**UNIVERSIDADE FEDERAL DE MINAS GERAIS**

Gean Guilherme dos Santos

# Uso de depthmaps para geração automática de animações

Belo Horizonte
2024

# Resumo

Este trabalho aborda a automação de parte do processo de criação de animações, com o objetivo de reduzir o esforço necessário para a produção de animações.

O trabalho visa atingir essa automação desenvolvendo técnicas para combinar ilustrações e valores gerados por depth maps para gerar uma ilustração que possa passar por movimentações simulando um movimento tridimensional.

Desse modo, são exploradas técnicas relacionadas a renderização 3D em tempo real e manipulação de objetos representados por meshes.

# Sumário

**1 Introdução** **3**
1.1 Motivação . . . 3
1.2 Objetivos . . . 7

**2 Renderização de objetos 3D** **8**
2.1 Meshes . . . 8
2.2 Shaders . . . 9
2.3 Fragment shader . . . 10
2.4 Vertex shader . . . 10
2.5 Coordenadas . . . 11

**3 Método Parallax Mapping** **13**
3.1 Parallax Mapping simples . . . 13
3.2 Espaço tangente . . . 15
3.3 Steep Parallax Mapping . . . 16
3.4 Parallax Occlusion Mapping . . . 18
3.5 Avaliação do método . . . 18
3.6 Código . . . 19

**4 Suavização de meshes** **21**
4.1 Suavização laplaciana . . . 22
4.2 Avaliação do método suavização laplaciana . . . 23
4.3 Aplicação do método . . . 24

**5 Conclusão** **25**
5.1 Reflexão sobre o desenvolvimento . . . 25
5.2 Comparação com animação manual . . . 26
5.3 Próximos passos . . . 26

**Referências Bibliográficas** **28**

# Capítulo 1

# Introdução

## 1.1 Motivação

A animação é o processo criar a ilusão de movimento em objetos, cenas e personagens. Animações envolvendo personagens são usada em diversas áreas, como por exemplo filmes e jogos. Enquanto isso, animações evolvendo objetos são importantes, por exemplo, para visualiação e design de máquinas, ferramentas, ou quaisquer outros objetos do mundo real. No entanto, a criação de animações é processo complexo que envolve várias etapas e habilidades técnicas, como por exemplo desenho, física, e programação. Existem atualmente vários métodos para criar animações, e cada um envolve ferramentas e técnicas diferentes.

Um destes métodos é o método tradicional de animação frame a frame. Este método envolve desenhar cada quadro à mão, a mesma técnica usada nas primeiras animações. Embora seja laboriosa, é uma técnica ainda muito utilizada em casos onde interatividade não é necessária, como filmes. Caso seja necessário que o usuário interaja de alguma forma com o objeto sendo animado, uma técnica diferente será necessária.

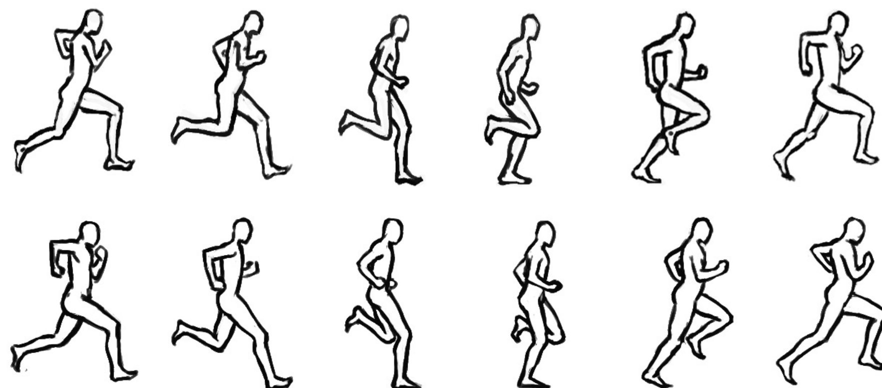

Figura 1.1: Animação tradicional. Fonte: https://citizenwoodward.wordpress.com/2014/03/14/frame-by-frame-run-cycle-animation/

Outra técnica é a animação 3D, que envolve a criação de modelos tridimensionais

que são manipulados digitalmente para simular movimento. A animação 3D é amplamente utilizada em filmes e jogos devido ao seu realismo e versatilidade, facilitando que personagens sejam exibidos de ângulos diferentes e em posições arbitrárias, aproveitando o trabalho existente. Métodos similares são necessários quando interatividade é um requerimento, por exemplo em jogos.

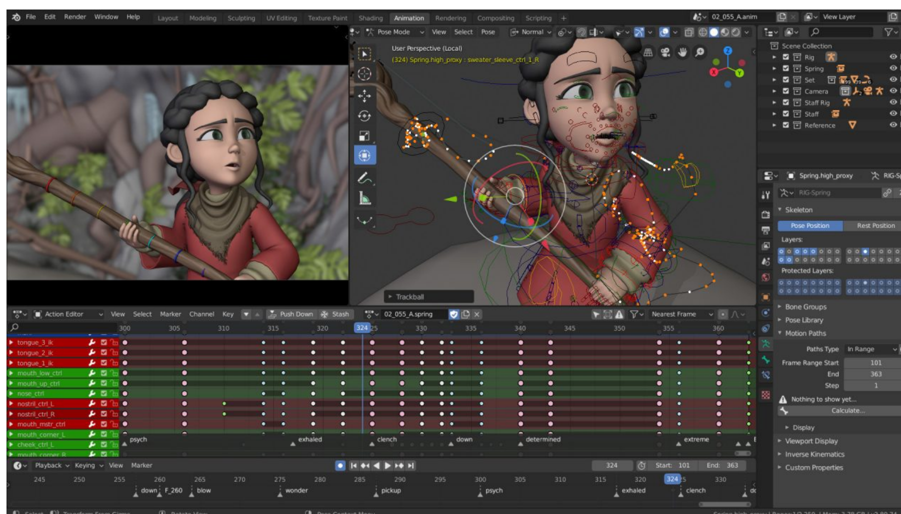

Figura 1.2: Animação 3D. Fonte: blender.org

Animações 3D tendem a ter custo de produção altamente elevado, por exemplo, existem filmes com animações 3D com custo de produção acima de 200.000.000 dólares [1]. Em casos em que um modelo tridimensional completo de um personagem não seja necessário, seria ideal criar animações a partir de uma ou poucas ilustrações de referência de um personagem. Existem técnicas para realizar o processo de transformar essas ilustrações em algo semelhante a um modelo tridimensional.

