FACULDADE DE TECNOLOGIA DE SÃO PAULO

PAULO GUILHERME GUARNERI COSTA

Ferramenta para Tesselação no Plano Hiperbólico

SÃO PAULO

2021

FACULDADE DE TECNOLOGIA DE SÃO PAULO

PAULO GUILHERME GUARNERI COSTA

Ferramenta para Tesselação no Plano Hiperbólico

Trabalho submetido como exigência parcial
para a obtenção do Grau de Tecnólogo em
Análise e Desenvolvimento de Sistemas
Orientador: Professor Doutor Silvio do Lago Pereira

SÃO PAULO

2021

FACULDADE DE TECNOLOGIA DE SÃO PAULO

PAULO GUILHERME GUARNERI COSTA

Ferramenta para Tesselação no Plano Hiperbólico

Trabalho submetido como exigência parcial para a obtenção do Grau de

Tecnólogo em Análise e Desenvolvimento de Sistemas.

Parecer do Professor Orientador

Aprovado

Conceito/Nota Final: 10,0

Atesto o conteúdo contido na postagem do ambiente TEAMS pelo aluno e assinada por mim para avaliação do TCC.

Orientador: Professor Doutor Silvio do Lago Pereira

SÃO PAULO, 02 de Dezembro de 2021.

Assinatura do Orientador

Assinatura do aluno

AGRADECIMENTOS

Agradeço primeiramente a Deus, por todas as oportunidades que me foram concedidas e por ter me sustentado durante toda minha vida.

À minha esposa e melhor amiga Ana, por compartilhar das minhas alegrias e preocupações, sempre me apoiando, me dando forças para prosseguir e tornando minha vida mais feliz a cada dia.

Aos meus pais Pedro Paulo e Cláudia, e meus irmãos Vinícius e Nathalya pelo apoio constante, pelos conselhos e pelo auxílio genuíno que sempre recebi.

Aos meus sogros Lourenço e Zilene e minha cunhada Victória pela companhia, amizade e apoio que sempre me deram.

Aos meus avós Pedro e Lurdes, e a meus parentes Maria e José Vitor, Pedro e Cida por terem me acolhido em suas casas quando precisei e terem me tratado como filho.

Aos meus amigos Alexandre, Eduardo H., Cláudio, Eduardo R., Welerson, Henrique e Bruna pelo incentivo, amizade, e apoio aos meus projetos.

Aos meus amigos Rubens, Julianna e Erick pela companhia, amizade e esforços que tivemos em conjunto.

Ao meu professor e orientador Silvio, que me proporcionou um conhecimento sólido, sobre o qual fundamento cada novo projeto, incluindo este.

RESUMO

A partir de princípios de geometria hiperbólica como os modelos do plano hiperbólico e as tesselações do plano hiperbólico, e de conhecimentos relacionados à computação gráfica, esse projeto visa desenvolver uma ferramenta capaz de reproduzir e movimentar mapas hiperbólicos ladrilhados. A ferramenta servirá de extensão para o motor gráfico Unity Engine e disponibilizada de forma gratuita.

Palavras-chaves: Geometria Hiperbólica, Plano Hiperbólico, Tesselações, Computação Gráfica, Unity Engine.

ABSTRACT

From hyperbolic geometry principles such as hyperbolic plane models and hyperbolic plane tessellations, and knowledge related to computer graphics, this project aims to develop a tool capable of reproducing and moving tiled hyperbolic maps. The tool will serve as an extension to the Unity Engine and will be available at no cost.

Keywords: Hyperbolic Geometry, Hyperbolic Plane, Tessellations, Computer Graphics, Unity Engine.

LISTA DE FIGURAS

Pág.

Figura 1 - Disco de Poincaré. 14
Figura 2 - Modelo Beltrami-Klein. 15
Figura 3 - Modelos e Projeções. 16
Figura 4: Tesselações Regulares. 19
Figura 5: Tesselação Hiperbólica {3,7}. 20
Figura 6: Tesselação Hiperbólica {4,5}. 20
Figura 7: Tesselação Hiperbólica {infinito,3}. 21
Figura 8: Mapa. 22
Figura 9: Mapa ladrilhado. 22
Figura 10: Ladrilhos individuais. 22
Figura 11: Ladrilhamento Hexagonal. 23
Figura 12: Malha poligonal subdividida. 25
Figura 13: Malha poligonal projetada. 27
Figura 14: Mapa hiperbólico. 28
Figura 15: Projeção da malha no Disco de Poincaré. 29
Figura 16: Mapa de ladrilhos. 30
Figura 17: Par ladrilho textura. 30
Figura 18: Experimento 1, sobreposição. 31
Figura 19: Experimento 1, ajuste. 32
Figura 20: Experimento 1, malha subdividida. 33
Figura 21: Experimento 2, translação. 34
Figura 22: Experimento 2, rotação. 34
Figura 23: Experimento 2, limites do disco. 35
Figura 24: Experimento 2, deformações. 36

Sumário

Pág.

1. INTRODUÇÃO **8**
1.1 Motivação 9
1.2 Objetivos 9
1.3 Metodologia 10

2. FUNDAMENTOS DE GEOMETRIA **11**
2.1 Geometria Euclidiana 11
2.2 Geometria Hiperbólica 12
2.2.1 Plano Hiperbólico 12
2.2.2 Modelos do Plano Hiperbólico 13
2.3 Transformações Geométricas 16
2.3.1 Transformações Afins 17
2.3.2 Transformações Hiperbólicas 17
2.4 Tesselações 18
2.4.1 Tesselações Regulares 19
2.4.2 Tesselações Hiperbólicas 19
2.4.3. Ladrilhamento de Mapas 21

3. DESENVOLVIMENTO DA FERRAMENTA **24**
3.1 Ambiente Unity Engine 24
3.2 Etapas de Desenvolvimento 24

4. RESULTADOS EXPERIMENTAIS **31**
4.1 Experimento I 31
4.2 Experimento II 33

5. CONCLUSÕES **37**

REFERÊNCIAS **38**

APÊNDICE A. CÓDIGO FONTE **40**

1. INTRODUÇÃO

Tesselação, também denominada ladrilhamento, consiste em recobrir uma superfície com formas geométricas, chamadas ladrilhos, sem que haja sobreposições nem lacunas entre elas. Um exemplo simples e concreto de tesselação é a pavimentação de pisos, ou o revestimento de paredes, com azulejos.

Quando a tesselação é feita por computador, seu resultado é representado sob a forma de imagens 2D ou 3D. Assim, é preciso ter ferramentas específicas para a renderizar tais imagens. Basicamente, renderização é o processo de síntese de imagem, usado em computação gráfica, para produzir imagens 2D ou 3D. A renderização pode ser usada tanto para gerar imagens em tempo real como, por exemplo, em jogos eletrônicos, quanto para produzir imagens pré-renderizadas com melhor qualidade como, por exemplo, filmes e animações.

A tesselação de planos tem várias aplicações no campo virtual. Por exemplo, num problema de navegação de um agente virtual, uma tesselação do plano em que o agente se encontra é necessária para que o algoritmo A* possa encontrar um caminho de menor custo heurístico entre o ponto em que o agente se encontra e o ponto que ele deseja alcançar. De fato, essa tarefa de busca de caminhos é fundamental em diversas aplicações de inteligência artificial, especialmente aquelas envolvendo a movimentação de personagens em jogos de computador. Outro exemplo, que será abordado mais detalhadamente nesta monografia, é a técnica de mapeamento por ladrilho, que serve para construir mapas para ambientes virtuais, com maior facilidade e melhor desempenho, como Camper (2012) descreve:

> A lógica básica por trás do mapeamento por ladrilho não é diferente da lógica por trás de LEGO ou de outros brinquedos comuns em que se constrói grandes estruturas partindo de um conjunto pequeno de peças individuais. Em vez de necessitar alocar memória para cada pixel, o programador necessita apenas armazenar o endereço de memória de cada ladrilho: os ladrilhos são numerados, com cada valor de ladrilho mapeado (daí o nome) para um tipo único de ladrilho de 8x8 pixels. Para desenhá-los na tela, o hardware primeiro lê o número do ladrilho, e então procura pelo padrão de pixels correspondente à aquele ladrilho. Do ponto de vista de performance de hardware, a primeira vantagem do tile-map sobre o framebuffer é uma exponencial redução da quantidade de memória necessária para armazenar uma quantidade de informações da tela, e por consequência, uma igual redução da quantidade de poder de

> processamento e tempo necessários para fazer mudanças nessa mesma tela. (CAMPER, 2012, p. 173, tradução nossa).

Existe ainda a diferença entre tesselações no plano euclidiano, que é basicamente o plano com que estamos acostumados, onde os exemplos usuais se encontram, e o plano hiperbólico, que é regido pelas leis da geometria hiperbólica possuindo características únicas e quase inexploradas.

Neste contexto, esta monografia propõe uma ferramenta computacional capaz de renderizar tesselações específicas em um plano hiperbólico.

1.1 Motivação

A geometria hiperbólica possui uma série de aplicações e mesmo não percebendo, faz parte de nosso cotidiano, seja por meio da descrição de modelos magnéticos, seja pela sua aplicação na teoria da relatividade de Einstein, assumindo até mesmo que a forma do nosso espaço-tempo pode ser hiperbólica, mesmo assim, não temos uma percepção intuitiva sobre como ela funciona ou ainda de como ela se parece.

A falta de representações visuais sobre o espaço e plano hiperbólicos ocorre principalmente pela escassez de ferramentas capazes de renderizar em tempo real objetos em espaços ou planos curvos, entretanto, partindo do conhecimento sobre certas noções sobre geometria não-euclidiana e computação gráfica, é possível desenvolver esse tipo de ferramenta, principalmente no momento atual, em que as unidades centrais de processamento (CPU) e as unidades de processamento gráficos (GPU) mais modernas possuem plena capacidade de processar e paralelizar execuções de cálculos geométricos mais complexos.

1.2 Objetivos

O principal objetivo desse projeto é desenvolver uma ferramenta que seja capaz de renderizar e movimentar o plano hiperbólico a partir do uso da técnica de mapeamento por ladrilho, utilizando a tesselação {4, 5} do plano hiperbólico para tal.

Para atingir esse objetivo, no entanto, é necessário ser capaz primeiramente de conseguir representar malhas poligonais como se fossem polígonos hiperbólicos, ou seja, basicamente uma conversão entre polígonos euclidianos em hiperbólicos.

Também será necessário utilizar um ambiente virtual para a renderização em tempo real que seja de fácil utilização por qualquer usuário interessado.

1.3 Metodologia

A primeira parte do projeto é destinada ao desenvolvimento de uma técnica capaz de representar uma malha poligonal subdividida com formato de quadrado (euclidiano) em um polígono hiperbólico de 4 lados.

A técnica se baseia em utilizar a proximidade da representação de retas do modelo Beltrami-Klein com a representação euclidiana e, após uma série de projeções e normalizações, chegar à uma representação do polígono original no modelo do Disco de Poincaré.

A segunda parte do projeto consiste em desenvolver classes e métodos que abstraiam as transformações hiperbólicas para que possam ser aplicadas ao polígono descrito anteriormente, nos dando a capacidade de movimentar o objeto em consistência com o modelo em que está projetado.

Após ser capaz de criar ladrilhos poligonais e movimentá-los no plano hiperbólico, aplicaremos a técnica de mapeamento por ladrilho, criando mapas a partir de ladrilhos diferentes por meio de um algoritmo capaz de posicionar corretamente no plano hiperbólico cada ladrilho individual.

2. FUNDAMENTOS DE GEOMETRIA

Como esse projeto está intimamente relacionado com questões e diferenças atribuídas à geometria euclidiana e geometria hiperbólica, há a necessidade de primeiramente explicar e exemplificar os fundamentos matemáticos e geométricos que são a base desse projeto.

