PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Interação e Visualização de objetos tetradimensionais

Guilherme Murad Heloui Teixeira

Projeto Final de Graduação

Orientação Prof. Waldemar Celes

Departamento de Informática

Curso de Graduação em Ciência da Computação

Rio de Janeiro

Junho de 2023

Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Murad Heloui Teixeira

Interação e Visualização de objetos tetradimensionais

Relatório de Projeto FInal

Relatório de Projeto Final, apresentado ao Programa de Ciência da Computação, do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Waldemar Celes

Rio de Janeiro
Junho de 2023

Resumo

Teixeira, Guilherme Murad. Celes, Waldemar. Interação e Visualização de objetos tetradimensionais. Rio de Janeiro, 2023. Projeto Final - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Este projeto tem como objetivo criar uma interação com objetos 4D. Para permitir ao usuário observar e intuir sobre o ambiente tetradimensional foram implementadas duas técnicas, o corte, onde fatias do objeto são criadas e manipuladas, e a projeção, onde o objeto é distorcido para ser totalmente visível em apenas 3 dimensões. Para isso o projeto explorou como conceitos de computação gráfica se estendem para um eixo adicional, formas de montar formas geométricas na quarta dimensão, assim como uma representação de objetos tridimensionais em um ambiente 2D.

Palavras-chave

Computação Gráfica, 4D, Primitivas, Fatiamento, Visualização

Abstract

Teixeira, Guilherme Murad. Celes, Waldemar. Interaction and Visualization of four-dimensional objects. Rio de Janeiro, 2023. Projeto Final - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

This project aims to create an interaction with 4D objects. In order to allow the user to observe and develop intuition about four-dimensional space, two techniques were implemented, the cross-section, where the object is sliced and manipulated, and the projection, where the object is distorted to be entirely visible in just 3 dimensions. To achieve this, the project explored how computer graphics concepts extended to an additional axis, ways of building shapes in the fourth dimension, as well as a representation of tridimensional objects in 2D.

Keywords

Computer Graphics, 4D, Primitives, Cross-Section, Visualization

1. Introdução

Nosso universo é composto por 3 dimensões espaciais, às quais comumente nos referimos como altura, largura e comprimento. Porém essa propriedade de nossa realidade não impediu matemáticos de séculos atrás de hipotetizar sobre o comportamento de um novo eixo, estendendo-se perpendicularmente aos três existentes. Muito de nosso conhecimento matemático podia ser aplicado, ou pelo menos estendido, para descrever o comportamento de uma quarta dimensão espacial. [1]

Curiosamente, essas hipóteses se provaram frutíferas e foram encontrados diversos benefícios de considerar uma quarta dimensão espacial. Inicialmente para áreas da matemática como a álgebra linear, porém eventualmente também para certos campos da física muitos anos depois. Nos dias atuais, essa utilidade também é frequentemente encontrada em setores da computação como a computação gráfica.

Entretanto, embora a matemática por trás possa ser compreendida por nós humanos, o mesmo não pode ser dito sobre a visualização de espaços tetradimensionais (4D). Nossa percepção do universo é limitada a 3 dimensões espaciais e imaginar uma extensão dessa realidade representa um enorme desafio, cuja superação poderia trazer grandes benefícios. Quando alguma intuição sobre o comportamento de objetos 4D é obtida, surge com ela uma facilidade de entendimento maior dos campos onde a existência de dimensões espaciais extras é relevante ou mesmo obrigatória.

Infelizmente essa intuição é notavelmente difícil de se alcançar. Este projeto procura, então, explorar a visualização em espaços 4D na forma de um programa interativo que auxilie o usuário a se acostumar com o comportamento de uma dimensão espacial adicional. Para alcançar esse objetivo, o projeto faz uso tanto da projeção quanto do corte. A projeção envolve mapear as coordenadas do espaço para um hiperplano tridimensional, similar a como um objeto 3D projeta uma sombra bidimensional em uma superfície. O corte, por sua vez, envolve a visualização apenas da interseção do mundo tetradimensional com o hiperplano, este podendo ser movimentado para que o espaço inteiro possa ser explorado. [2]

O projeto foi desenvolvido para o sistema Windows, utilizando a Engine de desenvolvimento open-source “Godot Engine” [3] em sua versão 4.0. A engine foi

escolhida por sua simplicidade e versatilidade, permitindo não apenas o acesso à interface gráfica “Vulkan” [4] como também facilitando todo o desenvolvimento de um programa interativo. A utilização do Godot trouxe não apenas todas as vantagens de poder se comunicar diretamente com a interface gráfica, permitindo o controle direto sobre os elementos fundamentais do Vulkan, como também a possibilidade de usufruir de inúmeras ferramentas que facilitam a implementação de elementos como movimentação de câmera e detecção de colisão. Se futuramente este projeto vier a ser estendido, a engine também pode ser útil para a implementação de funcionalidades mais complexas como a realidade virtual, cujo potencial de aplicação neste programa seria bem aproveitado.

Vale notar que muito do que será discutido não se aplica necessariamente apenas à existência de uma única dimensão espacial adicional. Certos campos de estudo consideram a possibilidade de inúmeras dimensões espaciais; todavia, o escopo deste projeto limita-se apenas ao espaço 4D.

2. Espaço tridimensional

Antes de representar um objeto 4D em um ambiente 3D, primeiro foi desenvolvida a capacidade de representar objetos tridimensionais com apenas duas dimensões, de forma que o usuário pudesse criar uma intuição maior sobre o comportamento de projeções e cortes em um contexto mais familiar. Outro benefício dessa escolha é o auxílio ao desenvolvimento do programa, uma vez que grande parte da lógica necessária na quarta dimensão pode ser extrapolada a partir das dimensões inferiores.

2.1 Primitivas

O primeiro passo envolveu decidir as primitivas gráficas. Ou seja, as formas geométricas mais simples que o programa seria capaz de representar, as quais devem compor qualquer outro objeto da cena. Com o intuito de facilitar a transformação de objetos entre os espaços de diferentes dimensões, para cada um foi escolhido o simplex de uma dimensão inferior, sendo o simplex a forma geométrica mais simples de seu espaço dimensional [5]. Ou seja, para os objetos

2D foi escolhida a linha (simplex unidimensional, ou 1-Simplex), e para objetos 3D foi escolhido o triângulo (simplex bidimensional, ou 2-Simplex).

2.2 Implementação

A cena é apresentada em duas vistas. A primeira exibe um objeto tridimensional que pode ser rotacionado e movido pelo usuário, assim como um plano abrangendo as coordenadas onde o valor y se iguala a 0. Já a segunda exibe o universo bidimensional representado pelo plano anteriormente mencionado. Nessa vista o usuário pode observar a fatia do objeto 3D que intercepta o plano, assim como a projeção que este tenha sobre o plano. O objetivo desejado com a projeção é ilustrado na Figura 2.1.

