UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Vitor Novaes Cantão

Uso de Métodos de Animação Procedural: Recomendações para Aplicação

Belém - PA

2021

Vitor Novaes Cantão

Uso de Métodos de Animação Procedural: Recomendações para Aplicação

Trabalho de Conclusão de Curso apresentado como um dos requisitos para a obtenção do título de Bacharel em Ciência da Computação pelo Curso de Bacharelado em Ciência da Computação da Universidade Federal do Pará.

Universidade do Pará – UFPA

Faculdade de Computação

Orientador: Prof. Dr. Sandro Ronaldo Bezerra Oliveira

Belém - PA

2021

Vitor Novaes Cantão

Uso de Métodos de Animação Procedural: Recomendações para Aplicação

Trabalho de Conclusão de Curso apresentado como um dos requisitos para a obtenção do título de Bacharel em Ciência da Computação pelo Curso de Bacharelado em Ciência da Computação da Universidade Federal do Pará.

Trabalho aprovado. Belém - PA, 21 de junho de 2021:

Prof. Dr. Sandro Ronaldo Bezerra Oliveira
Faculdade de Computação / UFPA - Orientador

Prof. MSc. Isaac Souza Elgrably
UFPA - Membro Externo

Prof. MSc. Igor Ernesto Ferreira Costa
UFPA - Membro Externo

Belém - PA
2021

Resumo

A ascensão da computação gráfica deu origem a uma nova geração de videogames, em que gráficos inigualáveis e animações meticulosas tornaram-se padrão. Além disso, com o crescimento da indústria, o interesse de novos desenvolvedores de jogos e de novas ferramentas também se expandiu. No campo da animação de videogame, as animações procedurais produzem resultados que seriam impossíveis ou extremamente custosos temporalmente, fornecendo resultados mais vívidos e conectados ao mundo do jogo. Este trabalho introduz novos desenvolvedores de jogos às técnicas de animação procedural, apresentando um guia com diferentes métodos e seções, contendo: como usar, quando aplicar e uso em cenários do mundo real. O processo consistiu na seleção das técnicas com base em critérios como relevância para o setor, qualidade de resultado e facilidade de implementação. Cada método foi estudado e aplicado, resultando nas seções do guia que contém discussões sobre as complexidades, as limitações e os pré-requisitos da implementação.

Palavras-chaves: animação procedural, recomendações para aplicação, jogos, *inverse kinematics*, unity.

Abstract

The rise of computer graphics gave birth to a new generation of videogames where unparallel graphics and meticulous animation became the standard. Furthermore, as the industry grew, the interest of new game developers and new tools have expanded as well. In the field of videogame animation, procedural animations yield results that would be otherwise impossible or extremely time-consuming, providing more vivid and game world-connected outcomes. This work introduces new game developers to procedural animation techniques, by presenting a guide with different methods and sections, containing: how to use, when to apply, and real-world scenario usage. The processes consisted of the selection of the techniques based on criteria such as industry relevance, outcome quality, and ease of implementation. Each method was studied and applied, resulting in the guide sections containing discussions about the implementation complexities, limitations, and pre-requisites.

Key-words: procedural animation, application guide, games, inverse kinematics, unity.

Sumário

1 Introdução

Neste capítulo, a introdução ao trabalho é apresentada, dando início com o contexto sob a qual o trabalho foi construído e a motivação por trás de sua realização. Posteriormente, a justificativa de tê-lo feito e os objetivos a serem alcançados através deste são apresentados. Por fim, são relatadas a metodologia do desenvolvimento deste trabalho e a estrutura do mesmo.

1.1 Contexto

Em seus 50 anos de história, os *videogames* cresceram de forma a se tornar um dos maiores meios de entretenimento (ZACKARIASSON; WILSON, 2012). Em seu primórdio, a animação em jogos consistia na troca de algumas poucas imagens 2D, conhecidas como *sprites*, em um fundo estático, especialmente pelo *hardware* extremamente limitado. A partir dos anos 90, no entanto, a geração de gráficos 3D tornou-se bem popular, impulsionada pelos gráficos impressionantes de títulos como *Super Mario 64* (WOLF, 2008).

Com o salto de 2D para 3D, as animações tornaram-se cada vez mais complexas, dado a dimensão adicional, o nível de possibilidades de deslocamento e interação num mundo 3D, e o aumento de poder computacional com o crescimento rápido de *hardware*. As expectativas para as grandes produções de jogos continuam a crescer a cada ano, com gráficos cada vez mais realistas, mundos maiores e, em especial, a qualidade da animação, que se torna cada vez mais meticulosa e próxima da realidade.

No entanto, apesar da recente democratização do desenvolvimento de jogos, principalmente pela abertura de poderosas *game engines*, como o *Unity* e *Unreal*, muitas práticas e técnicas ainda se encontram guardadas sob domínios da indústria, evidenciado pelos poucos autores veteranos que compartilham de seu conhecimento, como (GREGORY, 2009). Diante deste cenário, este trabalho visa apresentar algumas técnicas relevantes de animação procedural e democratizar seu acesso a partir de um guia de aplicação.

1.2 Motivação

A motivação deste trabalho é primariamente de cunho pessoal, nascido de um desejo de conhecer mais sobre esse tema e implementar diferentes métodos para construção de uma base de conhecimento, a fim de futuramente colocar em prática os aprendizados em produções de jogos reais.

A criação de um guia de aplicação surge como oportunidade de democratizar e

popularizar o debate sobre o tema. Espera-se, portanto, contribuir para o aprendizado de outros desenvolvedores, servindo como base para a entrada neste tema.

1.3 Justificativa

O mercado de jogos no Brasil, assim com em boa parte do mundo, está em rápida ascensão (ZACKARIASSON; WILSON, 2012). Por conta disso, é natural que haja uma busca cada vez maior por desenvolvedores de jogos. No entanto, são poucas universidades e faculdades que oferecem cursos mais específicos relacionados a *games*. Por conta disso, os profissionais desta área são, em parte, autodidatas (RUGGILL; MCALLISTER; NICHOLS, 2012), aprendendo a partir da abundante quantidade de materiais disponíveis na internet, como tutoriais, vídeos e artigos. Contudo, ainda há carência de conteúdos mais avançados, especialmente em português.

Portanto, este trabalho visa auxiliar desenvolvedores de jogos iniciantes que estejam interessados em técnicas de animação procedural, de forma a guiá-los a partir de um passo-a-passo e contribuir como uma referência futura. As escolhas dos métodos que são apresentados neste trabalho reforçam esta ideia, servindo com uma base de conhecimentos introdutórios.

1.4 Objetivos

A animação procedural permite a criação de animações que outrora seriam impossíveis de serem criadas de forma tradicional, ou extremamente inviáveis por razões temporais. Dessa forma, sua utilização na indústria dos *videogames* permite aumentar ainda mais a qualidade, o realismo, e interação com o mundo das grandes produções. Ademais, sua utilização por desenvolvedores independentes ainda implica em um grande benefício para esta categoria: a redução de esforço na criação de conteúdo.

Por conta disso, este trabalho tem como objetivo geral de fornecer um guia para a implementação de técnicas de animação procedural, com maior foco em animação de personagens.

Como objetivos específicos destacam-se:

1. Introduzir os conceitos de animação procedural, como base para aprendizados de múltiplas técnicas;
2. Identificar métodos de animação procedural que possuem relevância, bons resultados práticos e nível de complexidade relativamente baixo;

3. Explorar as técnicas de forma a guiar o leitor a entendê-las de forma prática, com a construção do pensamento por trás de sua aplicação;

4. Servir como um guia de referência para consultas futuras.

1.5 Metodologia

Como primeiro passo, foi feita uma pesquisa para identificar técnicas de animação procedural baseada em três critérios principais: relevância dentro da indústria e utilização em jogos, isto é, a técnica deve ter um alto grau de aplicabilidade; retorno significativo, isto é, dado seu grau de complexidade, espera-se uma notável melhora na animação; e facilidade de implementação, uma vez que o guia está estruturado em formato passo-a-passo, voltado para um leitor iniciante no tema.

Após isso, dentre as técnicas encontradas, as principais foram selecionadas para serem apresentadas neste trabalho. Com isso, para cada técnica escolhida, foi realizada uma nova pesquisa mais detalhada para ser implementada dentro do *Unity Engine*. Este passo de implementação da técnica foi fundamental não somente para um maior entendimento, mas para entender os prós e contras, bem como as dificuldades de implementação específicas das ferramentas utilizadas, o que veio a se tornar seções dentro do guia de aplicação.

Posteriormente, foi elaborada a estrutura das recomendações para aplicação, com base nas lições aprendidas da pesquisa e implementação. Por fim, o esboço para o guia passo-a-passo foi escrito.

1.6 Estrutura do Trabalho

O trabalho está estruturado da seguinte forma:

- Capítulo 1 - Introdução do trabalho. Apresenta o contexto em que o trabalho foi feito, sua motivação, os objetivos do mesmo e a metodologia para sua construção;
- Capítulo 2 - Fundamentação teórica por trás deste trabalho. Apresenta os conceitos fundamentais, sendo essencial para o entendimento e a aplicação das técnicas no capítulo posterior;
- Capítulo 3 - Apresenta as recomendações de aplicação dos métodos de animação procedural, incluindo: múltiplas técnicas de *Inverse Kinematics* e animação por Interpolação com *Bleendtree* em tempo real;
- Capítulo 4 - Conclusão do trabalho. Apresenta a visão geral do trabalho, contribuições do mesmo, limitações e trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo, a fundamentação teórica necessária para a aplicação dos métodos de animação procedural é apresentada. Primeiramente, discute-se os conceitos envolvidos no termo *animação procedural*. Após isso, são explanados os conceitos relacionados aos métodos aplicados. Em seguida, a ferramenta utilizada na aplicação é introduzida, bem como os principais termos relacionados a ela. Por fim, alguns conceitos matemáticos que serão amplamente utilizados são comentados.

2.1 Animação Procedural

Tradicionalmente, segundo (BRUDELIN, 1996), o processo de animação é uma tarefa tediosa, que envolve especificar inúmeros *key frames* (figura 1) para inúmeros graus de liberdade, a fim de obter uma movimentação desejada. Os métodos de *keyframing* proveem ferramentas para manipular ângulos de articulação apenas no nível mais baixo, deixando ao animador a tarefa de explicitamente considerar interações de alto nível baseado em sua experiência e habilidade.

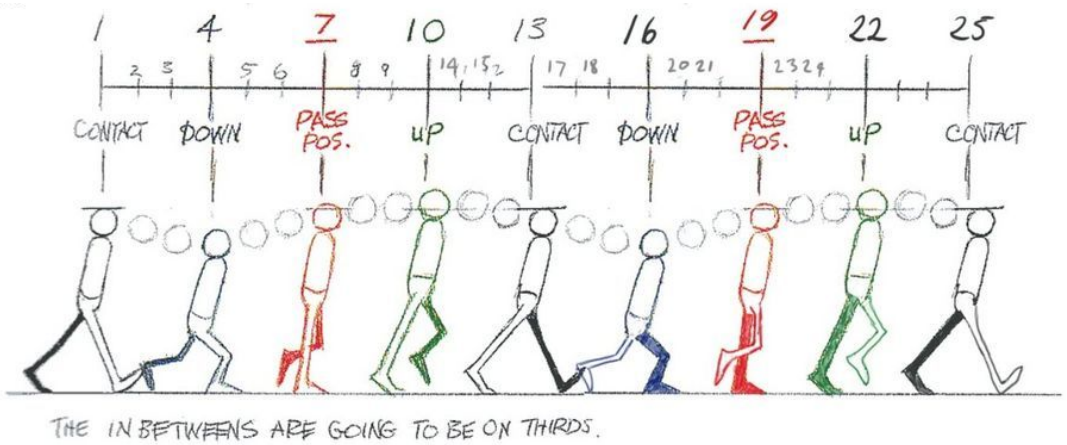

Figura 1 – *Keyframing* de um ciclo de caminhada (WILLIAMS, 2009).

Muitas vezes associada com o termo "geração aleatória", a geração procedural de conteúdo é um contraste ao tradicional método de criação de conteúdo, em que o artista manualmente fabrica cada *asset* do jogo. Tipicamente, a geração procedural envolve uma combinação de conteúdos produzidos por um artista, para se ter um maior controle, juntamente com uso de algoritmos e pseudo-aleatoriedade. A origem dessa técnica nos jogos deu-se nos gêneros de *Role Playing Game (RPG)* e *Roguelikes* (BROWN; SCIREA, 2020). A figura 2 exemplifica um mapa gerado de forma procedural no jogo *Rim World*.

Figura 2 – Exemplo de geração procedural. Mapa de jogo gerado aleatoriamente no jogo *Rim World*.

Dessa forma, entende-se o termo "animação procedural" como um tipo de animação digital gerada em tempo-real. Sua aplicação, dentro dos jogos, expande-se desde animação de personagens (como no jogo *Rain World* ilustrado na figura 3) a sistemas de partícula (como neve, fogo, fumaça), simulação de pano (como bandeiras e roupas), e física de ragdoll, que busca simular um personagem sem vida, colidindo com objetos e a cena de maneira natural.

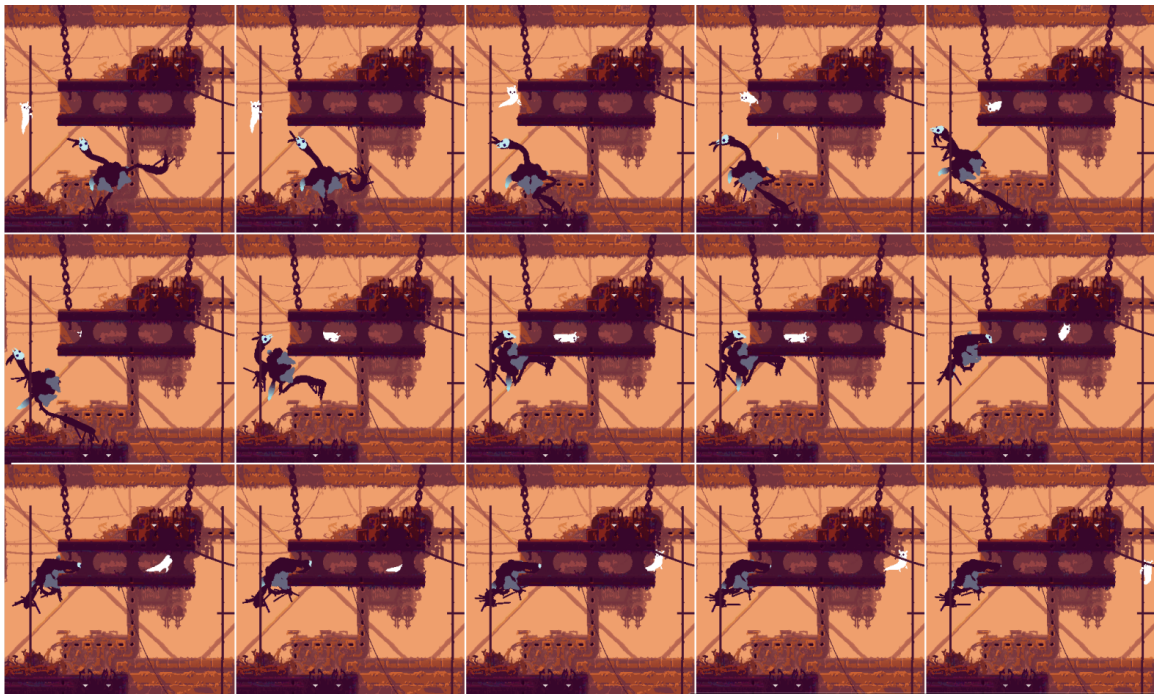

Figura 3 – Jogo *Rain World* que utiliza animação procedural para dar vida aos personagens de forma a reagir ao terreno.

2.2 Conceitos sobre os Métodos Analisados

As subseções a seguir apresentarão os conceitos dos métodos analisados nesta pesquisa de trabalho de conclusão de curso.

2.2.1 Forward Kinematics

Antes de iniciar a discussão sobre *Forward Kinematics* e *Inverse Kinematics*, é importante esclarecer o uso de alguns termos, uma vez que o conceito de ambos estão relacionados ao seu uso na robótica, além da animação, a saber:

- **Joint**: Também conhecido como eixos (*axis* na figura 4), os *joints* (articulações) são partes movíveis de um robô, ou modelo 3D, que causam movimento entre dois *links*;
- **Link**: Parte inflexível que conecta dois *joints*. Na animação, também é conhecido como **bone**;
- **Kinematic Chain**: Conjunto de *joints* e seus *links* que compõem um sistema;
- **Base**: Parte imóvel que dá início ao *kinematic chain*;
- **End-Effector**: A última parte de um *kinematic chain*. Dentro da animação de personagens, o *end-effector* é comumente a mão ou o pé do modelo.

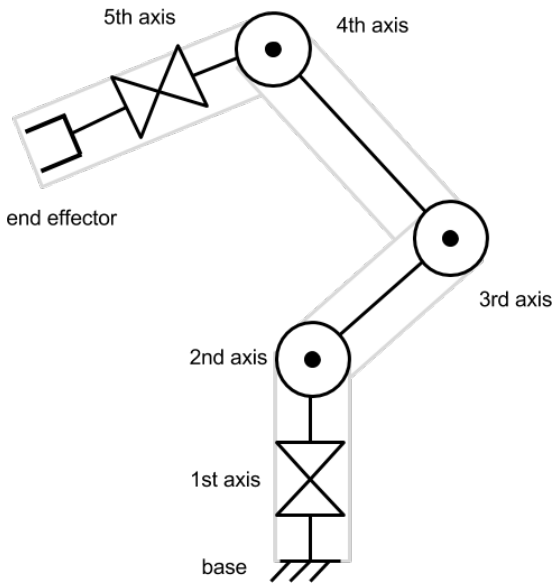

Figura 4 – Esquemática de braço robótico

De acordo com (BEGGS, 1983), a cinemática é o estudo da geometria do movimento. É uma área da física que estuda o movimento dos corpos e objetos sem considerar as forças que atuaram no objeto para que este se movesse.

O *Forward Kinematics* (FK) por sua vez, é uso da cinemática para computar a posição do *end-effector* com base nos parâmetros de sua corrente cinemática. O conceito de *FK* pode ser intuitivo aos animadores e aqueles familiarizados com *Game Engines*, uma vez que ao mover e rotacionar objetos que possuem relação de parentesco, o resultado é efetivamente a aplicação do conceito de *FK*, como exemplificado na figura 5. Note que após aplicada a rotação, todos os objetos filhos têm sua posição e rotação calculada a partir da rotação e posição aplicada ao seu ancestral.

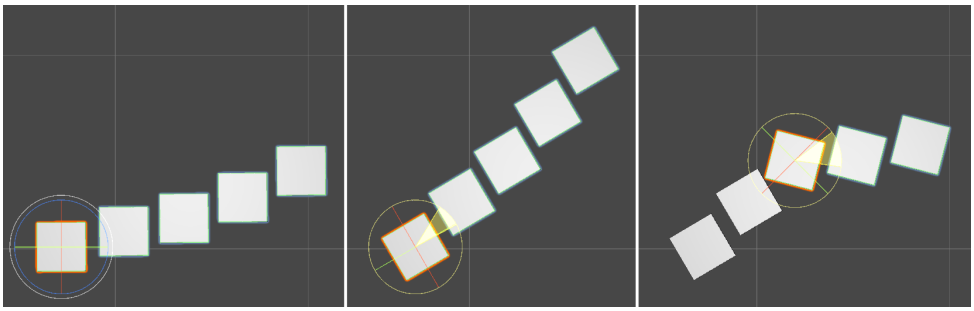

Figura 5 – Rotação de objetos em relação de parentesco (Unity).

2.2.2 Inverse Kinematics (IK)

Ao entender o conceito de *Forward Kinematics*, o *Inverse Kinematics* pode ser facilmente entendido como o processo inverso (KUCUK; BINGUL, 2006): no *FK*, temos como resultado a posição do *end-effector* a partir dos ângulos dos *joints*; já no *IK*, dado uma coordenada em que se deseja posicionar o *end-effector*, temos como resultado os ângulos dos *joints* da cadeia cinemática. Esse conceito está ilustrado na figura 6.

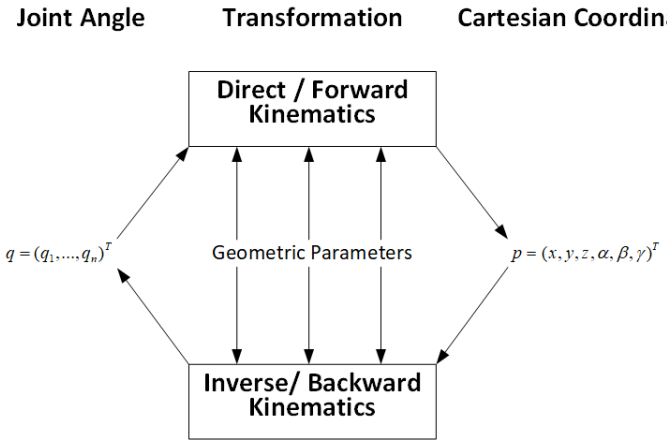

Figura 6 – Comparação entre *Forward Kinematics* e *Inverse Kinematics*.

O método de *IK* é muito utilizado em jogos e em animação 3D para conectar personagens fisicamente ao mundo. Um exemplo disso é o ajuste do modelo para que esteja realisticamente sobre o terreno. A figura 3 utilizou-se dessa técnica.

Há diversas soluções para o problema de *inverse kinematics*, desde soluções analíticas a soluções numéricas. Durante a aplicação da técnica de *IK*, será utilizado uma solução heurística conhecida como *FABRIK* (ARISTIDOU; LASENBY, 2011). A teoria e aplicação passo-a-passo será explicada na seção referente.

2.3 Unity

Unity é uma *Engine 3D* para a criação de *games* e produtos, sendo até mesmo adotada por indústrias fora do ramo de videogames, como arquitetura, engenharia, filmes e automobilísticos. Por se tratar de uma *Engine* grátis, poderosa e de fácil acesso, com centenas de conteúdos educativos, sua escolha torna-se ideal para a aplicação das técnicas introdutórias de animação procedural.

Como veremos no capítulo 3, há para cada método uma subseção contendo pré-requisitos para implementação da técnica, dessa forma, é esperado do leitor um certo grau de conhecimento e experiência com o *Unity*, variando de acordo com o método abordado. No entanto, há de serem esclarecido os termos utilizados referentes ao editor *Unity*, uma vez que serão utilizados exaustivamente durante a aplicação dos métodos.

2.3.1 Janelas do editor Unity

Esta subseção terá a finalidade de apresentar o editor da *engine* Unity (figura 7), a partir de uma explicação de cada item que compõe este editor a seguir:

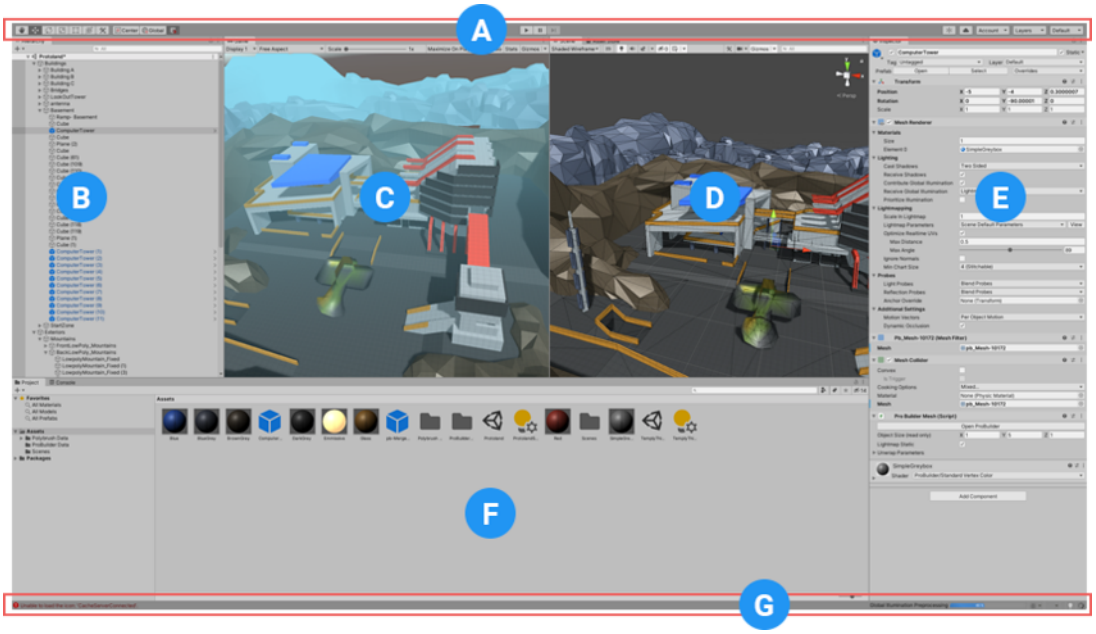

Figura 7 – As janelas mais comuns (em suas posições padrões) no editor Unity (UNITY'S..., b).