2.1 Geometria Euclidiana

É do senso comum atribuirmos à Geometria Euclidiana, a alcunha de ser a geometria que representa o nosso mundo seja em nível tridimensional (espaço) ou em nível bidimensional (plano). Sejam artes conceituais, plantas arquitetônicas, ou até mesmo jogos eletrônicos 2D ou 3D estão representados na geometria euclidiana. Entretanto, não é de conhecimento geral quais são as regras que constituem essa geometria, e esse é o primeiro ponto a ser abordado, pois para explicar o plano hiperbólico, é preciso primeiro saber quais são as bases da geometria euclidiana.

Segundo Santos (2011), por volta do ano 300 a.C., o matemático grego Euclides escreveu um tratado matemático geométrico compilado em 13 livros, chamado “Os Elementos”. Nesse tratado, Euclides descreveu precisamente as bases que sustentam a geometria a qual estudava, e Santos também enumera essas bases, que são os cinco postulados de Euclides:

> Postulado 1. Pode-se traçar uma (única) reta ligando quaisquer dois pontos.
> Postulado 2. Pode-se continuar (de uma única maneira) qualquer reta finita continuamente em uma reta.
> Postulado 3. Pode-se traçar um círculo com qualquer centro e com qualquer raio.
> Postulado 4. Todos os ângulos retos são iguais.
> Postulado 5. Sejam duas retas m e n cortadas por uma terceira reta r. Se a soma dos ângulos formados (ver figura) é menor do que 180 graus, então m e n não são paralelas. Além disso, elas se intersectam do lado dos ângulos cuja soma é menor do que 180 graus. (SANTOS, 2011, p. 15 - 18)

Ainda segundo Santos, com exceção dos 4 primeiros postulados de Euclides, que possuem uma natureza verbal axiomática, o quinto postulado possui uma aparência mais próxima de um teorema do que de um axioma, e Euclides teria

tardado a aceitar descrevê-lo como um postulado, tendo falhado em encontrar uma demonstração.

2.2 Geometria Hiperbólica

Mesmo após Euclides, as buscas pela prova do seu quinto postulado não se encerraram, tendo gerado muitas tentativas ao longo dos séculos que se seguiram, sem nunca trazer sucesso. Suspeitou-se então de que o quinto postulado fosse independente dos demais postulados, e portanto não poderia ser deduzido a partir dos outros quatro. Eves (2002) explica que a real independência do quinto postulado veio apenas com a demonstração da consistência de geometrias não-euclidianas, geradas a partir de negações do quinto postulado, tendo essas demonstrações sido produzidas pelos matemáticos: Beltrami, Gauss, Riemann, Arthur Cayley, Felix Klein, Henri Poincaré e outros. Assim, em 1871 Felix Klein batiza as geometrias geradas pelas demonstrações de Lobachevsky e de Riemann como: geometria hiperbólica e geometria elíptica respectivamente.

2.2.1 Plano Hiperbólico

Diferente do plano euclidiano, com o qual estamos acostumados, o plano hiperbólico, que está fundamentado na geometria hiperbólica plana, possui características particularmente distintas e de difícil visualização, como por exemplo: a soma dos ângulos de um triângulo é sempre menor que 180 graus, a razão entre o perímetro de uma circunferência e seu diâmetro é maior do que π (pi), entre outras características.

Entretanto, para melhor entendermos sobre o tema, é preciso apresentar antes a definição do plano hiperbólico. Segundo Holme (2010), temos como definição do plano hiperbólico:

> O plano hiperbólico é caracterizado por satisfazer todos os axiomas e postulados do plano euclidiano, exceto pelo quinto postulado, o qual é substituído pela afirmação:
> Dados uma linha e um ponto fora dessas linhas, então existem pelo menos duas linhas que passam pelo ponto e que não se cruzam com a linha (HOLME, 2010, p. 283, tradução nossa).

2.2.2 Modelos do Plano Hiperbólico

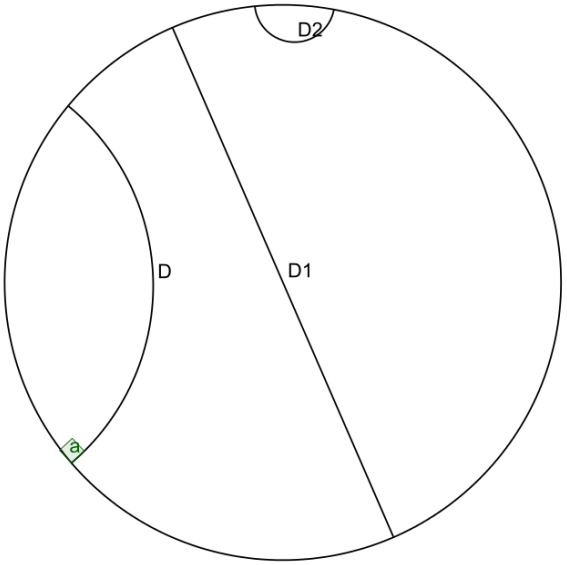
Embora tenhamos definido o plano hiperbólico, ainda necessitamos de maneiras para representá-lo, e o principal desafio nessa representação é o fato de que os objetos que utilizamos são construções geométricas euclidianas, como por exemplo: a tela do computador, ou ainda uma folha de papel. Para fazer com que representações de geometrias não-euclidianas pudessem ser visualizadas em ambiente euclidiano, e assim ser capaz de fazer construções e demonstrações geométricas, desenvolveram-se alguns modelos de representação.

Para representações do plano hiperbólico, o modelo mais famoso e comumente utilizado é o Modelo do Disco de Poincaré, que segundo Weisstein (2021?) é descrito da seguinte forma:

> O disco de Poincaré é um modelo para a geometria hiperbólica em que uma linha é representada como um arco de uma circunferência cujos limites são perpendiculares ao limite do disco. Dois arcos que não se cruzam, correspondem à retas paralelas, arcos que se cruzam ortogonalmente correspondem à retas perpendiculares, e arcos que cruzam no limite do disco são um par de retas limite. (WEISSTEIN, 2021?, tradução nossa).

Utilizando o Disco de Poincaré facilmente conseguimos verificar a estranheza causada pela geometria hiperbólica. Na Figura 1, podemos observar 3 retas paralelas: D, D1 e D2, ainda podemos verificar que o ângulo entre uma das extremidades de D com o limite do disco, é um ângulo reto.

Figura 1: Disco de Poincaré



Fonte: https://www.wikiwand.com/en/Poincar%C3%A9_disk_model

A representação do disco de Poincaré tem como principais propriedades a não distorção de ângulos e circunferências, mas em contrapartida, utiliza arcos para representar as retas.

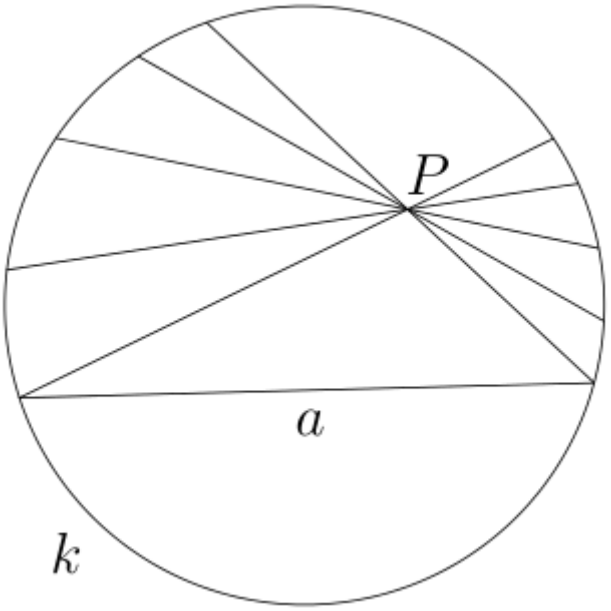
Existe ainda um outro modelo de disco, cuja representação contrasta com o disco de Poincaré justamente por representar retas como cordas e distorcer circunferências e ângulos. Trata-se do modelo projetivo, também conhecido como Modelo Beltrami-Klein ou Modelo Klein-Beltrami, que segundo Weisstein pode ser descrito da seguinte maneira:

> O modelo Klein-Beltrami da geometria hiperbólica consiste em um disco aberto no plano euclidiano em que cordas abertas correspondem a retas hiperbólicas. Duas linhas ***l*** e ***m*** são consideradas paralelas se suas cordas falham em se intersectar e são perpendiculares se satisfazerem as seguintes condições:
> 1. Se ao menos uma das duas linhas, ***l*** ou ***m*** for um diâmetro do disco, elas serão perpendiculares se e somente se forem perpendiculares na concepção euclidiana.
> 2. Se nenhuma for um diâmetro do disco, ***l*** é perpendicular a ***m*** se e somente se a extensão euclidiana da linha ***l*** cruzar o polo de ***m*** (definido como ponto de intersecção das tangentes do disco nas extremidades de ***m***). (WEISSTEIN, 2021?, tradução nossa).

Na Figura 2 podemos observar uma representação visual da negação do quinto postulado de euclides no modelo de disco de Beltrami-Klein. Onde pelo ponto

P passam inúmeras retas que são paralelas à reta a.

Figura 2: Modelo Beltrami-Klein



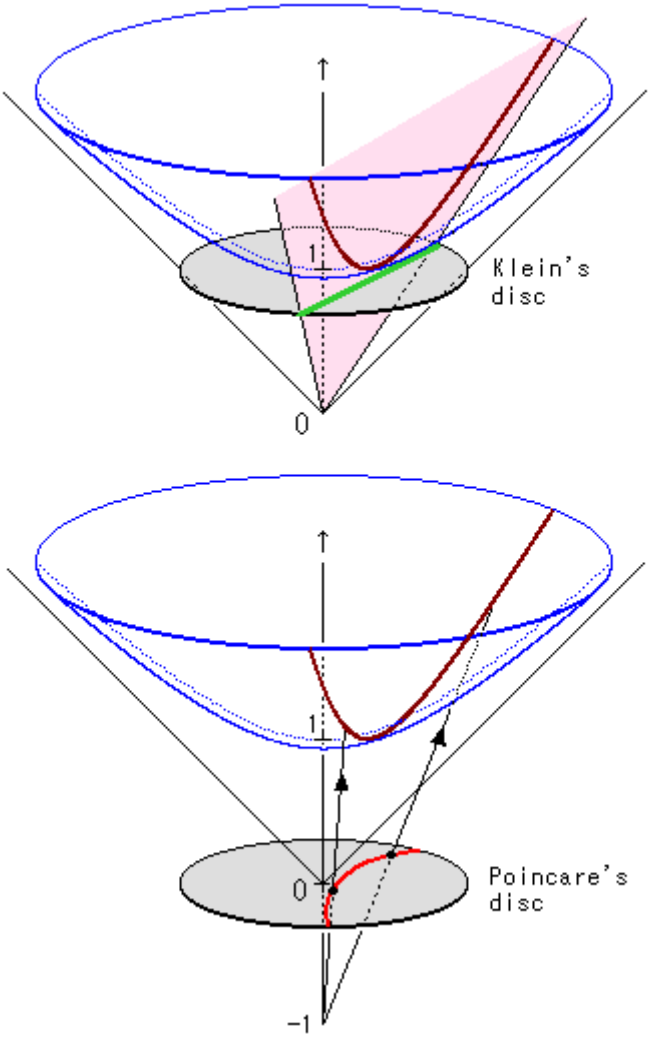
Fonte: https://www.wikiwand.com/en/Beltrami%E2%80%93Klein_model

Ainda existe um outro modelo para o plano hiperbólico, mas que não se trata de uma representação bidimensional em formato de disco, e sim tridimensional, que é o Modelo do Hiperbolóide, também conhecido como Modelo do Hiperbolóide de Minkowski ou Modelo do Hiperbolóide de Lorentz, que surge da ideia de usar a folha anterior de um hiperbolóide para representar todo o plano hiperbólico. Solheim (2012) define o modelo hiperbolóide da seguinte forma:

> O modelo final a que estamos chegando veio da física. Aqui, movemos o disco de Klein para o plano (z, t) = (z, 1) então o centro do disco é agora (0, 1) e pela origem nós projetamos o disco de Klein sobre o hemisfério superior de um hiperbolóide de duas folhas. (SOLHEIM, 2012, p. 126, tradução nossa).