Uma dessas técnicas é o método de animação esquelética. A animação esquelética é um método que utiliza um sistema de "ossos" para animar personagens e objetos. O animador posiciona ossos de certa maneira que represente a estrutura interna do personagem. Para personagens humanos e animais, a estrutura resultante assemelha-se a estrutura óssea do ser representado, mas ossos também podem ser usados em qualquer parte da ilustração que necessite deformação. Quando os ossos se movimentam, o sistema automaticamente deforma as ilustrações de maneira correspondente, criando a ilusão de movimento. As animações resultantes podem ser facilmente reutilizadas e adaptadas para diferentes personagens, aumentando a eficiência do processo de produção.

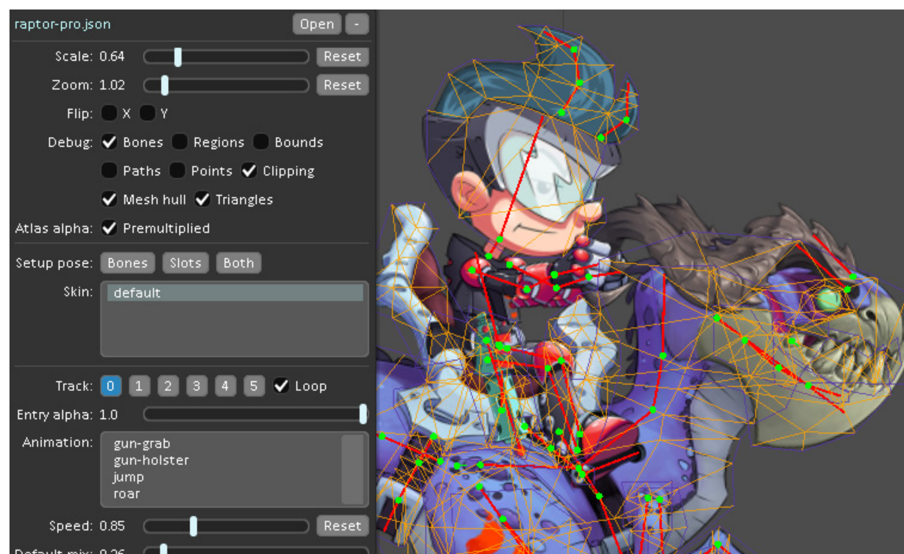


Figura 1.3: Animação esquelética. Fonte: https://esotericsoftware.com/

Esta técnica também pode ser utilizada para simular um efeito tridimensional. A imagem a seguir exemplifica como um número pequeno de imagens pode ser manipulado para gerar a ilusão de um objeto tridimensional, diminuindo significativamente o esforço necessário. Isso ocorre porque não é preciso criar um modelo tridimensional completo do objeto. O efeito resultante pode ser altamente convincente, criando a ilusão de que estamos realmente observando um objeto tridimensional.

Outro método, que é o foco principal deste trabalho, é a animação por deformação de malha ou *mesh*. Este método utiliza uma malha aplicada sobre a imagem, permitindo que os animadores deformem e manipulem a imagem original de diversas maneiras.

Por exemplo, o animador pode deformar a ilustração de modo que o personagem vire a cabeça, ou realize outro movimento tridimensional. A partir da imagem original e da imagem modificada, uma interpolação de posições intermediárias é realizada automaticamente, criando uma animação suave. Dessa forma, o animador modifica imagens de referência para criar movimento e expressão, resultando em consideravelmente menos trabalho manual, pois uma animação pode ser gerada a partir de poucas imagens, ou até uma única imagem em certos casos.

Esse método é especialmente útil para criar animações focadas em expressões faciais de personagens, por exemplo, se o personagem precisa olhar para várias direções, será necessário criar somente modificações onde o personagem olha para cima, para baixo, e pela simetria da maior parte dos personagens, para a direita ou esquerda.

Tradicionalmente, esse processo é realizado manualmente, onde o animador manipula a imagem para gerar a ilusão de rotação. Embora esse método reduza o tempo e o esforço necessários, o processo ainda é laborioso.

Como o custo e quantidade de trabalho envolvidos no processo ainda podem ser altos, seria interessante utilizar ferramentas para automatizar parte desse processo. Con-

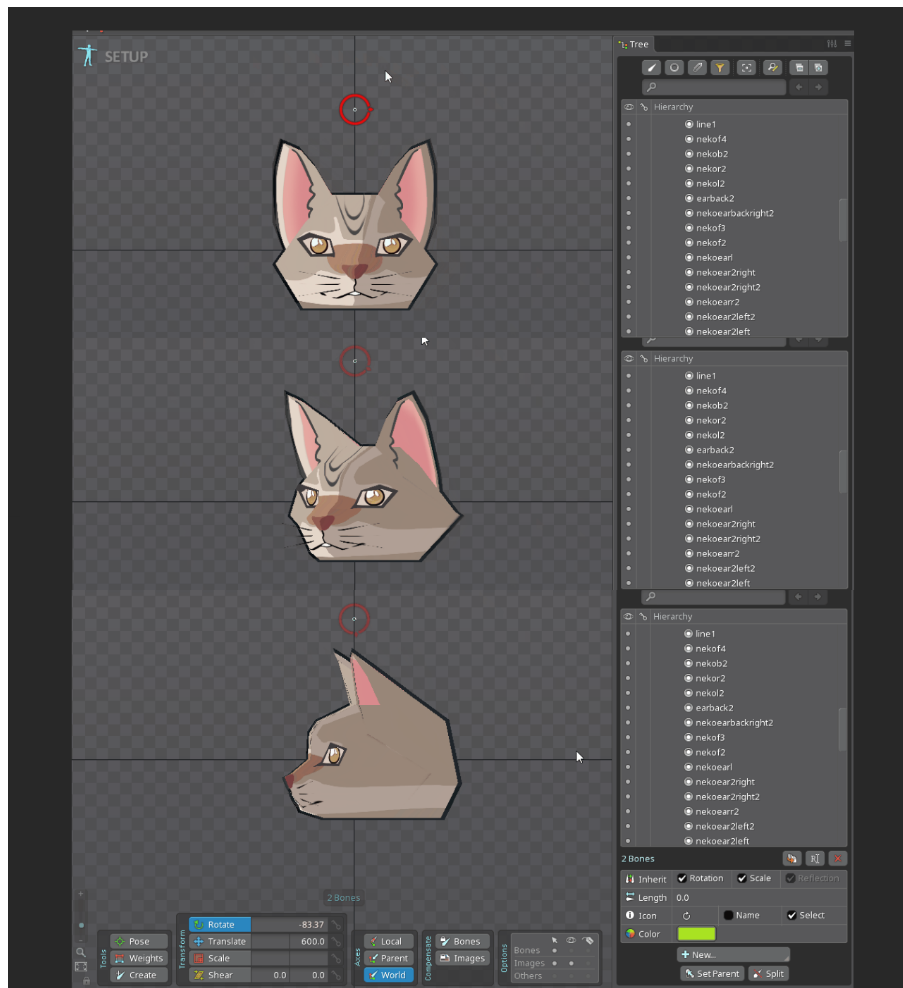


Figura 1.4: Simulação de um modelo 3D com animação esquelética.

