3}M_s\left(\frac{2}{5}R^2 + \frac{H^2}{4}\right) + 2M_c * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right)\\
\mathcal{I}_{xz} &= \mathcal{I}_{s,xz} + 2(\mathcal{I}_{c,xz}) = \frac{2}{5}M_sR^2 + \frac{2}{3}M_cR^2\\
\mathcal{I}_{xw} &= \mathcal{I}_{s,xw} + 2(\mathcal{I}_{c,xw}) = \frac{2}{5}M_sR^2 + \frac{2}{3}M_cR^2\\
\mathcal{I}_{yz} &= \mathcal{I}_{s,yz} + 2(\mathcal{I}_{c,yz}) = \frac{1}{3}M_s\left(\frac{2}{5}R^2 + \frac{H^2}{4}\right) + 2M_c * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right)\\
\mathcal{I}_{yw} &= \mathcal{I}_{s,yw} + 2(\mathcal{I}_{c,yw}) = \frac{1}{3}M_s\left(\frac{2}{5}R^2 + \frac{H^2}{4}\right) + 2M_c * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right)\\
\mathcal{I}_{zw} &= \mathcal{I}_{s,zw} + 2(\mathcal{I}_{c,zw}) + \frac{2}{5}M_sR^2 + \frac{2}{3}M_cR^2
\end{aligned}$$

As shown above, M_s corresponds to the mass of the spherinder and M_c corresponds to the mass of a hemispheric cap. Since the mass M of the hypercapsule is uniformly distributed throughout, the values of M_s and M_c can be calculated accordingly.

Since the hypervolume of a hypercapsule is $\frac{4}{3}\pi R^2H + \frac{1}{2}\pi^2R^4$, the uniform density ρ of a hypercapsule can be calculated as:

$$\rho = \frac{M}{V} = \frac{M}{\frac{4}{3}\pi R^2H + \frac{1}{2}\pi^2R^4}$$

The value of ρ can now be plugged into the equations for a spherinder's mass and a cap's mass to calculate M_s and M_c respectively:

$$M_s = \rho * V_s = \frac{M}{\pi R^2(\frac{4}{3}H + \frac{1}{2}\pi R^2)} * \frac{4}{3}\pi R^2 H = \frac{M}{1 + \frac{3\pi R^2}{8H}} \tag{C.1}$$

$$M_c = \rho * V_c = \frac{M}{\pi R^2(\frac{4}{3}H + \frac{1}{2}\pi R^2)} * \frac{1}{2}\pi^2 R^4 = \frac{M}{\frac{8H}{3\pi R^2} + 1} \tag{C.2}$$

C.2.1 The Inertia of the Spherinder Component

In 3D space, a vertical cylinder can be generated by sweeping a disk (that is parallel to the xz-plane) along the y-axis. Notably, the moment of inertia of a cylinder with respect to the xz-plane is equal to the inertia of a disk (i.e., $\frac{MR^2}{2}$). Additionally, the moments of inertia of a cylinder with respect to the xy and yz-planes are equal ($\mathcal{I}_{xy} = \mathcal{I}_{yz}$). Like a vertical cylinder in 3D space, a vertical spherinder in 4D space can be generated (with radius $R^2 = x^2 + z^2 + w^2$ and height H) by sweeping a sphere along the y-axis.

Calculating a Spherinder's Inertia about Perpendicular Planes

To begin calculating a the moment of inertia of the spherinder component of a hypercapsule, the spherinder's differential mass with respect to its radius should be calculated.

The hypervolume of a spherinder can be calculated by multiplying the volume of its spherical base by its height: $V = \frac{4}{3}\pi R^3 H$ [6]. Additionally, the hyper-surface area of a hypercapsule's spherinder component can be calculated by multiplying the surface area of its spherical base by its height: is $4\pi R^2 H$ [6]. Thus, the density of a hypercapsule's spherinder component (with uniformly distributed mass M, radius R, and height H) can be calculated as:

$$\rho = \frac{M}{\frac{4}{3}\pi R^3 H} = \frac{3M}{4\pi R^3 H}$$

Additionally, the spherinder's differential hypervolume (with respect to its radius) can be calculated as:

$$dV_r = 4\pi r^2 H dr$$

Thus, the differential mass of a spherinder (with respect to its radius) is:

$$dM_r = \rho * dV_r = \frac{3M}{4\pi R^3 H} * 4\pi r^2 H dr = \frac{3Mr^2 dr}{R^3}$$