A. O *Toolbar*, localizado na parte superior, contém ferramentas para manipular os gameobjects (melhor detalhados na subseção 2.3.2) contidos na janela *Scene* (D), além de botões de *play* (para dar início), *pause* e *step* (para avançar um frame);

B. O *Hierarchy*, localizado à esquerda, é uma representação hierárquica de cada *gameObject* presente na cena (Scene). Revela a relação entre *gameObject* que, por ser possível a relação de pai-filho, está relacionado com o conceito de *Forward Kinematics*, visto na seção 2.2.1;

C. O *Game View* simula o visual renderizado do jogo através da câmera;

D. A *Scene View* permite navegar visualmente pela cena do jogo, servindo de palco para os *gameObjects*;

E. O *Inspector* permite inspecionar os detalhes de um *gameObject* selecionado, bem como modificar valores e adicionar/remover propriedades e scripts;

F. A janela *Project* armazena a biblioteca de *assets* e scripts.

2.3.2 Gameobject e outros termos importantes

Nesta seção são abordados os principais termos (UNITY'S. . . , a) relacionados ao Unity que serão utilizados durante as recomendações de uso de métodos de animação procededural (visto no Capítulo **??**).

- **Gameobjects**: São os objetos fundamentais do Unity, que podem representar personagens, câmeras, cenários, etc. Sua funcionalidade é composta por seus componentes (scripts, por exemplo);
- **Transform**: Componente de um *gameObject* que permite a manipulação no espaço, com sua posição, rotação e escala, além de sua relação com os outros *transforms*;
- **Vector**: Classes para expressar e manipular vetores (ver seção 2.4.2);
- **Quaternion**: Classe que provê métodos de criação e manipulação de quatérnios (ver seção 2.4.3), representando rotação absolutas ou relativas;
- **Collider**: Componente invisível que define a forma de um *gameObject* em termos de colisão. É essencial para que haja detecção de colisões;
- **Rigidbody**: Componente que permite ao *gameObject* agir sob o controle da física, sendo capaz de receber forças de forma a mover-se realisticamente.

2.4 Conceitos Matemáticos

As subseções a seguir apresentarão aos conceitos matemáticos para entendimento dos métodos analisados nesta pesquisa de trabalho de conclusão de curso.

2.4.1 Interpolação

Na matemática, a interpolação é a determinação ou estimativa de um valor de $f(x)$ ou uma função de x, para certos valores conhecidos da função. Caso $x_0 < ... < x_n$ e $y_0 = f(x_0), ..., y_n = f(x_n)$ sejam conhecidos, e se $x_0 < x < x_n$, então o valor estimado de $f(x)$ é dito ser uma interpolação. Em outras palavras, a interpolação é um tipo de estimação, em que novos dados são gerados a partir de uma série de pontos discretos conhecidos (PRESS et al., 1986).

Há diferentes métodos de interpolação, diferindo no grau de complexidade, velocidade, acurácia, suavidade, etc. A seguir, os principais métodos utilizados são comentados com base nos dados da figura 8.

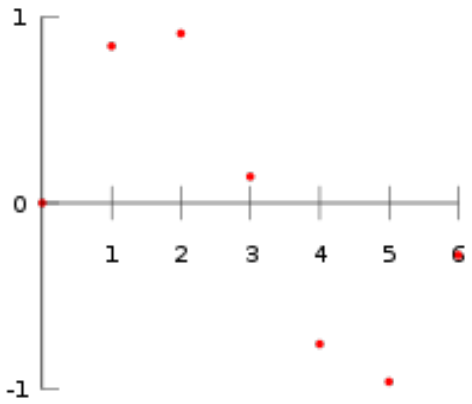

Figura 8 – Plotagem dos dados a serem interpolados.

2.4.1.0.1 Linear

A interpolação linear é uma das mais utilizadas nos jogos e animações, também conhecida como *LERP*. A imagem 9 ilustra o resultado gráfico da interpolação Matematicamente, o interpolante linear é uma linha reta entre dois pontos.

2.4.1.0.2 Polinomial

Como uma generalização da interpolação linear, a interpolação polinominal utiliza uma função polinomial de menor grau que passa pelos pontos dos dados. O resultado de uma interpolação polinomial é bem mais suave se comparada à linear, e pode ser utilizada

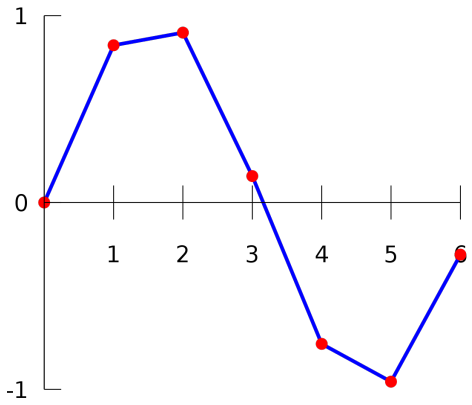

Figura 9 – As janelas mais comuns (em suas posições padrões) no editor Unity (UNITY'S..., b).

na animação quando deseja-se movimentos menos "robotizados". A figura 10 ilustra o resultado gráfico.

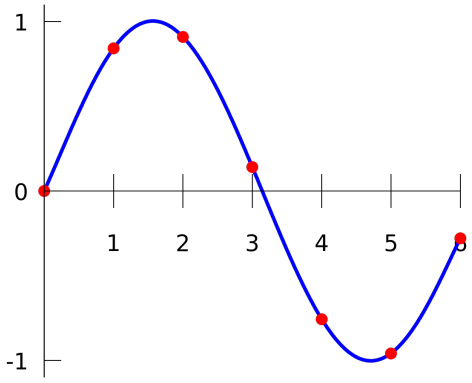

Figura 10 – Exemplo gráfico de adição de vetores.

2.4.2 Vetores

Segundo (HALLIDAY; RESNICK; WALKER, 2010), um vetor possui magnitude (comprimento) assim como direção, seguindo certas regras de combinação. Geometricamente, vetores são representados por uma seta, em que sua direção é igual a sua quantidade, e seu comprimento é proporcional à magnitude.

Em jogos, assim como na *Unity Engine*, os vetores são fundamentais para descrever propriedades, como a posição de personagens, velocidade de um carro em movimento, ou distância entre objetos.

A seguir, serão abordados alguns dos principais conceitos e aritméticas relacionados a vetores, que serão usados extensivamente durante a aplicação das técnicas de animação procedural.

2.4.2.0.1 Componentes

Segundo (HALLIDAY; RESNICK; WALKER, 2010), O componente de um vetor é sua projeção sob um eixo. Em três dimensões, um vetor pode possuir três componentes, de forma a pensar que cada componente exerce influência em uma direção, como pode ser visto na figura 11.

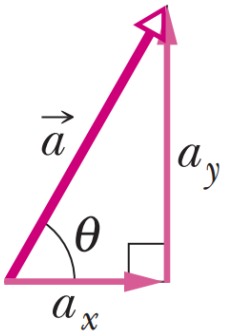

Figura 11 – Vetor $\vec{a}$ com seus componentes a_x e a_y.

No *Unity*, há diversas classes de vetores que diferem na quantidade de seus componentes. *Vetor2* possui dois componentes (x, y), e de forma semelhante o *Vector3* possui três componentes (x, y, z).

2.4.2.0.2 Vetor unitário

Um vetor unitário é um vetor que possui magnitude exatamente igual a 1, não possuindo dimensão e unidade. Seu propósito é unicamente especificar uma direção (HALLIDAY; RESNICK; WALKER, 2010). No *Unity*, podemos transformar um vetor em um vetor unitário através do processo de normalização.

2.4.2.0.3 Adição

Matematicamente, a soma vetorial é a soma de seus componentes, sendo uma operação comutativa e associativa. Geometricamente (como visto na figura 12), entende-se a adição como um deslocamento sucessivo dos vetores.

2.4.2.0.4 Subtração

De maneira semelhante à adição, na subtração a operação é realizada para cada componente. Isto é, dado dois vetores de dois componentes, $\vec{a}$ e $\vec{b}$, o resultado de $\vec{a} - \vec{b}$ será um noto vetor de componentes: $a_x - b_y$ e $a_y - b_y$. Importante notar que, diferentemente da adição, a subtração vetorial não é comutativa, como pode ser visto na figura 13.

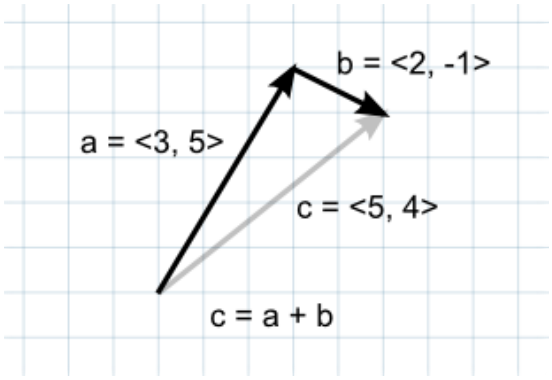

Figura 12 – Exemplo geométrico de adição de vetores (IMPORTANT...,).

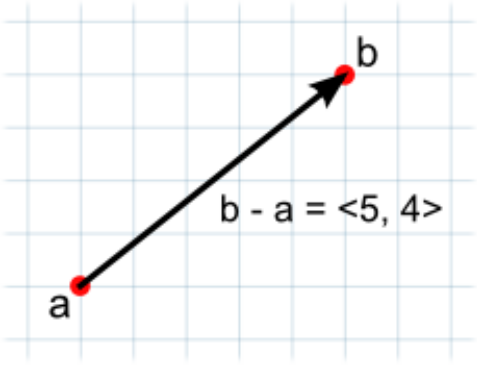

Figura 13 – Exemplo geométrico de subtração de vetores (IMPORTANT...,).

2.4.2.0.5 Produto escalar

Diferentemente das outras operações entre vetores, o produto escalar, como o nome implica, retorna um escalar. É possível entendê-lo como um valor que representa o crescimento direcional do vetor original, como pode ser visto na figura 14. Matematicamente, temos: $\vec{a} \cdot \vec{b} = ab\cos\theta$.

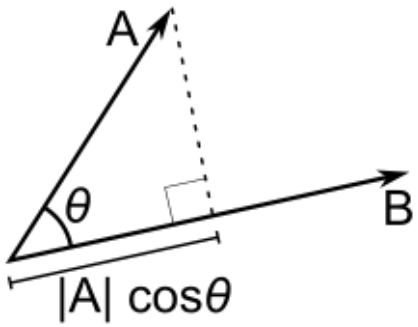

Figura 14 – Exemplo geométrico do produto escalar.

2.4.2.0.6 Produto vetorial

O produto vetorial é definido apenas em vetores de três dimensões. De forma geral, o produto vetorial de dois vetores $\vec{a}$ e $\vec{b}$ retorna um vetor $\vec{c}$ perpendicular aos dois vetores originais, dado uma direção, como pode ser visto na figura 15. Matematicamente temos: $\vec{a} \times \vec{b} = \|a\|\|b\| \sin \theta n$.

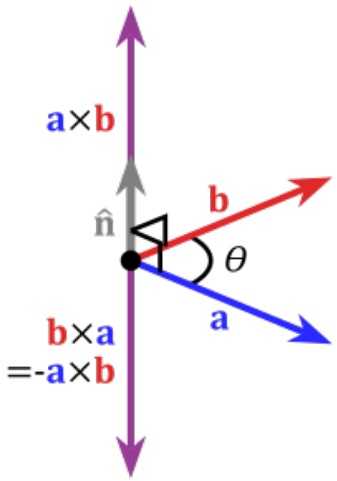

Figura 15 – Exemplo geométrico de um produto vetorial.

2.4.3 Quatérnios

No campo da computação gráfica, os *Quaternions* possuem três aplicações principais (GOLDMAN, 2011):

1. Reduzir espaço e aumentar a velocidade da cálculos envolvendo rotações. Em contraste à matriz de rotações (3×3), quatérnios são representados por apenas quatro escalares;
2. Evitar distorções que ocorrem por inexatidão de ponto flutuante envolvendo rotações, uma vez que quatérnios podem ser facilmente renormalizados;
3. Em animação de *key frames*, é possível interpolar suavemente entre duas rotações representadas por unidade de quatérnios com uma interpolação linear esféria (*SLERP*).

Baseado em números complexos, os quatérnios não são um conceito fácil de entender intuitivamente. Além disso, seu uso em jogos não requer um entendimento matemático, e casos em que se modificam os componentes individuais dos quatérnios são incomuns. Na maior parte, trabalha-se com rotações já existentes, utilizando-as para construir novas rotações, como pode ser visto na figura 16.

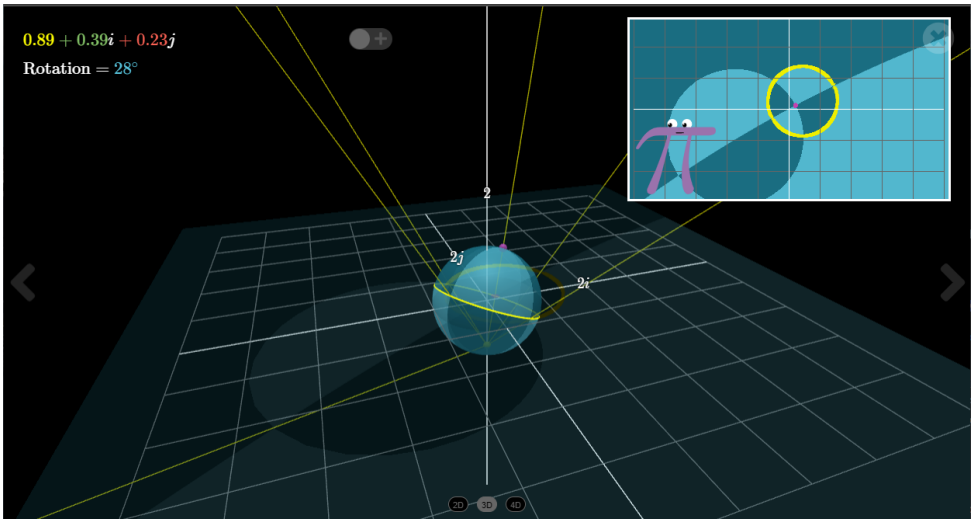

Figura 16 – Visualização 3D de um *Quaternion* (SANDERSON; EATER,).

O *Unity* provê muitos métodos que facilitam e abstraem a complexidade de quatérnios. É possível, por exemplo, trabalhar diretamente com **ângulos de Euler**. Neste caso, se for de interesse girar um objeto por trinta graus no eixo x, basta adicionar $x + 30$. A conversão para quatérnios é feita pela *engine*.

3 Recomendações de Uso de Métodos de Animação Procedural

Este capítulo será dedicado a apresentar o guia de recomendações de uso dos métodos para a implementação de animação procedural. Desta forma, cada método está estruturado em oito seções principais, a saber:

1. **O que é**: Nesta seção, a técnica e seus conceitos básicos são introduzidos ao leitor;
2. **Quando**: Ilustra cenários em que a técnica pode ser aplicada, contrastando com métodos tradicionais de animação;
3. **Como**: A aplicação teórica é discutida e o algoritmo apresentado. É fundamental para a aplicação prática a ser descrita na seção 6 desta estrutura;
4. **Complexidade de Implementação**: Fornece ao leitor uma breve discussão acerca da complexidade de implementação do método;
5. **Pré-requisitos**: Esta seção lista os requisitos necessários para que se possa iniciar a aplicação da técnica. Espera-se que o leitor tenha um certo conhecimento prévio, dependendo do método a ser apresentado;
6. **Exemplo de aplicação**: Conduz o leitor a um passo-a-passo da aplicação da técnica. Inicialmente, descreve os preparativos no *Unity* e, posteriormente, o *scripting*;
7. **Uso Real**: Ilustra um cenário de uso real em que a técnica é aplicada;
8. **Limitações**: Esclarece as limitações da técnica aplicada.

3.1 Escolha dos Métodos

Como discutido na seção 2.1, há inúmeras áreas para aplicação da animação procedural em jogos. Por conta disso, é natural que haja uma vasta gama de técnicas. Os métodos que serão apresentados foram escolhidos com base em dois critérios principais: i) Relevância, em outras palavras, sua aplicabilidade dentro da indústria e efetividade do método; e ii) Facilidade de implementação, por se tratar de um passo-a-passo, técnicas mais complexas exigiriam um conhecimento mais elevado do leitor.

3.2 FABRIK

As subseções a seguir tratarão sobre as orientações de uso do método FABRIK.

3.2.1 O que é?

Como em diversos problemas computacionais, há inúmeras soluções para o Inverse Kinematics, sendo não apenas importante a corretude do problema, mas também sua complexidade temporal e facilidade de implementação.

O *Forward And Backward Reaching Inverse Kinematics* (FABRIK), é um método heurístico para resolução do problema de Inverse Kinematics, apresentado por Andreas Aristidou em seu artigo (ARISTIDOU; LASENBY, 2011).

A ideia do algoritmo, que será explorada mais a fundo na seção 3.2.3, é tratar o problema de achar a posição de cada *joint* como um problema de achar um ponto em um uma linha.

3.2.2 Quando?

O método FABRIK pode ser utilizado para grande parte dos problemas envolvendo Inverse Kinematics. Não obstante, é uma solução eficiente, como pode ser visto na comparação a outros métodos na figura 17, e que produz resultados visuais realísticos, na comparação da figura 18.

	Number of Iterations	Matlab exe. time (*sec*)	Time per iteration (in *msec*)	Iterations per second
FABRIK	15.461	0.01328	0.86	1164
CCD	26.308	0.12356	4.69	213
Jacobian Transpose	1311.190	12.98947	9.90	101
Jacobian DLS	998.648	10.48051	10.49	95
Jacobian SVD-DLS	808.797	9.29652	11.50	87
FTL	21.125	0.02045	0.97	1033
Triangulation	1.000	0.05747	57.47	21

Figura 17 – Média de resultados para um único kinematic chain com 10 joints. Adaptado de “FABRIK: A fast, iterative solver for the Inverse Kinematics problem”, por Aristidou, Andreas.

3.2.3 Como?

Como o nome implica, o FABRIK é dividido em duas partes principais: a etapa Forward, e a etapa Backward. No entanto, é possível pular o algoritmo e chegar a um resultado rapidamente caso o alvo esteja fora do alcance da cadeia de joints.

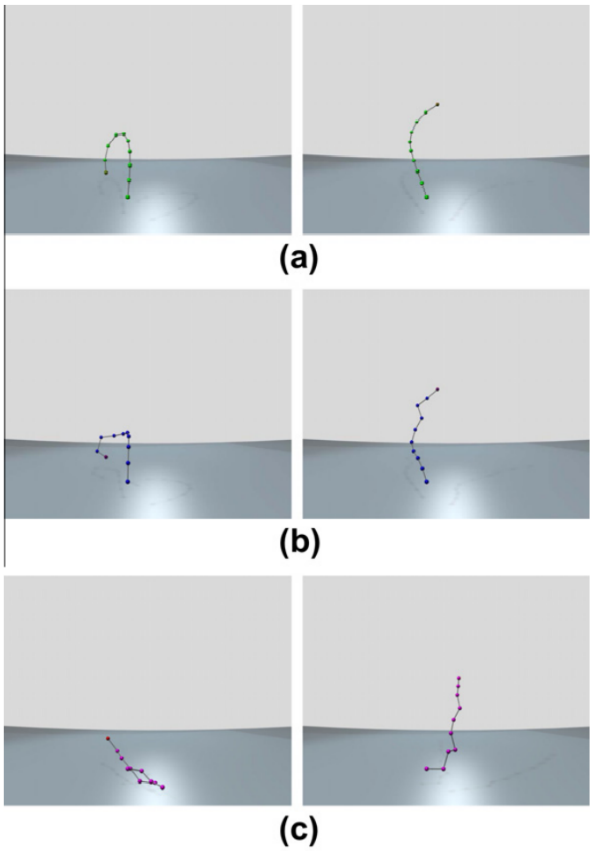

Figura 18 – a) FABRIK, b) CCD, c) J. DLS. Adaptado de "FABRIK: A fast, iterative solver for the Inverse Kinematics problem", por Aristidou, Andreas.

Para verificar se é possível alcançar o alvo, basta somar o comprimento dos Links, e caso o resultado seja menor que a distância do Início ao Alvo, traça-se uma linha imaginária do ponto inicial ao Alvo. Movemos então cada Joint para a linha, preservando o comprimento de cada link. Como resultado, teremos o end-effector o mais próximo possível do alvo em apenas uma iteração. Este processo é demonstrado na figura 19.

Caso contrário, o alvo está dentro do alcance do *end-effector*. Iniciaremos as etapas principais do FABRIK a serem discutidas a seguir.

3.2.3.1 Etapa Forward

Nesta etapa, o objetivo consiste em avançar o end-effector até o alvo, porém sem alterar o comprimento dos links. Para isso, movemos o end-effector (P4) para a posição do alvo e traçamos uma linha de sua nova posição até o Joint anterior (P3), como mostra os quadros 1 e 2, respectivamente, da figura 20.

Movemos P3 ao longo desta linha de forma que que o comprimento entre P3 e P4 (d3) mantenha-se o mesmo, dando origem ao ponto P3', como mostra o quadro 3 da figura 20. Com P3' em sua posição, faremos o mesmo com o Joint anterior (P2), traçaremos uma linha entre P3' e P2, deslocando P2 na linha de forma a manter seu comprimento (d2). Da

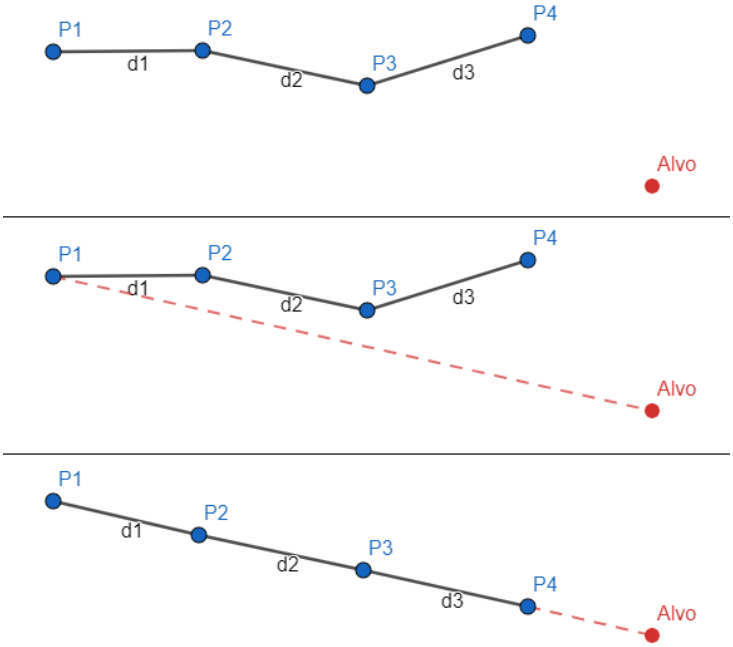

Figura 19 – Demonstração visual do algoritmo caso o alvo esteja fora do alcance.