Figura 1.5: Animação por deformação de mesh. Fonte: https://www.live2d.com/en/learn/sample/

siderando os avanços recentes em inteligência artificial, como a geração de modelos 3D a partir de ilustrações e a estimação de profundidade em imagens, talvez seja possível automatizar a criação de rotações tridimensionais de um personagem. Assim, para gerar um personagem virtual com o qual um usuário possa interagir, o animador poderia criar manualmente as expressões faciais iniciais, enquanto os movimentos poderiam ser gerados

automaticamente. Após a geração da animação pelo sistema, o animador faria ajustes manuais para atingir um nível de qualidade satisfatório.

## 1.2 Objetivos

O principal objetivo deste trabalho é reduzir ainda mais o esforço necessário para a criação de animações, aproveitando os avanços recentes em inteligência artificial, que possivelmente podem automatizar parte do processo. É importante notar que, embora as ferramentas desenvolvidas possam automatizar algumas etapas, o trabalho manual ainda será necessário para refinar as animações produzidas.

Portanto, os materiais produzidos devem não apenas servir para renderizar uma animação rudimentar ou servir como referência para a realização do processo manual, mas também oferecer uma base sólida para refinamentos manuais. Idealmente, os resultados gerados devem ser integráveis com ferramentas existentes, facilitando o fluxo de trabalho do animador.

Deste modo, o objetivo é desenvolver um sistema que, dado uma ilustração de um personagem, gere meshes para cada parte do personagem, criando versões deformadas de cada mesh. Ao interpolar entre o estado original e o estado deformado, deve-se produzir uma ilusão de movimento tridimensional. A malha deve ser simples, com uma quantidade razoável de vértices, permitindo ajustes manuais precisos. Além disso, a deformação deve gerar movimentos suaves dos vértices, de modo que pequenos ajustes resultem em modificações sutis na animação final.

Além de simplificar o processo de animação, o sistema deve garantir a qualidade do movimento tridimensional, de modo que o trabalho manual se concentre na melhoria da animação, enquanto as tarefas mais repetitivas e demoradas serão automatizadas, resultando em um processo mais eficiente e produtivo.

# Capítulo 2

# Renderização de objetos 3D

Este capítulo tem como objetivo expor as técnicas básicas de renderização e fornecer o contexto relevante para a compreensão do processo.

## 2.1 Meshes

Meshes 3D são uma estrutura de dados fundamental na renderização de objetos tridimensionais. Cada mesh (ou malha) contém, no mínimo, um conjunto de pontos no espaço, geralmente chamados de vértices, e informações sobre quais combinações de vértices determinam as faces da superfície a ser renderizada. Cada face pode ser um polígono arbitrário, no entanto, mais comumente, cada face será um triângulo, pois facilitam a manipulação em diversos casos.

Para que cada face possa ser visualizada, cada mesh pode também conter informações sobre sua aparência. Por exemplo, podemos definir uma cor associada a cada vértice. Se todos os vértices associados a uma face possuem a mesma cor, um triângulo pode ser renderizado com essa cor.

Para que seja possível renderizar um objeto em diferentes tamanhos ou ângulos, ou em dispositivos com resoluções variadas, cada mesh não contém informação sobre a cor específica de cada pixel na renderização final. Assim, um triângulo pode ser renderizado contendo pixels de diversas cores em seu interior. Por exemplo, se cada vértice é associado a uma cor diferente, uma interpolação pode ser feita para determinar a cor de cada pixel no resultado final.

Esse processo de interpolação é realizado por um programa denominado fragment shader. Os fragment shaders pertencem a uma família de diferentes shaders, que são programas executados em dispositivos gráficos. Os shaders desempenham um papel importante na definição da aparência final dos objetos renderizados, permitindo a criação de efeitos que seriam dificeís de excutar de maneira eficiente fora de um dispositivo gráfico.

Além da cor, os fragment shaders podem manipular outros atributos das faces,

como aplicar uma textura, por exemplo, que permite mapear imagens bidimensionais sobre as superfícies, adicionando detalhes sem a necessidade de aumentar a complexidade de cada mesh.

## 2.2 Shaders

Para que os dados contidos em cada mesh possam ser transformados em uma imagem final, eles passam por vários tipos de shaders, além dos shaders mencionados anteriormente. Cada estágio é executado em sequência, recebendo como entrada o output do estágio anterior. O resultado final será a cor exibida em cada pixel do dispositivo.

A seguinte figura ilustra os estágios necessários para que o conjunto de vértices possa ser transformado em uma imagem renderizada.

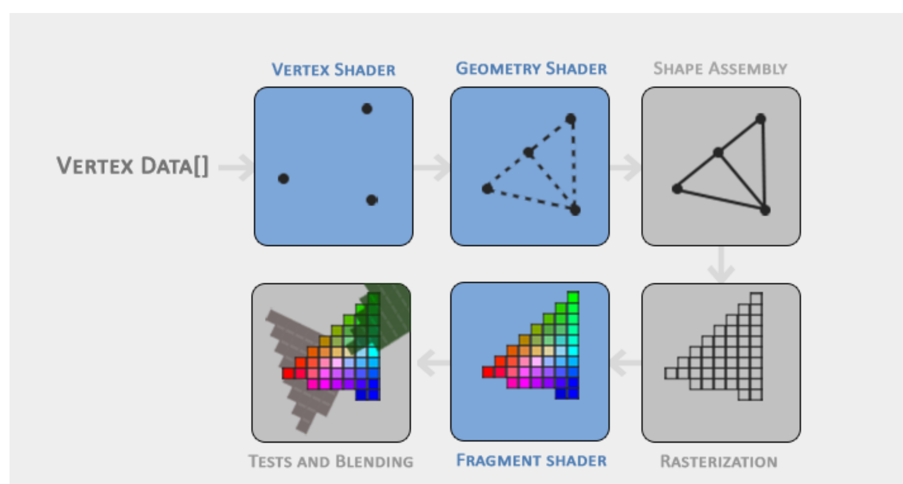


Figura 2.1: Shader Pipeline. Fonte[3]

Entre os diversos estágios de processamento, o vertex shader e o fragment shader, se destacam por desempenhar papéis importantes na customização da aparência dos objetos a serem renderizados.