Due to spherinders' symmetry, the moments of inertia of a vertical spherinder with respect to planes that are perpendicular to the y-axis are equal to each other ($\mathcal{I}_{xz} = \mathcal{I}_{xw} = \mathcal{I}_{zw}$). Thus, the moments of inertia of a vertical spherinder (with respect to its center of mass) about a perpendicular plane is:

$$\begin{aligned}3\mathcal{I}_{cm,y\perp} &= \mathcal{I}_{cm,xz} + \mathcal{I}_{cm,xw} + \mathcal{I}_{cm,zw}\\ &= \int (x^2+z^2)dm_r + \int (x^2+w^2)dm_r + \int (z^2+w^2)dm_r\\ &= \int (2x^2+2z^2+2w^2)dm_r\\ &= 2\int (x^2+z^2+w^2)dm_r\end{aligned}$$

Since the radius of the spherinder is $R^2 = x^2 + z^2 + w^2$, the integral above becomes:

$$3\mathcal{I}_{cm,y\perp} = 2\int r^2 * \rho * dV_r = 2\int r^2 * \frac{3Mr^2dr}{R^3} = \frac{6M}{R^3}\int r^4 dr$$

Integrating this from $r = 0$ to $r = R$ (where R is the radius of the hypersphere) produces:

$$3\mathcal{I}_{cm,y\perp} = \frac{6M}{R^3} * \left|\frac{1}{5}r^5\right|_0^R = \frac{6M}{R^3} * \frac{1}{5}R^5$$

This can be further simplified to be:

$$\mathcal{I}_{cm,y\perp} = \frac{2}{5}MR^2$$

Notably, this is identical to the moment of inertia of a sphere.

Calculating a Spherinder's Inertia about Parallel Planes

Next, the moment of inertia of a spherinder about parallel planes should be calculated [7]. To start, the spherinder's differential mass with respect to its height should be calculated. As mentioned previously, the density of a spherinder (with uniformly distributed mass M, radius R, and height H) is $\rho = 3M/4\pi R^3H$. A spherinder's differential hypervolume (with respect to its height) can be calculated by adding infinitesimally short spherinders with a height of dh:

$$dV_h = \frac{4}{3}\pi R^3 dh$$

Putting this together, the differential mass of a hypersphere (with respect to its height) can be calculated as:

$$dM_h = \rho * dV_h = \frac{3M}{4\pi R^3 H} * \frac{4}{3}\pi R^3 dh = \frac{Mdh}{H}$$

With this value, the spherinder's moment of inertia with respect to its parallel planes can be calculated. Given that $\mathcal{I}_{cm,y\perp} = (2/5)MR^2$, the Perpendicular Axis Theorem [4] can be generalized to 4D to show that:

$$d\mathcal{I}_{cm,y\perp} = d\mathcal{I}_{xy} + d\mathcal{I}_{yz} + d\mathcal{I}_{yw}$$

Due to the spherinder's symmetry, its moments of inertia about planes that are parallel to the y-axis are equal to each other ($\mathcal{I}_{xy} = \mathcal{I}_{yz} = \mathcal{I}_{yw}$). Using this insight and generalizing the Parallel Axis Theorem and Perpendicular Axis Theorem, the value of $d\mathcal{I}_{cm,y\perp}$ can be expressed as:

$$\begin{aligned} d\mathcal{I}_{cm,y\perp} &= 3(d\mathcal{I}_{cm,y\parallel} - dMh^2) \\ \Rightarrow d\mathcal{I}_{cm,y\parallel} &= \frac{1}{3} * d\mathcal{I}_{cm,y\perp} + dM_h h^2 \end{aligned}$$

Plugging in the value $d\mathcal{I}_{cm,y\perp}$ that was calculated previously results in:

$$\begin{aligned} d\mathcal{I}_{cm,y\parallel} &= \frac{1}{3} * \frac{2}{5} dM_h R^2 + dM_h h^2 \\ &= \frac{2}{15} dM_h R^2 + dM_h h^2 \end{aligned}$$

Integrating with respect to the spherinder's height H results in:

$$\begin{aligned} \mathcal{I}_{cm,y\parallel} = \int d\mathcal{I}_{cm,y\parallel} &= \int_{-\frac{H}{2}}^{\frac{H}{2}} \frac{2}{15} * \frac{Mdh}{H} R^2 + \frac{Mdh}{H} h^2 \\ &= \frac{2}{15} * \frac{MR^2}{H} \left| dh \right|_{-\frac{H}{2}}^{\frac{H}{2}} + \frac{M}{H} \left| h^2 dh \right|_{-\frac{H}{2}}^{\frac{H}{2}} \\ &= \frac{2MR^2}{15H} * H + \frac{M}{H} * \frac{1}{12} * H^3 \end{aligned}$$

This can be further simplified to produce:

$$\mathcal{I}_{cm,y\parallel} = \frac{1}{3}M\left(\frac{2}{5}R^2 + \frac{1}{4}H^2\right)$$