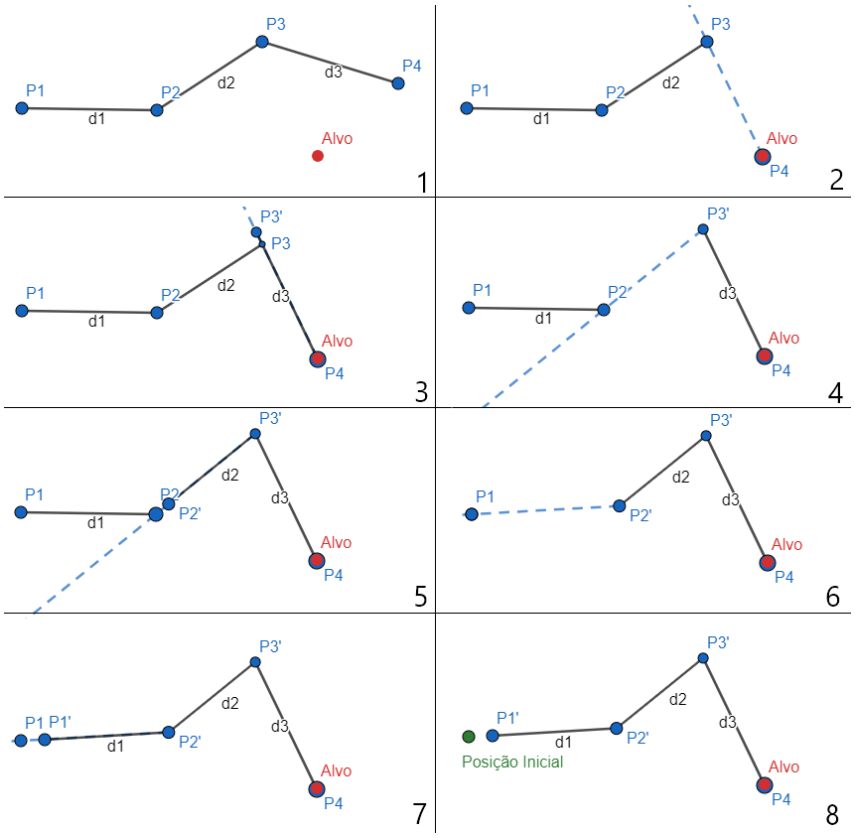

Figura 20 – Demonstração visual do algoritmo na etapa *Forward*.

mesma forma, esse processo é feito até que todos os links mantenham a distância correta.

O resultado final da etapa Forward pode ser visto no quadro 8 da figura 20. Note

que P1' não se encontra mais em sua posição inicial. Imaginando que o *joint* raíz (P1) é a base de um braço robótico, não seria correto deslocarmos esse ponto de sua posição inicial. Para resolver esse problema, veremos a etapa Backward.

3.2.3.2 Etapa Backward

Nesta etapa, o objetivo consiste em retornar á base até o ponto inicial, de forma muito semelhante à etapa anterior, porém de forma inversa. Começamos movendo a base (P1') para a posição inicial e traçamos uma linha de sua nova posição até o Joint posterior (P2'), como mostra o quadro 2 da figura 21.

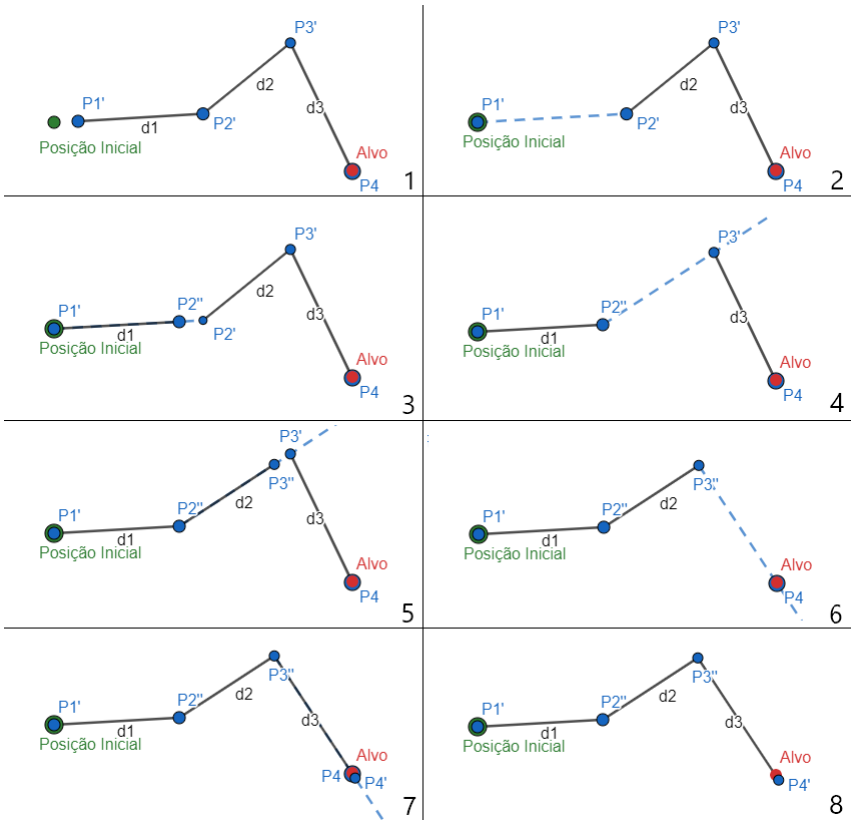

Figura 21 – Demonstração visual do algoritmo na etapa *Backward*.

Movemos P2 ao longo desta linha de forma que que o comprimento entre P1' e P2' (d1) mantenha-se o mesmo, dando origem ao ponto P2", como mostra o quadro 3 da figura 21. Com P2" em sua posição, faremos o mesmo com o Joint posterior (P3'), traçaremos uma linha entre P2' e P3", deslocando P3' na linha de forma a manter seu comprimento (d2). Da mesma forma, esse processo é feito até que todos os links mantenham a distância correta, como na etapa anterior.

O fim desta etapa marca o término de uma iteração do algoritmo FABRIK, fazendo com o que o end-effector esteja mais próximo do Alvo, como pode ser visto no quadro 8 da figura 21. Cada iteração subsequente do algoritmo reduzirá a distância do alvo, de

forma que ao definir uma margem de erro, é possível chegar a um bom resultado com relativamente poucas iterações.

3.2.3.3 Pseudo-código

Seguindo a lógica descrita anteriormente, apresentamos o pseudo-código da figura 22.

```
input  : As posições do joints p_i para i = 1, ..., n,
         a posição do alvo t e
         a distancia entra cada joint d_i = |p_{i+1} − p_i| para i = 1, ..., n.
output: As novas posições dos joints p_i para i = 1, ..., n − 1.
initialization;
if target is in range then
    for i ← 1 to maxIteration by 1 do
        if distance to goal < offset then
            break;
        else
            forwardReach();
            backwardReach();
        end
    end
else
    stretchToTarget();
end
```

Figura 22 – Pseudocódigo da iteração completa do algoritmo FABRIK.

Caso o alvo esteja dentro do alcance, faremos as etapas forward e backward até que o número máximo de iterações seja atingido, ou o end-effector tenha alcançado o alvo. Caso contrário, utilizaremos o método stretchToTarget, seguindo a ideia apresentada na figura 19.

O método stretchToTarget pode ser visto com mais detalhes no pseudo-código da figura 23. Sendo r a distância entre o alvo t e a posição inicial da base s. Iteramos, a partir do segundo joint, e atribuímos sua posição á posição do joint anterior em direção ao alvo t, mantendo a distância d do link.

$$\mathbf{r} = \|t - s\|$$
$$\textbf{for } i \leftarrow 1 \textbf{ to } n \textit{ by } 1 \textbf{ do}$$
$$\quad \mathbf{p}_i = p_{i-1} + (r * d_{i-1})$$
$$\textbf{end}$$

Figura 23 – Pseudocódigo do método *stretchToTarget*.

O método forwardReach e backwardReach também seguem a mesma ideia apresentada anteriormente. No pseudo-código da figura 24, igualamos o end-effector p_n a posição do alvo t. Com isso, iteramos do penúltimo *joint* ao joint raíz, atribuindo ao joint anterior p_{i-1} a soma da posição do joint atual p_i, e a direção r_i mantendo a distância do link d_{i1} através da multiplicação da distância.

$$
\begin{aligned}
&\mathrm{p}_n = t \\
&\textbf{for } i \leftarrow n-1 \textbf{ to } 1 \; by \; -1 \textbf{ do} \\
&\quad \mathrm{r}_i = |p_{i-1} - p_i| \\
&\quad \mathrm{p}_{i-1} = p_i + (r_i * d_{i-1}) \\
&\textbf{end}
\end{aligned}
$$

Figura 24 – Pseudocódigo do método *forwardReach*.

De forma semelhante, no método backwardReach apresentado no pseudo-código da figura 25, igualamos o joint raíz p_1 a posição do inicial s. Com isso, iteramos em ordem crescente até o penultimo joint $p_{n-}2$, atribuindo ao joint posterior p_{i+1} a soma da posição do joint atual p_i, e a direção r_i mantendo a distância do link d_i através da multiplicação da distância.

$$
\begin{aligned}
&\mathrm{p}_1 = s \\
&\textbf{for } i \leftarrow 1 \textbf{ to } n-2 \; by \; -1 \textbf{ do} \\
&\quad \mathrm{r}_i = |p_{i+1} - p_i| \\
&\quad \mathrm{p}_{i+1} = p_i + (r_i * d_i) \\
&\textbf{end}
\end{aligned}
$$

Figura 25 – Pseudocódigo do método *backwardReach*.

3.2.4 Complexidade de Implementação

Como pôde ser visto nos algoritmos e trechos de código apresentados até então, a complexidade de implementação do FABRIK é relativamente baixa. Por ser essencialmente um problema de encontrar um ponto em uma linha, envolve conceitos básicos e bem conhecidos dentro do campo do desenvolvimento de jogos, como geometria analítica e manipulação de vetores.

Além disso, é possível expandir o algoritmo, como adicionar constraints (que será demonstrado em uma seção posterior), sem que haja modificação em sua construção principal.

3.2.5 Pré-Requisitos

Para a implementação do FABRIK no Unity é necessário que o programador tenha um bom entendimento dos conceitos da geometria analítica e sinta-se confortável ao trabalhar com vetores. Não há, no entanto, nenhum conceito avançado de uso da engine Unity, mas faz-se necessário o entendimento básico de criação de scripts em C#.

3.2.6 Exemplo de aplicação

A seguir o exemplo de aplicação da técnica será apresentado, divido em subseções que representam partes da implementação.

3.2.6.1 Unity

Antes de iniciarmos a implementação do algoritmo, iremos definir a forma como os *joints* e *links* ficarão estruturados na cena do Unity e no código.

A fim de simplificar o processo, iremos definir cada *joint* como uma simples esfera, e o comprimento de cada *link* será automaticamente calculado com base na distância entre as esferas. Para visualizar cada *link*, iremos utilizar Gizmos, uma classe com utilitários para *debug* visual.

Como mostra a figura 26, cada esfera será filha de sua antecessora, formando uma hierarquia. Apesar de não importar para o algoritmo FABRIK como os joints estão estruturados na cena, a estrutura hierárquica é comum de ser utilizada, uma vez que modelos 3D normalmente usam essa disposição.

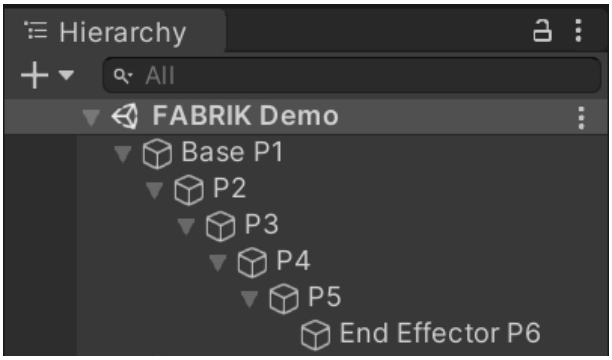

Figura 26 – Estrutura do objeto na *Hierarchy*.

Na cena, iremos dispor cada esfera em linha reta, como pode ser visto na figura 27, para simplificar o processo, lembrando que a distância entre cada esfera será mantida quando iniciarmos o algoritmo. No entanto, sinta-se à vontade para definir sua própria disposição dos *joints*.

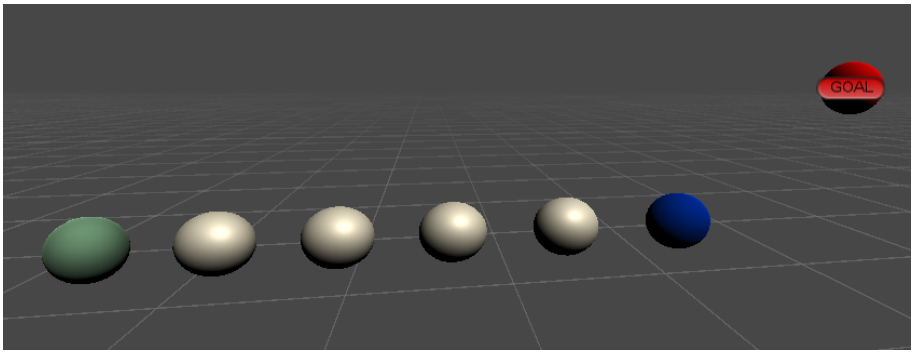

Figura 27 – A esfera verde corresponde a *Base P1*, enquanto a esfera azul corresponde ao *End Effector P6*, conforme a figura 26.

3.2.6.2 Implementação

Com isso, podemos finalmente criar o *script*, que nomearemos de *BasicFABRIK*, e adiciona-lo ao joint raiz (Base P1), como mostra a figura 28.

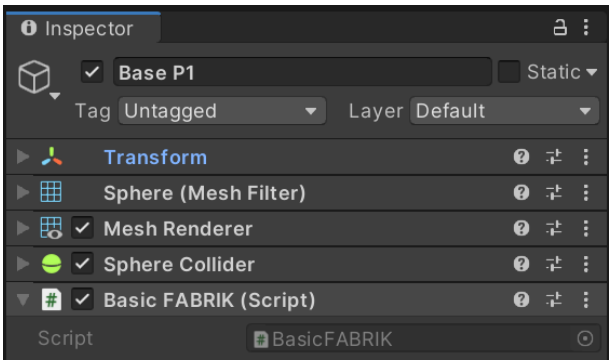

Figura 28 – Componentes do objeto "Base P1" após adição do *script*.

Conforme discutido na seção 3.2.3, há alguns dados que iremos manipular e, portanto, devemos tê-los como campos de nossa classe, a saber:

- Joints.
- Alvo a ser alcançado.
- Posição inicial do joint raiz.
- Distância entre cada par de joints.
- Margem de erro.
- Número máximo de iterações.

Para o campo joints, utilizaremos uma lista de Transforms, como pode ser visto na linha 1 do *code snippet* da figura 29. Utilizaremos também o atributo SerializeField,

expondo esse campo ao editor do Unity para que possamos preencher a lista de Joints manualmente, começando do joint raiz.

Da mesma forma, adicionaremos o alvo como um campo *Transform*, como pode ser visto na linha 3. A posição inicial do *joint* raiz será armazenado num Vector3, descrito na linha 5. A distância de cada link será armazenado num vetor de floats, como escrito na linha 6 da figura 29.

Por fim, armazenamos a margem de erro (offset) em um float, como mostra na linha 8 da figura 29. Definimos o valor de 0.01, que permite uma aproximação com pouquíssimo impacto no resultado final. Na linha 9, armazenamos o número máximo de iterações e atribuímos o valor de 10, que deve ser suficiente na maior parte dos casos.

```
1 [SerializeField] private List<Transform> joints = new List<Transform>();
2 [SerializeField] private Transform target;
3
4 private Vector3 startPosition;
5 private float[] jointsDistance;
6
7 private const float Offset = 0.01f;
8 private const int MaxIterations = 10;
```

Figura 29 – *Code snippet* dos *fields* da classe recém criada.

Desse modo, podemos preencher em ordem hierárquica a lista de joints no campo Joints do script, além de atribuir o alvo ao campo Target.

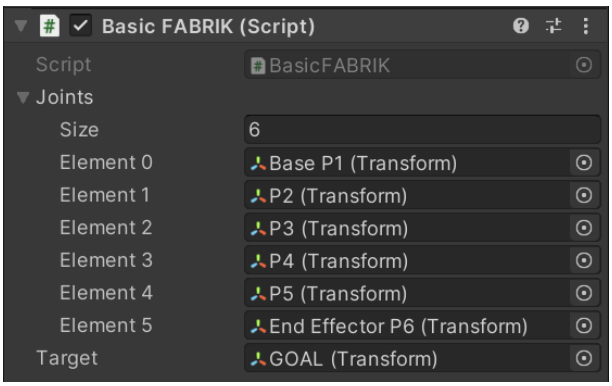

Figura 30 – Resultado do preenchimento da lista de *Transforms* no *script*.

Como mencionado anteriormente, faremos o uso da classe Gizmos para criar uma linha que servirá de representação visual dos links. Para isso, utilizaremos o método *OnDrawGizmos*, como mostra o code snippet da figura 31. Considerando o número de joints = n, teremos n-1 iterações, que é o número de links, conforme a linha 3 da figura da figura 31. Na linha 4, utilizaremos o método Gimoz.DrawLine que recebe dois vetores, sendo esses o joint atual e o joint posterior. O resultado pode ser visto na figura 32.

```
1 void OnDrawGizmos()
2 {
3     for (int i = 0; i < joints.Count - 1; i++)
4         Gizmos.DrawLine(joints[i].position, joints[i + 1].position);
5 }
```

Figura 31 – *Code snippet* do método *OnDrawGizmos*.

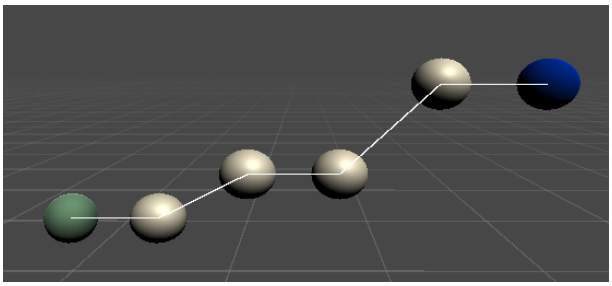

Figura 32 – Resultado visual do links.

Devemos então inicializar os campos restantes. Para o campo startPosition, atribuimos a posição do primeiro elemento (o joint raiz) da lista joints, como mostra a linha 3 do code snippet da figura 33.

Para o campo jointsDistance, inicializamos o array com o número de links (n° de joints - 1), na linha 5 da figura 33. Percorrendo cada índice do array, calculamos a distância de um par de joints como sendo a magnitude (comprimento) do vetor resultante da diferença entre a posição do joint posterior e joint atual, como pode ser visto na linhas 8 e 9 da figura 33.

```
 1 private void Awake()
 2 {
 3     startPosition = joints[0].position;
 4
 5     jointsDistance = new float[joints.Count - 1];
 6     for (int i = 0; i < jointsDistance.Length; i++)
 7     {
 8         jointsDistance[i] = (joints[i + 1].position -
 9                              joints[i].position).magnitude;
10     }
11 }
```

Figura 33 – *Code snippet* do método *Awake*.

Com isso, podemos implementar a parte principal do algoritmo que deve rodar a cada *frame*, devendo estar dentro do método Update. Com base no algoritmo apresentado

na figura 22, apresentamos o code snippet da figura 34.

```
1 private void Update()
2 {
3     if (IsTargetInRange())
4     {
5         for (int iteration = 0; iteration < MaxIterations; iteration++)
6         {
7             if (DistanceToTarget(joints.Last().position) < Offset)
8                 break;
9
10            ForwardReach();
11            BackwardReach();
12        }
13    }
14    else
15    {
16        StretchToTarget();
17    }
18 }
```

Figura 34 – *Code snippet* do método *Update*.

Começaremos pelo método *StretchToTarget*, que será rodado toda vez que o alvo não está dentro do alcance do end-effector.

```
1 private void StretchToTarget()
2 {
3     Vector3 direction = (target.position - startPosition).normalized;
4
5     for (int i = 1; i < joints.Count; i++)
6     {
7         joints[i].position = joints[i - 1].position +
8                              (direction * jointsDistance[i - 1]);
9     }
10 }
```

Figura 35 – *Code snippet* do método *StretchToTarget*.

Primeiramente, devemos encontrar a direção do joint raiz até o alvo, como mostra a linha 3 do code snippet da figura 35. Para isso, subtraímos a posição do alvo pela posição do joint raíz (startPosition). Normalizamos o vetor resultante, visto que só estamos interessados na direção.

Conforme as linhas 5-9 da figura 35, iteramos sobre cada *joint* a partir do segundo, atribuindo sua posição à posição do joint anterior, somando o produto entre a direção e a distância de seu link. A figura 36 ilustra o código em funcionamento.

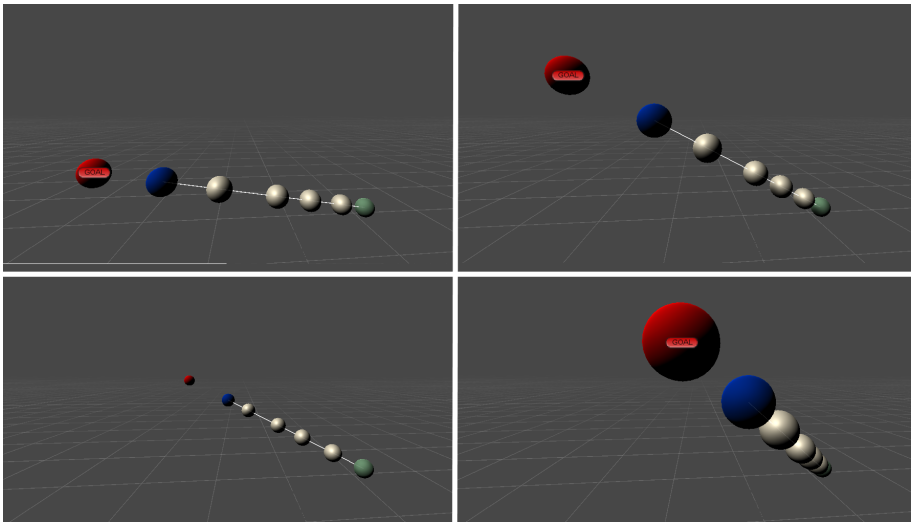

Figura 36 – Diferentes resultados do método *StretchToTarget*.

O próximo passo será a criação do método IsTargetInRange, que, como discutido na seção 3.2.3, é dado pela comparação da distância até o alvo do joint raíz, e a soma da margem de erro com o comprimento total dos links.

```
1 private bool IsTargetInRange()
2 {
3     return DistanceToTarget(startPosition) < TotalJointsDistance() + Offset;
4 }
```

Figura 37 – *Code snippet* do método *IsTargetInRange*.

O método DistanceToTarget é definido como a magnitude (comprimento) da diferença entre a posição do alvo e uma posição p, que nesse caso é a posição inicial. Já o método TotalJointsDistance é simplesmente a soma do array jointsDistance, como visto na figura 37.

```
1 private float DistanceToTarget(Vector3 p)
2 {
3     return (target.position - p).magnitude;
4 }
5
6 private float TotalJointsDistance()
7 {
8     return jointsDistance.Sum();
9 }
```

Figura 38 – *Code snippet* dos métodos *DistanceToTarget* e *TotalJointsDistance*.

Por fim, há apenas dois métodos restantes: ForwardReach e BackwardReach. Seguiremos a mesma ideia apresentada nas seções 3.2.3.1 e 3.2.3.2.