## 2.3 Fragment shader

A função do fragment shader é determinar a cor de um fragmento. Cada fragmento representa basicamente um pixel do dispositivo. Ao escrever um fragment shader, definimos como cada fragmento deve ser colorido com base nos dados de entrada, estebelecendo a lógica que determina a cor em cada vértice, e a determinação das cores para os pixels intermediários é feita automaticamente pelo pipeline gráfico.

Um exemplo simples de um fragment shader em HLSL (High-Level Shading Language):

```
float4 frag (Vertex vertex) : SV_TARGET
{
 return vertex.color;
}
```

Neste exemplo, o fragment shader retorna a cor associada ao vértice. Em hlsl cada vértice é representado por um vetor, caso queiramos associar uma cor ao vértice devemos definir uma estrutura de dados adequada, nesse caso uma estrutura chamada Vertex, para que possa ser passada para o fragment shader.

## 2.4 Vertex shader

O vertex shader tem a função de passar dados de cada vértice para os próximos estágios. Ele pode realizar modificações arbitrárias nas posições de cada vértice, modificando o formato ou tamanho do objeto, e também associando dados adicionais a cada vértice, por exemplo cores, ou texturas, para renderizar objetos mais interessantes.

Um exemplo de um vertex shader simples:

```
VertexOutput vert(VertexInput input)
{
 output.pos = WorldToClip(ObjectToWorld(input.pos.xyz));
 output.uv = TRANSFORM_TEX(input.uv, _MainTex);
```

```
}
```

Esse shader realiza uma transformação nas coordenadas da posição do vértice e associa um segundo vetor de coordenadas, o vetor de coordenadas uv, necessários para que objeto tenha tenha aparência determinada por uma imagem arbitrária, que define qual parte da imagem está associada com quais vértices.

## 2.5 Coordenadas

A primeira operação do vertex shader exibido é uma conversão de coordenadas. Essa conversão acontece porque cada mesh tem dados da posição relativos a uma origem arbitrária. Para que cada mesh possa ser movimentada e posicionada relativamente a outros objetos, precisamos de um segundo sistema de coordenadas em que cada objeto é posicionado relativo a uma origem comum

Para que os objetos possam então ser exibidos na tela, criamos um terceiro sistema de coordenadas em que os objetos são projetados em um plano, que pode ser interpretado como a tela do dispositivo. O fragment shader trabalha nesse sistema de coordenadas, então convertemos os vértices antes de passar para o fragment shader.

Criamos ainda mais um sistema de coordenadas, que associa cada vértice a uma parte arbitrária de uma textura, para que a textura possa ser aplicada ao objecto da maneira esperada.

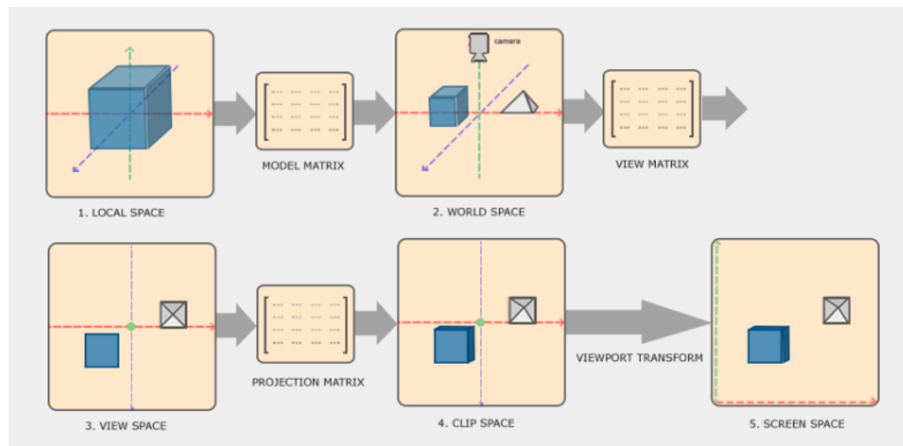


Figura 2.2: Sistemas de coordenadas. Fonte[3]

A figura mostra as conversões de coordenadas que acontecem para que cada shader possa realizar operações necessárias em cada vértice de maneira conveniente, tendo acesso a todos os dados necessários.

```
VertexOutput vert(VertexInput input){
 output.pos = WorldToClip(ObjectToWorld(input.pos.xyz));
 output.uv = TRANSFORM_TEX(input.uv, _MainTex);
}

float4 frag (VertexOutput input) : SV_TARGET {
 return tex2D(_MainTex, input.uv);
}
```

Esse exemplo mostra dois shaders, que trabalham em conjunto para exibir objetos com textura. A função tex2D é uma função em Hlsl que retorna a cor de um pixel da textura de acordo com as coordenadas uv do vértice.

# Capítulo 3

# Método Parallax Mapping

## 3.1 Parallax Mapping simples

Em uma primeira tentativa, uma maneira de gerar perspectivas diferentes é simplesmente criar uma malha com um vértice para cada pixel da ilustração. Assim cada vértice pode ter sua coordenada z ajustada de acordo com o valor gerado, e a malha pode ser renderizada normalmente, e diferentes rotações podem ser geradas. Esse método tem vários problemas. Primeiramente, é um método muito caro computacionalmente, sendo que a malha resultante pode ter milhões de vértices. Além disso, meshes com quantidade elevadas de vértices são difíceis de modificar manualmente.

Uma solução para o primeiro problema é o método parallax mapping. A ideia por trás do método é que ao invés de manipular vértices, apenas as coordenadas uv são modificadas de modo que aproxime a cor que seria vista naquele ponto se a superfície realmente tivesse estrutura tridimensional.

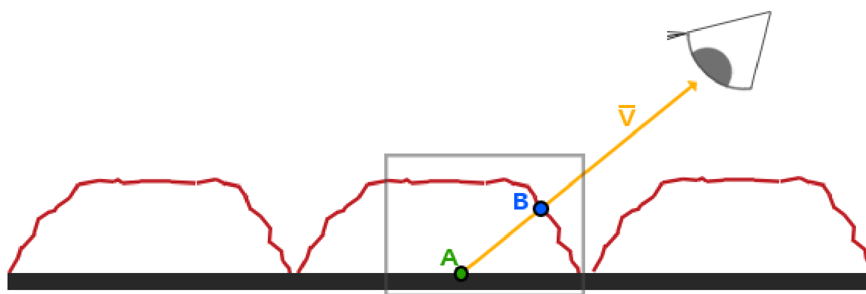


Figura 3.1: Parallax Mapping. Fonte[3]

Na figura, a linha vermelha mostra altura de cada ponto dado pelo depth map, o vetor V tem a mesma direção na qual a superfície é vista. O problema que tentamos resolver é como estimar as coordenadas uv do ponto B. O método parallax mapping simples estima as coordenadas projetando um vetor P, que tem magnitude igual à altura no ponto A, e direção igual a V, na superfície.