Combining the Inertia About Parallel and Perpendicular Planes

The moments of inertia about the parallel and perpendicular planes can be combined to produce the spherinder's moments of inertia. As a result, the moments of inertia of a spherinder (with mass M, radius R, and height H) about each plane are:

$$\mathcal{I}_{xy} = \mathcal{I}_{yz} = \mathcal{I}_{yw} = \mathcal{I}_{cm,y\parallel} = \frac{1}{3}M\left(\frac{2}{5}R^2 + \frac{1}{4}H^2\right)$$
$$\mathcal{I}_{xz} = \mathcal{I}_{xw} = \mathcal{I}_{zw} = \mathcal{I}_{cm,y\perp} = \frac{2}{5}MR^2$$

C.2.2 The Inertia of a Half-Hypersphere Cap

In 3D, the moment of inertia of a half-sphere about any plane with respect to its base is equal to the moment of inertia of a sphere [34]. By extension, the moment of inertia of a half-hypersphere, in 4D, about any plane with respect to base is equal to the moment of inertia of a hypersphere:

$$\mathcal{I}_{base,xy} = \mathcal{I}_{base,xz} = \mathcal{I}_{base,xw} = \mathcal{I}_{base,yz} = \mathcal{I}_{base,yw} = \mathcal{I}_{base,zw} = \frac{1}{3}MR^2$$

Notably, the moments of inertia (with respect to the center of mass) about planes that are parallel to the half-hypersphere's base is equal to the moments of inertia of those planes with respect to the base:

$$\mathcal{I}_{cm',xz} = \mathcal{I}_{base,xz} \qquad \mathcal{I}_{cm',xw} = \mathcal{I}_{base,xw} \qquad \mathcal{I}_{cm',zw} = \mathcal{I}_{base,zw}$$

However, the moments of inertia about planes that are perpendicular to the base do not have the same moments of inertia with respect to the half-hypersphere's center of mass. As a result, Steiner's rule must be used to find the value of the moments of inertia with respect to the half-hypersphere's center of mass [34]:

$$\mathcal{I}_{cm',xz} = \mathcal{I}_{base,xz} - MD^2 \qquad \mathcal{I}_{cm',xw} = \mathcal{I}_{base,xw} - MD^2 \qquad \mathcal{I}_{cm',zw} = \mathcal{I}_{base,zw} - MD^2$$

In the equations above, the variable D corresponds to the distance from the base of the cap to the cap's center of mass.

Just as a hemisphere can be created by adding infinitesimally short cylinders, a half-hypersphere can be created by adding infinitesimally short spherinders. Notably, the radius of a hyperspherical cap is $R^2 = x^2 + y^2 + z^2 + w^2$ and that the radius of an infinitesimally short spherinder is $r^2 = x^2 + z^2 + w^2$. Thus, the equation for the radius of a spherinder can be substituted into the equation of the hyperspherical cap, such that $r^2 + y^2 = R^2$. This equation can be rearranged to produce $r = \sqrt{R^2 - y^2}$. Plugging in the value of r into this equation yields: $dV = \frac{4}{3}\pi\left(\sqrt{R^2 - y^2}\right)^3 dy$. Additionally, the hypervolume of a half-hypersphere can be calculated by halving the hypervolume of a hypersphere: $V = \frac{1}{2} * \frac{1}{2}\pi^2 R^4 = \frac{1}{3}\pi^2 R^4$. Putting this together,the half-hypersphere's differential mass with respect to its height can be calculated as follows:

$$dM_y = \rho * dV_y = \frac{M}{\frac{1}{4}\pi^2 R^4} * \frac{4}{3}\pi(\sqrt{R^2 - y^2})^3 dy = \frac{16M}{3\pi R^4}(\sqrt{R^2 - y^2})^3 dy$$

In order to find the distance D from the center of mass to the base sphere, the following integral can be taken:

$$\begin{aligned} D &= \frac{1}{M}\int (y) dM_y = \frac{1}{M}\int y * \frac{16}{3\pi R^4}(\sqrt{R^2 - y^2})^3 dy \\ &= \frac{16}{3\pi R^4}\int y * (\sqrt{R^2 - y^2})^3 dy = \frac{16R}{15\pi} \end{aligned}$$

With the moments of inertia of the caps with respect to their centers of mass, the height of the hypercapsule can now be incorporated by using Steiner's rule again [34].

$$\begin{aligned} \mathcal{I}_{cm,y\|} &= \mathcal{I}_{cm',y\|} + M\left(\frac{H}{2} + D\right)^2 = (\mathcal{I}_{base,y\|} - MD^2) + M\left(\frac{H}{2} + D\right)^2 \\ &= \mathcal{I}_{base,y\|} - MD^2 + M\frac{H^2}{4} + MDH + MD^2 = \mathcal{I}_{base,y\|} + M\frac{H^2}{4} + MDH \\ &= \frac{1}{3}MR^2 + M\frac{H^2}{4} + MDH \end{aligned}$$