No método ForwardReach começaremos atribuindo a posição do alvo à posição do end-effector, como mostra a linha 3 da figura 39. Para cada joint em ordem reversa, atribuímos sua posição à posição do joint posterior, somando o produto da direção e da distância de seu link, para que mantenhamos sempre a mesma distância. Para calcularmos a direção, basta normalizarmos a diferença entre a posição de dois joints, $i-1$ e i.

```
1  private void ForwardReach()
2  {
3      joints.Last().position = target.position;
4
5      for (int i = joints.Count - 1; i > 0; i--)
6      {
7          Vector3 direction = (joints[i - 1].position -
8                               joints[i].position).normalized;
9          Vector3 position = joints[i].position +
10                              (direction * jointsDistance[i - 1]);
11
12         joints[i - 1].position = position;
13     }
14 }
```

Figura 39 – *Code snippet* do método *ForwardReach*.

O método BackwardReach opera de maneira similar ao ForwardReach, porém de forma inversa, como discutido na seção 3.2.3.2, e ilustrado na figura 40.

Com isso, o algoritmo está implementado e seu resultado gráfico em diferentes situações pode ser visto na figura 41.

3.2.7 Aplicação Real

Por ser um algoritmo generalista dentro do problema de Inverse Kinematics, faremos, nas próximas seções, exemplos mais específicos que conterão o uso deste método em situações reais.

3.2.8 Limitações

Conforme descrito pelo autor do FABRIK (ARISTIDOU; LASENBY, 2011), nenhuma limitação significativa desse método foi encontrada. Há, no entanto, pequenas limitações para casos específicos de tipos de joints.

```
1  private void BackwardReach()
2  {
3      joints.First().position = startPosition;
4
5      for (int i = 0; i < joints.Count - 2; i++)
6      {
7          Vector3 direction = (joints[i + 1].position -
8                               joints[i].position).normalized;
9          Vector3 position = joints[i].position +
10                              (direction * jointsDistance[i]);
11
12         joints[i + 1].position = position;
13     }
14 }
```

Figura 40 – *Code snippet* do método *BackwardReach*.

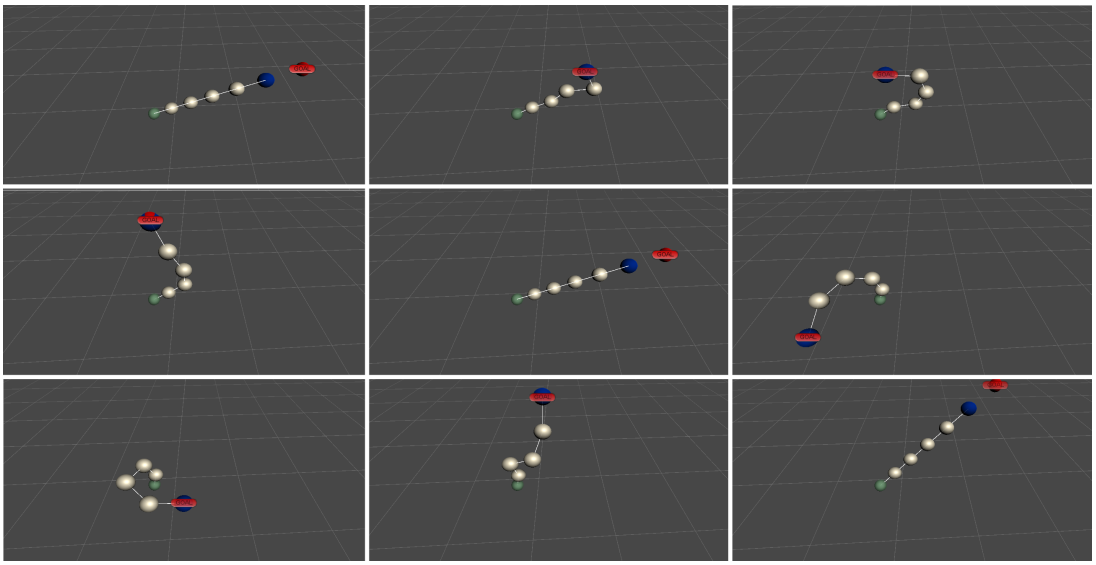

Figura 41 – Resultado final da implementação do algoritmo FABRIK.

Como não iremos abordar esses poucos casos, o algoritmo aqui apresentado não possui restrições para a resolução do problema de Inverse Kinematics.

3.3 Multiple End-Effectors FABRIK

As subseções a seguir tratarão sobre as orientações de uso do método *Multiple End-Effectors FABRIK*.

3.3.1 O que é?

Esta técnica é uma expansão do algoritmo FABRIK, apresentado no capitulo 3.2. A técnica permite a capacidade de lidar com mais de um end-effector e, consequentemente, mais de um alvo.

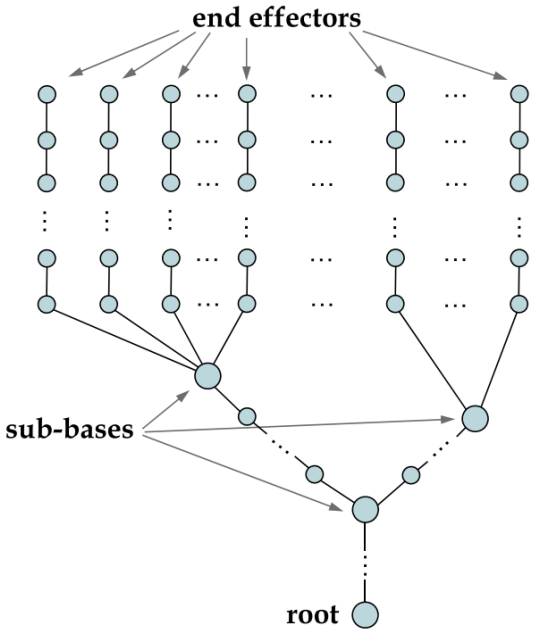

Figura 42 – Adaptado de "FABRIK: A fast, iterative solver for the Inverse Kinematics problem", por Aristidou, Andreas.

Conforme pode ser visto na figura 42, a adição de múltiplos end-effectors resulta em um número de cadeias igual ao número de end-effectors presentes, sendo um número maior possível caso haja cadeias intermediarias. Quando um Joint possui mais de um caminho, ou *link*, o chamamos de **sub-base**. A sub-base indicará o começo de uma nova cadeia.

3.3.2 Quando?

De forma isolada, a técnica FABRIK simples, discutida na seção 3.2, é suficiente para resolver o problema de Inverse Kinematics. No entanto, no mundo real temos modelos mais complexos, como os modelos multicorpos.

Como exemplo de modelo multicorpo, temos o corpo humano, que é constituído de várias cadeias cinemáticas, cada uma podendo ser controlada separadamente, mas em grande parte afetando o corpo como um todo. Por conta disso, esta técnica é necessária para resolver problemas que envolvam tanto múltiplos *end-effectors*, quanto múltiplos alvos (pontos de controle).

3.3.3 Como?

Para ilustrar o funcionamento do FABRIK com múltiplos end-effectors, utilizaremos um corpo com três cadeias cinemáticas, conforme o primeiro quadro da figura 43.

Como no algoritmo FABRIK apresentado no capítulo anterior, este método também possui duas partes. Na primeira parte aplicaremos o algoritmo FABRIK começando de cada end-effector em direção ao joint raíz, como demonstrado nos quadros 2, 3 e 4 da figura 43.

Após isso, teremos para cada sub-base o número de posições igual ao número de end-effectors de sua cadeia. Como o Joint D possui duas cadeias, terá duas posições resultantes da aplicação do algoritmo ($\mathrm{D}_{sub1} e D_{sub2}$).

Da mesma forma, o Joint B, com três cadeias, resultará nas posições B_{sub1}, B_{sub2} e B_{sub3}. A partir disso, a posição final de cada sub-base será a centroide de todas as suas posições, como é ilustrado nos quadros 5 e 6.

A segunda parte do algoritmo irá na direção inversa, começando do joint raíz, aplicaremos o FABRIK para cada sub-base, como representado nos quadros 7, 8 e 9. Quando todos as cadeias com sub-base forem atualizadas, iremos aplicar o algoritmo para cada end-effector até sua sub-base, como demonstrado nos quadros 10 e 11. Repetimos o processo até que o todos os end-effectors tenham atingido seus alvos ou caso não haja mudanças significativas entre sua posição prévia e atual.

Para simplificar o problema, discutiremos aqui o método FABRIK para múltiplos *end-effectors* utilizando apenas uma sub-base. O motivo disso é que para trabalhar com mais de uma sub-base é necessário o uso de algoritmos com características recursivas, o que dificultaria o entendimento desta técnica.

Como pode ser visto na figura 44, o algoritmo será dividido em três partes:

- **RootChainReach:** corresponde a primeira parte do algoritmo discutido anteriormente, onde começaremos aplicando o FABRIK para cada end-effector;
- **SubBaseUpdate:** após a aplicação do método *rootChainReach*, como visto na figura 44, haverá N posições para a sub-base igual ao número de end-effectors presentes. Aqui atualizaremos sua posição calculando a centroide de todas as posições geradas;
- **SubBaseChainReach:** corresponde a segunda parte do algoritmo discutido anteriormente, começaremos aplicando o FABRIK para cada end-effector, porém até a sua sub-base mais próxima, utilizando o valor atualizado do método anterior.

No método rootChainReach, descrito no algoritmo da figura 46, iteramos sobre cada cadeia e aplicamos a mesma lógica do FABRIK simples discutida na seção 3.2. No

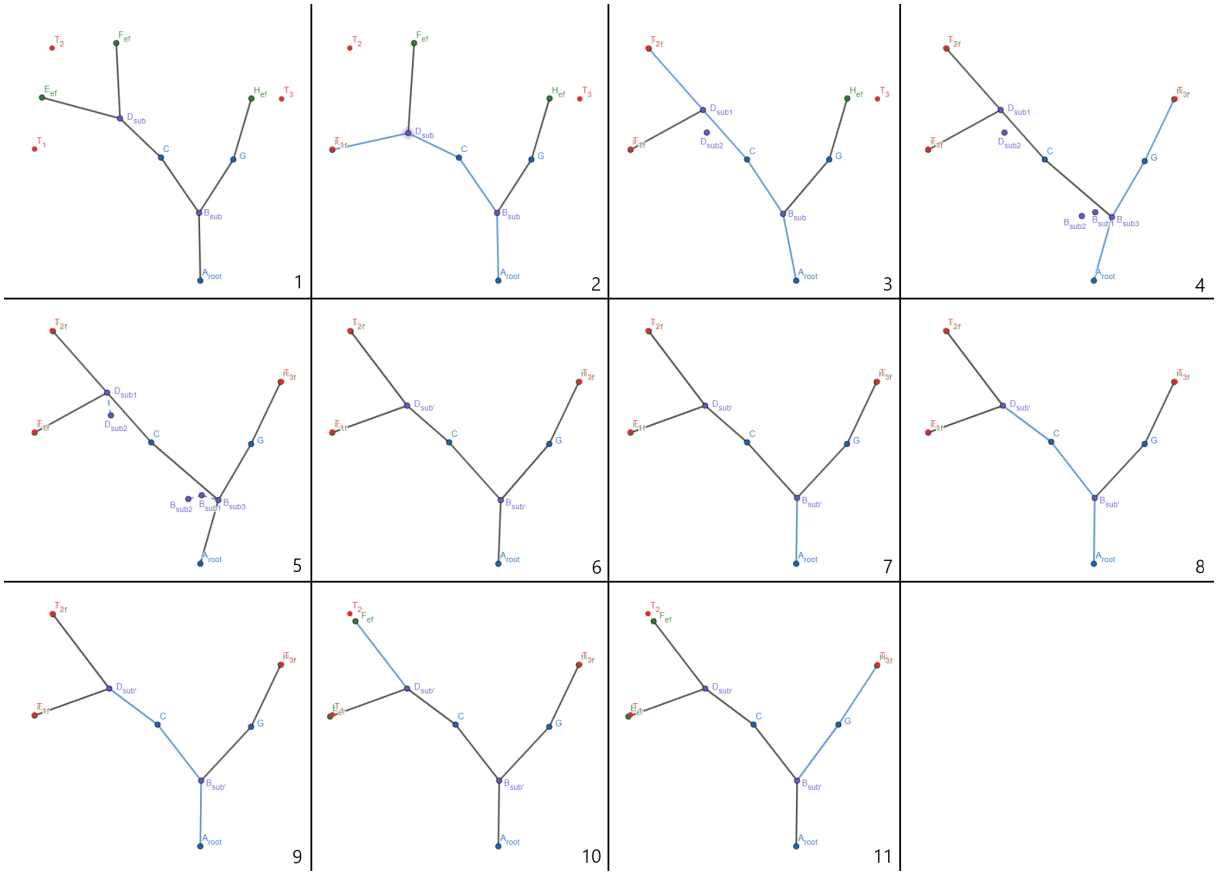

Figura 43 – Representação dos passos envolvidos no método “multi-FABRIK”.

```
input  : As cadeias FABRIK
output: As novas posições para cada cadeia FABRIK
initialization;
rootChainReach();
subBaseUpdate();
subBaseChainReach();
```

Figura 44 – Pseudocódigo da iteracão do algoritmo FABRIK para múltiplos end-effectors com um único *sub-base*.

entanto, ao invés de atualizarmos diretamente sua posição, a armazenaremos em uma variável (por isso o prefixo compute). Também guardaremos a posição da sub-base para cada cadeia iterada.

No método subBaseUpdate, descrito no algoritmo da figura 47, utilizaremos os dados das posições da sub-base coletadas no algoritmo da figura 46, e calcularemos sua nova posição a partir da centroide destas posições. Aplicaremos então o algoritmo FABRIK (*forward* e *backward*) e atualizaremos as posições dos joints.

No método subBaseChainReach, descrito no algoritmo da figura 48, primeiramente atualizaremos a posição de cada **joint**. Apóis isso, de forma muito semelhante ao algoritmo

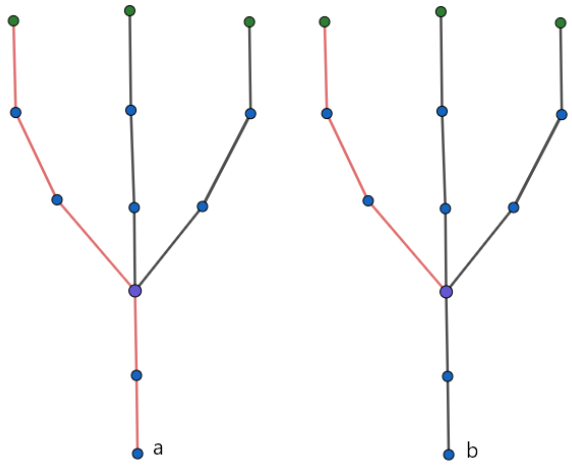

Figura 45 – a) Representa o alcance do método "rootChainReach", enquanto b) representa o alcance do método "subBaseChainReach".

```
for each chain in chains do
    subPosition = [];
    if chain target is in range then
        chain.computeForward();
        chain.computeBackward();
    else
        chain.computeStretchToTarget();
    end
    subPosition = posição da subbase da chain;
end
```

Figura 46 – Pseudocódigo do método *rootChainReach*.

```
subBase.positions.last() = CalculateCentroid();
subBase.ComputeForward();
subBase.ComputeBackward();
subBase.ApplyComputed();
subBase.UpdatePositionsArray();
```

Figura 47 – Pseudocódigo do método *subBaseUpdate*.

da figura 46, aplicaremos o algoritmo FABRIK simples, porém com uma versão modificada do método que limitará o alcance até a sub-base. No fim de cada iteração, aplicaremos a mudança em cada joint.

3.3.4 Complexidade de Implementação

Visualizar o funcionamento do problema de múltiplas cadeias, com múltiplos end-effectors e alvos, torna-se cada vez mais difícil conforme o número de cadeias envolvidas.

```
for each chain in chains do
    chain.UpdatePositionsArray();
    if chain target is in subbase to end-effector range then
        chain.computeForwardSubBase();
        chain.computeBackwardSubBase();
    else
        chain.computeStretchToTargetSubBase();
    end
    chain.applyComputedSubBase();
end
```

Figura 48 – Pseudocódigo do método *subBaseChainReach*.

No entanto, a sua implementação não foge do conceito inicial apresentado na seção 3.2.

Por conta disso, uma vez entendida a proposta do algoritmo FABRIK, a expansão para a técnica apresentada pode ser entendida com relativa facilidade. O desafio dá-se pela complexidade da adição de novos métodos, variáveis e controle externo de cada cadeia FABRIK.

3.3.5 Pré-Requisitos

Por ser uma expansão do algoritmo FABRIK, é necessário apenas que se tenha domínio do conteúdo apresentado na seção 3.2. Faremos uso extenso dos códigos e do conhecimento adquirido para implementar novos métodos e modificar funções existentes.

3.3.6 Exemplo de Aplicação

A seguir o exemplo de aplicação da técnica será apresentado, divido em subseções que representam partes da implementação.

3.3.6.1 Unity

Como na seção 3.2, iremos começar organizando a cena do Unity. Seguindo a mesma ideia da estrutura hierárquica, agora faremos três cadeias (A, B, C) a partir do ponto P3, que se tornará uma sub-base. Como teremos três end-effectors, faremos um alvo para cada cadeia, o que nos permite controlá-las de forma individual.

O resultado pode ser visto na figura 49, que exibe a estrutura na *Hierarchy*, e na figura 50, que representa a disposição dos objetos na cena.

Por fim, adicionaremos para cada end-effector (P6 A, P6 B e P6 C) o *script* BasicFABRIK, criado no capítulo 3.2.

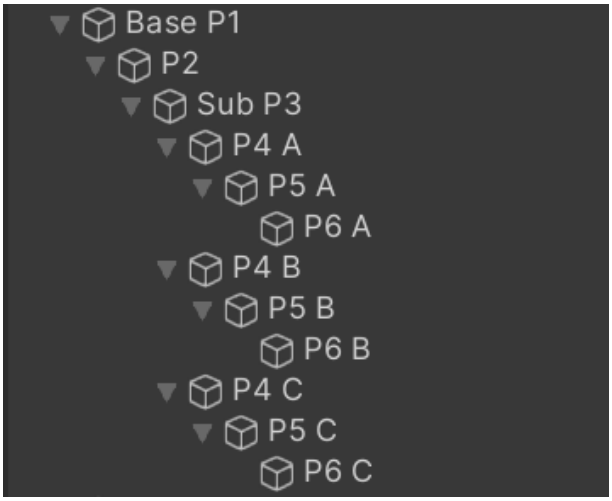

Figura 49 – Estrutura do objeto na *Hierarchy*.

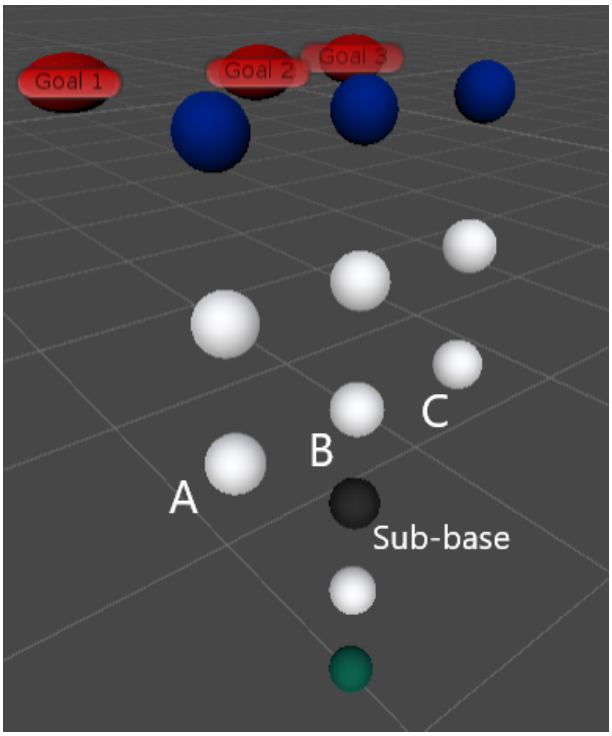

Figura 50 – Organização gráfica do objeto (com legendas) na cena.

3.3.6.2 Implementação

Diferentemente do método FABRIK simples, a posição de um *joint* pode não depender somente de sua cadeia. Por conta disso, as posições não devem ser atualizadas a medida em que são calculadas, mas armazenadas em uma variável para ser atualizada posteriormente.

Delegaremos então a função de atualizar as cadeias a uma nova classe, que chamaremos de MultipleEndEffectorsFABRIK. Esta classe pode ser atribuída a qualquer gameObject, mas por motivos de organização, atribuiremos ao joint raíz, como mostra a

figura 51

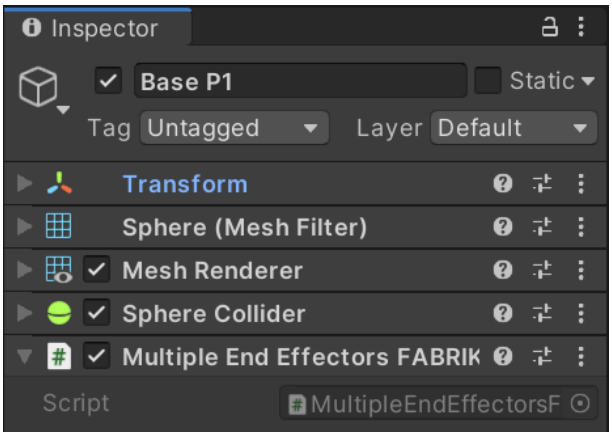

Figura 51 – Resultado dos componentes após a adição do *script*.

A classe MultipleEndEffectorsFABRIK terá dois campos, como mostra o code snippet da figura 52. O primeiro campo, endEffectorsChains, será uma lista de BasicFABRIK contendo cada cadeia de nosso modelo. O segundo campo será do tipo BasicFABRIK e conterá a cadeia da sub-base.

```
1 [SerializeField] private List<BasicFABRIK> endEffectorChains =
2                             new List<BasicFABRIK>();
3 [SerializeField] private BasicFABRIK subBaseChain;
```

Figura 52 – *Code snippet* dos *fields* adicionados.

Com isso, podemos preencher os campos através do Inspector, como mostra a figura 53.

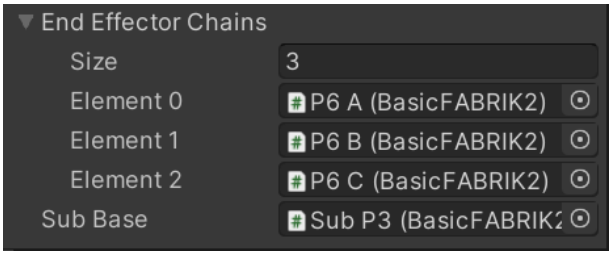

Figura 53 – Resultado do componente após o preenchimento da lista de *end-effectors*.

O próximo passo será a implementação do algoritmo da figura 44. Como este código deve ser executado a cada *frame*, o construiremos dentro do método Update.

Para simplificarmos, rodaremos o algoritmo 10 vezes a cada frame, como visto na linha 3 da figura 54. Na linha 5, executaremos o método RootChainReach e armazenaremos as posições retornadas na variável subBasePositions. Na linha 6, ainda da figura 54,

usaremos a variável subBasePositions como parâmetro para o método SubBaseUpdate. Por fim, aplicaremos o método SubBaseChainReach na linha 7.

```
1 void Update()
2 {
3     for (int j = 0; j < 10; j++)
4     {
5         var subBasePositions = RootChainReach();
6         SubBaseUpdate(subBasePositions);
7         SubBaseChainReach();
8     }
9 }
```

Figura 54 – *Code snippet* do método *Update*.

Começaremos a implementação dos métodos pelo método RootChainReach, descrito no code snippet da figura 55. Esta função retornará uma lista de posições (Vector3), como pode ser visto na declaração do método na linha 1.