Todos os três modelos apresentados possuem equivalência, e podem ser convertidos entre si através de projeções. Na Figura 3 podemos observar que tanto o modelo de disco de Beltrami-Klein quanto o disco de Poincaré podem ser projetados no hiperbolóide, assim como podem receber a projeção do hiperbolóide.

Figura 3: Modelos e Projeções



Fonte: http://web1.kcn.jp/hp28ah77/us3_poinc.htm

2.3 Transformações Geométricas

Todo movimento realizado por algum objeto no espaço euclidiano pode ser descrito pela sua translação, rotação e escala. É baseado nesses três pilares que, hoje em dia, funciona qualquer representação gráfica tridimensional digital. Esses pilares são denominados transformações, que consistem em transformações lineares que podem ser representadas por meio de operações envolvendo matrizes e vetores.

2.3.1 Transformações Afins

As transformações afins são um conjunto de transformações lineares que correspondem a movimentações que podem ser aplicadas a um ponto ou polígono, desde que esteja se utilizando de coordenadas homogêneas. Ao representar essas transformações em formato matricial, pode-se encontrar o ponto resultante da transformação simplesmente multiplicando a matriz pelo ponto inicial.

House (2017) e Keyser (2017) descrevem algumas dessas matrizes: a matriz (1) descrita abaixo corresponde à transformação de translação, ou seja, deslocamento de ponto inicial com uma variação distinta para cada eixo. A matriz (2) corresponde à mudança de escala em cada um dos eixos, e por fim a matriz (3) corresponde à rotação em torno do eixo z.

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2)$$

$$\begin{bmatrix} cos\Theta_z & -sen\Theta_z & 0 & 0 \\ sen\Theta_z & cos\Theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3)$$

2.3.2 Transformações Hiperbólicas

Como visto na Seção 2.3.1, transformações podem ser representadas a partir de matrizes e ao multiplicá-las por um ponto inicial (sob representação vetorial), temos como resultado um ponto final. Tais transformações são utilizadas para movimentar um ponto no espaço euclidiano tridimensional, para este projeto, no entanto, o foco é conseguir realizar transformações no plano hiperbólico.

Para compreendermos melhor como podemos realizar transformações no plano hiperbólico, podemos imaginar o seguinte cenário: um paralelo fácil de compreender seria a movimentação de um avião sobre o globo terrestre, ao realizar uma viagem de avião, estamos basicamente rotacionando um ponto sobre a superfície do globo terrestre tendo como origem da rotação o centro do globo, ou seja, embora na prática, se trate de uma translação, tal transformação pode ser descrita como uma rotação em um plano esférico. Da mesma forma, translações no plano hiperbólico podem ser descritas como rotações sobre o hiperbolóide de Minkowski.

As matrizes (4) (5) e (6) representam as rotações sobre o hiperbolóide de Minkowski.

$$\begin{bmatrix} cos\alpha & -sen\alpha & 0 \\ sen\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4)$$

$$\begin{bmatrix} cosh\beta & 0 & senh\beta \\ 0 & 1 & 0 \\ senh\beta & 0 & cosh\beta \end{bmatrix} \quad (5)$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cosh\gamma & senh\gamma \\ 0 & senh\gamma & cosh\gamma \end{bmatrix} \quad (6)$$

É perceptível a semelhança entre a matriz de rotação (4) e a matriz de rotação (3), isso ocorre pelo fato do hiperbolóide possuir simetria de rotação em torno do eixo z, o que significa que uma rotação do hiperbolóide em torno do eixo z pode ser descrita como uma rotação euclidiana comum.

2.4 Tesselações

Tesselações nada mais são do que a divisão de uma superfície em polígonos sem que existam lacunas ou sobreposições, assim como um mosaico, em que cada

pedra representa uma pequena fração da superfície do mosaico. Segundo Shephard (1986) uma definição formal para tesselações é:

> Uma tesselação planar T é uma família enumerada de conjuntos fechados T = {T1, T2, … } que cobrem o plano sem lacunas ou sobreposições. Mais explicitamente, a união dos conjuntos T1, T2, … (que são conhecidos como ladrilhos de T) deve ser o plano todo. (SHEPHARD, 1986, p. 16, tradução nossa).

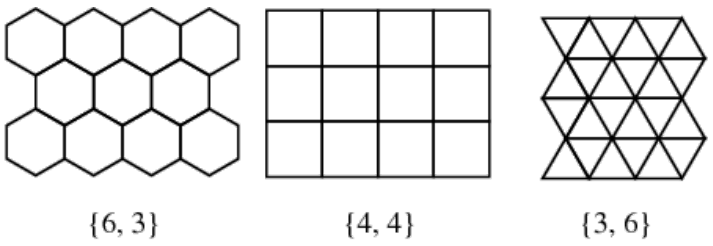
2.4.1 Tesselações Regulares

Tesselações regulares são, segundo Tacoronte (2021), tesselações formadas por polígonos regulares e congruentes entre si, existe ainda uma notação para classificar essas tesselações, Tacoronte explica:

> Dada uma tesselação regular, podemos representá-la por $\{p, q\}$, sendo q a quantidade de polígonos regulares de p lados que estão ao redor de cada vértice ou nó.

Podemos observar alguns exemplos então na Figura 4, onde temos três tesselações regulares: {6,3}, {4,4} e {3,6}, da esquerda para a direita respectivamente.

Figura 4: Tesselações Regulares



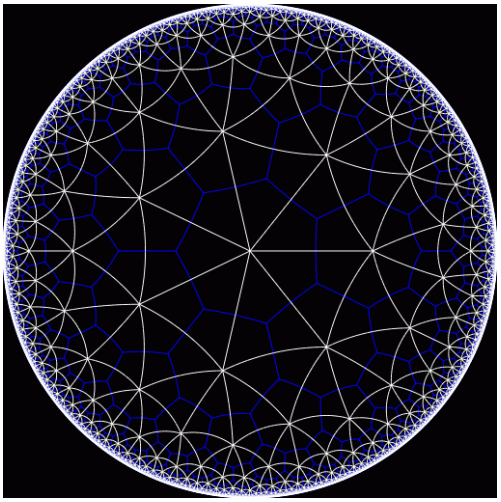
Fonte: https://mathworld.wolfram.com/Tessellation.html

2.4.2 Tesselações Hiperbólicas

Assim como o plano euclidiano pode ser tesselado, como vimos anteriormente, existem tesselações que podem ser feitas sobre superfícies curvas, e também são classificadas utilizando a mesma notação que se utiliza para tesselações regulares no geral.

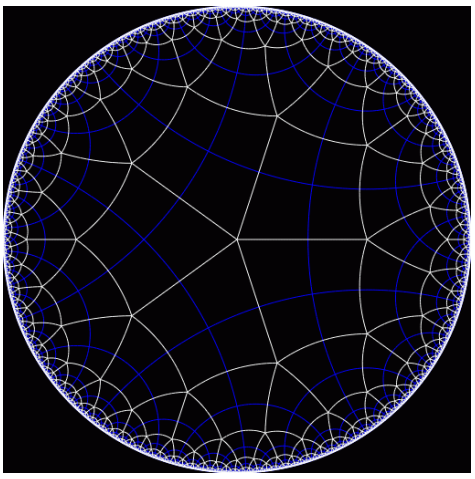
Existe também uma inequação que descreve a possibilidade da existência de qualquer tesselação do plano hiperbólico. Segundo Hatch (2002), a inequação é descrita por: (p - 2) . (q - 2) > 4. Para os valores de p e q que a satisfazem, significa que existe uma tesselação do plano hiperbólico para esses valores, como podemos observar na Figura 5, a tesselação {3,7} que resulta em 5 > 4, na Figura 6, a tesselação {4,5} que resulta em 6 > 4 e na Figura 7, onde a tesselação {infinito, 3} que resulta em infinito > 4.

Figura 5: Tesselação Hiperbólica {3,7}



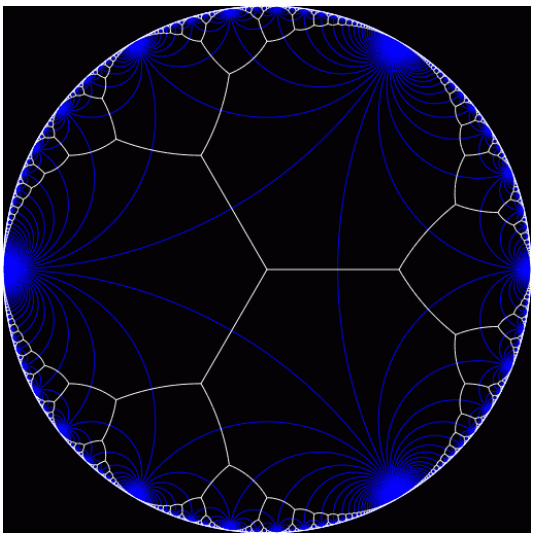
Fonte: http://www.plunk.org/~hatch/HyperbolicTesselations

Figura 6: Tesselação Hiperbólica {4,5}



Fonte: http://www.plunk.org/~hatch/HyperbolicTesselations

Figura 7: Tesselação Hiperbólica {infinito,3}



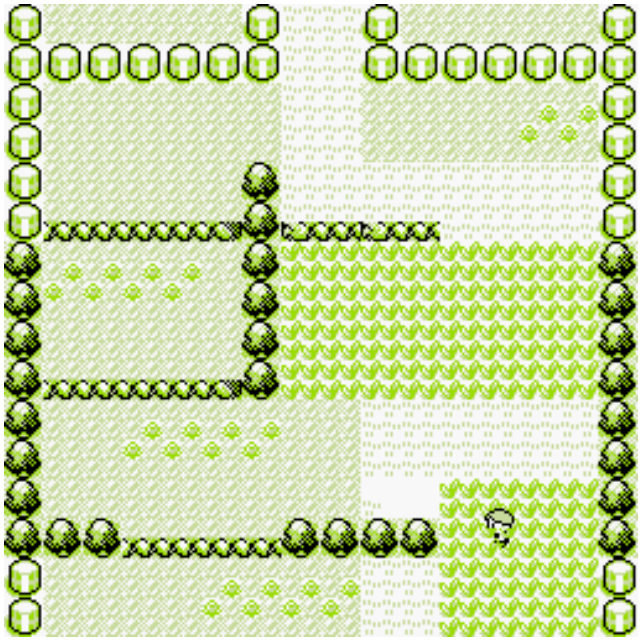
Fonte: http://www.plunk.org/~hatch/HyperbolicTesselations

2.4.3. Ladrilhamento de Mapas

O ladrilhamento é uma técnica utilizada na computação que está intimamente relacionada às tesselações regulares. Essa técnica consiste em: a partir de um conjunto limitado de tipos de ladrilhos formados pelo mesmo polígono, preencher um pedaço limitado do plano a partir desses ladrilhos, dessa forma, pode-se gerar mapas distintos e de grande extensão a partir do mesmo conjunto de ladrilhos.