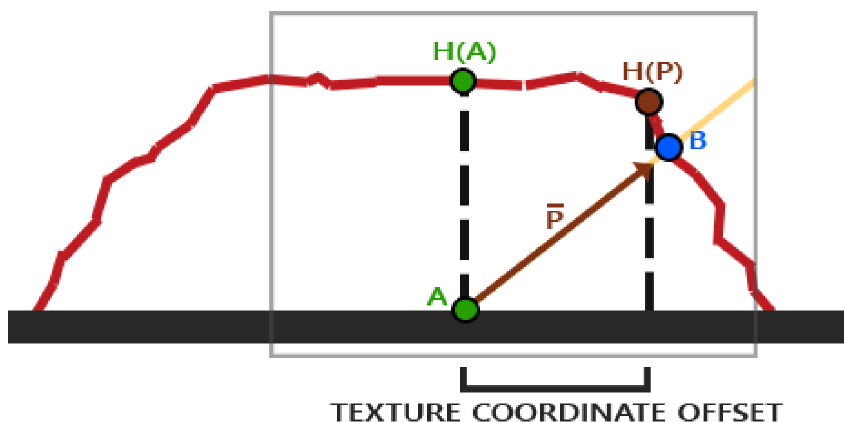


Figura 3.2: Parallax Mapping. Fonte[3]

A expectativa é que ao olharmos para a superfície em direção ao ponto A, veremos uma cor que aproxima a cor no ponto B. O método tem resultados bons para superfícies que não tem mudanças bruscas de cor ou altura, que nem sempre é o caso para ilustrações de personagens. O exemplo seguinte mostra como o resultado geralmente não é adequado para personagens.

Figura 3.3: Método parallax simples aplicado a um personagem.Vídeo: https://youtu.be/1dSrdxQHBZM

O método encontra um problema quando acontece uma rotação da superfície. Como estamos realizando todos os cálculos usando coordenadas uv, não é possível projetar o vetor P na superfície rotacionada sem nenhuma informação adicional. Uma maneira conveniente de solucionar o problema é convertendo as coordenadas do vetor V para o espaço tangente da superfície.

## 3.2 Espaço tangente

Para calcular uma base para o espaço tangente em cada vértice, determinamos três vetores: normal, tangente e bitangente (ou tangente dupla). O vetor normal é perpendicular à superfície do triângulo. O vetor tangente é paralelo à superfície e aponta na direção positiva do eixo u das coodenadas uv. Calclamos o vertor bitangente como o produto vetorial dos vetores normal e tangente, para completar a base ortogonal. Isso é feito porque um ponto arbitrário na superfície, dentro de um triângulo, pode ser escrito exatamente como uma combinação linear dos vetores tangente e bitangente.

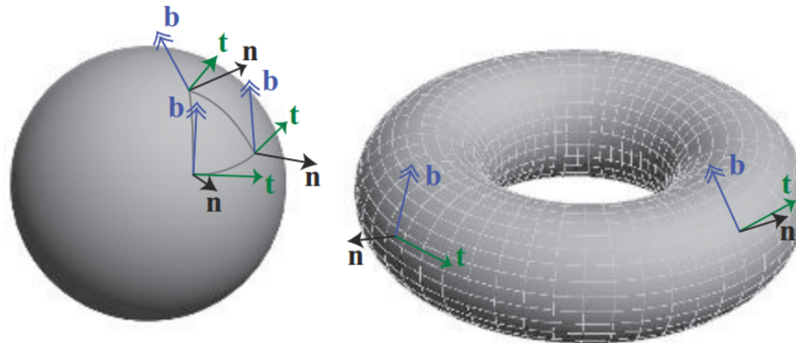


Figura 3.4: Vetores normal, tangente,e bitangente Fonte[2]

Para calcular as coordenadas usamos o seguinte método [4]

Para um vértice Q, escrevemos os outros dois vértices do triângulo, Q1,Q2 como combinação linear de T e B

$$Q_1 = s_1\mathbf{T} + t_1\mathbf{B}$$
$$Q_2 = s_2\mathbf{T} + t_2\mathbf{B}$$

Ou em forma matricial

$$\begin{pmatrix} (Q_1)_x & (Q_1)_y & (Q_1)_z \\ (Q_2)_x & (Q_2)_y & (Q_2)_z \end{pmatrix} = \begin{pmatrix} s_1 & t_1 \\ s_2 & t_2 \end{pmatrix} \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix}$$

Multiplicando pela inversa da matriz st

$$\begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix} = \frac{1}{s_1t_2 - s_2t_1} \begin{pmatrix} t_2 & -t_1 \\ -s_2 & s_1 \end{pmatrix} \begin{pmatrix} (Q_1)_x & (Q_1)_y & (Q_1)_z \\ (Q_2)_x & (Q_2)_y & (Q_2)_z \end{pmatrix}$$

Assim encontramos os vetores T e B, que são relativos ao triângulo definido por Q,Q1,Q2. Para calcular os vetores T e B para o vértice Q somente, calculamos a média de T e B para todos os triangles que Q pertence. O resultado é a seguinte matriz:

$$\begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$

que converte vetores do espaço tangente para o espaço objeto. Para realizar a conversão inversa, simplesmente usamos a inversa da matriz.

## 3.3 Steep Parallax Mapping

O método Steep Parallax Mapping é uma melhoria do parallax mapping. A ideia principal é dividir a superfície em n camadas, e escolhendo o primeiro múltiplo de P/n que atravessa a superfície, como ilustrado na figura. Importante observar que nesta figura, ao contrário das anteriores o plano da imagem é representado pela reta superior, e os valores do depth map representam profundidade e não altura, mas os princípios são os mesmos e uma simples inversão das cores da imagem converte profundidade para altura.