Plugging in the value of D results in:

$$\begin{aligned}\mathcal{I}_{cm,y\|} &= \frac{1}{3}MR^2 + M\frac{H^2}{4} + MDH \\ &= \frac{1}{3}MR^2 + M\frac{H^2}{4} + \frac{16R}{15\pi} * MH \\ &= M * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right)\end{aligned}$$

As a result, the moments of inertia of one cap of a hypercapsule (with mass M, radius R, and height H) are:

$$\begin{aligned}\mathcal{I}_{xy} &= M * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right) & \mathcal{I}_{xz} &= \frac{1}{3}MR^2 \\ \mathcal{I}_{xw} &= \frac{1}{3}MR^2 & \mathcal{I}_{yz} &= M * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right) \\ \mathcal{I}_{yw} &= M * \left(\frac{1}{3}R^2 + \frac{H^2}{4} + \frac{16RH}{15\pi}\right) & \mathcal{I}_{zw} &= \frac{1}{3}MR^2\end{aligned}$$

C.3 The Moment of Inertia of a Hyperbox

The method for calculating the inertia of a hyperbox is similar to calculating the inertia of a 3D box. To begin deriving a hyperbox's moment of inertia, the hyperbox's differential mass (with respect to its sides) should first be calculated. Since the hypervolume of a hyperbox with scale vector s is $s_x * s_y * s_z * s_w$, the density of a hypersphere (with uniformly distributed mass M) can be calculated as:

$$\rho = \frac{M}{s_x * s_y * s_z * s_w}$$

Additionally, the differential hypervolume of a hypercube with respect to its sides is simply $dV = dx * dy * dz * dw$. As a result, the differential mass of a hypercube is:

$$dM = \rho * dV = \frac{M}{s_x * s_y * s_z * s_w} * dx * dy * dz * dw$$

With these values, the moment of inertia of a hyperbox about the xy-plane (with respect to its center of mass) can be calculated as:

$$\begin{aligned}
\mathcal{I}_{xy} &= \int (x^2+y^2)dM = \iiiint (x^2+y^2)\frac{Mdxdydzdw}{s_x s_y s_z s_w} \\
&= \frac{M}{s_x s_y s_z s_w} \iiiint (x^2+y^2)dxdydzdw \\
&= \frac{M}{s_x s_y s_z s_w} (\int x^2 dx \int dy \int dz \int dw + \int dx \int y^2 dy \int dz \int dw) \\
&= \frac{M}{s_x s_y s_z s_w} (\int_{\frac{-Sx}{2}}^{\frac{Sx}{2}} x^2 dx \int_{\frac{-Sy}{2}}^{\frac{Sy}{2}} dy \int_{\frac{-Sz}{2}}^{\frac{Sz}{2}} dz \int_{\frac{-Sw}{2}}^{\frac{Sw}{2}} dw \\
&\quad + \int_{\frac{-Sx}{2}}^{\frac{Sx}{2}} dx \int_{\frac{-Sy}{2}}^{\frac{Sy}{2}} y^2 dy \int_{\frac{-Sz}{2}}^{\frac{Sz}{2}} dz \int_{\frac{-Sw}{2}}^{\frac{Sw}{2}} dw) \\
&= \frac{M}{s_x s_y s_z s_w} \left(\left| \frac{1}{3} x^3 \right|_{-Sx}^{Sx} s_y s_z s_w + \left| \frac{1}{3} y^3 \right|_{-Sy}^{Sy} s_x s_z s_w \right) \\
&= \frac{M}{s_x s_y s_z s_w} \left(\frac{1}{12} (s_x)^3 s_y s_z s_w + \frac{1}{12} (s_y)^3 s_x s_z s_w \right) \\
&= M \left(\frac{1}{12} s_x^2 + \frac{1}{12} s_y^2 \right) = \frac{1}{12} M (s_x^2 + s_y^2)
\end{aligned}$$

The moments of inertia for the xz, xw, yz, yw, and zw-planes can be derived similarly, resulting in the following moments of inertia for a hyperbox with uniformly distributed mass M and scale vector s:

$$\begin{aligned}
\mathcal{I}_{xy} &= \frac{1}{3} M (s_x^2 + s_y^2) & \mathcal{I}_{xz} &= \frac{1}{3} M (s_x^2 + s_z^2) & \mathcal{I}_{xw} &= \frac{1}{3} M (s_x^2 + s_w^2) \\
\mathcal{I}_{yz} &= \frac{1}{3} M (s_y^2 + s_z^2) & \mathcal{I}_{yw} &= \frac{1}{3} M (s_y^2 + s_w^2) & \mathcal{I}_{zw} &= \frac{1}{3} M (s_z^2 + s_w^2)
\end{aligned}$$

Appendix D

The Sutherland-Hodgman Clipping Algorithm

One approach to generating the contact manifold between two colliding 3D boxes is to use the Sutherland-Hodgman clipping algorithm, which clips the vertices of one box's face with the vertices of another box's face [26]. To generate the contact manifold between two colliding hyperboxes, this project implements a 4D version of the Sutherland-Hodgman Algorithm, which clips the vertices of one hyperbox's 3-cell with another hyperbox's 3-cell. This section of the appendix is dedicated to providing a broad overview of this project's 4D implementation of the Sutherland-Hodgman algorithm.