Na linha 3 da figura 55, declaremos a variável subBasePositions como uma lista de Vector3, que será preenchida a cada iteração na linha 15, e retornada ao fim do método na linha 18.

Nas linhas 4-12 da figura 55, temos a aplicação muito semelhante do algoritmo FABRIK simples, no entanto, como comentado anteriormente, usaremos versões modificadas dos métodos que apenas armazenarão as posições obtidas. É importante notar que estes métodos serão implementados posteriormente nesta seção.

No método SubBaseUpdate, representado no code snippet da figura 56, utilizaremos como parâmetro a lista de posições geradas pelo método anterior.

Nas linhas 4-5 da figura 56, atribuímos a posição da sub-base, no array de posições, o valor da centroide das posições recebidas do método anterior. Para isso, na linha 3 primeiro encontramos o índice da sub-base neste array, que sempre será o último índice (uma vez que começamos a contar do end-effector para dentro).

Feito isso, precisamos aplicar o método forward e backward para corrigir a posição dos demais joints, como visto nas linhas 7 e 8 da figura 56. Por fim, aplicamos as mudanças diretamente aos joints no método da linha 10, e atualizamos as novas posições no array na linha 11.

O método CalculateCentroid, descrito no code snippet da figura 57, recebe uma lista de Vector3 com posições e retorna a soma dos vetores dividido pela quantidade de posições.

```
1 private List<Vector3> RootChainReach()
2 {
3     List<Vector3> subBasePositions = new List<Vector3>();
4     foreach (BasicFABRIK2 chain in endEffectorChains)
5     {
6         if (chain.IsTargetInRange())
7         {
8             chain.ComputeForward();
9             chain.ComputeBackward();
10        }
11        else
12            chain.ComputeStretchToTarget();
13
14        var subBasePosition = chain.positions[chain.SubBaseIndex];
15        subBasePositions.Add(subBasePosition);
16    }
17
18    return subBasePositions;
19 }
```

Figura 55 – *Code snippet* do método *RootChainReach*.

```
1 private void SubBaseUpdate(List<Vector3> subBasePositions)
2 {
3     var subBaseIndex = subBaseChain.positions.Length - 1;
4     subBaseChain.positions[subBaseIndex]
5                     = CalculateCentroid(subBasePositions);
6
7     subBaseChain.ComputeForward();
8     subBaseChain.ComputeBackward();
9
10    subBaseChain.ApplyComputed();
11    subBaseChain.UpdatePositionsArray();
12 }
```

Figura 56 – *Code snippet* do método *SubBaseUpdate*.

```
1 private Vector3 CalculateCentroid(List<Vector3> positions)
2 {
3     Vector3 sum = new Vector3();
4     foreach (Vector3 p in positions)
5         sum += p;
6
7     return sum / positions.Count;
8 }
```

Figura 57 – *Code snippet* do método *CalculateCentroid*.

Para completar a classe MultipleEndEffectorsFABRIK, temos o método SubBaseChainReach, descrito no code snippet da figura 58.

Iteramos sobre cada cadeia de forma muito semelhante ao code snippet da figura

55. Na linha 5 da figura 58, iremos atualizar o array de posições, uma vez que as posições foram modificadas no método *SubBaseUpdate*. Nas linhas 6-12 aplicaremos o FABRIK simples, porém utilizando o método modificado que percorrerá até a sub-base, e será explicado mais adiante. Por fim, na linha 14, aplicaremos o resultado da iteração a nova posição do joint.

```
1 private void SubBaseChainReach()
2 {
3     foreach (BasicFABRIK2 chain in endEffectorChains)
4     {
5         chain.UpdatePositionsArray();
6         if (chain.IsTargetInRangeSubBase())
7         {
8             chain.ComputeForwardSubBase();
9             chain.ComputeBackwardSubBase();
10        }
11        else
12            chain.ComputeStretchToTargetSubBase();
13
14        chain.ApplyComputedSubBase();
15    }
16 }
```

Figura 58 – *Code snippet* do método *SubBaseChainReach*.

3.3.6.3 Modificação do *Script BasicFABRIK*

A próxima etapa será a modificação do script **BasicFABRIK** , bem como a criação de diversos métodos utilizados pela classe MultipleEndEffectorsFABRIK. Faremos as seguintes modificações nesta seção:

- Remoção do conteúdo do método Update;
- Adição de uma variável para armazenar as posições dos joints (*positions*);
- Adição de variáveis de informação da sub-base;
- Reestruturar os métodos para utilizar a nova variável, ao invés de mudar diretamente a posição do joint;
- Adição de novos métodos que operam sobre a nova variável de posição;
- Adição de Métodos FABRIK que vão até a sub-base.

Removeremos o método Update como um todo, uma vez que delegaremos a chamada dos métodos à classe recém criada MultipleEndEffectorsFABRIK. Após isso, como descrito

no code snippet da figura 59, iremos criar o array de posições (linha 1), um field para a sub-base (linha 3), uma propriedade para o índice da sub-base (linha 5) e, por fim, um Vector3 para armazenar a posição inicial da sub-base (linha 7).

```
1 public Vector3[] positions;
2
3 [SerializeField] private BasicFABRIK2 subBaseChain;
4
5 public int SubBaseIndex { get; private set; } = -1;
6
7 private Vector3 subBaseStartPosition;
```

Figura 59 – *Code snippet* dos *fields* da classe recém criada.

Iremos, então, atualizar os métodos StretchToTarget, BackwardReach e ForwardReach, que foram apresentados na seção 3.2. No método ComputeStretchToTarget, a diferença se dá na linha 7 da figura 60, em que substituimos a posição do joint pelo array positions. Do mesmo modo, iremos fazer alteração nas linhas 14, 19, 20, 28, 32 e 33.

É importante notar que no método *ComputeForward* só atribuímos a posição do alvo à última posição caso (linhas 27-28 da figura 60) caso haja uma sub-base.

Como visto no code snippet da figura 56, necessitamos de dois métodos: UpdatePositionsArray e *ApplyComputed*.

O *code snippet* da figura 61 apresenta o método UpdatePositionsArray, que itera sobre o *array* de posições e o atualiza com os dados das posições dos joints.

Já o método ApplyComputed, representado no code snippet da figura 62, funciona da maneira inversa. Iteramos sobre as posições e atribuímos a posição de cada joint sua posição armazenada no array positions correspondente.

Também temos um método para encontrar o índice da sub-base (*GetSubBaseIndex*), representado no code snippet da figura 63. Iteramos sobre a lista de joints (excluindo o último, já que sempre será o end-effector) e retornamos o índice atual caso o joint possua um componente de BasicFABRIK. O método retorna -1 caso não haja sub-base.

Com isso, podemos inicializar as novas variáveis no método Start, como pode ser visto no code snippet da figura 64. As reticências na linha 3 indicam que há mais trechos de código que foram ocultados, conforme visto na seção 3.2.

Para finalizar, precisamos implementar as versões "sub-base"dos métodos, que foram utilizados no code snippet da figura 58. Como os métodos serão muito semelhantes às funções utilizadas no code snippet da figura 60, faremos um comparativo entre os dois.

Iniciaremos pelo método ComputeStretchToTargetSubBase. A primeira diferença

```
1 public void ComputeStretchToTarget()
2 {
3     Vector3 direction = (target.position - startPosition).normalized;
4
5     for (int i = 1; i < Joints.Count; i++)
6     {
7         positions[i] = positions[i - 1]
8                     + (direction * jointsDistance[i - 1]);
9     }
10 }
11
12 public void ComputeBackward()
13 {
14     positions[0] = startPosition;
15
16     for (int i = 0; i < Joints.Count - 2; i++)
17     {
18         Vector3 direction =
19                         (positions[i + 1] - positions[i]).normalized;
20         positions[i + 1] = positions[i] +
21                         (direction * jointsDistance[i]);
22     }
23 }
24
25 public void ComputeForward()
26 {
27     if (subBaseChain)
28         positions[positions.Length - 1] = target.position;
29
30     for (int i = Joints.Count - 1; i > 0; i--)
31     {
32         Vector3 direction = (positions[i - 1] - positions[i]).normalized;
33         positions[i - 1] = positions[i] +
34                         (direction * jointsDistance[i - 1]);
35     }
36 }
```

Figura 60 – *Code snippet* dos métodos atualizados.

```
1 public void UpdatePositionsArray()
2 {
3     for (int i = 0; i < Joints.Count; i++)
4         positions[i] = Joints[i].position;
5 }
```

Figura 61 – *Code snippet* do método *UpdatePositionsArray*.

pode ser vista nas linhas 3-4 da figura 65, em que substituirmos a variável startPosition (a posição raiz), pela última posição da cadeia da sub-base. A outra diferença é que

```
1 public void ApplyComputed()
2 {
3     for (int i = 0; i < positions.Length; i++)
4         Joints[i].position = positions[i];
5 }
```

Figura 62 – *Code snippet* do método *ApplyComputed*.

```
1 public int GetSubBaseIndex()
2 {
3     for (int i = Joints.Count - 2; i >= 0; i--)
4     {
5         if (Joints[i].GetComponent<BasicFABRIK2>())
6             return i;
7     }
8
9     return -1;
10 }
```

Figura 63 – *Code snippet* do método *GetSubBaseIndex*.

```
1 void Start()
2 {
3     ...
4
5     SubBaseIndex = GetSubBaseIndex();
6     if (subBaseChain)
7         subBaseStartPosition = Joints[SubBaseIndex].position;
8
9     UpdatePositionsArray();
10 }
```

Figura 64 – *Code snippet* do método *Start*.

começaremos a iteração (na linha 6) pelo **joint** posterior à sub-base.

Já no método *ComputeBackwardSubBase*, substituímos o primeiro índice pelo índice da sub-base, bem como a variável de posição inicial pela variável de posição inicial da sub-base, na linha 14 da figura 65. Também começaremos a iteração pelo índice da sub-base, como demonstra a linha 16.

Por fim, no método *ComputeForwardSubBase*, ainda na figura 65, atualizamos a variável subBaseStartPosition com a posição armazenada da sub-base, conforme a linha 25. Na linha 28, ao invés de iterarmos sobre toda a cadeia, interrompemos a iteração no índice da sub-base.

```
public void ComputeStretchToTargetSubBase()
{
    Vector3 direction = (target.position -
                         subBaseChain.positions.Last()).normalized;

    for (int i = SubBaseIndex + 1; i < Joints.Count; i++)
        positions[i] = positions[i - 1] +
                            (direction * jointsDistance[i - 1]);
}


public void ComputeBackwardSubBase()
{
    positions[SubBaseIndex] = subBaseStartPosition;

    for (int i = SubBaseIndex; i < Joints.Count - 2; i++)
    {
        Vector3 direction = (positions[i + 1] - positions[i]).normalized;
        positions[i + 1] = positions[i] + (direction * jointsDistance[i]);
    }
}

public void ComputeForwardSubBase()
{
    subBaseStartPosition = positions[SubBaseIndex];
    positions[positions.Length - 1] = target.position;

    for (int i = Joints.Count - 1; i > SubBaseIndex; i--)
    {
        Vector3 direction = (positions[i - 1] - positions[i]).normalized;
        positions[i - 1] = positions[i] + (direction * jointsDistance[i - 1]);
    }
}
```

Figura 65 – *Code snippet* dos novos método.

Também necessitamos do método IsTargetInRangeSubBase, apresentado no *code snippet* da figura 66. A diferença para o método o*IsTargetInRange* dá-se pelo uso do método SubBaseJointsDistance (ao invés do *TotalJointsDistance*), e por passar como parâmetro a posição da sub-base (ao invés da posição raiz) para o método *DistanceToTarget*.

```
public bool IsTargetInRangeSubBase()
{
    return DistanceToTarget(positions[SubBaseIndex]) <
                    SubBaseJointsDistance() + Offset;
}
```

Figura 66 – *Code snippet* do método *IsTargetInRangeSubBase*.

O método *SubBaseJointsDistance*, descrito no *code snippet* da figura 67, será simplesmente a soma de todas as distâncias dos joints até a sub-base.

Encerrando os métodos necessários, temos a função ApplyComputedSubBase,

```
1 private float SubBaseJointsDistance()
2 {
3     float sum = 0;
4     for (int i = SubBaseIndex; i < Joints.Count - 1; i++)
5         sum += jointsDistance[i];
6     return sum;
7 }
```

Figura 67 – *Code snippet* do método *SubBaseJointsDistance*.

descrita no *code snippet* da figura 68. De forma muito semelhante ao método *ApplyComputed*, atribuímos as posições dos *joints* os valores armazenados no array de posições, no entanto iteramos até a sub-base, como mostra a linha 3.

```
1 public void ApplyComputedSubBase()
2 {
3     for (int i = SubBaseIndex; i < positions.Length; i++)
4         Joints[i].position = positions[i];
5 }
```

Figura 68 – *Code snippet* do método *ApplyComputedSubBase*.

Finalmente, a figura 69 ilustra a técnica em funcionamento em situações diferentes.

3.3.7 Aplicação Real

Do mesmo modo que o FABRIK simples, a heurística aqui apresentada serve como base para realização de muitas técnicas de animação que envolvem Inverse Kinematics. Como discutido na seção 3.6.2, este método é especialmente interessante para modelos multicorpos, como personagens humanos.

A figura 70 demonstra uma aplicação real utilizando técnicas que envolvem o método FABRIK para múltiplos *end-effectors*. Note que, ao movimentarmos a mão esquerda, há uma mudança em todo o corpo apesar de existirem outros pontos de controle.

3.3.8 Limitações

Como discutido em (ARISTIDOU; CHRYSANTHOU; LASENBY, 2016), há um número crescente de casos especiais à medida que a cadeia torna-se mais complexa. Na figura 71 temos um caso em que a distância entre os alvos é maior que a abertura entre os dois *end-effectors*. Sendo assim, há três possibilidades de resolução: o *end-effector* esquerdo alcança o alvo esquerdo, o *end-effector* direito alcança o alvo direito, ou ambos não alcançam o alvo.

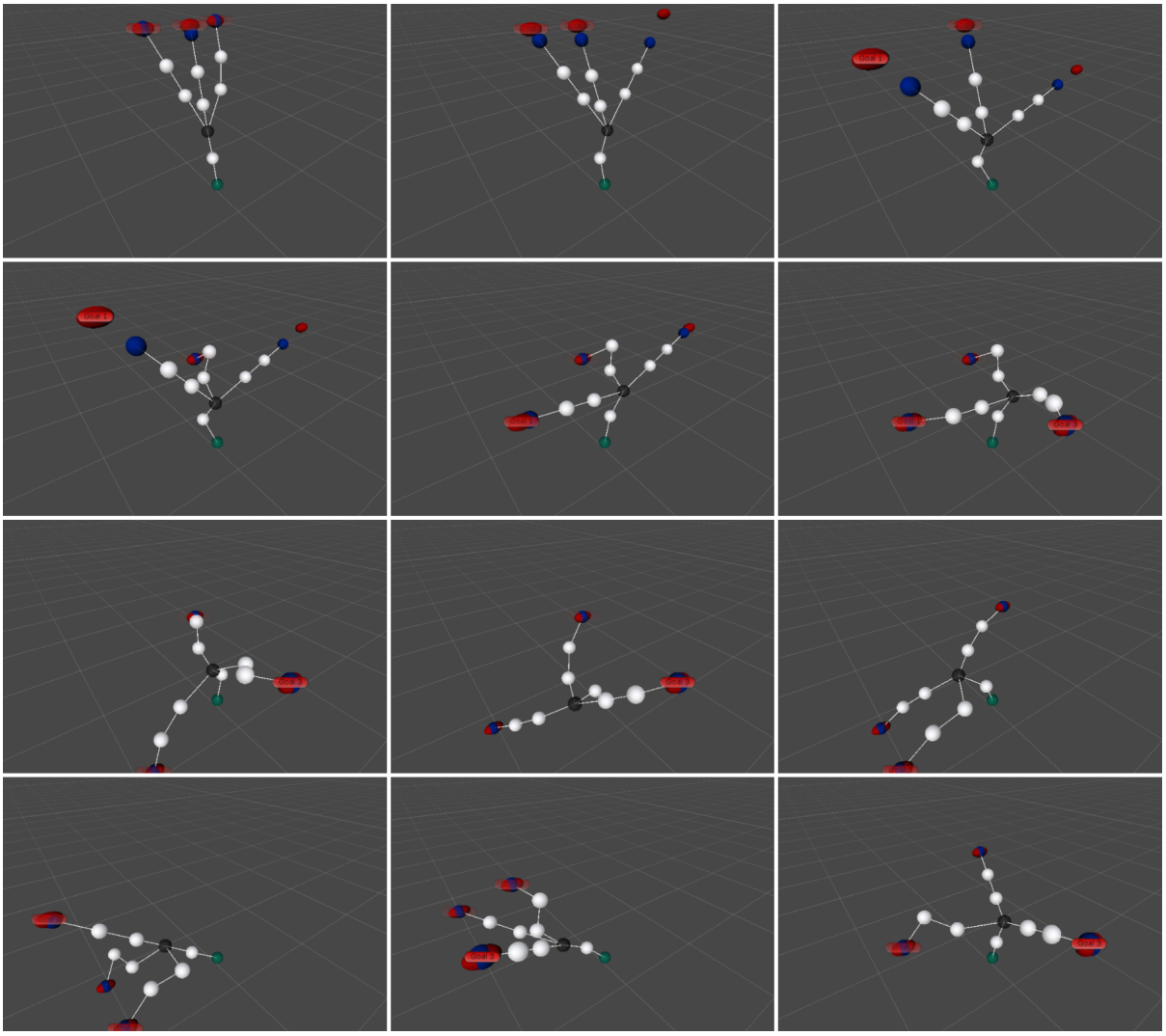

Figura 69 – Resultado final da técnica de múltiplos end-effectors FABRIK.

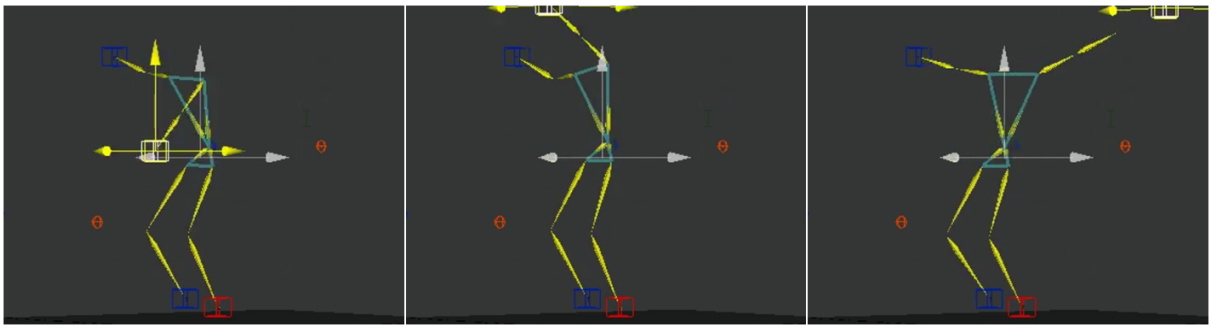

Figura 70 – Esqueleto humano no software "FABRIC Engine 2".

Portanto, não é difícil imaginar a quantidade de parâmetros possíveis para um maior controle da cadeia de múltiplos *end-effectors*. Entretanto, o método apresentado nesta seção é um bom ponto inicial, podendo ser utilizado em uma vasta gama de situações.

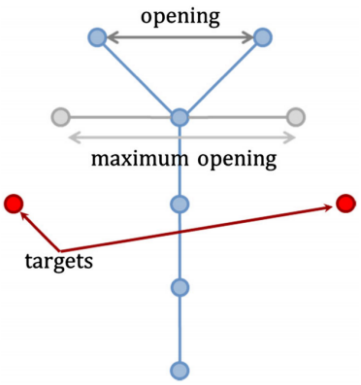

Figura 71 – Adaptado de "Extending FABRIK with model constraints", por Aristidou, Andreas.

3.4 Foot Inverse Kinematics

As subseções a seguir tratarão sobre as orientações de uso do método *Foot Inverse Kinematics*.

3.4.1 O que é?

Nesta seção iremos aplicar os conceitos de *Inverse Kinematics* (IK) a um contexto mais próximo da realidade, o chamado *Foot IK*, em que iremos utilizar a técnica em um modelo humanoide para melhorar o realismo do posicionamento de seus pés em relação ao terreno.

3.4.2 Quando?

Sempre que se deseja aumentar o realismo de um personagem ou criatura em relação à interatividade com o mundo a sua volta, o *Foot IK* é sempre bem-vindo. Na figura 72 podemos notar a diferença da mesma cena com a técnica ligada e desligada. Pequenos detalhes como esse ajudam a construir a imersão de um jogo.

3.4.3 Como?

Uma vez que temos uma solução de IK, a implementação do *Foot IK* resume-se a movermos o alvo até a altura correta do chão. Para adicionarmos ainda mais realismo, também rotacionaremos o pé rente à superfície de contato. Essa ideia está ilustrada na figura 73.

O primeiro passo então é achar a posição da superfície para qual deslocaremos o pé. Para isso, utilizaremos os **Raycasts**. Um *ray* funciona de forma semelhante ao um

Figura 72 – Exemplo da técnica de *Foot IK* ligado e desligado no jogo *The Elder Scrolls Online*.

input : Posição do alvo
output: Nova posição do alvo
initialization;
if *ground is in range* **then**
    ikPosition = groundLevel;
    ikRotation = rotateToGroundNormal;
end

Figura 73 – Pseudocódigo do *foot IK*.

vetor, onde definimos um ponto inicial e uma direção. Caso seu comprimento não seja definido, um *ray* torna-se uma semirreta.

Um *raycast* é o uso de um *ray*, como por exemplo um raio de luz de um laser, para coletar informações de colisão, como distância, superfície normal, ou o próprio objeto colidido. É uma técnica muito utilizada em jogos de tiro, como exemplificado na figura 74.

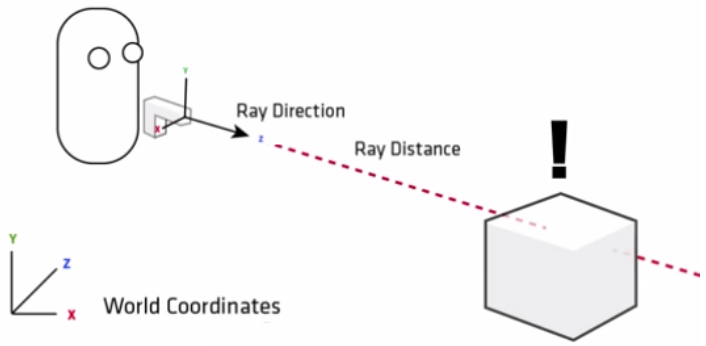

Figura 74 – Esquema de *raycast* sendo utilizado para projéteis.

Com isso, podemos verificar a condição *groundisinrange*. O que resta é mudar o

alvo do IK, como vimos anteriormente, para a posição em que se encontra a superfície. Para dar ainda mais realismo, rotacionamos o pé para ficar rente à superfície, utilizando sua normal, como mostra a figura 75.

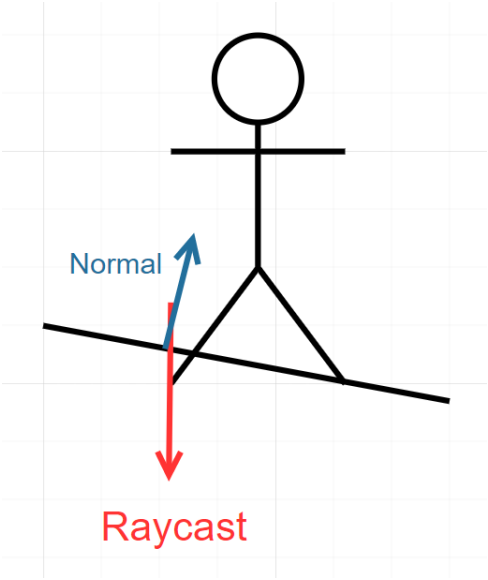

Figura 75 – Esquema demonstrando os vetores de *raycast* e normal.