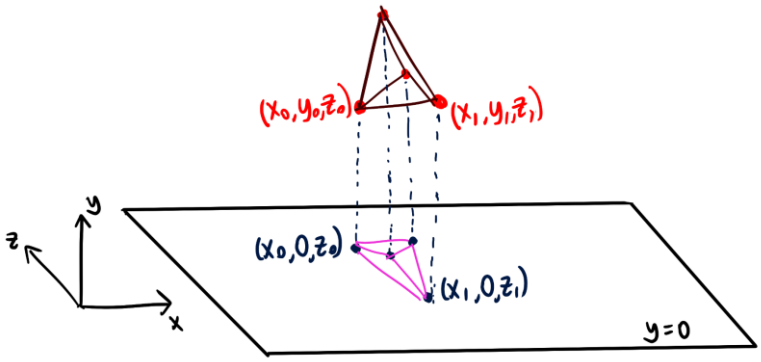

Figura 2.1 Projeção ortogonal de um tetraedro sobre o plano $y = 0$

Para garantir total controle sobre os objetos tridimensionais, estes não foram criados usando as ferramentas dedicadas que a engine disponibiliza, mas sim definidos manualmente como a lista de vértices, assim como uma outra lista que relaciona se cada vértice com cada primitiva que o contém, permitindo que o programa se comunique diretamente com a interface gráfica para desenhá-los nas telas. Como pode ser observado no Trecho de Código 2.1, também há uma lista para as cores de cada face e uma última para as arestas, que são importantes para a projeção.

```
if index == SHAPE.TETRAHEDRON:
        vertices = [
```

```
            Vector3(-1./3,  0.0       ,  sqrt(8./9)),
            Vector3(-1./3,  sqrt(2./3), -sqrt(2./9)),
            Vector3(-1./3, -sqrt(2./3), -sqrt(2./9)),
            Vector3( 1.0 ,  0.0       ,  0.0)
      ]
      tris = [
            [0, 2, 1],
            [0, 1, 3],
            [0, 3, 2],
            [1, 2, 3],
      ]
      colors = [
            # Cor de cada face
            Color(1, 1, .5),
            Color(0, 0, 1),
            Color(1, 0, 0),
            Color(0, 1, 0)
      ]
      wireframe = [
            [1,2,3],  # v0
            [2,3],    # v1
            [3],       # v2
            []         # v3
      ]
```

Trecho de Código 2.1 Representação de um tetraedro no código

A projeção é alcançada sem muitas complicações, transformando as arestas de cada triângulo que compõe o objeto tridimensional em primitivas bidimensionais, descartando suas coordenadas y no processo. O resultado é uma simples projeção ortogonal, agindo como uma espécie de sombra do objeto. Na Figura 2.2, observamos o cubo posicionado acima do plano, com sua projeção visível no canto esquerdo superior.

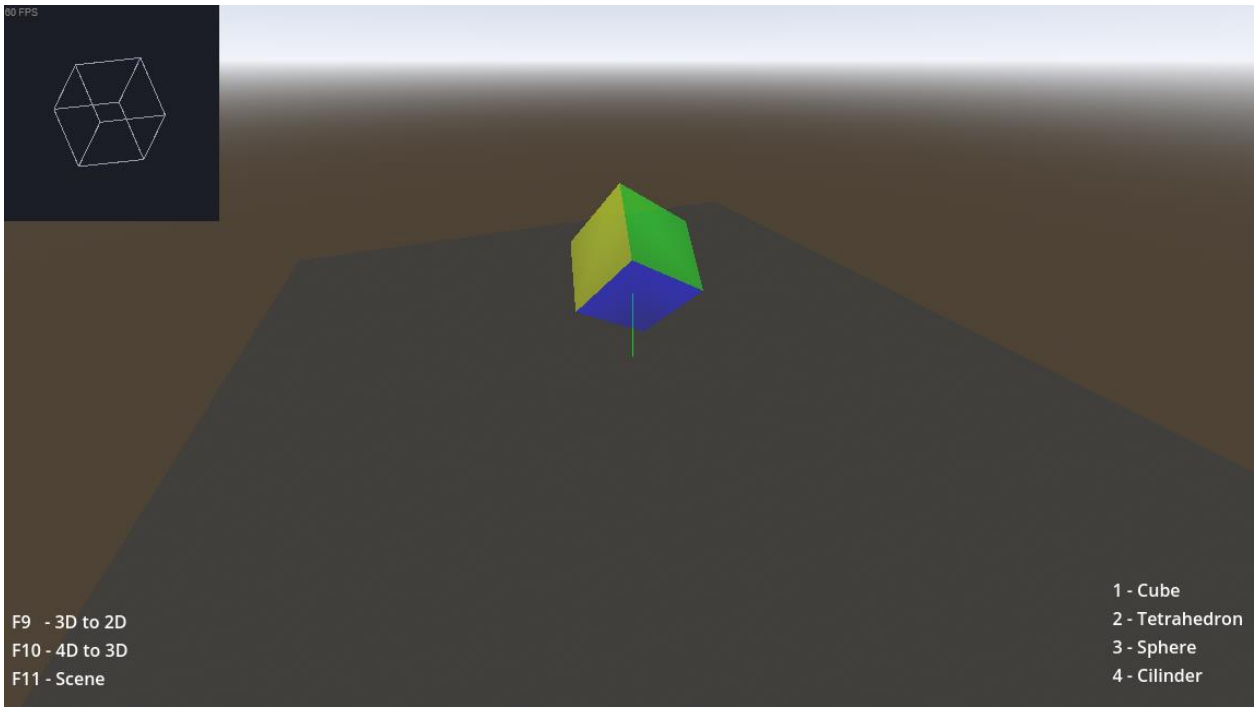

Figura 2.2 Cubo tridimensional e sua projeção ortogonal sobre o plano

Já o corte envolve uma lógica mais avançada. Para cada triângulo que compõe o objeto, é preciso descobrir como que este triângulo intercepta o plano. Sendo esta intercepção composta necessariamente das primitivas utilizadas na dimensão abaixo (neste caso, a linha do espaço bidimensional).