Once a collision between two hyperboxes has been detected (and the collision normal has been found), the Sutherland-Hodgman algorithm can be used to generate the contact manifold. To begin, the algorithm looks for the reference 3-cell, which is the 3-cell whose normal is most parallel to the collision normal compared to all of the other 3-cells of the two hyperboxes. To find the reference 3-cell, the 3-cells with the most parallel normals in the first hyperbox and second hyperbox should be found. Then, these two 3-cells should be compared to select the 3-cell with the most parallel normal to the collision normal. The selected 3-cell can then treated as the reference 3-cell.

The following pseudocode shows how to find a hyperbox's 3-cell with the most parallel normal to the collision normal:

Algorithm 16 Getting the 3-Cell with the Most Parallel Normal

```
1: Input
2:     $n$      the collision normal
3:     $C$      the list of the 3-cells of a hyperbox
4: Output
5:     $c^+$    the 3-cell with the most parallel normal in the given list
6: procedure GetMostParallel
7:     $n := v \;/\; \|v\|$
8:     $c^+ := C[0]$
9:     $s_{max} := -\infty$
10:    for each $c$ in $C$ do
11:        $s := c.n \cdot n$        ▷ Dot product of collision normal and the 3-cell's normal
12:        if $s > s_{max}$ then
13:            $s_{max} := s$
14:            $c^+ := c$
15:        end if
16:    end for
17:    return $c^+$
18: end procedure
```

The next step of the Sutherland-Hodgman algorithm is to find the incident 3-cell, which is the 3-cell with the most anti-parallel normal vector to the collision normal. In particular, the incident 3-cell should be part of the hyperbox that does not have the reference 3-cell. In other words, if the first hyperbox has the reference 3-cell, then the algorithm should search the list of 3-cells in the second hyperbox to find the incident 3-cell, and vice-versa.

The following pseudocode shows how to find a hyperbox's 3-cell with the most anti-parallel normal to the collision normal:

Algorithm 17 Getting the 3-Cell with the Most Anti-Parallel Normal

```
1:  Input
2:      $n$      the collision normal
3:      $C$      the list of the 3-cells of a hyperbox
4:  Output
5:      $c^-$    the most 3-cell with the most anti-parallel normal in the given list
6:  procedure GetMostAntiParallel
7:      $c^- := C[0]$
8:      $s_{min} := \infty$
9:      for each $c$ in $C$ do
10:         $s := c.n \cdot n$        ▷ Dot product of collision normal and the 3-cell's normal
11:         if $s < s_{min}$ then
12:             $s_{min} := s$
13:             $c^- := c$
14:         end if
15:     end for
16:     return $c^-$
17: end procedure
```

With both the reference and incident 3-cells, the algorithm should then use the adjacent 3-cells of the reference 3-cell to clip the faces of the incident 3-cell. This can be done by treating the adjacent 3-cells as hyperplanes and calculating the distance of each vertex in the incident 3-cell to the hyperplanes. If two vertices of an edge have a positive distance from a hyperplane, then they both lie outside of the hyperplane, and the edge should be discarded; if both vertices have a negative distance from the hyperplane, then they both lie within the hyperplane, so the edge should be kept; if one vertex has a positive distance while the other has a negative distance from the hyperplane, then the edge is intersecting the hyperplane, which means that the vertex with a positive distance from the hyperplane should be replaced with the intersection point between the hyperplane and the edge.

The following pseudocode demonstrates how to calculate the point of intersection between an edge and a hyperplane [35]:

Algorithm 18 Point of Intersection Between an Edge and a Hyperplane

```
1: Input
2:     $n$      the normal of the plane
3:     $x$      a point on the plane
4:     $v_1$    the first vertex of the edge
5:     $v_2$    the second vertex of the edge
6: Output
7:     $p$      the point of intersection
8: procedure IntersectingPoint
9:     $u := v_2 - v_1$
10:    $d := n \cdot x$
11:    $t := (d - n \cdot v_1)/(n \cdot u)$
12:    $p := v_1 + t * n$
13:    return $p$
14: end procedure
```

Since this project represents the edges of a face implicitly (by ordering the face's vertices in a clockwise fashion), the algorithm for clipping the faces of a 3-cell must preserve the clockwise-ordering of each face's vertices, even when replacing vertices with intersection points.