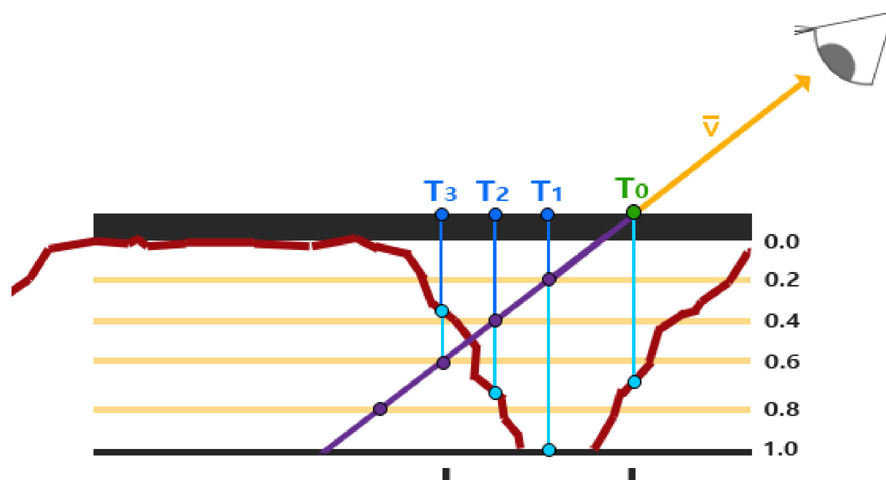


Figura 3.5: Método steep parallax mapping.

A figura mostra um exemplo utilizando o método:

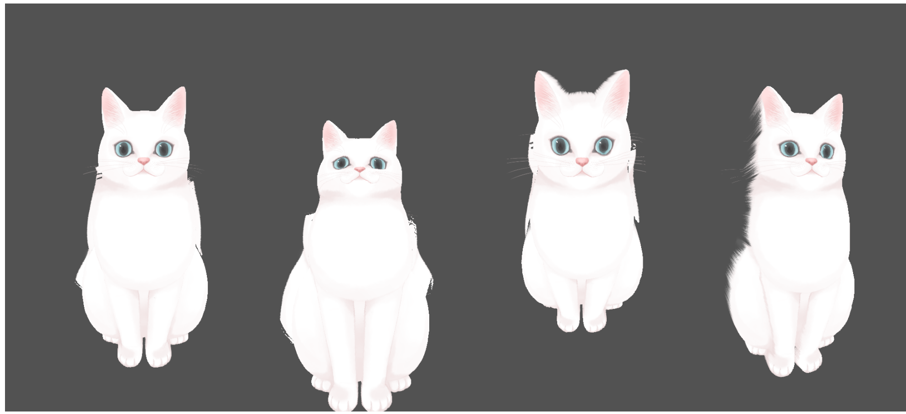

Figura 3.6: Método steep parallax mapping. Apresenta artefatos ao redor do personagem. Vídeo: https://youtu.be/-ozzzy8J094

O método apresenta uma melhoria do parallax simples, mas ainda apresenta alguns artefatos, como planos repetidos, porque vetores de ângulos diferentes podem acabar sendo projetados na mesma posição

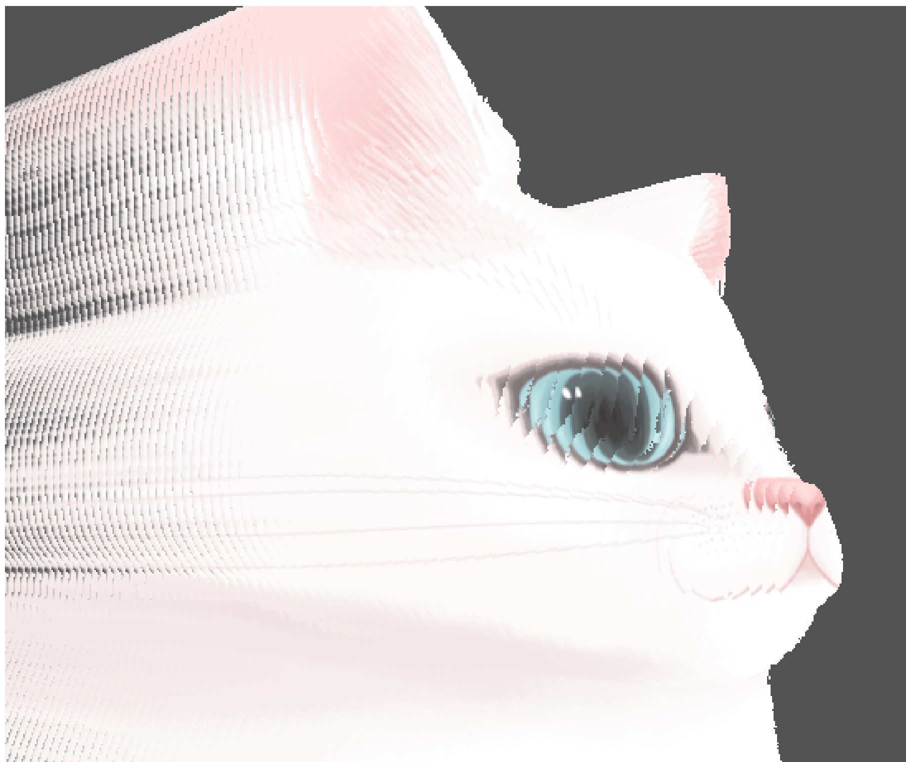

Figura 3.7: Personagem com artefatos em forma de camadas visíveis.

## 3.4 Parallax Occlusion Mapping

O método Parallax Occlusion Mapping é uma pequena melhoria no método anterior, e tenta evitar alguns artefatos. Isso é alcançado por meio de uma pequena alteração em como escolhemos as coordenadas finais. Calculamos uma média das coordenadas obtidas dos passos antes de depois de atravessar a superfície, assim conseguindo em média um valor mais próximo das coordenadas do ponto que deveria ser realmente observado.

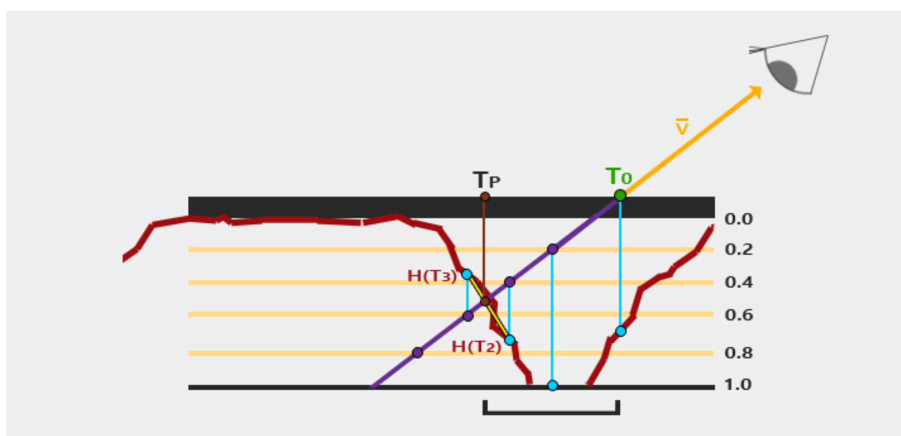


Figura 3.8: Parallax Occlusion Mapping

## 3.5 Avaliação do método

O método gera resultados interessantes para visualização e avaliação de depth maps. Para o uso em geração de animações, o próximo passo seria desenvolvimento de um método para mapear os resultados gerados pelo parallax mapping em deformações de mesh.