Foi então elaborada uma lista com os possíveis arranjos que os vértices de um triângulo poderiam assumir em relação a um plano. O caso onde todos os seus vértices estão no mesmo lado do plano resulta em um corte vazio, porém, no outro extremo, se todos os vértices forem contidos pelo plano, então todas as arestas do triângulo pertencem ao corte. Nos casos intermediários, onde vértices estão em lados diferentes do plano, onde apenas alguns vértices estão contidos no plano ou em casos onde ambas as coisas acontecem, foi necessário levar em consideração comportamentos mais específicos como os novos vértices criados pela interseção de arestas com o plano. Todos os casos únicos possíveis (ou seja, descartando os casos que forem espelhados de outros já considerados) estão representados na Figura 2.3.

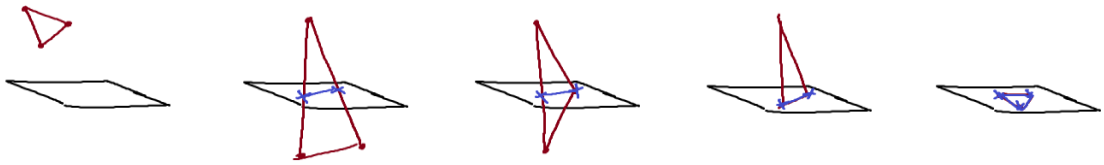

Figura 2.3 Configurações de um triângulo em torno de um plano de corte e seus resultados

O resultado do corte pode ser visto na Figura 2.4, também no canto esquerdo superior.

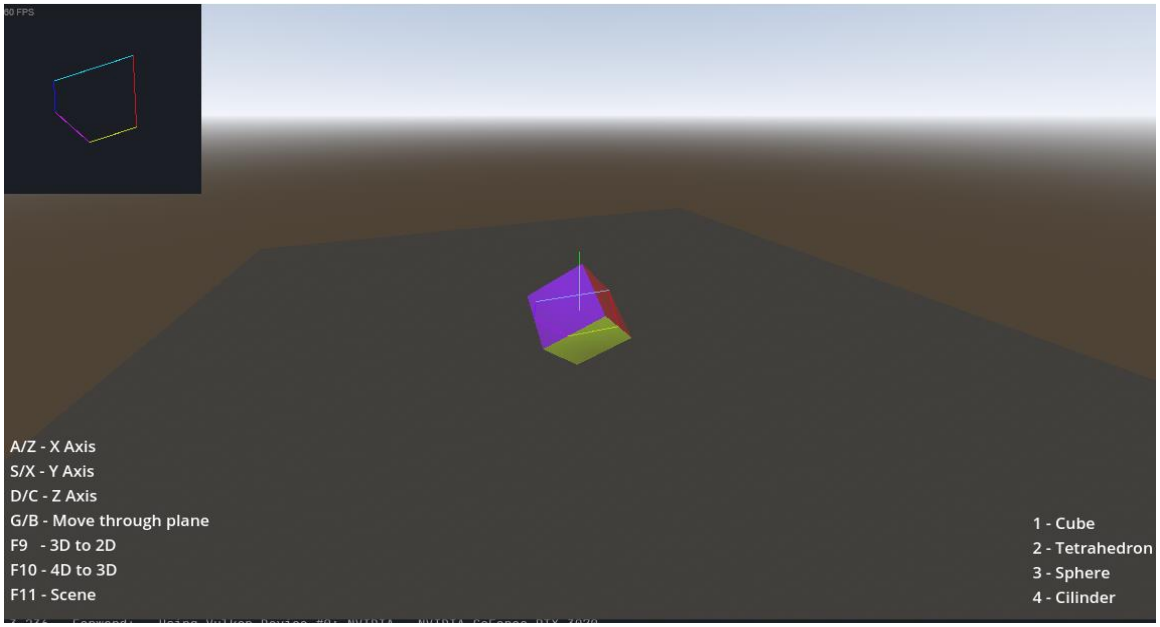

Figura 2.4 Cubo tridimensional e sua fatia resultante pela interseção com o plano de corte

3. Espaço tetradimensional

Com a implementação anterior bem sucedida o projeto pôde focar em sua premissa principal, a representação 3D de um universo 4D. Embora muita da lógica utilizada para isso seja uma extensão natural da etapa anterior, esta adaptação não é trivial.

Para auxiliar com a introdução de novos conceitos, foi montada a Tabela 3.1 que será referenciada ao longo deste documento.

Dimensão	0D	1D	2D	3D	4D
Simplex	Ponto	Linha	Triangulo	Tetraedro	5-Célula
Primitiva		Ponto	Linha	Triangulo	Tetraedro
n-face	Vértices	Arestas	Faces	Células	4-Faces
n-cubo			Quadrado	Cubo	Tesserato
n-esfera			Círculo	Esfera	3-Esfera

Tabela 3.1 Tabela de diversos conceitos em cada espaço dimensional

3.1 Implementação

Para esta nova cena, deve existir um objeto 4D, um hiperplano tridimensional de corte, e a fatia resultante. Entretanto, apenas esta é exibida ao usuário, em uma única tela de exibição.

Estendendo a lógica que foi escolhida para a representação de primitivas anteriormente, foi decidido que a primitiva usada seria o tetraedro (Simplex tridimensional ou 3-Simplex), como exibido na Tabela 3.1. Isso significa que para realizar o corte do objeto, é necessário considerar todos os casos em que um tetraedro pode estar posicionado em relação a um hiperplano tridimensional, e quais seriam os triângulos resultantes de cada corte. Enquanto para os triângulos este passo era intuitivo (devido a nossa capacidade de imaginar o espaço tridimensional), desta vez é necessária uma lógica mais elaborada.