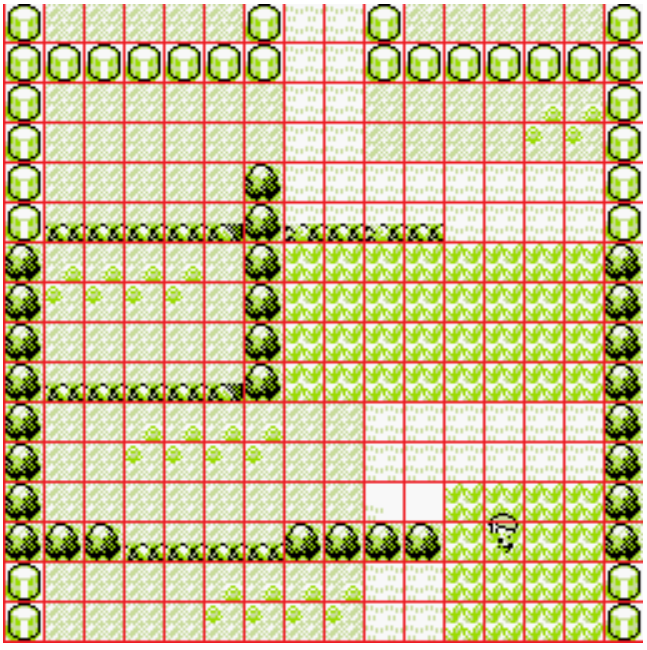
Essa técnica é amplamente utilizada na criação de jogos digitais 2D, e permite que se poupe memória por utilizar referências aos ladrilhos. A Figura 8 corresponde a uma pequena parte de um mapa ladrilhado utilizado no jogo Pokémon Red (1996). Já a Figura 9 nos mostra como esse mapa está representado sobre uma tesselação {4,4} e por fim a Figura 10 nos mostra alguns dos ladrilhos fundamentais que compõem o mapa em questão.

Figura 8: Mapa



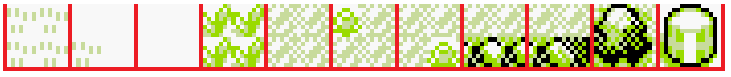
Fonte: Autor (2021)

Figura 9: Mapa ladrilhado



Fonte: Autor (2021)

Figura 10: Ladrilhos individuais



Fonte: Autor (2021)

A mesma técnica pode ser aplicada a outras tesselações regulares do plano. O jogo Sid Meier's Civilization 5 (2010) por exemplo se utiliza de ladrilhos hexagonais, ou seja, tesselação {6,3} como podemos ver na Figura 11.

Figura 11: Ladrilhamento Hexagonal

Fonte: https://civilization.fandom.com/pt-br/wiki/Civilization_V

Partindo desses princípios, tanto de como funcionam as tesselações regulares, quanto em como funciona o ladrilhamento de mapas, podemos inferir que é possível criar mapas hiperbólicos ladrilhados, esse que é um dos objetivos desse projeto, para o qual será utilizada a tesselação hiperbólica {4,5} que já foi apresentada na Figura 6.

3. DESENVOLVIMENTO DA FERRAMENTA

A ferramenta desenvolvida é uma extensão em formato de pacote de customização para o motor gráfico Unity Engine, compatível com a versão 2019.4.28f1 e superiores, com código fonte e binários publicados de forma gratuita no GitHub.

3.1 Ambiente Unity Engine

Unity Engine (Unity Technologies, 2019) é não só um motor gráfico, como também um dos ambientes de desenvolvimento de software customizado mais utilizados no mercado atualmente, oferece licenças gratuitas e suporte para criação de aplicações mobile (tanto Android quanto iOS), realidade aumentada, realidade virtual, sendo capaz de compilar executáveis para Windows, MacOS, Linux, além dos consoles mais populares atualmente: Nintendo Switch, Xbox One e PlayStation 4. Além da versatilidade de sistemas compatíveis, o Unity Engine ainda possui um conjunto de ferramentas e APIs de fácil utilização para criar extensões do seu editor.

Motores gráficos são softwares especializados em renderização em tempo real, fazendo a comunicação entre a aplicação e a Unidade de Processamento Gráfico (GPU), além de detectar inputs de teclado, mouse, joysticks, a depender da plataforma para qual a aplicação será compilada. Com relação à APIs gráficas, o Unity atualmente conta com suporte a DirectX, OpenGL, Vulkan e Metal, sendo capaz de compilar também seus respectivos sombreadores, que são programas capazes de processar paralelamente vértices para obter efeitos geométricos desejados e pixels normalmente para cálculo de iluminação.

O projeto aqui apresentado se trata de uma ferramenta de extensão para o editor do Unity Engine, que o torna capaz de renderizar e movimentar representações tridimensionais do plano hiperbólico, podendo ser posteriormente compilado para qualquer uma das plataformas citadas.

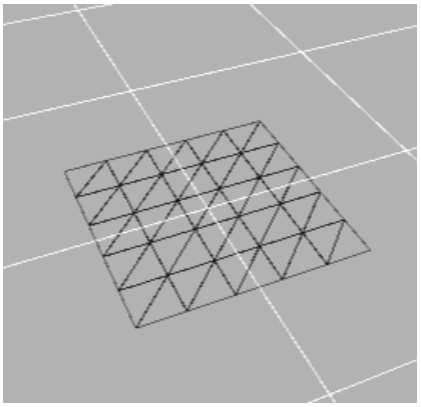
3.2 Etapas de Desenvolvimento

Atualmente, o Unity Engine possui suporte apenas para desenvolvimento de aplicações utilizando a linguagem C#, e também possui suporte para utilização das

linguagens de sombreadores: HLSL, GLSL e Nvidia CG. Esse projeto então foi desenvolvido utilizando-se da linguagem C# e Nvidia CG. A escolha da linguagem do sombreador foi feita unicamente por ter mais familiaridade, mas essas linguagens possuem sintaxes muito próximas, sendo todas elas baseadas em C.

A primeira etapa do projeto trata da conversão de uma malha poligonal de um quadrado subdividido para ser representado nos modelos do plano hiperbólico, ou seja, não estamos tratando de representar objetos euclidianos em um plano hiperbólico, e sim assumindo que esse polígono pertence já ao plano hiperbólico. A malha poligonal foi gerada utilizando o software Blender (Blender Foundation, 2021), e é apenas um quadrado subdividido como podemos ver na Figura 12, salvo em um arquivo no formato Filmbox (extensão .fbx).

Figura 12: Malha poligonal subdividida



Fonte: Autor (2021)

Uma malha poligonal é um objeto que representa um modelo tridimensional de algum objeto. Essa representação é obtida através do armazenamento da posição de todos os vértices dos polígonos que formam a malha, assim como quais são as formas geométricas primitivas que geram a figura final. Tais primitivas podem ser pontos, linhas, triângulos ou quadriláteros, sendo que na computação gráfica a melhor forma de representar um objeto do mundo real é através da subdivisão em triângulos, como pode ser observado na Figura 12.

Uma vez que possuímos a malha poligonal, temos acesso a cada um de seus vértices, e podemos ler e alterar suas posições. Como estamos assumindo que esse quadrilátero representado pela malha poligonal é um objeto do plano hiperbólico, podemos assumir que ele está representado no modelo Beltrami-Klein,

que é o modelo em que as retas não são distorcidas, sendo assim uma representação de um quadrado hiperbólico. Entretanto, como vimos anteriormente, para podermos utilizar as transformações hiperbólicas, precisamos trabalhar sobre o modelo do hiperbolóide de Minkowski e, portanto, precisamos de um algoritmo que faça a conversão entre os modelos. Como os modelos de Beltrami-Klein e o disco de Poincaré são projeções simples do hiperbolóide de Minkowski, podemos obter a conversão entre eles utilizando apenas a equação da reta no espaço. O método que descreve a conversão de um ponto no espaço vetorial representado no modelo Beltrami-Klein para o modelo do hiperbolóide de Minkowski é escrito da seguinte forma, onde VectorD3 é uma estrutura de dados que representa um ponto no espaço vetorial composto por 3 doubles:

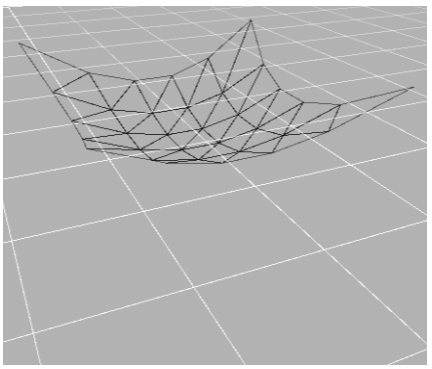
```
public static VectorD3 BeltramiKleinToMinkowski(VectorD3 P)
{
     double factor = MathD.Sqrt(1.0 / (1.0 - ((P.x * P.x) + (P.y * P.y))));
     double x1 = P.x * factor;
     double y1 = P.y * factor;
     double z1 = factor;
     return new VectorD3(x1, y1, z1);
}
```

Utilizando desse método, a conversão de uma malha se dá simplesmente por um loop que percorre todos os seus vértices e altera suas respectivas posições:

```
  Mesh mesh = meshObject.GetComponent<MeshFilter>().mesh;
  Vector3[] vertices = mesh.vertices;

  for (int i = 0; i < vertices.Length; i++)
  {
      // Obtém a posição global do vértice através da posição local
      Vector3 worldV = LocalToWorld(transformReference,
mesh.vertices[i],meshObject.transform.localScale);
      Vector3 worldVUp = new Vector3(worldV.x, worldV.z, worldV.y);
      VectorD3 worldVCartesian = Math.Math.NormalizeToCartesianCoordinates(worldVUp);

      //Converte o ponto do modelo de Klein para Minkowski
      VectorD3 result = Math.Math.BeltramiKleinToMinkowski(worldVCartesian);
      Vector3 res = result.ToUnity();
      vertices[i] = WorldToLocal(transformReference, new Vector3(res.x, res.z, res.y),
meshObject.transform.localScale);
  }
  mesh.vertices = vertices;
```

Como resultado, a mesma malha apresentada na Figura 11 agora fica com a aparência apresentada na Figura 13.

Figura 13: Malha poligonal projetada



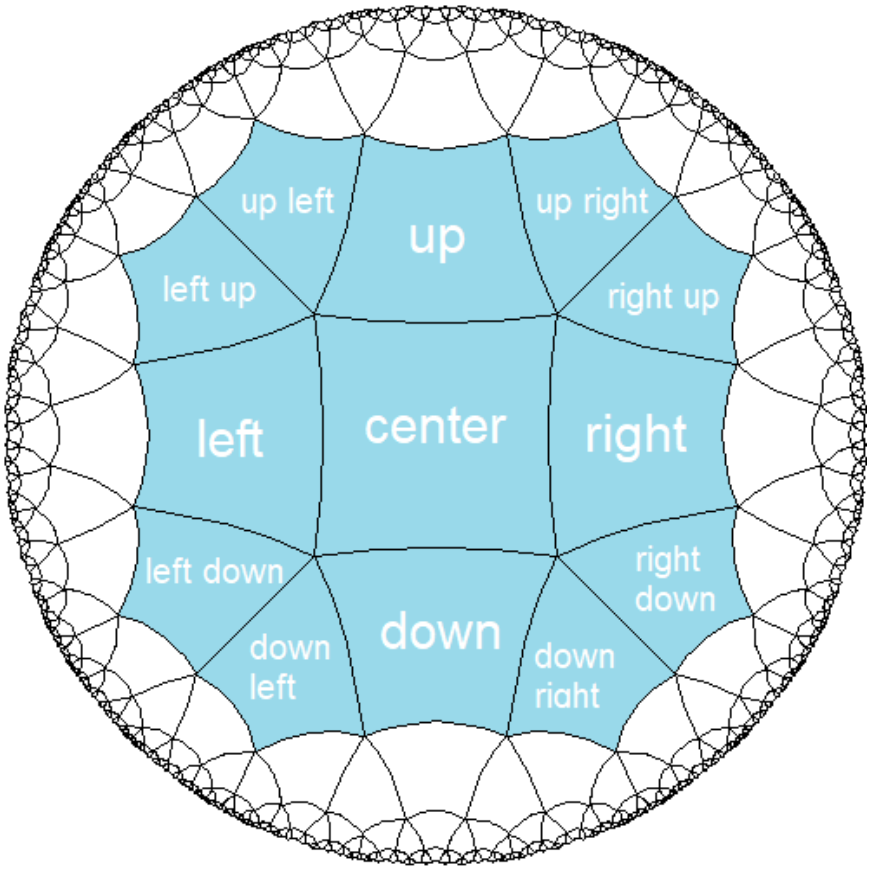
Fonte: Autor (2021)