The following algorithm can be used to clip the face of a 3-cell while preserving the clockwise-ordering of its vertices:

Algorithm 19 Clipping a Face with a Hyperplane

```
1: Input
2:     $c$      the clipping 3-cell
3:     $f$      the face of interest
4: Output
5:     $f_c$    the clipped face
6: procedure ClipFaces
7:     $n := c.n$                                  ▷ The normal of the clipping 3-cell
8:     $x := c.v$                                  ▷ A vertex of the clipping 3-cell
9:     V := f.V                                    ▷ Gets the list of vertices in the face
10:    initialize $U$ as an empty list of vertices
11:    i := 0                                      ▷ Initializes the current index
12:    repeat
13:        $j := (i+1)$ % Length($V$)  ▷ Gets the index of the next vertex of the face
14:        $v_i := V[i]$
15:        $v_j := V[j]$
16:        $d_i :=$ DistanceFromPlane($n, x, v_i$)
17:        $d_j :=$ DistanceFromPlane($n, x, v_j$)
18:        if $d_i < 0$ and $d_j < 0$ then         ▷ Both vertices are inside the hyperplane
19:            append $v_i$ to $U$
20:        else if $d_i < 0$ then           ▷ Only the first vertex is inside the hyperplane
21:            append $v_i$ to $U$
22:            append IntersectingPoint($n, x, v_i, v_j$) to $U$
23:        else if $d_j < 0$ then         ▷ Only the second vertex is inside the hyperplane
24:            append IntersectingPoint($n, x, v_i, v_j$) to $U$
25:        end if
26:    until $i == $ Length($V$)
27:    initialize a new face $f_c$
28:    $f_c.ID := f.ID$
29:    $f_c.V := U$
30:    return $f_c$
31: end procedure
```

After clipping the incident 3-cell's faces with the adjacent 3-cells of the reference 3-cell, the final step of the clipping algorithm is to discard all of the remaining vertices that have a positive distance from the reference-cell. This can be done quite simply by treating the reference 3-cell as a hyperplane, calculating the distance of each vertex to the hyperplane, and keeping track of a list of the vertices with a negative distance from the hyperplane:

Algorithm 20 Remove Vertices Outside of the Reference 3-Cell

```
1: Input
2:     $c_R$     the reference 3-cell
3:     $F$       the list of the faces in the clipped incident 3-cell
4: Output
5:     $CP$      a list of contact point objects
6: procedure RemoveOutsideVertices
7:     $n := c_R.n$                    ▷ The normal of the reference 3-cell
8:     $x := c_R.v$                    ▷ A vertex of the reference 3-cell
9:     initialize $CP$ as an empty list of contact points
10:    for each $f$ in $F$ do
11:        for each $v$ in $f.V$ do
12:            $s :=$ DistanceFromHyperplane$(n, x, v)$
13:            if $s < 0$ then
14:                initialize a new contact point $cp$
15:                set the $n$ field of $cp$ to be $n$
16:                set the $p$ field of $cp$ to be $p$
17:                set the $s$ field of $cp$ to be $s$
18:                append $cp$ to $CP$
19:            end if
20:        end for
21:    end for
22:    return $CP$
23: end procedure
```

Each of these steps can be combined into the following algorithm for generating the contact manifold for two colliding hyperboxes:

Algorithm 21 Generating the Contact Manifold Between Hyperboxes HB_1 and HB_2

```
1: Input
2:     $n$      the collision normal
3:     $C_1$    the list of 3-cells in $HB_1$
4:     $C_2$    the list of 3-cells in $HB_2$
5: Output
6:     $CP$     the list of contact points
7: procedure ClipHyperboxes
8:     $c_1 := $ GetMostParallel$(n, C_1)$
9:     $c_2 := $ GetMostParallel$(n, C_2)$
10:    if $(c_1.n \cdot n > c_2.n \cdot n)$ then
11:        $c_R := c_1$                                   ▷ Sets the reference 3-cell
12:        $c_I := $ GetMostAntiParallel$(n, C_2)$        ▷ Gets the incident 3-cell
13:    else
14:        $c_R := c_2$                                   ▷ Sets the reference 3-cell
15:        $c_I := $ GetMostAntiParallel$(n, C_1)$        ▷ Gets the incident 3-cell
16:    end if
17:    set $C_{aR}$ to be a list containing the adjacent 3-cells of $c_R$
18:    set $G$ to be a copy of the list of faces in $c_I$
19:    for each $c$ in $C_{aR}$ do
20:        for each $g$ in $G$ do
21:            replace $g$ in $G$ with ClipFace$(c, f)$
22:        end for
23:    end for
24:    $CP := $ RemoveOutsideVertices$(c_R, G)$
25:    return $CP$
26: end procedure
```

Appendix E

Signed Distance Functions for 4D Primitives

To implicitly define 4D primitives with signed distance functions, the signed distance functions of their 3D counterparts (as described by Quilez [37]) can be modified.