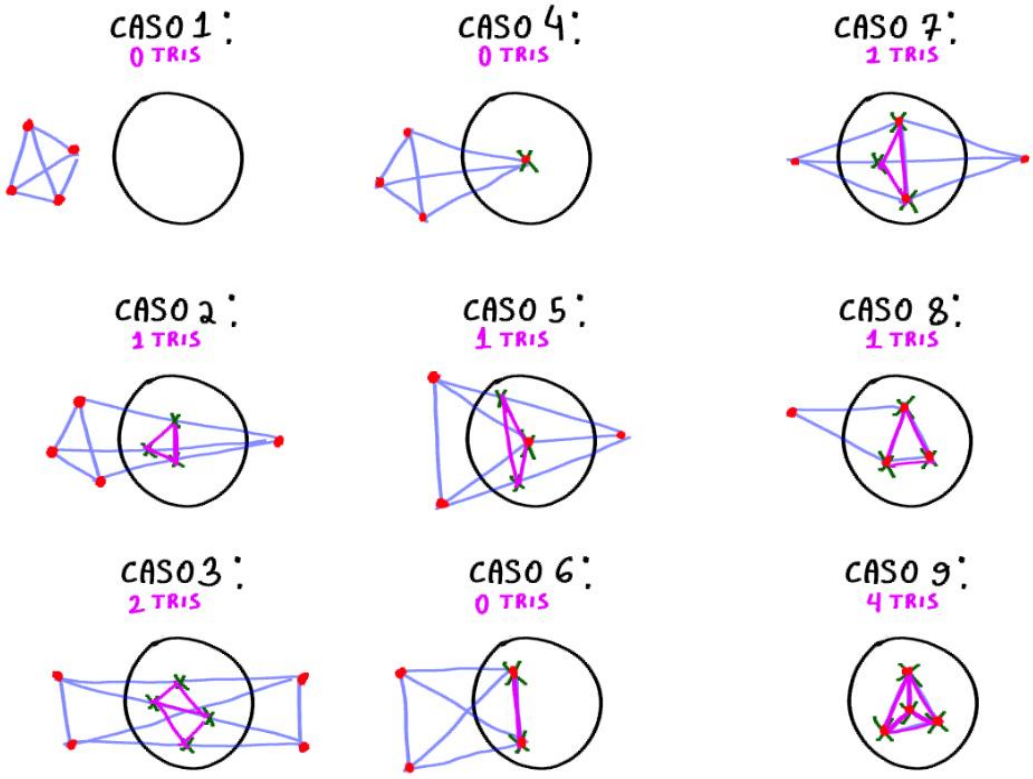

Figura 3.1 Configurações de um tetraedro em torno de um hiperplano de corte e seus resultados

Na Figura 3.1, os círculos pretos são uma representação arbitrária do hiperplano de corte. Os pontos vermelhos representam os vértices do tetraedro, estes podendo estar de cada lado do hiperplano assim como contidos no mesmo. Para os propósitos deste diagrama, a posição específica dos pontos não é relevante, apenas em qual desses 3 setores cada um se encontra. Assim como no capítulo anterior, apenas os casos únicos são considerados, e qualquer caso que seja equivalente (ou espelhado) aos já representados pela figura foram descartados.

Para o caso 1, onde todos os vértices estão em um único lado, é evidente que o tetraedro não é cortado pelo plano e por consequência não há triângulos resultantes no corte, nos casos 4 e 6 também temos cortes vazios porém menos intuitivos. Nestes, os cortes seriam respectivamente um ponto e uma linha, formas inexistentes em nosso ambiente tridimensional, uma vez que são inferiores à nossa primitiva (o triângulo). Mesmo se fôssemos representá-las, uma ou mais de suas dimensões seriam nulas e portanto não seriam visíveis. Por esses motivos, o programa trata todos os três casos como vazios.

Nos casos 2 e 3, o resultado é obtido a partir dos pontos onde as arestas do tetraedro interceptam com o hiperplano (representados por um X verde). Os triângulos resultantes são então os que possuem esses pontos de interseção como vértices. Pelo fato do tetraedro ser convexo, o arranjo de 2 triângulos escolhidos para os 4 vértices do caso 3 é irrelevante.

Nos casos 8 e 9, temos que os triângulos resultantes fazem parte do próprio tetraedro, enquanto nos casos 5 e 7 existem tanto pontos que já faziam parte do tetraedro quanto novos pontos de interseção.

Para lidar com a projeção, a abordagem foi similar à utilizada no capítulo anterior, com a grande diferença sendo o descarte da coordenada w, ao invés da coordenada y. Tal como na cena anterior, esta projeção depende da existência de uma lista de arestas, distinta da lista de primitivas, uma vez que idealmente as arestas internas de uma célula não devem ser projetadas.

Para que a projeção pudesse melhor auxiliar a compreensão do que o corte representa, suas arestas foram pintadas de vermelho e azul de acordo com suas posições relativas ao hiperplano de corte, ou seja, cada aresta projetada informa se ela localiza-se no lado positivo ou negativo do hiperplano. Suas opacidades também foram reduzidas de acordo com a distância para o mesmo. Essas adições resultam em uma projeção que ajuda o usuário a adquirir uma intuição sobre como a posição do plano de corte afeta o objeto tridimensional exibido. Podemos observar a projeção e o corte em ação simultaneamente na Figura 3.2, onde um hiperplano de corte atravessa um objeto 4D.

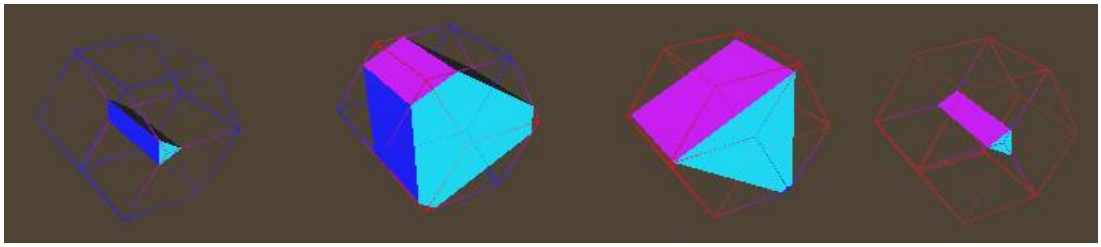

Figura 3.2 Hiperplano de corte atravessando um tesserato

3.2 Representação de formas

Um grande desafio para a implementação de uma espaço tetradimensional é a incapacidade humana de visualizar o ambiente 4D e, por consequência, a criação de qualquer objeto precisa ser cautelosamente feita de uma forma que garanta que sua forma realmente seja a desejada. Foi decidido que a primitiva utilizada seria o tetraedro, mas representar um objeto qualquer com tetraedros não é uma tarefa intuitiva.

Foi importante que o projeto fosse desenvolvido mantendo em mente o conceito de células. Como visto na Tabela 3.1, um objeto 4D é composto de células 3D da mesma forma que faces compõem objetos tridimensionais. Independente da escolha da primitiva, não estamos limitados a manter cada célula como exclusivamente um tetraedro.

Para exemplificar o comentado, vamos considerar o processo para criar um tesserato, isto é, a forma 4D análoga a um cubo, comumente chamada de hipercubo.