A próxima etapa do desenvolvimento é utilizar malhas como ladrilhos individuais de uma tesselação {4,5}, e para isso é necessário que a malha poligonal apresentada na Figura 12 tenha as dimensões correspondentes à essa tesselação para não sobrepor nenhum outro ladrilho ou deixar lacunas entre os ladrilhos. Utilizando a malha poligonal em questão e representado-a por um quadrado, há como se saber qual o tamanho necessário da diagonal desse quadrado, assim como o valor do seu lado, Dunham (1986) define esse cálculo como o cálculo da região fundamental. O trecho seguinte de código é a implementação desse cálculo:

```
public static VectorD3[] HypotenuseOfFundamentalRegionEndPoint(int p, int q)
{
    double cosh2 = Cot(MathD.PI / (double) p) * Cot(MathD.PI / (double) q);
    double sinh2 = MathD.Sqrt(cosh2 * cosh2 - 1);

    double coshq = MathD.Cos(MathD.PI / (double) q) / MathD.Cos(MathD.PI / (double) p);
    double sinhq = MathD.Sqrt(coshq * coshq - 1);

    double rad2 = sinh2 / ( cosh2 + 1 );
    double x2pt = sinhq / ( coshq + 1 );

    double xqpt = MathD.Cos( MathD.PI / (double) p ) * rad2;
    double yqpt = MathD.Sin( MathD.PI / (double) p ) * rad2;

    return new VectorD3[]{new VectorD3(xqpt, yqpt, 0f), new VectorD3(x2pt, 0f, 0f)};
}
```

Tomando então cada malha poligonal como sendo um ladrilho do plano hiperbólico, a próxima etapa é criar um mapa a partir desses ladrilhos, posicionando corretamente cada um deles. O mapa proposto é composto por apenas 13 ladrilhos, sendo um ladrilho central cercado pelos outros 12 ladrilhos vizinhos, como pode ser observado na Figura 14.

Figura 14: Mapa hiperbólico



Fonte: Autor (2021)

Para gerar esse mapa então, é preciso gerar a malha poligonal central e movimentá-la para suas posições utilizando das transformações hiperbólicas já apresentadas.

Para fazer essa movimentação sem resultar em sobreposição das malhas ou lacunas entre elas, apenas transladamos esses ladrilhos utilizando os valores encontrados no trecho de código anterior, ou seja, para posicionar um ladrilho à direita do ladrilho central, basta gerar um ladrilho central e movimentá-lo para a direita na quantidade equivalente ao valor de seu lado, e assim sucessivamente para todos os outros ladrilhos.

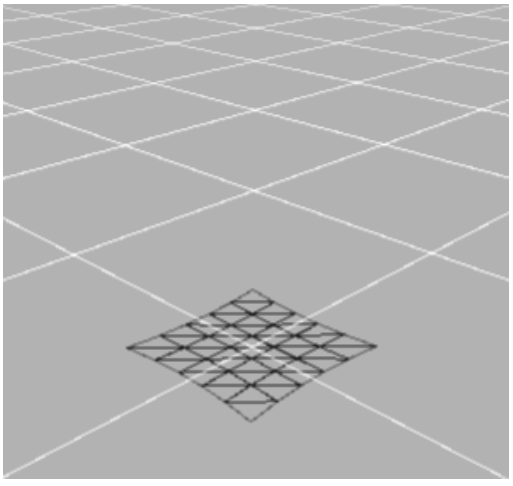
A próxima etapa do projeto então trata da projeção desse mapa para o modelo do disco de Poincaré, pois ainda estamos utilizando o modelo do Hiperbolóide de Minkowski que possui uma visualização menos intuitiva de um plano, e sua vantagem maior para o projeto é apenas a possibilidade de aplicar as transformações hiperbólicas para movimentação.

Para evitar de criar mais um loop que percorre todos os vértices para realizar a projeção, usamos então um sombreador de vértices, que calcula paralelamente e diretamente na Unidade de Processamento Gráfico uma nova posição para cada vértice. O trecho de código a seguir implementa essa projeção do hiperbolóide de Minkowski sobre o disco de Poincaré utilizando a linguagem do sombreador, ou seja, Nvidia CG.

```
v2f vert (appdata v)
{
    v2f o;
    float4x4 customModel;
    customModel = unity_ObjectToWorld;
    float4 Pe = mul(customModel, v.vertex);
    float3 P0 = float3(_P0.x, _P0.y - _Height, _P0.z);
    float interHeight = sqrt(pow(Pe.x, 2) + pow(Pe.z, 2) + 1);
    float3 Ps = float3(Pe.x, interHeight, Pe.z);
    float3 V = Pe - P0;
    float t = P0.y/V.y;
    float pX = P0.x + (t * V.x);
    float pZ = P0.z + (t * V.z);
    float3 projectedP = float3(pX, 0.0, pZ);
    float4 resultVertex = mul(unity_WorldToObject, projectedP);
    o.vertex = UnityObjectToClipPos(resultVertex);

    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}
```

A aplicação desse sombreador resulta então na re-projeção da malha apresentada na Figura 13, no modelo do disco de Poincaré como podemos ver na Figura 15.

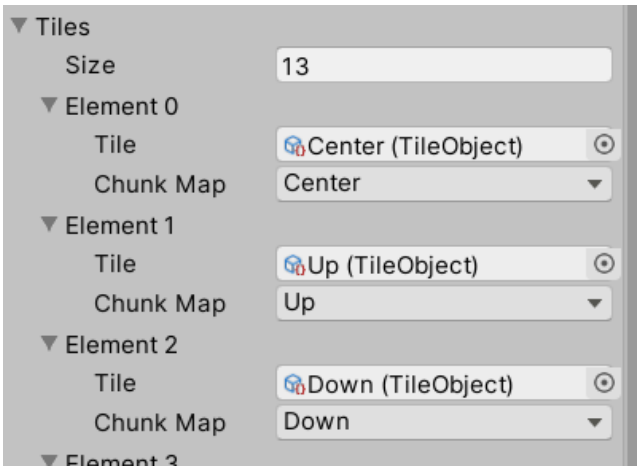
Figura 15: Projeção da malha no Disco de Poincaré



Fonte: Autor (2021)

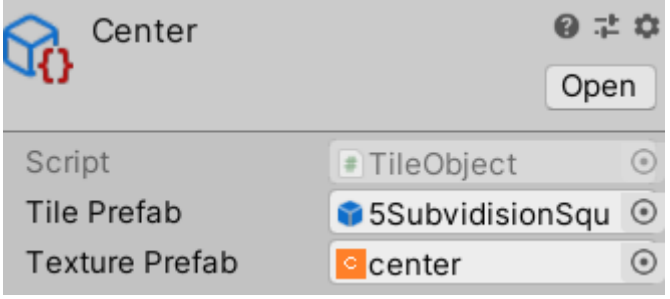
Como última etapa então, ainda falta a implementação do ladrilhamento de mapas. Como estamos utilizando mapas de apenas 13 ladrilhos, armazenamos todos os ladrilhos em uma lista de 13 ladrilhos, cada ladrilho nada mais é do que uma malha poligonal com sua textura respectiva, também contendo a posição relativa ao ladrilho central, podendo ser repetida dentro do mapa. No ambiente da Unity Engine, listas podem ser armazenadas como arquivos editáveis por meio de um artifício chamado Scriptable Objects, que segundo sua documentação, são contêineres de dados que você pode usar para guardar grandes quantidades de dados independente das instâncias de classe. Tanto o par (ladrilho, textura) quanto a lista de ladrilhos que corresponde ao mapa são armazenados como Scriptable Objects. A Figura 16 nos mostra como o mapa fica disponível no editor para ser editado, enquanto que a Figura 17 mostra como o par (ladrilho, textura) fica disponível no editor.

Figura 16: Mapa de ladrilhos



Fonte: Autor (2021)

Figura 17: Par ladrilho textura



Fonte: Autor (2021)

4. RESULTADOS EXPERIMENTAIS

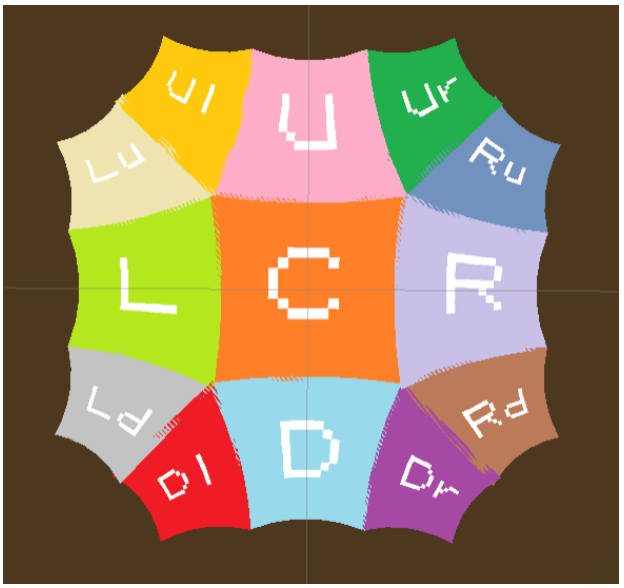
No decorrer do desenvolvimento da ferramenta, alguns testes foram aplicados com o objetivo de verificar se a ferramenta está a produzir resultados coerentes com sua proposta. Este capítulo expõe tais experimentos, bem como seus resultados.

4.1 Experimento I

Esse primeiro experimento tem como objetivo analisar como a organização do mapa ladrilhado através do posicionamento dos ladrilhos individuais se comporta, se há sobreposições ou lacunas e se esses ladrilhos estão sendo posicionados de forma coerente com o que foi proposto. Para melhor visualização, cada par (ladrilho, textura) foi anotado com uma abreviatura indicando sua posição (tais abreviaturas são formadas a partir das letras iniciais das palavras Center, Left, Right, Up and Down, de modo similar ao que é observado na Figura 14).

O resultado obtido em primeiro momento pode ser observado na Figura 18.

Figura 18: Experimento 1, sobreposição



Fonte: Autor (2021)

Podemos observar na Figura 18 que os ladrilhos foram posicionados corretamente, ou seja, a movimentação dos ladrilhos a partir do ladrilho central se

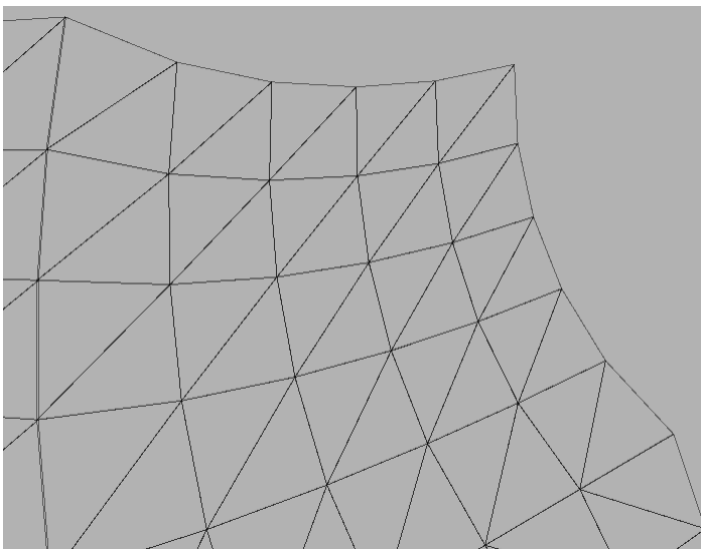
comportou de maneira esperada. Entretanto, podemos observar também que existem pequenos trechos de sobreposição entre os ladrilhos. A origem dessas sobreposições pode estar relacionada com a propagação de erros proveniente da escala dos ladrilhos ou mesmo proveniente do valor de deslocamento, ambos oriundos da implementação do algoritmo de cálculo da região fundamental de Dunham. Entretanto, como essas sobreposições aparentam descrever um erro sistemático, um ajuste simples é suficiente para trazer o efeito desejado, como podemos ver na Figura 19.