3.4.4 Complexidade de Implementação

Uma vez que temos um algoritmo de *Inverse Kinematics* realizado, a implementação da técnica demonstrada será relativamente bem simples. No entanto, para casos mais complexos com diversos *edge cases*, é necessário bem mais atenção e uso de técnicas mais elaboradas.

3.4.5 Pré-Requisitos

Como pré-requisito desta seção, temos apenas o entendimento da técnica de *Inverse Kinematics* apresentado nas seções anterioes.

3.4.6 Exemplo de Aplicação

Para esta seção utilizaremos a modelo oficial do Unity, a Unity-Chan (figura 76), por ser um bom exemplo de modelo encontrados em produções reais. Além disso, usaremos o algoritmo de *Inverse Kinematics* presente no próprio sistema de animação humanóide do Unity (*Mecanim*).

A Unity-Chan pode ser encontrada em sua página na *asset store*: <https://assetstore.unity.com/packages/3d/characters/unity-chan-model-18705>. Basta importar e o modelo junto de suas animações ficará pronto para uso.

Figura 76 – A modelo oficial do Unity: Unity-Chan!

3.4.6.1 Unity

Com o modelo importado, colocaremos em cena o *prefab* "unitychan", que se encontra dentro do pacote importado, destacado na figura 77.

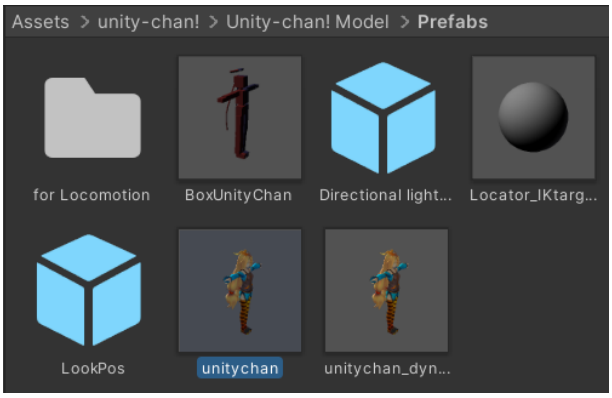

Figura 77 – Modelo após importação na janela *Project*.

O próximo passo é ativarmos o *IK Pass*, que permitirá o uso da função *OnAnimatorIK*. Para isso, vá até o componente *Animator* do modelo recém colocado na cena e clique duas vezes sobre o campo *Controller*, que abrirá uma janela do *Animator*. No canto superior à direita, dentro da categoria *Layers*, temos o ícone de configuração do *Base Layer*, ao clicarmos basta ativarmos o *checkbox* do *IK Pass*. O processo está ilustrado na figura 78.

3.4.6.2 Scripting

Com tudo pronto no editor, criaremos um *script* responsável pelo controle do *Foot IK*, que chamarei de **MecanimFootIK**. Essa classe terá 4 *fields*:

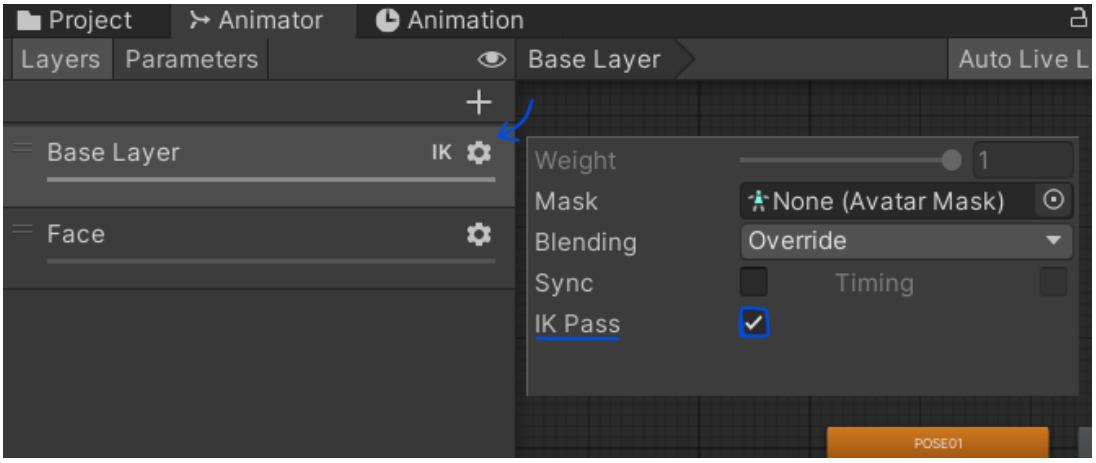

Figura 78 – Ativação do *IK Pass*.

- Animator: fará referência ao **Animator** para que seja possível controlar a posição e rotação do *Inverse Kinematics Target*. Como pode ser visto na linha 12 da figura 79;
- MaxHeight: parâmetro responsável por definir a altura máxima em que o pé irá aderir a superfície. Também define o ponto inicial do *raycast*. Pode ser visto na linha 7 do *code snippet* da figura 79;
- MaxDistance: define o comprimento do *raycast*, em outras palavras define quão longe o sistema de *Inverse Kinematics* irá ser ativado. Declarado na linha 10 do *code snippet* da figura 79;
- Offset: define a margem de deslocamento do ponto de contato. Declarado na linha 4 do *code snippet* da figura 79.

```
1  public class MecanimFootIK : MonoBehaviour
2  {
3      [SerializeField] [Range(0, 1f)]
4      private float offset = 0.05f;
5
6      [SerializeField] [Range(0, 2f)]
7      private float maxHeight = 0.5f;
8
9      [SerializeField] [Range(0, 2f)]
10     private float maxDistance = 0.65f;
11
12     private Animator animator;
```

Figura 79 – *Fields* da classe recém criada.

Como pode ser visto na figura 80, o componente agora possui três campos editáveis com um *slider*. Os valores padrões foram bons valores encontrados por mim, mas variam por situação e modelo.

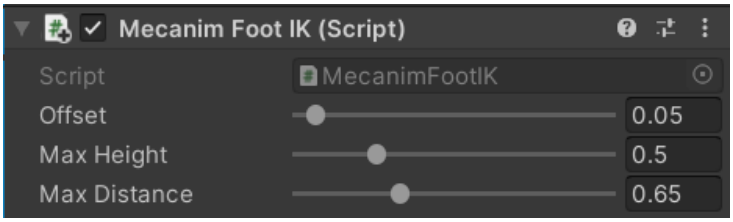

Figura 80 – *Mecanim Foot IK script* com *fields* serializados.

O primeiro passo é inicializar o *animator*. Para isso, basta pegarmos o componente através do método *Awake*, como mostra o *code snippet* da figura 81.

```
1 private void Awake()
2 {
3     animator = GetComponent<Animator>();
4 }
```

Figura 81 – *Code snippet* do método *Awake*.

Com isso, daremos início à parte principal do script. O código será escrito dentro do método *OnAnimatorIK*, que é chamado após o processo de animação, porém antes das transformações serem efetuadas, como ilustrado na figura 82. O método possui um parâmetro *layerIndex* que não será utilizado nessa demonstração, a declaração pode ser vista na linha 1 do *code snippet* 84.

Primeiramente, na linha 3, instanciaremos um objeto que conterá as informações da colisão do *raycast*. Utilizaremos esse objeto mais adiante. Na linha 5 pegaremos a posição do alvo do *Inverse Kinematics* (funcionando de forma bem semelhante ao que vimos nas seções anteriores), através do método *GetIKPosition* do animator. Nesse exemplo, utilizaremos apenas o pé esquerdo, por isso passaremos o *enum AvatarIKGoal.LeftFoot* como parâmetro.

O próximo passo é calcular a origem do *raycast*. Um primeiro pensamento seria de posicionar a origem na mesma posição do pé, como nas imagens *a*) e *b*) da figura 83. A ideia funcionaria para a imagem *a*), em que a superfície está abaixo do pé, no entanto, para casos como a figura *b*), em que o pé encontra-se abaixo (ou dentro) da superfície, a técnica não funcionaria. Por conta disso, moveremos a origem para cima de acordo com a variável $maxHeight$, como visto na imagem *c*).

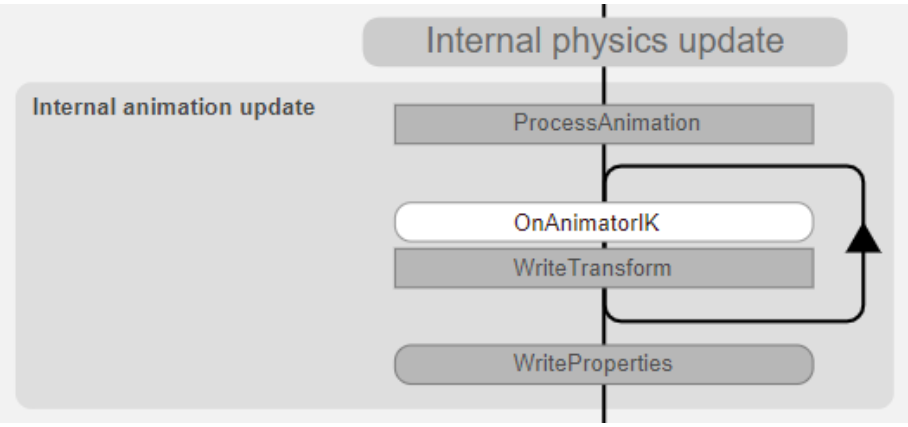

Figura 82 – Ordem de execução da atualização interna de animação.

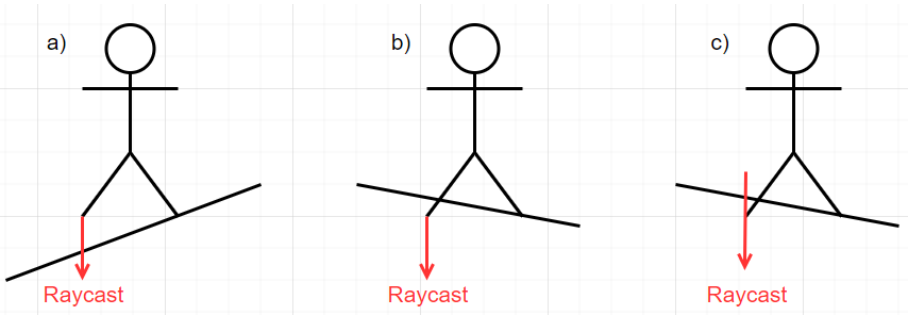

Figura 83 – Esquema demonstrando diferentes posições da origem do *raycast*.

Dessa forma, como visto na linha 7 da figura 84, definimos a origem como a soma da posição do alvo previamente calculada com a altura $maxHeight$. Com isso, temos o necessário para o *raycast*, através do método *Physics.Raycast*, em que o primeiro parâmetro é a origem, o segundo é a direção (para baixo, em direção ao solo), o terceiro é o $hitInfo$ por referência, e por fim o quarto é a distância máxima.

Ainda na figura 84, uma vez que o método *Physics.Raycast* tem retorno booleano, se houve colisão ou não, o colocamos dentro de um *if case*. As informações da colisão ficam armazenadas no $hitInfo$. Com isso, nas linhas 11 e 12 atribuiremos um peso ao método IK do *animator*. O primeiro parâmetro é o *target* do IK, nesse caso estamos usando o pé esquerdo, e o segundo é um valor normalizado (entre 0 e 1), que representa a porcentagem em que o IK afeta a animação (0 sendo sem efeito, 1 sendo 100% de efeito, e seus valores intermediários). A nova posição do *target IK*, visto na linha 14, será o ponto de contato da colisão, através do $hitInfo.point$. Além disso, adicionaremos um *offset* para cima, a fim de ajustar a posição do pé. Com isso, na linha 15, atribuiremos a nova posição utilizando o método *animator.SetIKPosition*.

Por fim, faremos o mesmo quanto à rotação, calculamos a nova rotação como um *Quaternion* que representa a rotação da direção do pé atual ($Vector3.up$) para a direção da normal da colisão ($hitInfo.normal$), como visto nas linhas 17-18 da figura 84. Por fim,

basta aplicar utilizando o método *animator.SetIKRotation*, como mostra a linha 19.

```
1  public void OnAnimatorIK(int layerIndex)
2  {
3      var hitInfo = new RaycastHit();
4
5      Vector3 targetPosition = animator.GetIKPosition(AvatarIKGoal.LeftFoot);
6
7      Vector3 origin = targetPosition + Vector3.up * maxHeight;
8
9      if (Physics.Raycast(origin, Vector3.down, out hitInfo, maxDistance))
10     {
11         animator.SetIKPositionWeight(AvatarIKGoal.LeftFoot, 1);
12         animator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, 1);
13
14         Vector3 newPosition = hitInfo.point + Vector3.up * offset;
15         animator.SetIKPosition(AvatarIKGoal.LeftFoot, newPosition);
16
17         Quaternion newRotation =
18             Quaternion.FromToRotation(Vector3.up, hitInfo.normal);
19         animator.SetIKRotation(AvatarIKGoal.LeftFoot, newRotation);
20     }
21 }
```

Figura 84 – *Code snippet* do método *OnAnimatorIK*.

3.4.6.3 Resultados

Na figura 85 podemos ver o comparativo entre o personagem em terreno plano (mais à esquerda), e com diferentes simulações de terreno nas imagens a direita. Já na figura 86, podemos ver o *IK* comportando-se num terreno angulado plano. Por fim, na figura 87 temos um exemplo mais exagerado.

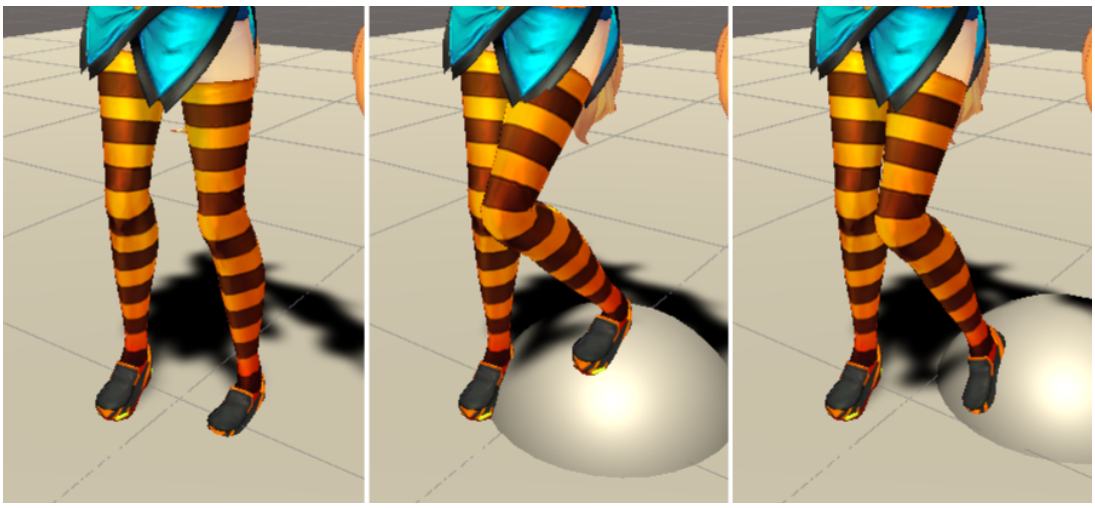

Figura 85 – Exemplo de posicionamento e rotação dos pés em uma alta superfície arredondada.

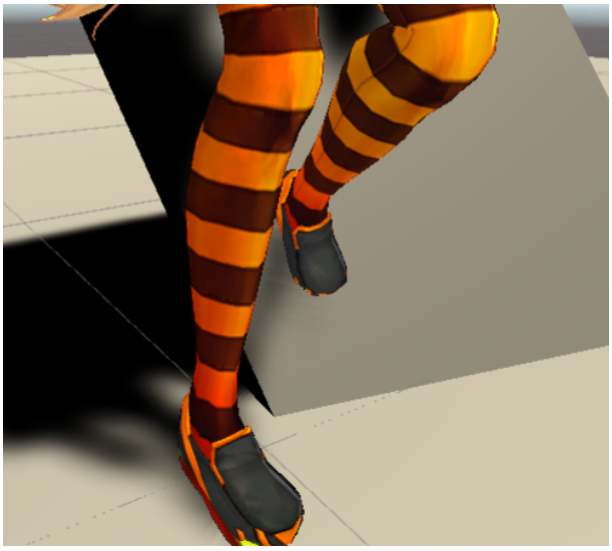

Figura 86 – Exemplo de posicionamento e rotação dos pés utilizando um plano diagonal.

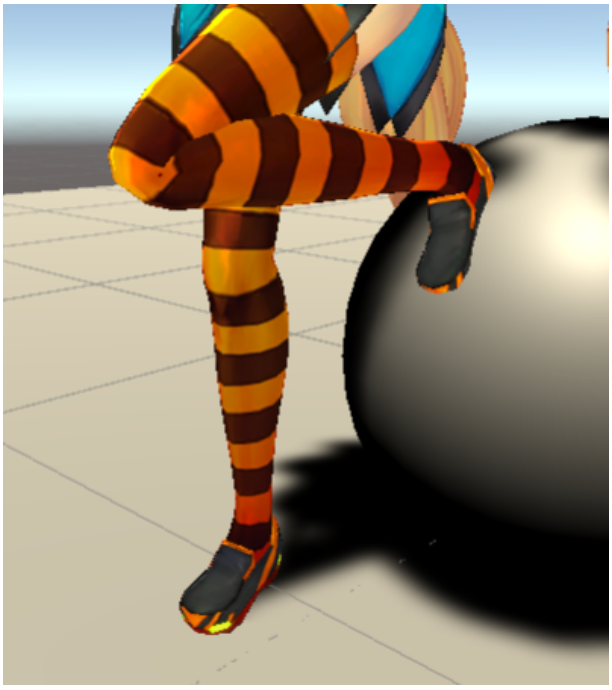

Figura 87 – Exemplo de posicionamento e rotação dos pés em uma alta superfície arredondada.

3.4.7 Aplicação Real

Tanto no campo das animações quanto no campo dos jogos, há diversos exemplos de mídias que utilizam esta técnica, tornando-se até mesmo uma solução esperada pelos jogadores em jogos mais modernos. A figura 88 ilustra a utilização em três jogos AAA bem famosos: *Sekiro: Shadows Die Twice*; *The Legend of Zelda: The Wind Waker*; e *Tomb Raider*, respectivamente.

Figura 88 – Exemplo de jogos recentes e antigos que usam a técnica de Inverse Kinematics.

3.4.8 Limitações

Por utilizarmos o módulo *Mecanim* do *Unity* nesta seção, estamos limitados a um modelo humanoide que possua um *Avatar* válido. Para mais informações sobre como importar um modelo válido e configurar o *Avatar*, deve-se acessar o link: <https://docs.unity3d.com/Manual/ConfiguringtheAvatar.html>.

3.5 Hand Inverse Kinematics

As subseções a seguir tratarão sobre as orientações de uso do método *Hand Inverse Kinematics*.

3.5.1 O que é?

Assim como na seção anterior, quando abordamos uma aplicação real do *Inverse Kinematics*, aqui iremos mostrar uma outra forma de aplicação: o uso de *IK* nas mãos para, de forma procedural, deslocar as mãos de uma personagem quando aproxima-se de uma parede.

3.5.2 Quando?

A técnica de *Hand IK* é muito versátil, sendo constantemente utilizada sempre que se deseja ter uma interatividade do personagem com objetos a sua volta. Nesta seção iremos explorar apenas uma ideia de seu uso. Para outras aplicações reais, veja a seção 3.5.7.

3.5.3 Como?

Esta técnica funciona de forma muito semelhante ao problema de *Foot IK* da seção 3.4, uma vez que possuímos a solução de *IK*, atribuiremos ao alvo correspondente da mão do personagem a um objeto. Neste caso, faremos com que o personagem, ao se aproximar de uma parede, coloque a mão sobre ela de forma procedural. Utilizaremos apenas a mão direita para ilustrar.

input : Posição do alvo
output: Nova posição do alvo
initialization;
if *wall is in range* **then**
    ikPosition = closest point of wall;
    ikRotation = rotate palm to face wall;
end

Figura 89 – Pseudocódigo do método *Hand IK*.

Como pode ser visto na figura 89, primeiramente devemos detectar se existe uma parede próxima ao personagem, caso positivo, devemos verificar se a mesma encontra-se numa distância igual ou menor ao comprimento do braço. Para isso, diferentemente da último seção que utilizamos *Raycasts*, utilizaremos o método de *OverlapSphere*.

O método de *OverlapSphere*, ilustrado na figura 90, consiste em retornar os objetos que colidem com uma esfera, estando totalmente em sua área ou apenas parcialmente em contato.

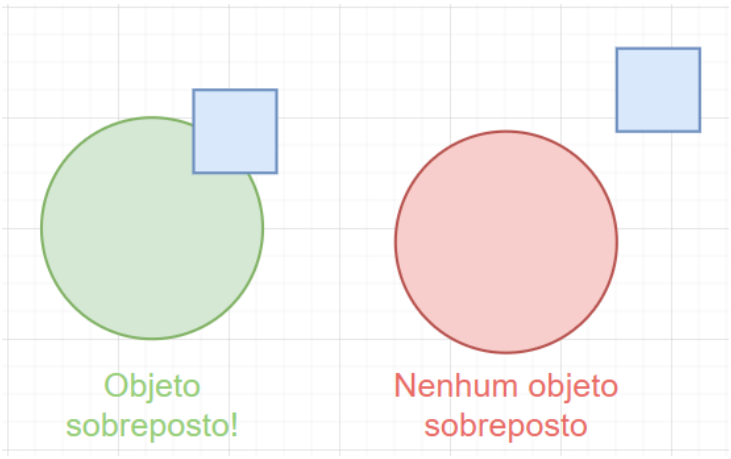

Figura 90 – Esquema 2D ilustrando o uso de *OverlapSphere* para detectar objetos.

Utilizando a *OverlapSphere* podemos verificar a condição *wall is in range*. Com isso, movemos a mão para o ponto mais próximo do objeto retornado. Para dar ainda mais realismo, rotacionamos a mão de forma que ela pareça pousar sobre a parede.

3.5.4 Complexidade de Implementação

Como na seção anterior, a complexidade reside apenas em manipular o alvo *IK* de maneira a buscar a melhor qualidade visual. Nesta seção aplicaremos alguns conceitos de álgebra linear para obter informações que permitam escolher e manipular o melhor *IK goal*.

3.5.5 Pré-Requisitos

Como pré-requisito desta seção, temos apenas o entendimento da técnica de *Inverse Kinematics* apresentado nas seções anteriores.

3.5.6 Exemplo de Aplicação

A seguir o exemplo de aplicação da técnica será apresentado, divido em subseções que representam partes da implementação.

3.5.6.1 Unity

Continuando da seção anterior, também utilizaremos a *Unity-Chan* como nosso modelo. Para essa seção precisamos de uma parede e, para isso, utilizarei apenas um cubo com dimensões ajustadas, conforme ilustrado na figura 91

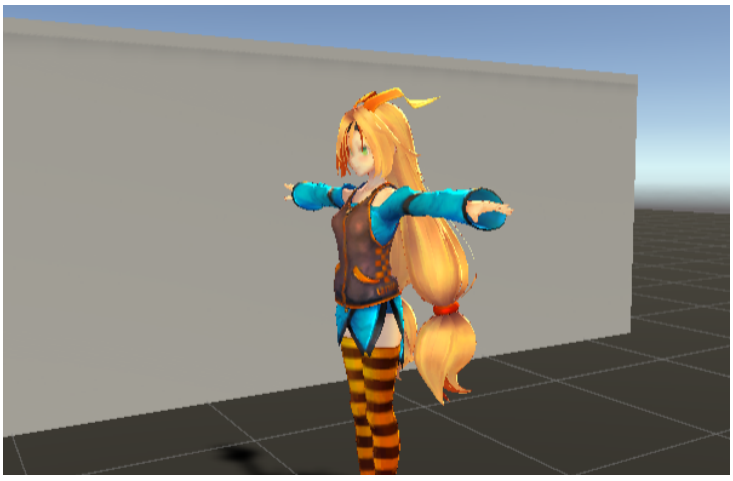

Figura 91 – Cena que será utilizada nesta seção.