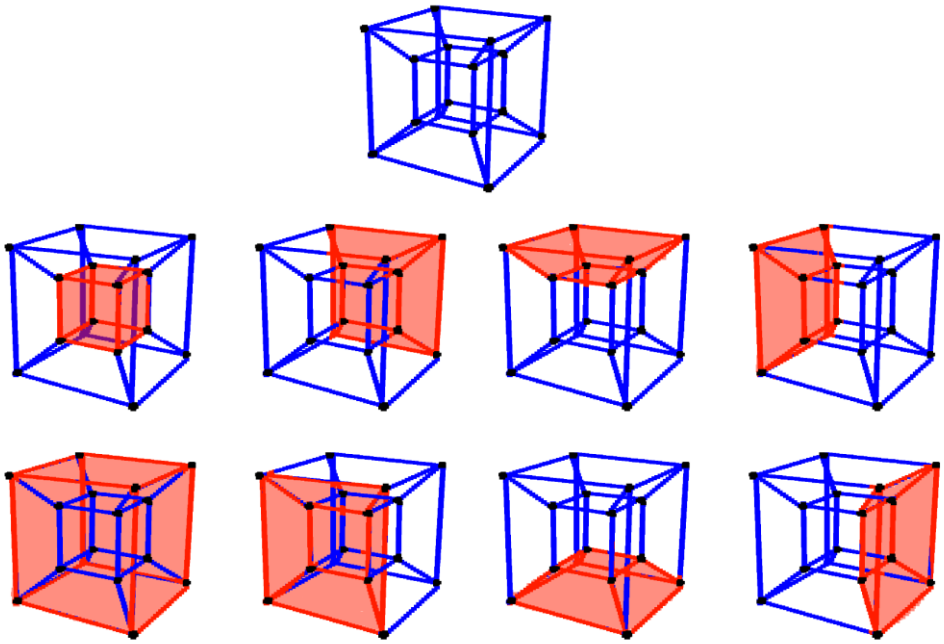

Figura 3.3 Projeção de um tesserato com suas oito células cúbicas destacadas

Na Figura 3.3, podemos observar como um hipercubo é composto de 8 células cúbicas (as quais aparentam estar distorcidas devido à projeção). Com esse conhecimento em mente foi necessário apenas representar um cubo com tetraedros e então usar 8 destes para compor a forma desejada.

Aproveitando a situação, esta separação em células também foi usada como uma maneira de colorir o objeto, colorindo cada célula com uma cor diferente para poder identificá-las no corte. Podemos novamente ver o fatiamento de um tesserato em ação na Figura 3.2, e na Figura 3.4 temos um Icositetrachoron (ou 24-Célula) com suas células coloridas.

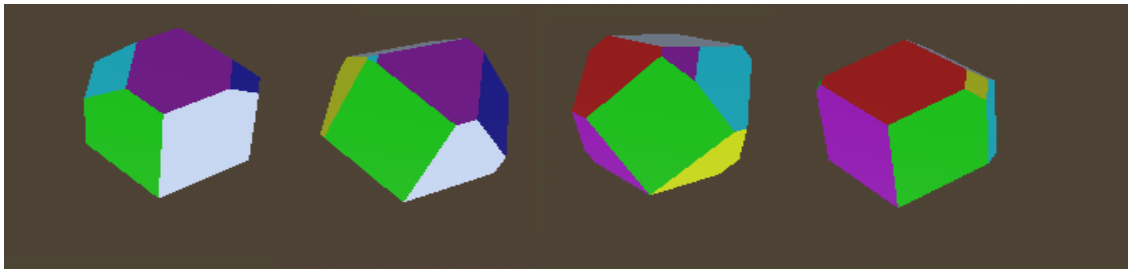

Figura 3.4 Rotação de um Icositetrachoron ao redor dos eixos Z e W, com cada uma de suas 24 células coloridas de cores diferentes.

3.3 Extensão de formas tridimensionais

Com o intuito de ter mais opções para a criação de formas tetradimensionais, também foi desenvolvido um método para poder estender formas tridimensionais no eixo *w*, da mesma forma que podemos estender um quadrado em um cubo ou um círculo em um cilindro.

Para isso foi utilizado o processo de “tetrahedralization”, como descrito em *“Direct construction of a four-dimensional mesh model from a three-dimensional object with continuous rigid body movement”[6].*

No artigo mencionado, esta metodologia é utilizada para representar o movimento de um objeto tridimensional como um objeto 4D. Embora esse objetivo seja diferente, a maneira que a movimentação de um triângulo gera um prisma de tetraedros se encaixa perfeitamente nas necessidades deste projeto.

Dois tetraedros podem ser adquiridos conectando os triângulos de origem e destino ao centróide do prima, e então cada par de arestas paralelas compõe 4 tetraedros, totalizando 14 tetraedros. No nosso caso, o segundo triângulo é posicionado a uma distância fixa, e esse processo é repetido para todos os triângulos de nosso objeto. A Figura 3.5 destaca esses tetraedros.

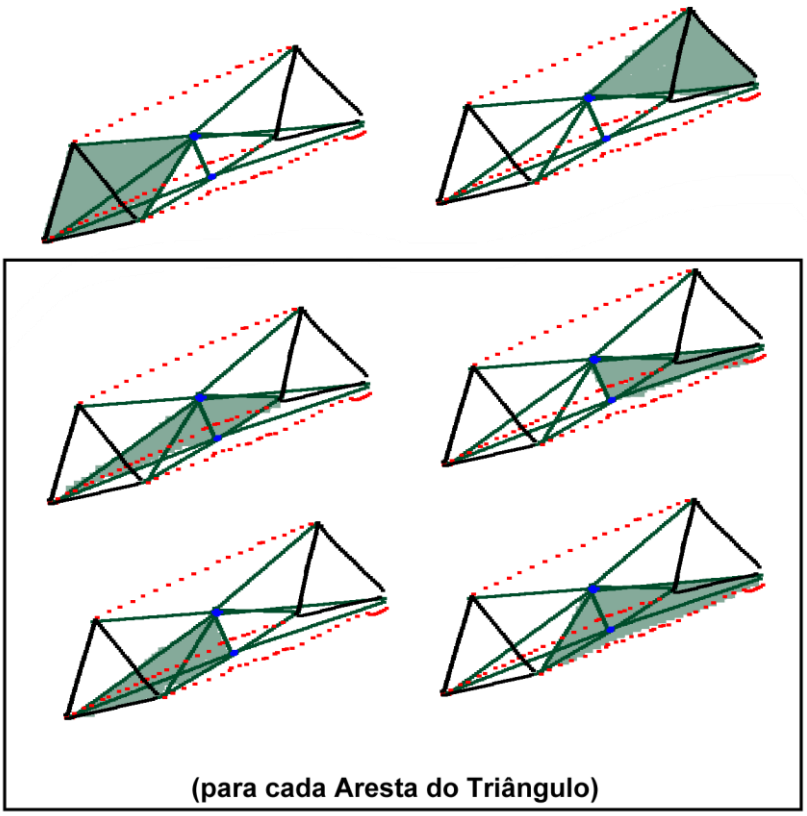

Figura 3.5 Os 14 tetraedros que conectam os dois triângulos

Porém, este processo não é suficiente para nosso programa. Uma vez que nossa primitiva é o tetraedro, o objeto tridimensional em cada ponta precisa também ser convertido para uma representação com a nova primitiva. Situação análoga a um círculo que, para ser estendido para um cilindro, precisa ser posteriormente fechado, uma vez que se apenas suas arestas fossem estendidas para triângulos, o resultado seria apenas um tubo

Esta etapa é notavelmente difícil de ser feita em formas côncavas, portanto é feita apenas para formas convexas, criando tetraedros a partir de todos os seus triângulos ligados em um único ponto no interior da forma, exemplificado na Figura 3.6 enquanto o resultado de todo o processo pode ser visto na Figura 3.7.