Figura 19: Experimento 1, ajuste

Fonte: Autor (2021)

Caso nos aproximemos muito, ainda é possível identificar pequenos trechos em que existem sobreposições ou lacunas. Entretanto, esses trechos são menos perceptíveis do que as próprias imperfeições da malha, pois a mesma possui poucas subdivisões e, portanto, em vez de descrever um arco perfeito como previsto pelo disco de Poincaré, descreve pequenos segmentos de reta que se aproximam do formato de um arco, como podemos ver na Figura 20.

Figura 20: Experimento 1, malha subdividida



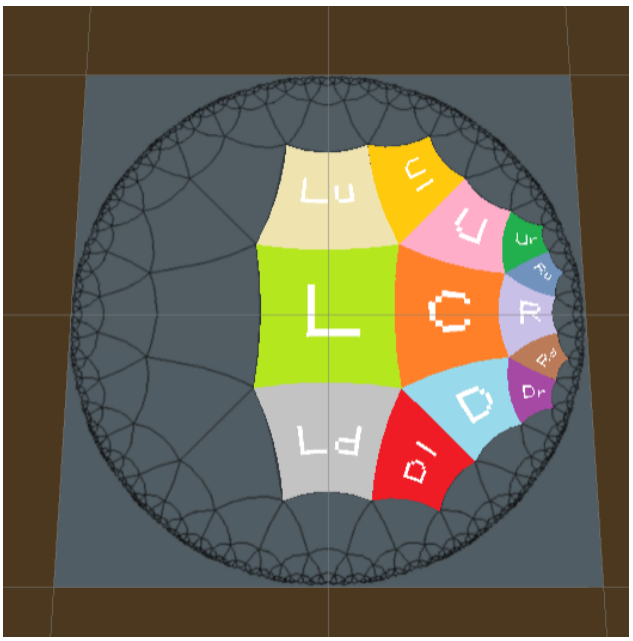
Fonte: Autor (2021)

4.2 Experimento II

Esse segundo experimento tem como objetivo analisar como é possível movimentar o mapa, se essa movimentação produz efeitos indesejados como deformações adicionais, se é possível realizar translação e rotação mantendo a coerência, e se em algum momento o mapa ultrapassa os limites do disco de Poincaré. Para realização desse experimento, além das texturas atribuídas a cada ladrilho como foi realizado no experimento anterior, também foi utilizada uma imagem de referência representando o disco de Poincaré assim como sua tesselação {4,5}.

O resultado obtido ao transladar todo o mapa pode ser observado na Figura 21.

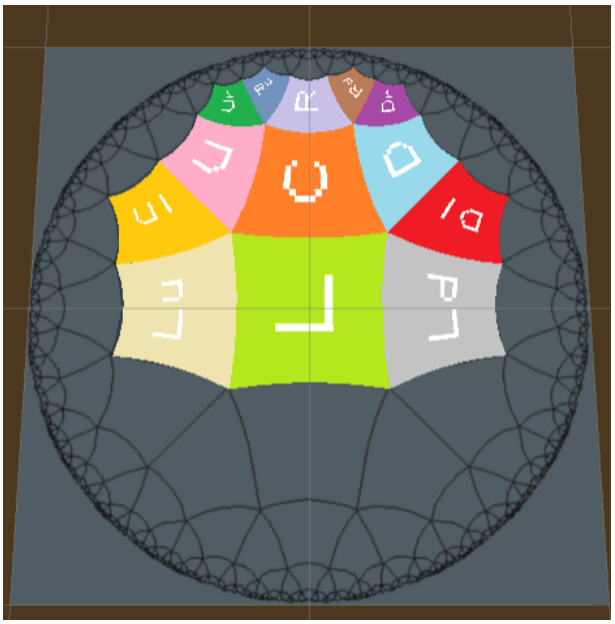
Figura 21: Experimento 2, translação



Fonte: Autor (2021)

Como podemos observar, a translação do mapa é bastante coerente com relação à imagem de referência, o que confirma a consistência desse movimento com o modelo em que está projetado. Partindo do mesmo estado, ao rotacionar no sentido anti-horário, obtemos o resultado apresentado pela Figura 22.

Figura 22: Experimento 2, rotação

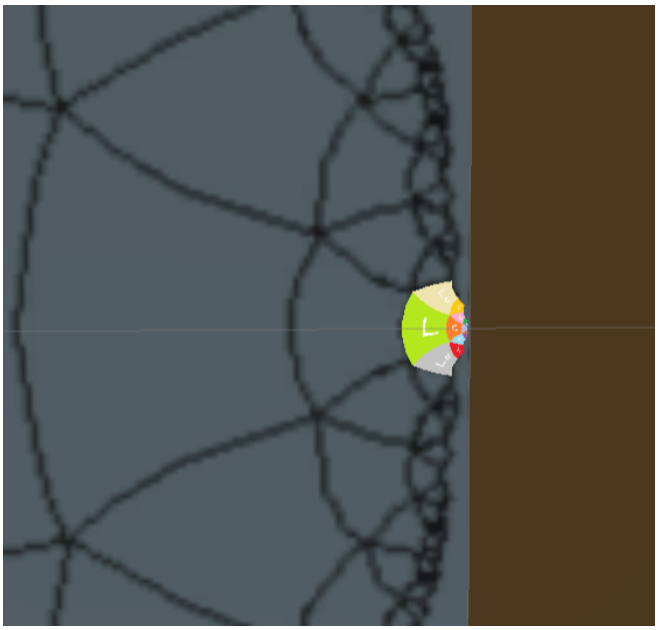


Fonte: Autor (2021)

Observamos então que também há coerência do movimento de rotação com relação à imagem de referência, confirmando também a consistência dessa transformação quando aplicada a todo o mapa.

Após grandes deslocamentos, ainda podemos observar que o mapa permanece dentro dos limites do disco de Poincaré, como podemos ver na Figura 23.

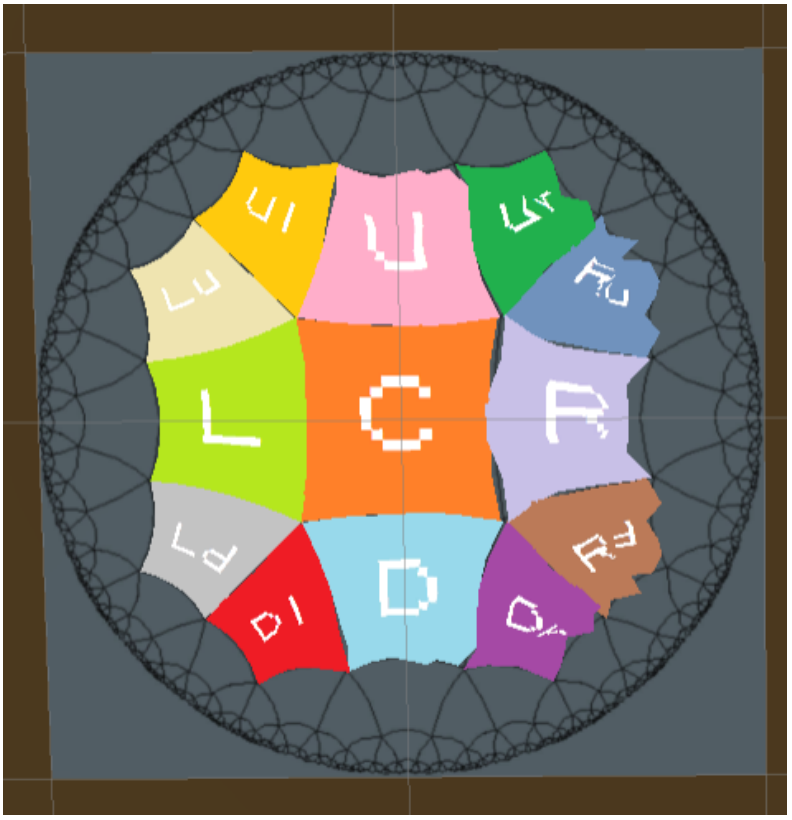
Figura 23: Experimento 2, limites do disco



Fonte: Autor (2021)

A partir do estado anterior, apresentado na Figura 23, deslocamos o mapa novamente para o centro de nossa visualização, e o resultado obtido é apresentado na Figura 24.

Figura 24: Experimento 2, deformações



Fonte: Autor (2021)

Podemos observar então um resultado indesejado, que é a deformação das malhas que constituem o mapa após realizar grandes deslocamentos. Podemos ainda observar que os ladrilhos mais a direita foram mais afetados. A origem dessas deformações pode estar associada ao fato de que a borda do disco de Poincaré representa o infinito, portanto, quanto mais próximo um vértice está da borda, mais amplificados serão os erros de arredondamento de ponto flutuante.

Uma solução imediata é evitar grandes deslocamentos, entretanto, a solução ideal seria produzir um algoritmo de correção.

5. CONCLUSÕES

A tesselação (cobertura de uma superfície com formas geométricas, sem que haja sobreposições nem lacunas entre elas) é uma operação de grande interesse teórico e prático. Para realizar a tesselação usando um computador, especialmente quando a superfície considerada é hiperbólica, precisamos ter ferramentas especializadas. Neste contexto, o objetivo deste trabalho foi introduzir os principais conceitos de geometria euclidiana e hiperbólica e mostrar como uma ferramenta computacional para tesselação pode ser implementada com base neles.

A tarefa de se produzir uma ferramenta que seja capaz de renderizar alguma geometria não euclidiana está fundamentada em princípios como as tesselações e as transformações, demonstrando ser possível desenvolver ferramentas ainda mais complexas, capazes de renderizar não apenas planos, mas espaços curvos, caso exista maior aprofundamento matemático e técnico.

De maneira geral, os resultados obtidos foram satisfatórios, pois de fato foi possível desenvolver tal ferramenta, as transformações se comportaram como previsto, as técnicas de projeção funcionaram muito bem, descrevendo os modelos de forma coerente, e ainda que sob a limitação dos mapas possuírem apenas 13 ladrilhos, o mapeamento por ladrilho também funcionou como esperado.

Em trabalhos futuros, poderia ser aplicado um algoritmo de correção para resolver o efeito indesejado observado no segundo experimento (resultante do acúmulo de erros de arredondamento nas operações efetuadas com números em ponto flutuante), a limitação de 13 ladrilhos por mapa também pode ser superada, utilizar outras formas geométricas para representar ladrilhos bem como outras tesselações também seria possível adicionar em versões futuras, sabendo que a escolha de se utilizar quadriláteros e a tesselação {4,5} foi somente uma maneira de manter um escopo enxuto para dar a atenção necessária a cada ponto. Além disso, também seria possível com pequenas adaptações, criar uma aplicação em realidade virtual para visualização desses mapas.

REFERÊNCIAS

BLENDER. Versão 2.9.3. [S.I.]: Blender Foundation, 2021.

DUNHAM, D. **Hyperbolic Symmetry.** Computers & Mathematics with Applications, Elsevier, Grã Bretanha, 1986, v. 12B, n. I/2, p. 139 - 153.

CAMPER, B. **Color-Cycled Space Fumes in the Pixel Particle Shockwave: The Technical Aesthetics of Defender and the Williams Arcade Platform.** Before the crash: early video game history, Wayne State University Press, Estados Unidos, 2012.

EVES, H. **Introdução à História da Matemática.** Tradução de H. H. Domingues. 1. ed., Brasil, 2002. Título original: Introduction to the History of Mathematics.

HATCH, D. **Hyperbolic Planar Tessellations.** [S.I.] [2002], Disponível em: <http://www.plunk.org/~hatch/HyperbolicTesselations>. Acesso em: 28 de Novembro de 2021.

HOLME, A. **Geometry Our Cultural Heritage.** 2. ed. Springer, Noruega, 2010.