Signed Distance Function for a Hypersphere

Given a vector p (that points from the ray's origin to the hypersphere's center) and the hypersphere's radius s, the following signed distance function can be used to implicitly define the hypersphere:

```
float sdHypersphere(float4 p, float s)
{
    return length(p) - s;
}
```

Signed Distance Function for a Vertical Hypercapsule

Given a vector p (that points from the ray's origin to the hypercapsule's center point), the hypercapsule's height h, and the hypercapsule's radius r, the following signed distance function can be used to implicitly define the hypercapsule:

```
float sdVerticalHypercapsule(float4 p, float h, float r)
{
    p.y -= clamp(p.y, 0.0, h);
    return length(p) - r;
}
```

Signed Distance Function for a Hyperbox

Given a vector p (that points from the ray's origin to the hyperbox's center) and the hyperbox's scale vector b, the following signed distance function can be used to implicitly define the hyperbox:

```
float sdHyperbox(float4 p, float4 b)
{
    float4 q = abs(p) - b;
    return length(max(q, 0.0))
        + min(max(q.x, max(q.y, max(q.z, q.w))), 0.0);
}
```

Signed Distance Function for a Hyperplane

Given the ray's origin p, the hyperplane's normal n, and the hyperplane's offset o, the following signed distance function can be used to implicitly define the hyperplane:

```
float sdHyperplane(float4 p, float4 n, float offset)
{
    return dot(p, n) - offset;
}
```

Signed Distance Function for the Union Operator

In order to combined the outputs of the signed distance functions into a single signed distance field, the union operator can be performed. Given distances $d1$ and $d2$ from the ray's origin to two different objects, the union operator can be defined as:

```
float4 opUnion(float d1, float d2)
{
    return min(d1, d2);
}
```

Appendix F

Link to Project Code

The code associated with this project can be found at https://github.com/EpiTorres/into-another-dimension.

Bibliography

[1] "Capsule." [Online]. Available: https://en.wikipedia.org/w/index.php?title=Capsule_(geometry)&oldid=1042983461

[2] Cuboid. [Online]. Available: https://www.math.net/cuboid

[3] Elastic collisions and contacts warm starting. [Online]. Available: https://gamedev.net/forums/topic/699326-elastic-collisions-and-contacts-warm-starting/5392665/

[4] Perpendicular axis theorem. [Online]. Available: https://byjus.com/jee/perpendicular-axis-theorem/

[5] Ray-tracing: Generating camera rays (generating camera rays). [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays

[6] "Spherinder." [Online]. Available: https://en.wikipedia.org/w/index.php?title=Spherinder&oldid=1078430624

[7] A08. Answer to "how to find the moment of inertia of a solid cylinder about transverse (perpendicular) axis passing through its center?". [Online]. Available: https://socratic.org/questions/how-to-find-the-moment-of-inertia-of-a-solid-cylinder-about-transverse-perpendic

[8] Acegikmo. Answer to "how many and which axes to use for 3d OBB collision with SAT". [Online]. Available: https://gamedev.stackexchange.com/a/92055

[9] M. t. Bosch, "Designing a 4d world: The technology behind miegakure [hide&reveal]." [Online]. Available: https://www.youtube.com/watch?v=vZp0ETdD37E

[10] ——. Let's remove quaternions from every 3d engine (an interactive introduction to rotors from geometric algebra). [Online]. Available: https://marctenbosch.com/quaternions/

[11] ——, "N-dimensional rigid body dynamics," vol. 39, no. 4, pp. 55–1, publisher: ACM New York, NY, USA.

[12] Britannica. Density. [Online]. Available: https://www.britannica.com/science/density

[13] ——. Gravity. [Online]. Available: https://www.britannica.com/science/gravity-physics/Acceleration-around-Earth-the-Moon-and-other-planets

[14] E. Catto, "Box2d-lite," original-date: 2019-01-03T00:20:12Z. [Online]. Available: https://github.com/erincatto/box2d-lite

[15] ——, "Fast and simple physics using sequential impulses," in *Gam es Developer Conference*, p. 9.

[16] ——, "Iterative dynamics with temporal coherence," in *Game developer conference*, vol. 2, issue: 5.

[17] E. Chisolm, "Geometric algebra." [Online]. Available: http://arxiv.org/abs/1205.5935

[18] K. S. Chong. Collision detection using the separating axis theorem. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169

[19] d000hg. Answer to “sphere plane collision”. [Online]. Available: https://www.gamedev.net/forums/topic/206825-sphere-plane-collision/2324187

[20] Y. Daoust. Answer to “cuboid inside generic polyhedron”. [Online]. Available: https://stackoverflow.com/a/24189921

[21] C. Doran, S. R. Gullans, A. Lasenby, J. Lasenby, and W. Fitzgerald, *Geometric algebra for physicists*. Cambridge University Press.