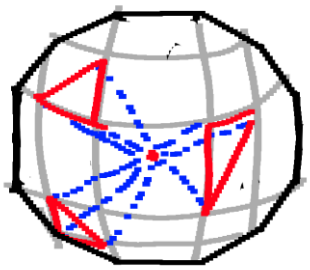

Figura 3.6 Tetraedros sendo criados a partir de cada triângulo de um objeto 3D

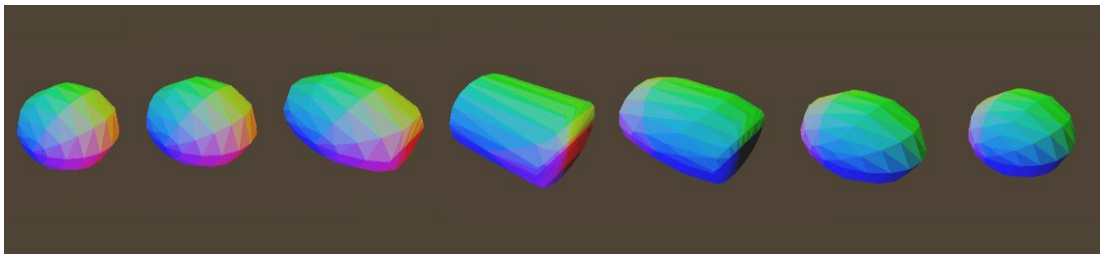

Figura 3.7 Objeto resultante da extensão de uma esfera sendo rotacionado 180 graus ao redor do eixo w

3.4 3-Esfera

A 3-Esfera é a forma tetradimensional análoga a uma esfera ou um círculo nos espaços dimensionais inferiores. Todos os pontos de sua superfície são equidistantes de seu centro, satisfazendo a equação $(x-a)^2+(y-b)^2+(z-c)^2+(w-d)^2=r^2$, onde (a,b,c,d) são as coordenadas do centro da hiperesfera.

Como previamente estabelecido, este projeto se beneficia demasiadamente da extensão de implementações dos casos análogos em 3 dimensões para que estas possam ser estendidas para incluir um eixo adicional. Portanto foi importante olhar para como a esfera foi construída no projeto.

```
var delta_phi = PI / num_rings
var delta_theta = 2 * PI / num_segments

    for i in range(num_rings + 1):
       var phi = i * delta_phi
       var sin_phi = sin(phi)
       var cos_phi = cos(phi)
```

```
    for j in range(num_segments + 1):
        var theta = j * delta_theta
        var sin_theta = sin(theta)
        var cos_theta = cos(theta)

        var x = cos_theta * sin_phi
        var y = sin_theta * sin_phi
        var z = cos_phi

        vertices.append(Vector3(x, y, z))
```

Trecho de Código 3.1 Criação dos vértices de uma esfera

No Trecho de Código 3.1, podemos observar como a esfera foi dividida em anéis e segmentos. Estes são percorridos pelas variáveis θ (theta) e φ (phi) como uma volta inteira e uma semi-volta, respectivamente, para gerar todos os vértices da esfera, como demonstrado na Figura 3.8.

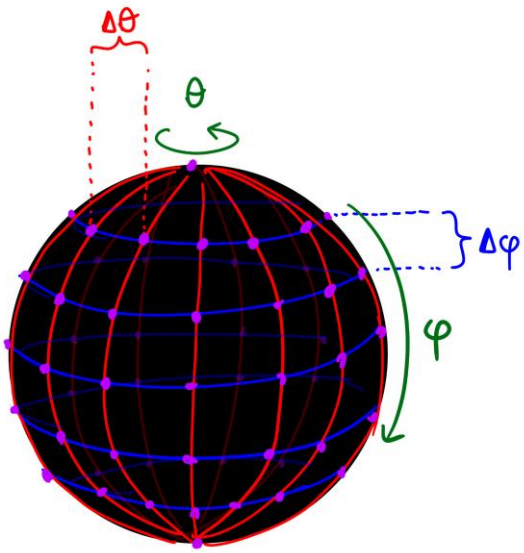

Figura 3.8: Vértices de uma esfera sendo formados a partir das variáveis θ e φ

Também na Figura 3.8, vemos como os anéis e segmentos formam quadriláteros que compõem a superfície esférica. Como dito durante a decisão das primitivas, o triângulo pode ser utilizado para montar qualquer forma mais

complexa e, neste caso, apenas dois triângulos são necessários para compor cada quadrilátero.

```
if i > 0 and j > 0:
      var v0 = (i - 1) * (num_segments + 1) + j - 1
      var v1 = i * (num_segments + 1) + j - 1
      var v2 = i * (num_segments + 1) + j
      var v3 = (i - 1) * (num_segments + 1) + j

      triangles.append([v0, v1, v2])
      triangles.append([v0, v2, v3])
```

Trecho de Código 3.2 Criação dos triângulos a partir dos vértices

No Trecho de Código 3.2, contido no loop apresentado no Trecho de Código 3.1, vemos como os dois triângulos são criados. Vale notar que, devido à escolha de nossa primitiva ser um polígono, qualquer forma curva é necessariamente uma aproximação. Porém, quanto maiores os números de anéis e segmentos, mais próximo o objeto resultante é de uma esfera.