Além disso, para facilitar no filtro dos objetos colididos, posteriormente, iremos adicionar um *Layer* único a nossa parede. Para isso, selecionamos o objeto no *Hierarchy* e na seção *Layer* no *Inspector* selecionamos a opção *Add Layer*, como visto na figura 92.

Feito isso, uma lista de *Layers* aparecerá, sendo reservados os *layers* de 1-7, incluiremos um novo no slot 8, que nomearei de *Wall*. O resultado pode ser visto na figura 93.

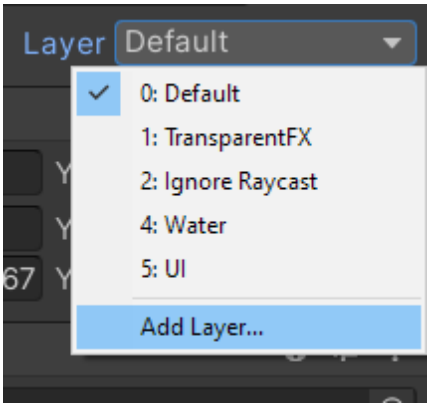

Figura 92 – Contexto para adicionar um novo *layer*.

Figura 93 – Novo *layer* adicionado no *slot* livre.

Para finalizar, basta adicionarmos o *layer* recém criado à nossa parede, como ilustrado na figura 94.

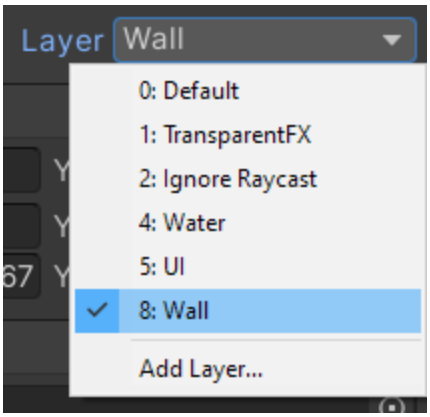

Figura 94 – Novo *layer* selecionado para o objeto *wall*.

3.5.6.2 Scripting

Com tudo pronto no editor, criaremos um *script* responsável pelo controle do *Hand IK*, que chamarei de **MecanimHandIK**. Essa classe terá 5 *fields*:

- Animator: fará referência ao **Animator** para que seja possível controlar a posição e rotação do *Inverse Kinematics Target*. Declarado na linha 3 do *code snippet*da figura

95;

- RightShoulder: fará referência ao *transform* do ombro da personagem. Será o ponto de partida para a esfera que encontrará por colisões. Pode ser visto na linha 6 do *code snippet* da figura 95;
- WallDistanceOffset: define o quão distante da parede estará a mão da personagem. Declarado na linha 9 do *code snippet* da figura 95;
- ArmLength: define o comprimento do braço do personagem, em outra palavras define o quão longe o alvo de *IK* terá efeito. Declarado na linha 12 do *code snippet* da figura 95;
- HeightOffset: controla a altura em que a mão pousará sobre a parede. Declarado na linha 15 do *code snippet* da figura 95.

```
1  public class MecanimHandIK : MonoBehaviour
2  {
3      private Animator animator;
4
5      [SerializeField]
6      private Transform rightShoulder;
7
8      [SerializeField] [Range(0, 0.1f)]
9      private float wallDistanceOffset = 0.035f;
10
11     [SerializeField] [Range(0f, 1f)]
12     private float armLength = 0.5f;
13
14     [SerializeField] [Range(-0.5f, 0.5f)]
15     private float heightOffset = 0f;
```

Figura 95 – *Fields* da classe recém criada.

Como pode ser visto na figura 96, o componente agora possui quatro campos editáveis com um *slider*. Os valores padrões foram bons valores encontrados por mim, mas variam por situação e modelo. Observe também que atribuímos o ombro direito ao campo *Right Shoulder*, sendo sua localização ilustrada na figura 97.

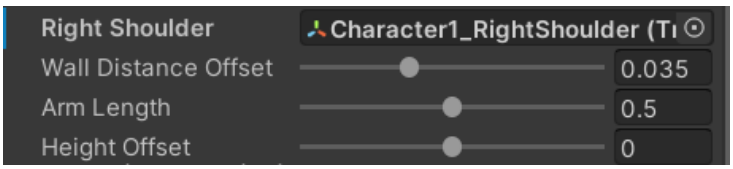

Figura 96 – Mecanim Hand IK script com fields serializados.

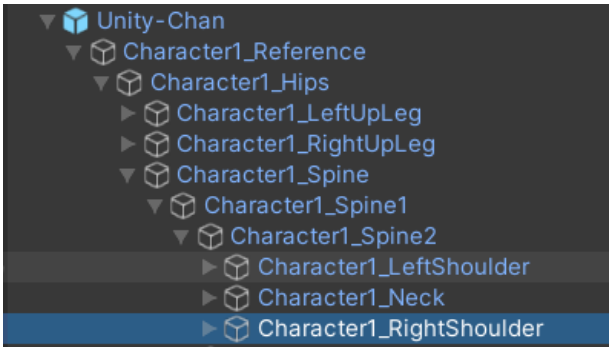

Figura 97 – Localização do ombro direito dentro da hierarquia.

```
1 private void Awake()
2 {
3     animator = GetComponent<Animator>();
4 }
```

Figura 98 – *Code snippet* do método *Awake*.

O primeiro passo é inicializar o *animator*. Para isso, basta pegar o componente através do método *Awake*, como mostra o *code snippet* da figura 98.

De forma bem semelhante ao *Foot IK*, daremos início à parte principal do script. O código será escrito dentro do método *OnAnimatorIK* e sua declaração pode ser vista na linha 1 do *code snippet* da figura 99.

O primeiro passo é verificar e encontrar se há alguma parede próxima o suficiente da personagem. Para isso, na linha 3 do *code snippet* da figura 99, chamamos o método *GetNearbyWallCollider*. Exploraremos então este método, ilustrado no *code snippet* da figura 101.

Primeiramente, devemos calcular a origem do *sphereOverlap*. Uma vez que iremos tratar a mão direita, podemos utilizar o ombro direito como origem, visto que ao esticar o braço, tanto o ombro quanto a mão alinham-se na mesma direção. Para isso, definimos a variável *spherePosition* (um *Vector3*) como a posição do ombro somado com um deslocamento à direita ($transform.right$). O valor multiplicado define o quão deslocado à direita será a esfera. Este valor pode ser parametrizado, no entanto, foi definido como 0.45 por ser um bom valor encontrado de forma empírica.

Como o método *overlapSphere* necessita, além de um ponto inicial, de um raio para a esfera, definimos a variável *radius* na linha 5, e atribuímos um valor que também pode ser parametrizado. Por fim, o parâmetro final da função é definido na linha 6, o $layerMask$ define que *layer* será filtrado. O funcionamento de *LayerMask* é um pouco diferente, sendo necessário utilizar operações bitwise, uma vez que seu tipo é definido

```
1 private void OnAnimatorIK(int layerIndex)
2 {
3     Collider wall = GetNearbyWallCollider();
4
5     if (wall == null) return;
6
7     Vector3 contactPoint = wall.ClosestPoint(rightShoulder.position)
8                            + (transform.up * heightOffset);
9
10    Vector3 direction =
11            (contactPoint - rightShoulder.position).normalized;
12    float distance =
13            Vector3.Distance(rightShoulder.position, contactPoint);
14
15    float angle = Vector3.Angle(transform.forward, direction);
16
17    if (distance < armLength && (angle > 10 && angle < 130))
18        SetIK(contactPoint, direction);
19 }
```

Figura 99 – *Code snippet* do método *OnAnimatorIK*.

como uma *flag* de 32 bits. Por utilizarmos o *slot* de número 8 para o *layer* da parede, selecionamos este através da operação $1 << 8$.

Agora que temos todas as variáveis de parâmetro, chamamos o método *OverlapSphere* e armazenamos seu resultado, um *array* de *collider*, como pode ser visto na linha 8. Uma representação visual pode ser visto na figura 100.

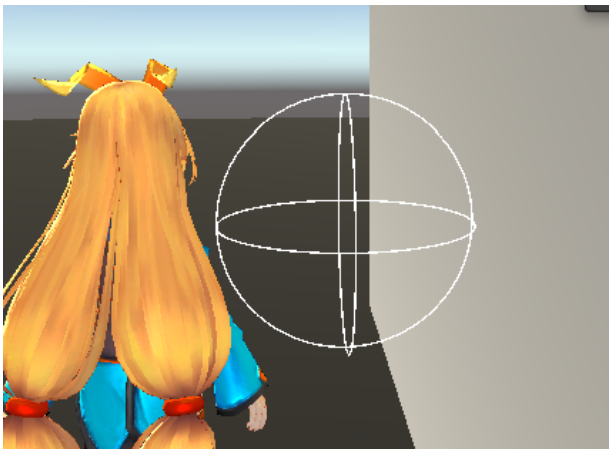

Figura 100 – Debug visual da *OverlapSphere*.

O que resta é retornarmos o valor do método. Caso o *array* de *collider* não esteja vazio (verificando se o tamanho é maior que 0), retornarmos o primeiro item. Caso não não haja *collider*, retornarmos *null*, como descrito na linha 11 do *code snippet* 101.

Com o método *GetNearbyWallCollider* conclúido, retornaremos à figura 99. Na

```
1  private Collider GetNearbyWallCollider()
2  {
3      Vector3 spherePosition = rightShoulder.position
4                               + transform.right * 0.45f;
5      const float radius = 0.3f;
6      const int layerMask = 1 << 8;
7
8      Collider[] hitColliders = Physics.OverlapSphere(spherePosition,
9                                                      radius, layerMask);
10
11     return hitColliders.Length > 0 ? hitColliders.First() : null;
12 }
```

Figura 101 – *Code snippet* do método *GetNearbyWallCollider*.

linha 5 vemos que caso não existe nenhuma parede próxima, retornaremos no método. Caso exista, o próximo passo é calcular o ponto de contato com a parede em que a mão irá se deslocar. Para isso, podemos usar o método *ClosestPoint* que nos retorna o ponto ($Vector3$) mais próximo ao parâmetro que passarmos. Neste caso, passaremos a posição do ombro direito, como pode ser visto na linha 7, ainda da figura 99. Já na linha 8 adicionaremos um deslocamento vertical baseado no campo $heightOffset$, para controlarmos a altura em que a mão repousará.

Com posse do ponto de contato, o próximo passo é calcular a direção da parede em relação ao ombro para calcularmos a distância e ângulo posteriormente. Para isso, basta subtrair o ponto inicial do ponto final. Neste caso, nosso ponto inicial é o ponto de contato $contactPoint$, e o ponto final a posição do ombro $rightShoulder.position$. Como não nos interessa a parte escalar do vetor, normalizamos o resultado, como mostra a linha 11 da figura 99.

Para calcular a distância é bem simples, basta utilizarmos o método *Vector3.Distance* e passar as variáveis de posição do ombro e ponto de contato, como pode ser visto nas linhas 12-13. Por fim, também calculamos o ângulo entre o vetor que aponta para frente ($transform.forward$) de nosso personagem e a direção do ponto de contato (que acabamos de calcular), o que pode ser feito com o método *Vector3.Angle*, conforme a linha 15 da figura 99.

Feito isto, podemos utilizar tais variáveis para calcular se a parede encontra-se numa distância menor que o tamanho do braço ($armLength$), e se a direção está dentro de um limite de ângulo (de forma que a personagem não tente mover o braço em um ângulo humanamente impossível). Neste caso, utilizei o limite $10 < angle < 130$. Este *check* encontra-se na linha 17 da figura 99.

Finalmente, caso o *check* da linha 17, ainda da figura 99, seja verdadeiro, iremos realizar a transformação no *IK* através do método *SetIK*, conforme o *code snippet* 102. Seu

funcionamento é bem semelhante ao visto no capítulo anterior, primeiramente devemos atribuir o peso 1 tanto à posição quanto à rotação, conforme as linhas 3 e 4 do *code snippet* da figura 102. Com isso, definimos a posição do alvo $goalPosition$ como o ponto de contato previamente calculado. Para termos mais controle, podemos subtrair pelo produto entre a direção e nossa variável que define o deslocamento da parede $wallDistanceOffset$, como pode ser visto na linha 6.

Na linha 7, ainda da figura 102, faremos a atribuição do $goalPosition$ ao sistema de *IK*, como vimos anteriormente também. Na linha 9 calculamos a rotação necessária para que a mão fique de maneira natural na parede. Para isso, o primeiro passo é rotacioná-la de forma que fique na direção do ponto de contato. Isso pode ser feito através do método *Quaternion.LookRotation*, utilizando o *direction* como parâmetro. Feito isso, rotacionaremos a mão de forma que a palma fique rente à superfície da parede, utilizando o método *Quaternion.Euler*, com componente $x = -80$, o efeito desejado é encontrado.

```
1 private void SetIK(Vector3 point, Vector3 direction)
2 {
3     animator.SetIKPositionWeight(AvatarIKGoal.RightHand, 1);
4     animator.SetIKRotationWeight(AvatarIKGoal.RightHand, 1);
5
6     Vector3 goalPosition = point - direction * wallDistanceOffset;
7     animator.SetIKPosition(AvatarIKGoal.RightHand, goalPosition);
8
9     Quaternion goalRotation = Quaternion.LookRotation(direction) *
10                              Quaternion.Euler(-80, 0, 0);
11    animator.SetIKRotation(AvatarIKGoal.RightHand, goalRotation);
12 }
```

Figura 102 – *Code snippet* do método *SetIK*.

Por fim, basta aplicarmos a rotação com o método *SetIKRotation* e finalizamos o *scripting*.

3.5.6.3 Resultados

Nas figuras 103 e 104 podemos observar os resultados sob diferentes perspectivas e diferentes distancias da parede. Já a figura 105 ilustra a animação da personagem aproximando-se cada vez mais da parede e como isso resulta em poses distintas.

3.5.7 Aplicação Real

O jogo *Read Dead Redemption III* é um grande exemplo de utilização desta técnica em situações diversas. A figura 106 demonstra a interação de um jogador ao buscar suprimentos, em que há todo um processo para abrir cada compartimento e pegar de

Figura 103 – Exemplo de resultado do *Hand IK*.

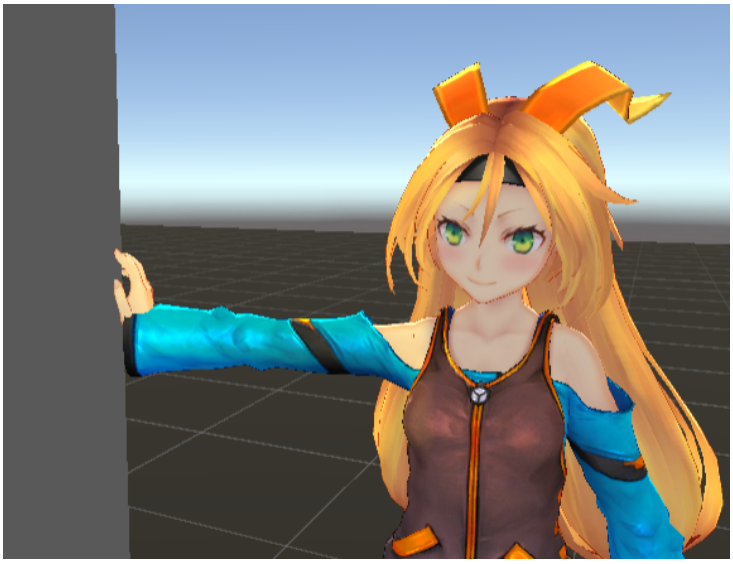

Figura 104 – Exemplo de resultado do *Hand IK*.

maneira realística cada objeto. Tal feito seria impossível de se animar em tempo hábil para todas as situações possíveis.

3.5.8 Limitações

Da mesma forma como na seção anterior, ao utilizarmos o módulo *Mecanim* do Unity nesta seção, estamos limitados a um modelo humanóide que possua um *Avatar* válido. Além do mais, em cenários mais complexos é necessário um tratamento mais rigoroso dos possíveis ângulos e rotações (uma parede com geometria mais complexa por

Figura 105 – Exemplo de resultado do *Hand IK* enquanto a personagem se move mais próximo da parede.

Figura 106 – *Read Dead Redemption II*. Animação em que o jogador abre um armário e coleta individualmente cada item, dependendo da escolha do jogador.

exemplo). Há também a limitação da corrente *IK* estar apenas nos membros, sem envolver o restante do corpo, o que pode ser um grande fator limitante para alguns casos.

3.6 Animação por Interpolação com *Blendtree* em tempo real

As subseções a seguir tratarão sobre as orientações de uso do método de interpolação com *blendtree*.

3.6.1 O que é?

Interpolar consiste em estimar novos dados a partir de dados já existentes. A animação por interpolação segue a mesma ideia, onde podemos criar novas animações com base em dois ou mais *key frames*, como ilustrado na figura 107.

Figura 107 – Ilustração de *frames* gerados a partir de *keyframes* já definidos.

Essa técnica não é nenhuma novidade, ao utilizarmos softwares de modelagem e animação 3D (como o *Blender*), definimos *keyframes* e deixamos o computador fazer o trabalho pesado: gerar novos frames entre os *keyframes* definidos. No entanto, o interessante é pensar na aplicação desta técnica de forma procedural, é possível gerar novas animações baseadas em curvas em tempo real, utilizando poucos *keyframes*. Mais do que interpolar entre *keyframes*, é possível também interpolar entre animações, estas também podendo ser geradas de forma procedural.

No Unity, as *blend trees* permitem-nos justamente fazer essas combinações de animações, utilizando variáveis que controlam essa mistura, como veremos nas próximas seções.

3.6.2 Quando?

Animar utilizando meios convencionais, apesar de preciso, demanda muito esforço. Por conta disso, esta técnica encaixa-se muito bem em situações em que animar de forma convencional seria inviável por conta do tempo, ou em casos de animações mais simples.

O interessante está em se tratar de um método muito versátil, podendo ser utilizado em inúmeros cenários, desde personagens a animações de objetos ou efeitos visuais.

3.6.3 Como?

O Unity fornece-nos uma ferramenta poderosa conhecida como *Blend tree*, ilustrada na figura 108. Essencialmente, é um método para combinar e misturar diferentes animações através de parâmetros unidimensionais ou bidimensionais. No entanto, ao invés de apenas utilizarmos animações completas, usaremos *keyframes* únicos, interpolando-os com diferentes métodos para a criação de animações.

Para termos maior controle sob a interpolação, utilizaremos curvas editáveis em tempo de execução. A ideia é utilizar um parâmetro unidimensional na *Blend Tree* que seguirá uma curva para cada animação. Para isso, dividiremos a curva em pontos equidistantes pelo número de animações na *Blend Tree*, sendo cada ponto associado a um *keyframe*.

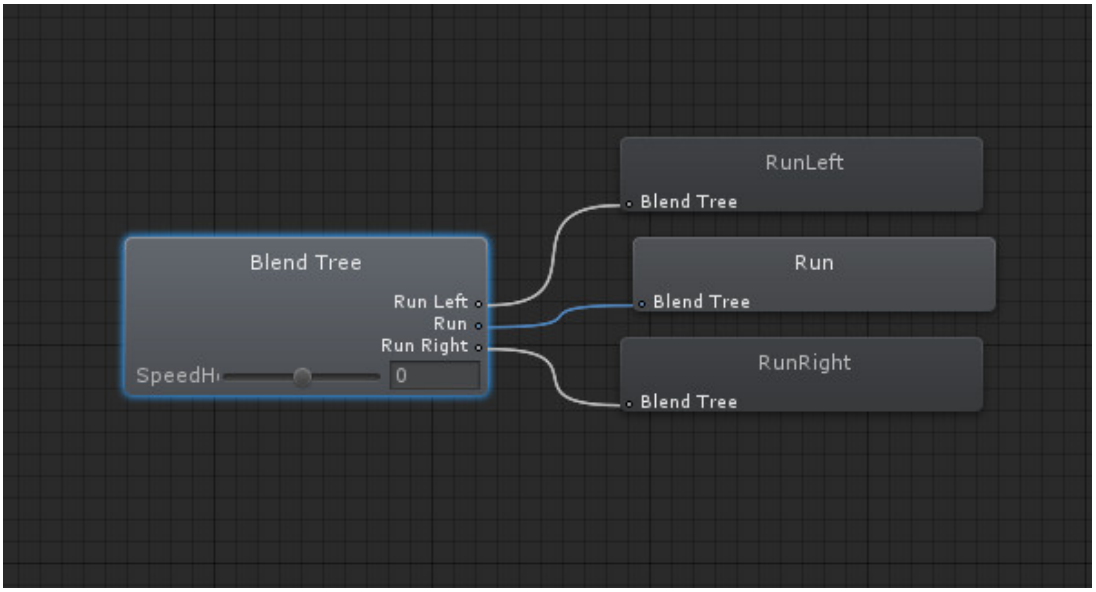

Figura 108 – Exemplo de *Blend Tree* com parâmetro unidimensional.

3.6.4 Complexidade de Implementação

A complexidade de implementação depende do escopo da animação e do entendimento matemático do desenvolvedor. No geral, como veremos nas próximas seções, a parte de programação requer pouco esforço, mas uma atenção especial deve ser dada aos *keyframes* e à escolha de um tipo de interpolação para alcançar melhores resultados.

3.6.5 Pré-Requisitos

É necessário um nível intermediário no Unity. O desenvolvedor necessita conhecer componentes como *Animator*, *Animation* e *Blend Tree*. Além disso, deve ser capaz de trabalhar com *coroutines* e manipular parâmetros de animação. Conhecimento matemático sobre interpolação e curvas é essencial para bons resultados.

3.6.6 Exemplo de Aplicação

A seguir o exemplo de aplicação da técnica será apresentado, divido em subseções que representam partes da implementação.

3.6.6.1 Modelo, *Keyframes* e *Blend Tree*

Antes de iniciarmos a aplicação da técnica, é necessário um modelo 3D , bem como *keyframes* para interpolar. Como modelo irei utilizar o definido em <https://assetstore.unity.com/packages/3d/characters/robots/space-robot-kyle-4696>.

A criação dos keyframes pode ser feita em ferramentas externas de modelagem, como o Maya e Blender, no entanto, utilizarei o animador interno do Unity. Após inserirmos o modelo na cena e o selecionarmos no *Hierarchy*, podemos criar uma animação nova (que no nosso caso conterá apenas um keyframe) ao clicar no botão **Create**.

Criaremos 3 *animation clips* distintos. A figura 109 ilustra os keyframes que iremos criar, sendo o 3° keyframe o espelhamento do keyframe 2, isto é, os membros direitos do personagem agora terão os mesmo valores dos membros esquerdos, e vice-versa (como visto no keyframe 4 da figura 110

Com isso, como podemos ver na figura 110, utilizando os 3 clips é possível criar uma animação cíclica, que estará em animação sempre que o jogador estiver andando. Os keyframes 1, 3 e 5 correspondem ao keyframe 1 da figura 109 e chamaremos de "Walk". Já o keyframe 2 corresponde ao keyframe 2 da figura 109, enquanto o keyframe 4 corresponde ao keyframe 2 espelhado. Estes serão chamados de "Walk L"e "Walk R", respectivamente.

Ao criarmos clips de animação para o nosso personagem, automaticamente é criado também um *Animator*, responsável por gerenciar e dar vida aos clips. É na janela do *Animator* também que podemos criar nossa *Blend tree*, basta clicarmos com o botão direito no fundo e ir em *Create State* > *From New Blend Tree*. Para a recém criada *Blend Tree* estar ativa assim que rodarmos a cena, clique com o botão direito sobre ela e vá em *Set as Layer Default State*.