[22] D. Eberly, “Dynamic collision detection using oriented bounding boxes.”

[23] E. Galin, E. Guérin, A. Paris, and A. Peytavie, “Segment tracing using local lipschitz bounds,” in *Computer Graphics Forum*, vol. 39. Wiley Online Library, pp. 545–554, issue: 2.

[24] R. Gaul. Answer to “question about collision resolution”. [Online]. Available: https://www.gamedev.net/forums/topic/703356-question-about-collision-resolution/5410761/

[25] ——, “qu3e,” original-date: 2014-10-13T04:37:57Z. [Online]. Available: https://github.com/RandyGaul/qu3e

[26] ——. Understanding sutherland-hodgman clipping for physics engines. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/understanding-sutherland-hodgman-clipping-for-physics-engines--gamedev-11917

[27] D. Gregorius. Answer to “separating axis theorem OBB vs sphere”. [Online]. Available: https://gamedev.stackexchange.com/a/163874

[28] ——, “Robust contact creation for physics simulations,” in *Game Developers Conference*, p. 947.

[29] E. Guendelman, R. Bridson, and R. Fedkiw, “Nonconvex rigid bodies with stacking,” vol. 22, no. 3, pp. 871–878, publisher: ACM New York, NY, USA.

[30] J. C. Hart, "Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces," vol. 12, no. 10, pp. 527–545. [Online]. Available: http://link.springer.com/10.1007/s003710050084

[31] C. H. Hinton, *The fourth dimension.* S. Sonnenschein & Company.

[32] D. E. Johnson, "Projection and view frustums," p. 5.

[33] T. Johnson. Answer to "what is the MTV (minimum translation vector) in SAT (seperation of axis)?". [Online]. Available: https://gamedev.stackexchange.com/a/32550

[34] B. Lovrovic. Capsule inertia tensor. [Online]. Available: https://gamedev.net/tutorials/programming/math-and-physics/capsule-inertia-tensor-r3856

[35] L. Motl. Answer to "plane intersecting line segment". [Online]. Available: https://math.stackexchange.com/a/47611

[36] P. Olthof. Raymarching shader tutorial [playlist]. [Online]. Available: https://www.youtube.com/playlist?list=PL3POsQzaCw53iK_EhOYR39h1J9Lvg-m-g

[37] I. Quilez. 3d signed distance functions. [Online]. Available: https://iquilezles.org

[38] ——. Numerical normals for SDFs. [Online]. Available: https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm

[39] ——. Soft shadows. [Online]. Available: https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm

[40] E. Siegel. Ask ethan: Does our universe have more than 3 spatial dimensions? Section: Science. [Online]. Available: https://www.forbes.com/sites/startswithabang/2021/06/04/ask-ethan-does-our-universe-have-more-than-3-spatial-dimensions/

[41] U. Technologies. Unity - manual: Rays from the camera. [Online]. Available: https://docs.unity3d.com/Manual/CameraRays.html

[42] ——. Unity - manual: Understanding the view frustum. [Online]. Available: https://docs.unity3d.com/Manual/UnderstandingFrustum.html

[43] M. ten Bosch, "4d toys," last accessed September 23, 2021. [Online]. Available: https://4dtoys.com/

[44] ——, "How to walk through walls using the 4th dimension [miegakure: a 4d game]," last accessed September 23, 2021. [Online]. Available: https://www.youtube.com/watch?v=9yW--eQaA2I

[45] ——, "Miegakure," last accessed September 23, 2021. [Online]. Available: https://miegakure.com/

[46] TravisG. Answer to "separating axis theorem: Finding which edge normals to use". [Online]. Available: https://stackoverflow.com/a/6244218

[47] TURANSZKIJ. Capsule collision detection. [Online]. Available: https://wickedengine.net/2020/04/26/capsule-collision-detection/

[48] user_123abc. Answer to "line and plane intersection in 3d". [Online]. Available: https://math.stackexchange.com/a/84088

[49] A. Vaxman, "Lecture 4 - collisions." [Online]. Available: https://www.cs.uu.nl/docs/vakken/mgp/2018-2019/Lecture%204%20-%20Collisions.pdf

[50] J. Vermandere, "4d explorer," last accessed September 23, 2021. [Online]. Available: https://www.youtube.com/watch?v=nUExziADzjc

[51] ——, "Making a 4d game - 4d explorer," last accessed September 23, 2021. [Online]. Available: https://jellever.itch.io/4dexplorer

[52] E. W. Weisstein. Capsule. Publisher: Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/Capsule.html

[53] ——. Hypercube. Publisher: Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/Hypercube.html

[54] ——. Hypersphere. Publisher: Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/Hypersphere.html

[55] ——. Plane. Publisher: Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/Plane.html

[56] ——. Surface area. Publisher: Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/SurfaceArea.html

[57] ——. Volume. Publisher: Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/Volume.html