HOUSE, D., KEYSER, J. C. **Foundations of Physically Based Modeling and Animation.** CRC Press, Estados Unidos, 2017.

SANTOS, A. R. S., VIGLIONI, H. H. B. **Geometria Euclidiana Plana.** UFS, Brasil, 2011.

SHEPHARD, G. C. **Tilings and Patterns.** W. H. Freeman and Company, Estados Unidos, 1986.

SOLHEIM, Z. S. L. **The hyperboloid model of hyperbolic geometry.** Tese (Mestrado em Ciência) - Eastern Washington University, Estados Unidos, 2012.

TACORONTE, F. **Escher e a divisão regular do plano.** Tese (Mestrado em Matemática) - Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, Brasil, 2021.

UNITY. Versão 2019.4.28f1. [S.I.]: Unity Technologies, 2019.

WEISSTEIN, E. W. **Klein-Beltrami Model**. MathWorld, A Wolfram Web Resource. [S.I.] [2021?]. Disponível em: <https://mathworld.wolfram.com/Klein-BeltramiModel.html>. Acesso em: 15 de novembro de 2021.

WEISSTEIN, E. W. **Poincaré Hyperbolic Disk**. MathWorld, A Wolfram Web Resource. [S.I.] [2021?]. Disponível em: <https://mathworld.wolfram.com/PoincareHyperbolicDisk.html>. Acesso em: 15 de novembro de 2021.

WEISSTEIN, E. W. **Tessellation.** MathWorld, A Wolfram Web Resource. [S.I.] [2021?]. Disponível em: <https://mathworld.wolfram.com/Tessellation.html>. Acesso em: 28 de novembro de 2021.

APÊNDICE A. CÓDIGO FONTE

Endereço eletrônico do repositório disponível em: https://github.com/paulogcosta/HyperbolicTileMapping

Math.cs

```
using UnityEngine;
using MathD = System.Math;

namespace Hyperbolic.Math
{
    public class Math
    {
        /// <summary>
        /// Retorna o ponto correspondente nas coordenadas cartesianas
        /// onde o eixo Z corresponde a altura, usando doubles.
        /// </summary>
        public static VectorD3 NormalizeToCartesianCoordinates(Vector3 u)
        {
            return new VectorD3((double)u.x, (double)u.z, (double)u.y);
        }

        /// <summary>
        /// Retorna a Cotangente de x em radianos.
        /// </summary>
        public static double Cot(double x)
        {
            return  1.0 / MathD.Tan(x);
        }

        /// <summary>
        /// Retorna o ponto correspondente ao vértice projetado no Disco de Poincaré
        /// a partir do Modelo do Hiperbolóide de Minkowski.
        /// </summary>
        public static VectorD3 MinkowskiToPoincare(VectorD3 P)
        {
            VectorD3 P0 = new VectorD3(0.0, 0.0, -1.0);
            double intersectionHeight = MathD.Sqrt((P.x * P.x) + (P.y + P.y) + 1.0);
            VectorD3 Ps = new VectorD3(P.x, P.y, intersectionHeight);
            VectorD3 V = P - P0;
            double t = P0.z / V.z;
            double pX = P0.x + (t * V.x);
            double pY = P0.y + (t * V.y);
            return new VectorD3(pX, pY, 0.0);
        }

        /// <summary>
        /// Retorna o ponto correspondente ao vértice projetado no hiperbolóide
        /// de Minkowski a partir da projeção de Beltrami-Klein.
        /// </summary>
        public static VectorD3 BeltramiKleinToMinkowski(VectorD3 P)
        {
            double factor = MathD.Sqrt(1.0 / (1.0 - ((P.x * P.x) + (P.y * P.y))));
            double x1 = P.x * factor;
            double y1 = P.y * factor;
            double z1 = factor;
            return new VectorD3(x1, y1, z1);
        }

        /// <summary>
```

```
        /// Retorna o ponto correspondente à hipotenusa da região fundamental
        /// da tesselação {p, q} descrita por Dunham.
        /// </summary>
        public static VectorD3[] HypotenuseOfFundamentalRegionEndPoint(int p, int q)
        {
            double cosh2 = Cot(MathD.PI / (double) p) * Cot(MathD.PI / (double) q);
            double sinh2 = MathD.Sqrt(cosh2 * cosh2 - 1);

            double coshq = MathD.Cos(MathD.PI / (double) q) / MathD.Cos(MathD.PI / (double)
p);
            double sinhq = MathD.Sqrt(coshq * coshq - 1);

            double rad2 = sinh2 / ( cosh2 + 1 );
            double x2pt = sinhq / ( coshq + 1 );

            double xqpt = MathD.Cos( MathD.PI / (double) p ) * rad2;
            double yqpt = MathD.Sin( MathD.PI / (double) p ) * rad2;

            return new VectorD3[]{new VectorD3(xqpt, yqpt, 0f), new VectorD3(x2pt, 0f,
0f)};
        }

        /// <summary>
        /// Retorna a matriz de de translação hiperbólica ao longo do eixo X.
        /// </summary>
        public static MatrixD4x4 HyperTranslateX(double angle)
        {
            MatrixD4x4 m = MatrixD4x4.Identity();
            m.m11 = MathD.Cosh(angle);
            m.m21 = MathD.Sinh(angle);
            m.m12 = MathD.Sinh(angle);
            m.m22 = MathD.Cosh(angle);
            return m;
        }

        /// <summary>
        /// Retorna a matriz de translação hiperbólica ao longo do eixo Y.
        /// </summary>
        public static MatrixD4x4 HyperTranslateY(double angle)
        {
            MatrixD4x4 m = MatrixD4x4.Identity();
            m.m00 = MathD.Cosh(angle);
            m.m02 = MathD.Sinh(angle);
            m.m20 = MathD.Sinh(angle);
            m.m22 = MathD.Cosh(angle);
            return m;
        }

        /// <summary>
        /// Retorna a matriz de rotação afim em torno do eixo X.
        /// </summary>
        public static MatrixD4x4 RotateZ(double angle)
        {
            MatrixD4x4 m = MatrixD4x4.Identity();
            m.m00 = MathD.Cos(angle);
            m.m10 = MathD.Sin(angle);
            m.m01 = -MathD.Sin(angle);
            m.m11 = MathD.Cos(angle);
            return m;
        }
    }
}
```

MatrixD4x4.cs

```
namespace Hyperbolic.Math
{
    /// <summary>
    /// Matriz quadrada de ordem 4 que utiliza doubles.
    /// </summary>
    public struct MatrixD4x4
    {
        public double m00, m01, m02, m03,
                      m10, m11, m12, m13,
                      m20, m21, m22, m23,
                      m30, m31, m32, m33;

        public MatrixD4x4(double m00, double m01, double m02, double m03,
                          double m10, double m11, double m12, double m13,
                          double m20, double m21, double m22, double m23,
                          double m30, double m31, double m32, double m33)
        {
            this.m00 = m00; this.m01 = m01; this.m02 = m02; this.m03 = m03;
            this.m10 = m10; this.m11 = m11; this.m12 = m12; this.m13 = m13;
            this.m20 = m20; this.m21 = m21; this.m22 = m22; this.m23 = m23;
            this.m30 = m30; this.m31 = m31; this.m32 = m32; this.m33 = m33;
        }

        /// <summary>
        /// Retorna a matriz identidade de ordem 4 usando doubles.
        /// </summary>
        public static MatrixD4x4 Identity()
        {
            return new MatrixD4x4(1.0, 0.0, 0.0, 0.0,
                                  0.0, 1.0, 0.0, 0.0,
                                  0.0, 0.0, 1.0, 0.0,
                                  0.0, 0.0, 0.0, 1.0);
        }

        /// <summary>
        /// Multiplicação entre uma Matriz quadrada de ordem 4 reduzida para
        /// ordem 3 e um Vetor de 3 dimensões utilizando doubles.
        /// </summary>
        public static VectorD3 operator *(MatrixD4x4 a, VectorD3 b)
        {
            double x = (a.m00 * b.x) + (a.m01 * b.y) + (a.m02 * b.z);
            double y = (a.m10 * b.x) + (a.m11 * b.y) + (a.m12 * b.z);
            double z = (a.m20 * b.x) + (a.m21 * b.y) + (a.m22 * b.z);
            return new VectorD3(x, y, z);
        }

        /// <summary>
        /// Multiplicação entre duas Matrizes quadradas de ordem 4 usando doubles.
        /// </summary>
        public static MatrixD4x4 operator *(MatrixD4x4 a, MatrixD4x4 b)
        {
            double m00 = (a.m00 * b.m00) + (a.m01 * b.m10) + (a.m02 * b.m20) + (a.m03 *
b.m30);
            double m01 = (a.m00 * b.m01) + (a.m01 * b.m11) + (a.m02 * b.m21) + (a.m03 *
b.m31);
            double m02 = (a.m00 * b.m02) + (a.m01 * b.m12) + (a.m02 * b.m22) + (a.m03 *
b.m32);
            double m03 = (a.m00 * b.m03) + (a.m01 * b.m13) + (a.m02 * b.m23) + (a.m03 *
b.m33);

            double m10 = (a.m10 * b.m00) + (a.m11 * b.m10) + (a.m12 * b.m20) + (a.m13 *
b.m30);
            double m11 = (a.m10 * b.m01) + (a.m11 * b.m11) + (a.m12 * b.m21) + (a.m13 *
b.m31);
```

```
            double m12 = (a.m10 * b.m02) + (a.m11 * b.m12) + (a.m12 * b.m22) + (a.m13 *
b.m32);
            double m13 = (a.m10 * b.m03) + (a.m11 * b.m13) + (a.m12 * b.m23) + (a.m13 *
b.m33);

            double m20 = (a.m20 * b.m00) + (a.m21 * b.m10) + (a.m22 * b.m20) + (a.m23 *
b.m30);
            double m21 = (a.m20 * b.m01) + (a.m21 * b.m11) + (a.m22 * b.m21) + (a.m23 *
b.m31);
            double m22 = (a.m20 * b.m02) + (a.m21 * b.m12) + (a.m22 * b.m22) + (a.m23 *
b.m32);
            double m23 = (a.m20 * b.m03) + (a.m21 * b.m13) + (a.m22 * b.m23) + (a.m23 *
b.m33);

            double m30 = (a.m30 * b.m00) + (a.m31 * b.m10) + (a.m32 * b.m20) + (a.m33 *
b.m30);
            double m31 = (a.m30 * b.m01) + (a.m31 * b.m11) + (a.m32 * b.m21) + (a.m33 *
b.m31);
            double m32 = (a.m30 * b.m02) + (a.m31 * b.m12) + (a.m32 * b.m22) + (a.m33 *
b.m32);
            double m33 = (a.m30 * b.m03) + (a.m31 * b.m13) + (a.m32 * b.m23) + (a.m33 *
b.m33);

            return new MatrixD4x4(m00, m01, m02, m03,
                                  m10, m11, m12, m13,
                                  m20, m21, m22, m23,
                                  m30, m31, m32, m33);
        }
    }
}
```

VectorD3.cs

```
using UnityEngine;

namespace Hyperbolic.Math
{
    /// <summary>
    /// Vetor de 3 dimensões que utiliza doubles.
    /// </summary>
    public struct VectorD3
    {
        public double x, y, z;
        public VectorD3(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        public override string ToString()
        {
            return $"({this.x}, {this.y}, {this.z})";
        }

        /// <summary>
        /// Realiza subtração de vetores usando doubles.
        /// </summary>
        public static VectorD3 operator -(VectorD3 a, VectorD3 b)
        {
            return new VectorD3(a.x - b.x, a.y - b.y, a.z - b.z);
        }

        /// <summary>
        /// Retorna o ponto correspondente nas coordenadas do Unity
```

```
        /// onde o eixo Y corresponde a altura, usando floats.
        /// </summary>
        public Vector3 ToUnity()
        {
            return new Vector3((float)this.x, (float)this.z, (float)this.y);
        }
    }
}
```