Porém, durante o desenvolvimento, foi observado que o método parallax gera uma quantidade muito grande de artefatos nas imagens quando rotacionadas a ângulos não muito pequenos. Assim foi preferível buscar outro método para alcançar os resultados desejados. O método ainda é útil porque funciona também em imagens que não estão divididas em camadas, assim foi essencial para determinar quais modelos de geração de depthmap são mais adequados para ilustrações e imagens não fotorrealistas, permitindo testes em quantidade maior de imagens.

## 3.6 Código

Implementação do método Parallax Occlusion Mapping em hlsl. O código foi desenvolvido e testado usando Unity, mas teoricamente funciona em qualquer engine ou framework com as alterações adequadas.

```
float3 SampleDepth(float3 pos,float2 uv)
{
 return float3(pos.x,pos.y,magnitude*tex2Dlod(heightTex, float4(uv, 0, 0)).r);
}

VertexOutput BaseVert(float3 pos, float3 normal, float4 tan,float2 uv)
{
 VertexOutput output;
 output.objPos = float4(pos,1);
 output.pos = WorldToClip(ObjectToWorld(pos.xyz));
 output.fragPos = float4(ObjectToWorld(pos.xyz),1);
 output.normal = mul((float3x3)transpose(UNITY_MATRIX_I_M), normal);
 output.uv = TRANSFORM_TEX(uv, baseTex);
 float3 T = normalize(mul(UNITY_MATRIX_M, tan.xyz));
 float3 B = normalize(mul(UNITY_MATRIX_M, cross(normal, tan.xyz)));
 float3 N = normalize(mul(UNITY_MATRIX_M, normal));
 float3x3 TBN = float3x3(T, B, N);
 output.tbn = TBN;
 return output;
}

VertexOutput vert(VertexInput input)
{
 VertexOutput output = BaseVert(input);
 float3 outPos = SampleDepth(output.objPos,output.uv);
 output.objPos = float4(outPos,1);
 return output;
}

float3 GetViewDir(VertexOutput input)
{
```

```
 float3 viewDir = normalize(viewPos - input.fragPos);
 return mul(input.tbn,viewDir);
}

float4 Parallax(VertexOutput input)
{
 float layers = 128;
 float3 viewDir = normalize(GetViewDir(input))/layers;
 float z = viewDir.z;
 float2 lastTex = input.uv;

 [loop]
 for (int i = 0; i < layers; ++i)
 {
  float2 p = viewDir*i*magnitude / (z*layers);
  p.x = -p.x;
  float2 texCoords =  input.uv + p.xy;

  if(1 - tex2D(heightTex, texCoords).r < i/layers)
  {
   return (tex2D(_MainTex,texCoords)+tex2D(_MainTex,lastTex))/2;
  }

  lastTex = texCoords;
 }

 return tex2D(_MainTex, input.uv);
}

float4 frag (VertexOutput input) : SV_TARGET
{
 return Parallax(input);
}
```

# Capítulo 4

# Suavização de meshes

Podemos também tentar modificar os vértices diretamente com os valores do depth map. Usando uma malha simples podemos evitar o problema da complexidade que tentamos evitar com método parallax mapping. Ao aplicar os valores diretamente, obtemos resultados não muito bons.

Figura 4.1: Aplicando diretamente valores do depthmap

Isso acontece porque utilizando apenas um pixel para determinar a posição de um vértice, obtemos resultados imprevisíveis com grande variação de altura. Além disso, se um vértice tem coordenadas uv localizadas no plano de fundo, ele terá um valor de profundidade baixo, levando a deformação indesejada de partes da imagem contidas no mesmo triângulo.

## 4.1 Suavização laplaciana

Uma maneira de evitar os problemas encontrados é modificar a malha gerada de modo a remover variações bruscas de altura. Isso pode ser alcançado por meio de diversos métodos de suavização de mesh. O método de suavização laplaciana é um dos mais simples, mesmo assim é altamente usado pela simplicidade de implementação e por não ser um método computacionalmente intensivo.

A ideia por trás do método é mover cada vértice de maneira que se desloque em direção a média das posições dos seus vizinhos. Considerando primeiro uma versão simplificada usando uma curva:

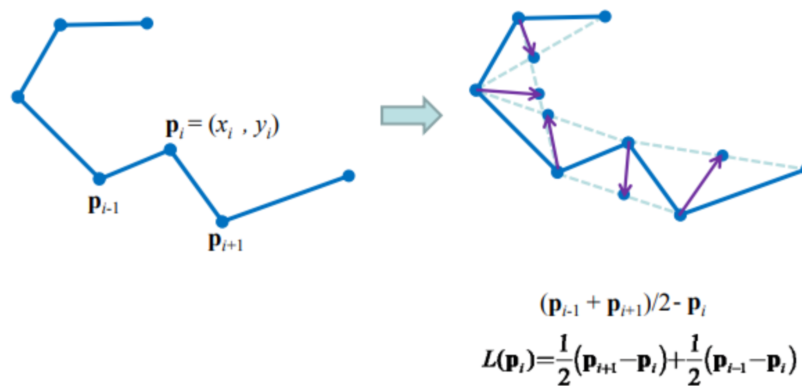


Figura 4.2: Suavização de uma curva. Fonte[5]

Assim para cada vértice $p_i$ denominamos por $L(p_i)$ o vetor que representa o deslocamento de $p_i$ até a média dos vizinhos. A cada iteração do algoritmo, deslocamos então o vértice até a média de $p_i$ e $L(p_i)$. Para permitir uma configuração em casos diferentes, podemos também realizar um deslocamento modificado por uma constante $\lambda$. Assim temos

$$\mathbf{p}_i^{(t+1)} = \mathbf{p}_i^{(t)} + \lambda L\left(\mathbf{p}_i^{(t)}\right)$$

$$L\left(\mathbf{p}_i\right) = \tfrac{1}{2}\left(\mathbf{p}_{i+1} - \mathbf{p}_i\right) + \tfrac{1}{2}\left(\mathbf{p}_{i-1} - \mathbf{p}_i\right)$$