Para que essa lógica se aplique a uma 3-Esfera, é necessário um novo parâmetro angular além de θ e φ. Foi adicionado um novo conceito além de anéis e segmentos, as colunas, ditadas pelo parâmetro ψ (psi) [7]. A implementação pode ser observada no Trecho de Código 3.3.

```
var delta_phi = 2 * PI / num_rings
var delta_theta = 1 * PI / num_segments
var delta_psi =   1 * PI / num_columns
for i in range(num_rings + 1):
   var phi = i * delta_phi
   var sin_phi = sin(phi)
   var cos_phi = cos(phi)

   for j in range(num_segments + 1):
      var theta = j * delta_theta
      var sin_theta = sin(theta)
      var cos_theta = cos(theta)
```

```
    for k in range(num_columns + 1):
        var psi = k * delta_psi
        var sin_psi = sin(psi)
        var cos_psi = cos(psi)

        var x = sin_psi * sin_phi * sin_theta
        var y = sin_psi * sin_theta * cos_phi
        var z = sin_psi * cos_theta
        var w = cos_psi
```

Trecho de Código 3.3 Criação dos vértices de uma 3-Esfera

Por fim, resta apenas conectar os vértices com nossa primitiva, os tetraedros. O processo é de maneira similar à como os quadriláteros foram criados anteriormente, porém com um paralelograma. Como vimos durante a criação de um tesserato (cujas células também eram paralelogramas), cada célula pode ser composta de 5 tetraedros. Este processo é então realizado para cada conjunto de 8 vértices próximos. Além disso, foi aproveitada a existência de 3 separações (Anéis, segmentos e colunas) para atribuir um dos canais de cor RGB para cada, resultando em uma coloração única para cada célula.

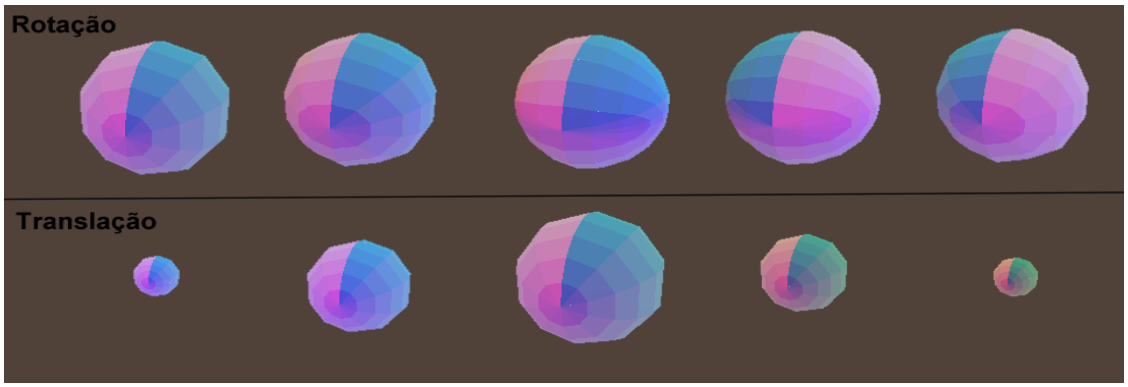

Figura 3.9: Transformações sobre uma hiperesfera

Na Figura 3.9 vemos como o resultado se comporta exatamente como esperado. A rotação de uma 3-Esfera não altera o formato da fatia, embora vemos como estamos observando células diferentes durante a rotação. Enquanto a translação através do hiperplano de corte altera o tamanho da fatia. Na Figura

3.10, com a projeção habilitada, observamos como o perímetro externo nunca é modificado.

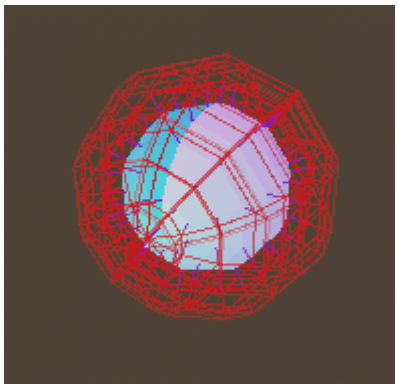

Figura 3.10 Projeção da hiperesfera habilitada

4. Ambiente de Desenvolvimento

Como mencionado anteriormente, o projeto foi desenvolvido na engine open-source “Godot”. Esta foi atualizada para sua versão 4.0 no início do desenvolvimento do projeto e foi decidido que o que já estava feito seria adaptado para as mudanças na engine pois embora a documentação ainda estivesse sendo atualizada, trazendo um desafio a mais para o desenvolvimento, haviam vários benefícios de se usar a versão nova. O mais significativo foi a mudança do uso da interface gráfica “OpenGL” [8] para a interface “Vulkan”, permitindo não apenas melhor performance quanto um controle maior sobre o programa, uma vez que é uma interface comparativamente mais baixo-nível.

4.1 Performance

Durante o desenvolvimento do projeto, houve uma grande preocupação com a performance. Enquanto um programa tradicional poderia lidar com formas muito mais complexas com eficiência, a necessidade deste projeto de realizar operações constantes sobre cada tetraedro, reconstruindo a configuração de triângulos resultantes após qualquer movimento, trouxe um grande desafio para o desenvolvimento.

Porém, uma grande vantagem do ambiente escolhido é a presença de um analisador de performance dentro da própria engine, permitindo observar em tempo real quais partes do programa demoravam mais ou usavam mais recursos,

auxiliando a detecção de partes problemáticas do código. Esse analisador pode ser observado na Figura 4.1.

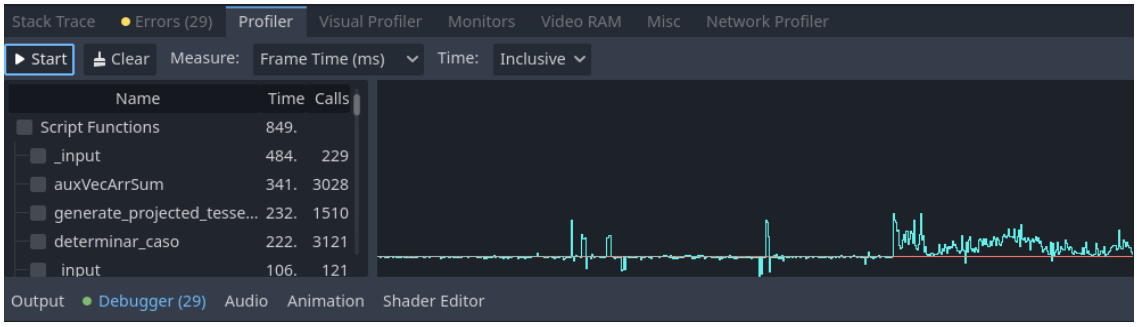

Figura 4.1 Analisador de execução da Godot Engine

Durante o desenvolvimento, com o auxílio da ferramenta acima, foi evidente que os cálculos sobre cada tetraedro do objeto tetradimensional eram o maior obstáculo para alcançar uma performance aceitável. Por isso, além de tentar otimizar o código em questão, foi decidido separar os tetraedros em threads separados. A lista de primitivas foi dividida em um dado número de threads de execução e cada sublista resultante foi processada em paralelo.

Este esforço foi recompensado com uma melhora drástica na velocidade de execução e taxa de quadros do programa.