ChunkObject.cs

```
using System.Collections.Generic;
using UnityEngine;

namespace Hyperbolic.Tile
{
    [CreateAssetMenu(fileName = "Chunk", menuName = "Hyperbolic/Chunk")]
    public class ChunkObject : ScriptableObject
    {
        [SerializeField] List<ChunkTile> tiles;

        public void Initialize()
        {
            GameObject reference = Instantiate(new GameObject(), new Vector3(0, 0, 0),
Quaternion.identity);
            reference.name = "Reference";
            foreach(ChunkTile tile in tiles)
            {
                GameObject obj = Instantiate(tile.tile.TilePrefab, new Vector3(0,0,0),
Quaternion.Euler(new Vector3(-90, 0, 0)));
                Tile objTile = obj.AddComponent<Tile>();
                objTile.TileObject = tile.tile;
                objTile.ChunkMap = tile.chunkMap;
            }
        }
    }

    [System.Serializable]
    public class ChunkTile
    {
        [SerializeField] public TileObject tile;
        [SerializeField] public ChunkMap chunkMap;
    }
}
```

TileObject.cs

```
using UnityEngine;

namespace Hyperbolic.Tile
{
    [CreateAssetMenu(fileName = "Tile", menuName = "Hyperbolic/Tile")]
    public class TileObject : ScriptableObject
    {
        [SerializeField] GameObject tilePrefab;
        [SerializeField] Texture texturePrefab;

        public GameObject TilePrefab => tilePrefab;
        public Texture TexturePrefab => texturePrefab;
    }

    public enum ChunkMap
    {
        center,
        up,
```

```
        down,
        left,
        right,
        upLeft,
        leftUp,
        upRight,
        rightUp,
        downLeft,
        leftDown,
        downRight,
        rightDown
    }
}
```

Tile.cs

```
using UnityEngine;

namespace Hyperbolic.Tile
{
    public class Tile : MonoBehaviour
    {
        [SerializeField] private TileObject tileObject;
        [SerializeField] private ChunkMap chunkMap;
        private GameObject referenceTransformObject;
        public TileObject TileObject { get => tileObject; set => tileObject = value; }
        public ChunkMap ChunkMap { get => chunkMap; set => chunkMap = value; }

        private void Start()
        {
            referenceTransformObject = GameObject.Find("Reference");
            AssignTexture();
            TileUtils.MeshKleinToMinkowski(gameObject, referenceTransformObject);
            AssignPosition((float)Math.Math.HypotenuseOfFundamentalRegionEndPoint(4,
5)[1].x * 4);
        }

        private void Update()
        {
            TileUtils.Move(gameObject, 1f, referenceTransformObject);
        }

        private void AssignTexture()
        {
            var texturePropertyBlock = new MaterialPropertyBlock();
            texturePropertyBlock.SetTexture("_MainTex", TileObject.TexturePrefab);
            GetComponent<Renderer>().SetPropertyBlock(texturePropertyBlock);
        }

        private void AssignPosition(float step)
        {
            step = step + 0.025f;
            switch (chunkMap)
            {
                case (ChunkMap.center):
                    TileUtils.MoveDiscrete(gameObject, 0, 0, false);
                    break;
                case (ChunkMap.up):
                    TileUtils.MoveDiscrete(gameObject, step, 0, false);
                    break;
                case (ChunkMap.left):
                    TileUtils.MoveDiscrete(gameObject, 0, step, false);
                    break;
                case (ChunkMap.down):
                    TileUtils.MoveDiscrete(gameObject, -step, 0, false);
                    break;
                case (ChunkMap.right):
```

```
                    TileUtils.MoveDiscrete(gameObject, 0, -step, false);
                    break;
                case (ChunkMap.leftUp):
                    TileUtils.MoveDiscrete(gameObject, step, step, false);
                    break;
                case (ChunkMap.upLeft):
                    TileUtils.MoveDiscrete(gameObject, step, step, true);
                    break;
                case (ChunkMap.rightDown):
                    TileUtils.MoveDiscrete(gameObject, -step, -step, false);
                    break;
                case (ChunkMap.downRight):
                    TileUtils.MoveDiscrete(gameObject, -step, -step, true);
                    break;
                case (ChunkMap.rightUp):
                    TileUtils.MoveDiscrete(gameObject, step, -step, false);
                    break;
                case (ChunkMap.upRight):
                    TileUtils.MoveDiscrete(gameObject, step, -step, true);
                    break;
                case (ChunkMap.leftDown):
                    TileUtils.MoveDiscrete(gameObject, -step, step, false);
                    break;
                case (ChunkMap.downLeft):
                    TileUtils.MoveDiscrete(gameObject, -step, step, true);
                    break;
                default:
                    break;
            }
        }
    }
}
```

TileUtils.cs

```
using UnityEngine;
using Hyperbolic.Math;

namespace Hyperbolic.Tile
{
    public class TileUtils
    {
        /// <summary>
        /// Retorna a posição global a partir de uma posição local assumindo
        /// um GameObject como referência e uma escala.
        /// </summary>
        public static Vector3 LocalToWorld(GameObject obj, Vector3 u, Vector3 scale)
        {
            Matrix4x4 localToWorld = obj.transform.localToWorldMatrix;
            Vector3 normalized = localToWorld.MultiplyPoint3x4(u);
            Matrix4x4 localNormalized = new Matrix4x4();
            localNormalized.m00 = normalized.x;
            localNormalized.m11 = normalized.y;
            localNormalized.m22 = normalized.z;
            return localNormalized * scale;
        }

        /// <summary>
        /// Retorna a posição local a partir de uma posição global assumindo
        /// um GameObject como referência e uma escala.
        /// </summary>
        public static Vector3 WorldToLocal(GameObject obj, Vector3 u, Vector3 scale)
        {
            Vector3 inverseScale = new Vector3(1f / scale.x, 1f / scale.y, 1f / scale.z);
            Matrix4x4 localToWorld = obj.transform.localToWorldMatrix;
            Vector3 normalized = localToWorld.MultiplyPoint3x4(u);
            Matrix4x4 localNormalized = new Matrix4x4();
```

```
            localNormalized.m00 = normalized.x;
            localNormalized.m11 = normalized.y;
            localNormalized.m22 = normalized.z;
            return localNormalized * inverseScale;
        }

        /// <summary>
        /// Projeção no Hiperbolóide de Minkowski de um plano que está no modelo
Beltrami-Klein
        /// utilizando um GameObject como referência global.
        /// </summary>
        public static void MeshKleinToMinkowski(GameObject meshObject, GameObject
transformReference)
        {
            Mesh mesh = meshObject.GetComponent<MeshFilter>().mesh;
            Vector3[] vertices = mesh.vertices;

            for (int i = 0; i < vertices.Length; i++)
            {
                // Obtém a posição global do vértice através da posição local
                Vector3 worldV = LocalToWorld(transformReference, mesh.vertices[i],
meshObject.transform.localScale);
                Vector3 worldVUp = new Vector3(worldV.x, worldV.z, worldV.y);
                VectorD3 worldVCartesian =
Math.Math.NormalizeToCartesianCoordinates(worldVUp);

                //Converte o ponto do modelo de Klein para Minkowski
                VectorD3 result = Math.Math.BeltramiKleinToMinkowski(worldVCartesian);
                Vector3 res = result.ToUnity();
                vertices[i] = WorldToLocal(transformReference, new Vector3(res.x, res.z,
res.y), meshObject.transform.localScale);
            }
            mesh.vertices = vertices;
        }

        /// <summary>
        /// Movimentação discreta no Hiperbolóide de Minkowski. Por conta da holonomia,
movimentar
        /// no eixo x e depois no eixo y resulta em uma posição diferente de movimentar
primeiro no
        /// eixo y e depois no eixo x.
        /// </summary>
        public static void MoveDiscrete(GameObject meshObject, float x, float y, bool
xFirst)
        {
            Mesh mesh = meshObject.GetComponent<MeshFilter>().mesh;
            Vector3[] vertices = mesh.vertices;
            for (var i = 0; i < vertices.Length; i++)
            {
                MatrixD4x4 result = MatrixD4x4.Identity();
                if (x == 0 && y == 0) result = MatrixD4x4.Identity();
                if (x == 0) result = Math.Math.HyperTranslateY(y);
                else if (y == 0) result = Math.Math.HyperTranslateX(x);
                else if (xFirst) result = Math.Math.HyperTranslateX(x) *
Math.Math.HyperTranslateY(y);
                else if (!xFirst) result = Math.Math.HyperTranslateY(y) *
Math.Math.HyperTranslateX(x);
                VectorD3 currentPos = new VectorD3(vertices[i].x, vertices[i].y,
vertices[i].z);
                VectorD3 resultPos = result * currentPos;
                Vector3 resultFloat = resultPos.ToUnity();
                vertices[i] = new Vector3(resultFloat.x, resultFloat.z, resultFloat.y);
            }
            mesh.vertices = vertices;
        }

        /// <summary>
```

```
        /// Movimentação contínua no Hiperbolóide de Minkowski. Utilizando o eixo
horizontal para rotacionar
        /// e o vertical para seguir em frente ou retornar.
        /// </summary>
        public static void Move(GameObject meshObject, float speed, GameObject
transformReference)
        {
            float x = (float)Input.GetAxis("Horizontal") * -speed * (float)Time.deltaTime;
            float y = (float)Input.GetAxis("Vertical") * -speed * (float)Time.deltaTime;

            Mesh mesh = meshObject.GetComponent<MeshFilter>().mesh;
            Vector3[] vertices = mesh.vertices;

            for (int i = 0; i < vertices.Length; i++)
            {
                MatrixD4x4 result;
                Vector3 worldV = LocalToWorld(transformReference, mesh.vertices[i],
meshObject.transform.localScale);
                result = Math.Math.RotateZ(x * 2) * Math.Math.HyperTranslateY(y);
                VectorD3 currentPos = new VectorD3(vertices[i].x, vertices[i].y,
vertices[i].z);
                VectorD3 resultPos = result * currentPos;
                Vector3 resultFloat = resultPos.ToUnity();
                vertices[i] = new Vector3(resultFloat.x, resultFloat.z, resultFloat.y);
            }
            mesh.vertices = vertices;
        }
    }
}
```

Poincare.shader

```
Shader "Custom/Poincare"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _P0 ("Origin", Vector) = (0.0, 0.0, 0.0)
        _Height ("Height of Hyperboloid", Float) = 1.0

    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
```

```
            float2 uv : TEXCOORD0;
            UNITY_FOG_COORDS(1)
            float4 vertex : SV_POSITION;
        };

        sampler2D _MainTex;
        float4 _MainTex_ST;
        float3 _P0;
        float _Height;

        v2f vert (appdata v)
        {
            v2f o;
            float4x4 customModel;
            customModel = unity_ObjectToWorld;
            float4 Pe = mul(customModel, v.vertex);
            float3 P0 = float3(_P0.x, _P0.y - _Height, _P0.z);
            float interHeight = sqrt(pow(Pe.x, 2) + pow(Pe.z, 2) + 1);
            float3 Ps = float3(Pe.x, interHeight, Pe.z);
            float3 V = Pe - P0;
            float t = P0.y/V.y;
            float pX = P0.x + (t * V.x);
            float pZ = P0.z + (t * V.z);
            float3 projectedP = float3(pX, 0.0, pZ);
            float4 resultVertex = mul(unity_WorldToObject, projectedP);
            o.vertex = UnityObjectToClipPos(resultVertex);

            o.uv = TRANSFORM_TEX(v.uv, _MainTex);
            UNITY_TRANSFER_FOG(o,o.vertex);
            return o;
        }

        fixed4 frag (v2f i) : SV_Target
        {
            // sample the texture
            fixed4 col = tex2D(_MainTex, i.uv);
            // apply fog
            UNITY_APPLY_FOG(i.fogCoord, col);
            return col;
        }
        ENDCG
    }
  }
}
```