Ou em notação matricial:

$$\mathbf{P}^{(t+1)} = \mathbf{P}^{(t)} - \lambda \mathbf{L}\mathbf{P}^{(t)}$$

$$\mathbf{P} = \begin{pmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix} \in \mathbb{R}^{n\times 2} \quad \mathbf{L} = \frac{1}{2} \begin{pmatrix} 2 & -1 & & & & -1 \\ -1 & 2 & -1 & & & \\ & & \ddots & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

Em que L é a matriz laplaciana. A matriz é constituída pela matriz de adjacência dos vértices, com o número de vizinhos na diagonal, que é sempre 1 ou 2 no caso de uma curva. Assim o algoritmo pode ser generalizado para meshes arbitrárias. Além de modificar a matriz L, a matriz P deve ser modificada para conter todos os vizinhos de cada vértice.

## 4.2 Avaliação do método suavização laplaciana

O método apresenta algumas fraquezas, como por exemplo pode remover detalhes finos, e também tende a convergir para um único ponto com número de passos elevado. Apesar desses problemas, o método resulta em uma melhora considerável dos resultados para meshes relativamente simples consideradas no trabalho.

Figura 4.3: Após realizar a suavização. Vídeo: https://youtu.be/E1mh8M-5NFM

Assim, métodos mais complexos não foram procurados, sendo que esse método obteve resultados satisfatórios.

## 4.3 Aplicação do método

Com as meshes suavizadas, cada mesh é renderizada em sequência para recriar a imagem original. Importante realizar a renderização com projeção ortográfica para que a ilustração original não seja deformada em seu estado neutro. Além da rotação, a magnitude de vértice mais distantes deve ser diminuída em quantidade proporcional ao ângulo de rotação, para simular um efeito de perspectiva mesmo no modo de projeção ortográfica.

Importante observar que nesse método é necessário que a imagem esteja dividida em camadas, já que uma rotação pode movimentar as partes superiores da imagem de modo a revelar partes inexistentes. Em geral isso representa um pequeno aumento do trabalho necessário para criação da ilustração, sendo que as partes inferiores não são exibidas com tanta frequência e podem ser menos detalhadas que outras partes da ilustração.

# Capítulo 5

# Conclusão

## 5.1 Reflexão sobre o desenvolvimento

O desenvolvimento do trabalho levou a exploração de novas áreas e técnicas tanto matemáticas quanto computacionais. Tais técnicas tem aplicações além das exibidas no trabalho. Por exemplo, um entendimento melhor sobre renderização 3D é importante não só para o desenvolvimento mas também para uso de software de modelagem 3D, com aplicações não só na indústria de entretenimento mas também em engenharia, arquitetura entre outras.

Os métodos desenvolvidos satisfazem parcialmente os objetivos iniciais, sendo que foi possível criar animações que simulam movimento tridimensional, mas uma quantidade maior do que a esperada de trabalho manual e ajustes são necessários para atingir resultados satisfatórios.

## 5.2 Comparação com animação manual

A seguinte imagem compara animação gerada automaticamente com uma animação feita manualmente. A animação automática é exibida com fundo cinza. É possível observar como a animação manual apresenta qualidade em geral melhor que a animação automática. Porém espera-se que a similaridade entre os resultados indique que as animações sirvam como base para um refinamento manual.

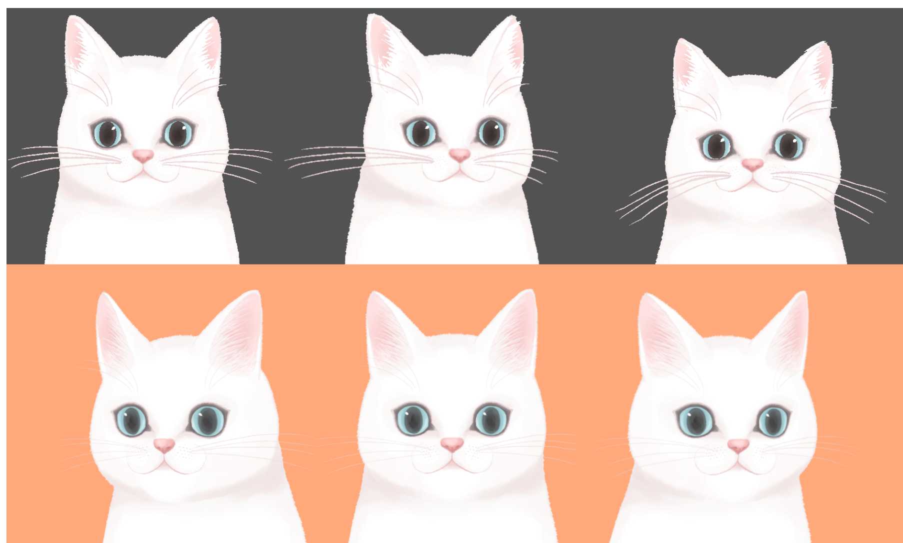

Figura 5.1: Comparação com animação feita manualmente. Vídeo: https://youtu.be/t1MnmyChvvE

## 5.3 Próximos passos

Em relação ao que pode ser feito para elevar o nível de automatização do processo, talvez o mais importante seria desenvolvimento de modelos de IA de geração de depth maps voltados especificamente para ilustrações, sendo que modelos atuais voltados para imagens fotorrealistas tem resultados imprevisíveis em ilustrações diferentes, necessitando testes com vários modelos diferentes para obter resultados satisfatórios para cada imagem.

Parte do processo que também pode ser automatizada é a divisão automática da imagem em camadas e geração automática das regiões vazias resultantes, sendo que testes realizados com modelos atuais de segmentação geraram resultados não satisfatórios.

Além disso, todas as modificações realizadas em outputs gerados pelas ferramentas

desenvolvidas para o trabalho foram feitas com simples ferramentas desenvolvidas para teste e iteração, extremamente limitadas quando comparadas com ferramentas dedicadas para este processo. A integração com ferramentas existentes será necessária para que os métodos desenvolvidos possam representar um passo em direção a automação do processo de criação de animações.

# Referências Bibliográficas

[1] https://www.the-numbers.com/movies.

[2] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A K Peters/CRC Press, 4th edition, 2018.

[3] Joey de Vries. *Learn OpenGL*. 2016.

[4] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*. Course Technology PTR, 3rd edition, 2011.

[5] Stanford University. Cs 468: Computational photography and videography (spring 2012), 2012.