4.2 Cena

O principal objetivo do trabalho era a criação de um espaço interativo, e limitar-se apenas a uma única cena com apenas um objeto em seu centro limitava demasiadamente o potencial do projeto. Portanto foi adicionada uma cena adicional, com o intuito de permitir ao usuário se movimentar por um espaço com múltiplos objetos tetradimensionais.

Para alcançar este objetivo foi aproveitado o modelo de licença livre “First Person Starter” [9] por Dimitar Dimitrov no GitHub. Este modelo é composto de um espaço bem simples com um personagem controlável em primeira pessoa, o modelo foi então adaptado para as necessidades do projeto, modificando propriedades da movimentação, câmera, detecção de colisão e modificação do ambiente para a inclusão das formas 4D.

Para simplificar o uso do programa para o usuário, ao invés de permitir que cada objeto fosse movimentado, estes se movem (ou não) por conta própria, enquanto o indivíduo controla a posição do hiperplano que corta todos eles com a

roda de rolagem do mouse, e um indicador na tela com a mesma coloração azul e vermelha usada para a projeção, de forma a auxiliar o usuário encontrar os objetos na cena. Na Figura 4.2 observamos o resultado.

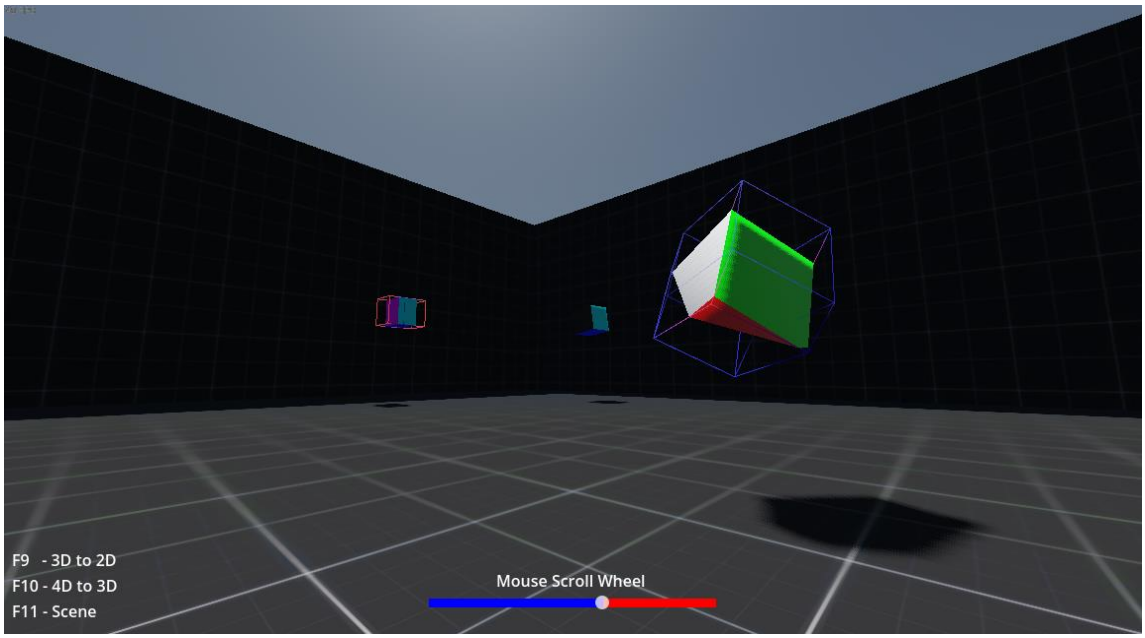

Figura 4.2 Parte da cena resultante, com um tesserato em movimento e alguns pêndulos tetradimensionais.

5. Considerações finais

Neste projeto, foi desenvolvido um programa que atende à premissa de aclimatar o usuário com a existência de um eixo adicional. A implementação foi bem-sucedida e o programa resultante cumpriu o desejado. Porém houve um desvio da proposta inicial apresentada no início de Projeto Final I, que inicialmente estava mais voltada para o desenvolvimento de algum desafio, como uma pista de obstáculos, um labirinto ou um quebra cabeça. O resultado alcançado, por sua vez, acabou dando maior foco para a inclusão de uma grande variedade de objetos com propósitos diferentes e métodos de implementação diferentes, sem necessariamente algum objetivo desafiador em mente para o usuário, que agora assume um papel mais próximo a um observador e não um jogador.

Esta mudança, embora não inicialmente planejada, foi intencional. Substituindo certos desafios que viriam na proposta inicial, como um maior foco

em garantir a performance, por novos desafios que foram julgados ser mais interessantes para o projeto.

Dito isto, alcançar maior interatividade seria um grande objetivo para possíveis trabalhos futuros, motivando o usuário a pensar com maior cuidado sobre o ambiente 4D, possivelmente revisitando a ideia de navegar em um labirinto tetradimensional. Além disso, seria extremamente benéfica a adição de compatibilidade com a tecnologia de realidade virtual, permitindo maior imersão e, por consequência, melhor intuição resultante. Por último, outra possível direção seria a inclusão de simulações físicas no espaço 4D, no projeto foi incluída a colisão das fatias com o mundo tridimensional, porém fora do eixo de corte não existe qualquer computação.

O projeto, assim como seu código fonte, está disponível em https://github.com/GraekTarmikos/4DSim para sistemas Windows.

6. Referências Bibliográficas

[1] **Four-dimensional space.** Disponível em: https://en.wikipedia.org/wiki/Four-dimensional_space. Acesso em Maio/2023

[2] KAUR, A; SHARMA, N. **A Review on 4D Visualization.** Disponível em: https://www.ijert.org/research/a-review-on-4d-visualization-IJERTCONV5IS03071.pdf. Acesso em Setembro/2022

[3] **GODOT ENGINE.** Disponível em: https://godotengine.org/

[4] **VULKAN.** Disponível em https://www.vulkan.org/

[5] WEISSTEIN, ERIC W. **"Simplex."** From MathWorld – A Wolfram Web Resource. Disponível em https://mathworld.wolfram.com/Simplex.html. Acesso em Abril/2023

[6] Ikuru Otomo, Masahiko Onosato, Fumiki Tanaka, **"Direct construction of a four-dimensional mesh model from a three-dimensional object with continuous rigid body movement."** Disponível em https://www.sciencedirect.com/science/article/pii/S2288430014500152. Acesso em Abril/2023

[7] **3-sphere.** Disponível em: https://en.wikipedia.org/wiki/3-sphere. Acesso em Abril/2023

[8] **OPENGL.** Disponível em: https://www.opengl.org.

[9] DIMITROV, D. **First Person Starter.** Disponível em: https://github.com/Whimfoome/godot-FirstPersonStarter. Acesso em Março/2023.