Clique duas vezes sobre o novo quadro gerado, isso levará a sua própria camada. Aqui, ao clicarmos sobre a *Blend Tree*, será possível a popularmos no *Hierarchy* ao inserirmos mais espaços no botão de adição. Seguiremos a mesma ordem apresentada na

Figura 109 – Os dois *keyframes* que iremos utilizar.

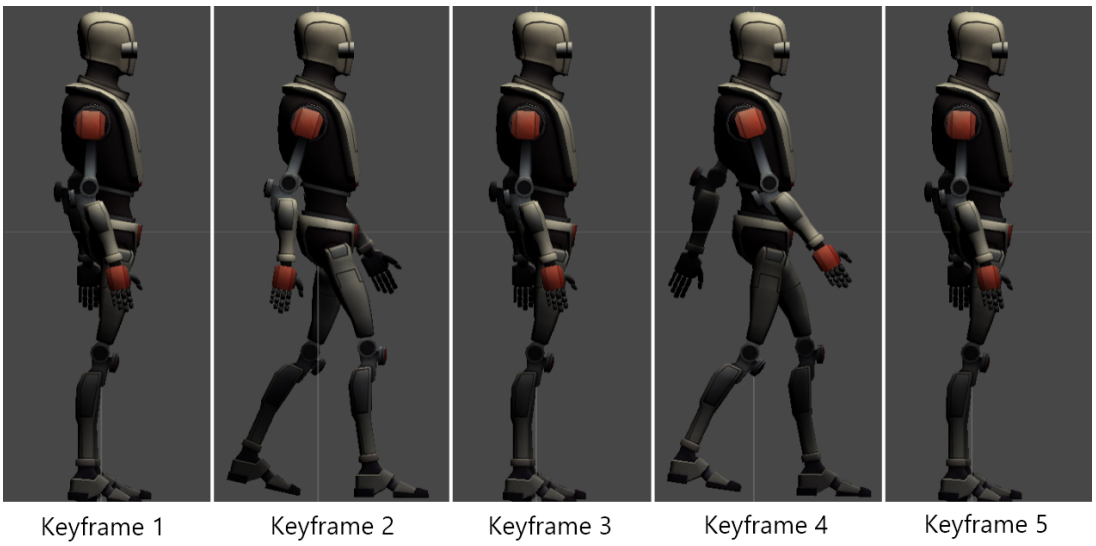

Figura 110 – *Keyframes* a serem usados na *Blend Tree*

figura 110, resultando na figura 111.

Como são usados 5 clips no total, a *Blend Tree* automaticamente calcula o *threshold* (o limite que define quando outro clip iniciará) dividindo em 5 partes iguais, começando de 0 e indo até 1. Os valores intermediários (os não *threshold*) são a interpolação entre dois clips. A figura 112 apresenta a *Blend Tree* após a inserção dos clips na ordem correta. Repare que há um *slider* com nome de Blend, este é o parâmetro responsável por interpolar entre os clips.

Para ilustrar o funcionamento, ao rodarmos a *Blend Tree* em uma iteração completa (parâmetro Blend de 0 a 1) de forma linear, obteremos um resultado semelhante à figura

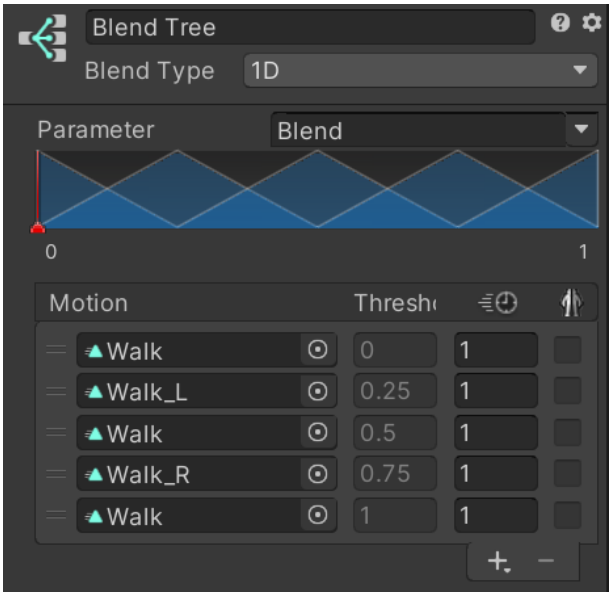

Figura 111 – Visão da *Blend Tree* no Hierarchy após a configuração.

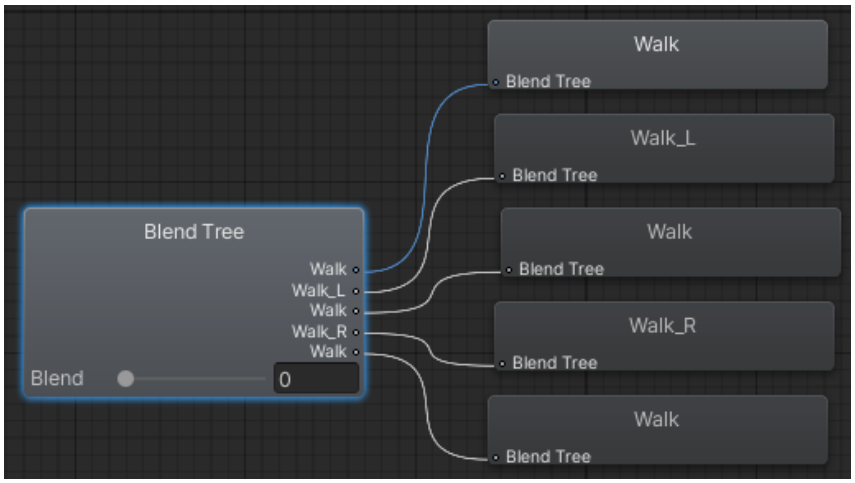

Figura 112 – *Blendtree* após inserção dos *clips*.

113.

3.6.6.2 Scripting

O próximo passo é a criação de um *script* responsável pela interpolação, que chamarei de **BlendTreeInterpolation**. Essa classe terá 3 *fields*:

- Animator: fará referência ao **Animator** para que seja possível controlar o parâmetro **Blend** da *blend tree*. Declarado na linha 4 do *code snippet* da figura 114;
- AnimationCurve: curva responsável pelo *sample* de valores da variável **Blend**. Declarado na linha 1 do *code snippet* da figura 114;

Figura 113 – Frames gerados pela interpolação dos *keyframes* na *Blend Tree*.

- AnimationDuration: um *float* que ditará a velocidade/duração da animação em segundos. Declarado na linha 2 do *code snippet* da figura 114.

```
1 [SerializeField] private AnimationCurve curve;
2 [SerializeField] [Range(0, 10f)] private float animationDuration = 1.25f;
3
4 private Animator animator;
```

Figura 114 – Frames gerados pela interpolação dos *keyframes* na *Blend Tree*.

Como pode ser visto na figura 115, o componente possui agora dois campos editáveis, a curva e a duração da animação. Ao clicarmos sobre a caixa ao lado do campo *Curve*, uma janela de edição de curva se abrirá. Discutiremos seu uso mais a frente.

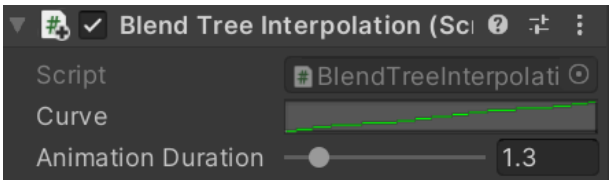

Figura 115 – Blend Tree Interpolation script com *fields* serializados

Devemos inicializar o animator que já se encontra presente em nosso *gameObject*, para isso basta pegarmos o componente assim que o script for inicializado, como pode ser visto no *code snippet* da figura 116.

```
1 private void Awake()
2 {
3     animator = GetComponent<Animator>();
4 }
```

Figura 116 – *Code snippet* do método *Awake*.

O próximo passo é a criação do método responsável pelo controle do parâmetro *Blend* seguindo o sample da curva. No entanto, como esta é uma tarefa assíncrona, que ocorre ao longo do tempo e não apenas em um único frame, utilizaremos **Coroutines**. Para isso, nosso método deve retornar um *IEnumerator*, como pode ser visto na linha 1 do *code snippet* da figura 117.

Na linha 3 da figura 117, guardaremos o tempo atual em que o método iniciou, a fim de calcularmos se o tempo decorrido já alcançou a duração da nossa animação. Na linha 5 faremos justamente isso, executaremos o método enquanto o tempo atual é menor que a soma do *timeStamp* e a duração da animação. Nas linhas 7-8 inicializaremos a variável $animationTimeFraction$, que será a fração atual da duração da animação, calculada como a diferença entre o tempo atual e o inicial, dividido pela duração da animação. Na linha 9 utilizaremos a variável $animationTimeFraction$ para encontrarmos a posição atual da animação na curva.

Finalmente, nas linhas 11-12, ainda da figura 117, iremos atualizar o parâmetro *Blend* do *animator* como a interpolação (sem limites) de $animationTimeFraction$ entre 0 e 1. Na linha 14, como nossa função não possui retorno, faremos o *yield* de *null* em cada iteração.

Para fins de testes, iremos também criar um *coroutine* que rodará o método $InterpolateWithCurve$ infinitamente. Da mesma forma, esse método deve ser uma *coroutine*, como mostra a linha 1 do *code snippet* da figura 118. Dentro do *loop* infinito da linha 3, iniciaremos a *coroutine* com o método *StartCoroutine* passando como parâmetro o método recém criado $InterpolateWithCurve$. Como devemos esperar até o fim da execução da *coroutine* atual para iniciar uma nova, faremos um *yield* que esperará até o fim da *coroutine* atual esperando a duração da animação, conforme pode ser visto na linha 6.

3.6.6.3 Curvas e Resultados

Agora que temos tudo funcionamos, basta definirmos uma curva que representará a forma em que os *keyframes* serão interpolados. Essa é uma parte importantíssima, uma vez que que uma boa animação depende tanto de bons keyframes quanto de uma boa interpolação. Ao clicarmos na caixa da curva, visto na figura 115, há alguns *presets*

```
1 private IEnumerator InterpolateWithCurve()
2 {
3     float timeStamp = Time.time;
4
5     while (Time.time < timeStamp + animationDuration)
6     {
7         float animationTimeFraction =
8                     (Time.time - timeStamp) / animationDuration;
9         animationTimeFraction = curve.Evaluate(animationTimeFraction);
10
11        animator.SetFloat("Blend",
12                    Mathf.LerpUnclamped(0f, 1f, animationTimeFraction));
13
14        yield return null;
15    }
16 }
```

Figura 117 – *Code snippet* do método *InterpolateWithCurve*.

```
1 private IEnumerator AnimationLoop()
2 {
3     while (true)
4     {
5         StartCoroutine(InterpolateWithCurve());
6         yield return new WaitForSeconds(animationDuration);
7     }
8 }
```

Figura 118 – *Code snippet* do método *AnimationLoop*.

disponíveis na parte de baixo. O primeiro *preset* é de uma curva linear, que gerará animações semelhantes ao demonstrado na figura 113. No entanto, poucos serão os casos em que uma interpolação linear entre *keyframes* gerará uma animação satisfatória, uma vez que criará animações abruptas, sem continuidade nos movimentos.

Ironicamente, nosso modelo é um robô, um bom candidato para utilizar interpolação linear. No entanto, para demonstrar a capacidade e flexibilidade desta técnica, utilizaremos uma interpolação cúbica. A figura 119 ilustra um comparativo entre 3 tipos de interpolação.

Para que possamos controlar a curva de acordo com os *keyframes*, devemos adicionar *keys* ao editor de curva para ter um maior controle. Dividindo as coordenadas em partes iguais pelo número de clips na *blend tree*, no nosso caso 5, teremos *keys* nas coordendas $(0, 0)$, $(0.25, 0.25)$, $(0.5, 0.5)$, $(0.75, 0.75)$ e $(1, 1)$. Essa tarefa pode ser automatizada por *script*, mas para ilustração faremos manualmente.

Criamos as *keys* ao clicar com o botão direito do mouse e $CreateKey$. Com as *keys*

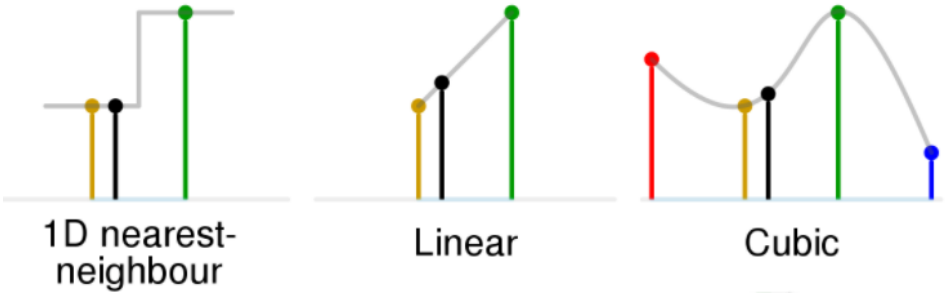

Figura 119 – Comparativo entre alguns tipos de interpolação de uma dimensão.

no lugar, iremos rotacionar o ponto de controle das *keys* 2 e 4, que correspondem ao clips "Walk L"e "Walk R", de modo que a linha de controle fique paralela ao eixo x, como pode ser visto na figura 120.

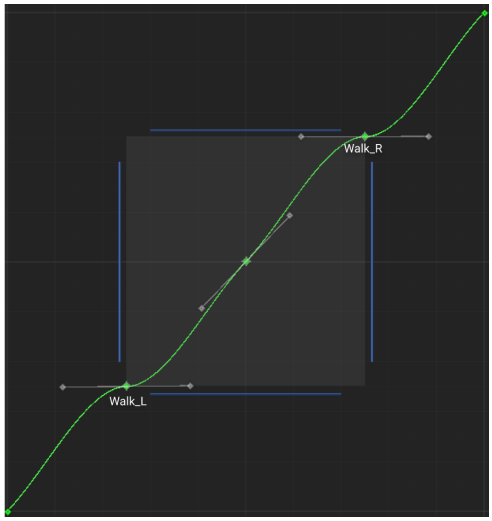

Figura 120 – Curva após a mudança nos pontos de controle.

Com isso temos a interpolação cúbica, em que a continuidade do movimento é preservada, tornando a animação mais natural. Os frames da animação com interpolação cúbica podem ser vistos na figura 123.

É possível utilizar o que foi apresentado nessa seção para casos mais complexos. A figura 121 ilustra a utilização da mistura de duas *Blend trees*, uma para animação de andar, e outra para animação de correr. Portanto, há a possibilidade de gerar frames entre os frames gerados entre animações, a figura 122 mostra o momento de transição entre as duas animações.

3.6.7 Aplicação Real

Diferentes curvas podem ser utilizadas para dar vida de forma diferente às animações. Por conta disso, esta técnica mostra-se muito flexível. Suas aplicações abrangem todo

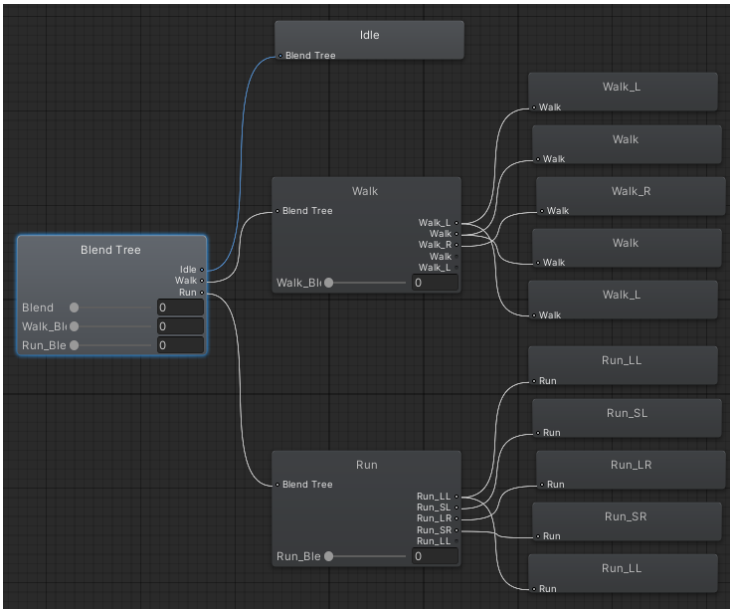

Figura 121 – Exemplo de utilização de múltiplas *Blend Trees*

Figura 122 – Exemplo de transição entre animações interpoladas.

tipo de animação, desde personagens a animações de objetos ou efeitos visuais. Porém, sua principal qualidade é a eficiência, com poucos *keyframes* é possível criar animações de alta qualidade, o que é muito vantajoso para estúdios de jogos menores. Há ainda a possibilidade de mesclar múltiplas *blend trees*, a fim de criar animações mais complexas que dependam de várias variáveis.

O jogo **Overgrowth**, desenvolvido apenas por duas pessoas, é um exemplo de

Figura 123 – Frames gerados pela interpolação cúbica dos *keyframes*

sucesso desta técnica. Ao combinar com outros métodos de animação, com poucos *keyframes* foi possível chegar a níveis impressionantes de animação, ilustrando sua eficiência. Como exemplo, temos a movimentação do personagem principal, sendo construída a partir de apenas 13 *keyframes*, como ilustra a figura 124.

3.6.8 Limitações

Por se tratar de uma técnica bem versátil, as limitações podem ser superadas ao combiná-la com outros métodos. No entanto, apesar de demandar poucos recursos, é notório que a qualidade de uma animação meticulosamente feita por um animador é dificilmente superada.

Figura 124 – Exemplo de *keyframes* usados juntamente com a técnica para gerar animações convincentes.

4 Conclusão

Neste capítulo, a conclusão do trabalho é apresentada, destacando primeiramente uma visão geral do trabalho, as contribuições para a área de animação procedural, as limitações do trabalho e, por fim, as propostas de trabalhos futuros.

4.1 Visão Geral do Trabalho

No primeiro capítulo foi apresentada a introdução do trabalho, com o contexto sob a qual o trabalho foi inserido, bem como a motivação, sua justificativa, os objetivos do trabalho, a metodologia na sua realização, e sua estruturação.

No segundo capítulo, a fundamentação teórica necessária não somente para conceber o trabalho, mas como base para as recomendações práticas do capítulo posterior, foi apresentada. Primeiramente o conceito de animação procedural foi introduzido, de forma a contextualizar e oferecer uma visão mais ampla antes da aplicação. Após isso, alguns conceitos mais concretos que foram utilizados em técnicas no capítulo 3 foram ilustrados. Termos e conceitos chaves da *engine* utilizada também foram apresentados. Por fim, os principais conceitos matemáticos utilizados no capítulo 3 foram explanados, incluindo interpolação, vetores e quatérnios.

O terceiro capítulo apresentou as recomendações de aplicação dos métodos de animação procedural. Os métodos que servem como base para aplicações mais complexas foram primeiramente apresentados, incluindo o algoritmo FABRIK e sua expansão para múltiplos *end-effectors*. Após isso, a implementação do FABRIK foi utilizada para apresentar os métodos de *Foot IK* e *Hand IK*. Por fim, diferentemente dos outros métodos que utilizam *inverse kinematics*, a técnica de Interpolação com *Bleendtree* é apresentada.

4.2 Contribuições

Como principal contribuição deste trabalho, evidencia-se a construção de um guia construído a partir da escolha de técnicas de animação procedural baseadas em três critérios: relevância na indústria de jogos, retorno significativo de sua aplicação e facilidade de implementação.

Além disso, o guia oferece um passo-a-passo focado não apenas na aplicação em si, como também na reflexão sobre o processo de implementação. Tal formato das recomendações ainda oferece a possibilidade de uso do guia para referências futuras, com seções dedicadas a como e quando utilizar os métodos, além de seções que discutem a

complexidade e uso em cenários reais. A partir desses critérios, o leitor terá uma base sólida do tema para compreender e aplicar técnicas de animação procedural.

4.3 Limitações

O presente trabalho expõe recomendações para a aplicação de múltiplas técnicas de animação procedural, incluindo seções que abordam quando, como e a complexidade da técnica a ser aplicada.

No entanto, sem um cenário de uso real para aplicação das técnicas, as nuances da implementação, como dificuldades únicas do contexto inserido, ou alteração da técnica para se encaixar em determinadas mecânicas ou limitações impostas pelo *game design*, são perdidas e dificultam a criação de um guia mais abrangente.

Além disso, o trabalho carece de técnicas que englobam outras áreas da animação procedural, como sistemas de partícula, simulação de tecido e pelagem, e dinâmicas de *rigidbodies*.

4.4 Trabalhos Futuros

Como uma proposta para trabalho futuro, tem-se a implementação de diversos outros métodos de animação procedural, com foco em outras áreas de atuação além da animação de personagens, como sistemas de partícula, simulação de tecido, e dinâmicas de *rigidbodies*.

Além disso, outra proposta é a inserção destes métodos que foram apresentados dentro de um cenário real, aplicada a jogos que aproveitem os benefícios e resultados das técnicas. Dessa forma, evidenciam-se as discussões feitas de quando, como e onde utilizar os métodos.

Referências

ARISTIDOU, A.; CHRYSANTHOU, Y.; LASENBY, J. Extending fabrik with model constraints. *Computer Animation and Virtual Worlds*, 2016. Citado na página 52.

ARISTIDOU, A.; LASENBY, J. Fabrik: A fast, iterative solver for the inverse kinematics problem. *Graphical Models*, 2011. Citado 3 vezes nas páginas 15, 24 e 36.

BEGGS, J. *Kinematics*. [S.l.]: Hemisphere Publishing Corporation, 1983. Citado na página 13.

BROWN, J.; SCIREA, M. Procedural generation for tabletop games: User driven approaches with restrictions on computational resources. In: _____. [S.l.: s.n.], 2020. Citado na página 11.

BRUDELIN, A. *Procedural Motion Control Techniques for Interactive Animation of Human Figures*. Tese (Doutorado), 1996. Citado na página 11.

GOLDMAN, R. Understanding quaternions. *Graphical Models*, 2011. Citado na página 21.

GREGORY, J. *Game Engine Architecture*. [S.l.]: CRC Press, 2009. Citado na página 8.

HALLIDAY, D.; RESNICK, R.; WALKER, J. *Fundamentals of Physics*. [S.l.: s.n.], 2010. Citado 2 vezes nas páginas 18 e 19.

IMPORTANT Classes - Vectors. <https://docs.unity3d.com/Manual/VectorCookbook.html>. Accessed: 2021-04-20. Citado na página 20.

KUCUK, S.; BINGUL, Z. Robot kinematics: Forward and inverse kinematics. 2006. Citado na página 14.

PRESS, W. et al. *Numerical Recipes: The Art of Scientific Computing*. [S.l.]: Cambridge University Press, 1986. Citado na página 17.

RUGGILL, J.; MCALLISTER, K.; NICHOLS, R. *Inside the Video Game Industry: Game Developers Talk About the Business of Play*. [S.l.]: Routledge, 2012. Citado na página 9.

SANDERSON, G.; EATER, B. *Visualizing quaternions: An explorable video series*. <https://eater.net/quaternions>. Disponível em: <https://eater.net/quaternions>. Citado na página 22.

UNITY'S important classes. <https://docs.unity3d.com/Manual/ScriptingImportantClasses.html>. Accessed: 2021-04-19. Citado na página 16.

UNITY'S interface Documentation. <https://docs.unity3d.com/Manual/UsingTheEditor.html>. Accessed: 2021-04-18. Citado 2 vezes nas páginas 15 e 18.

WILLIAMS, R. *The Animator's Survival Kit–Revised Edition: A Manual of Methods, Principles and Formulas for Classical, Computer, Games, Stop Motion and Internet Animators*. [S.l.]: Faber Faber, Inc., 2009. Citado na página 11.

WOLF, M. *The Video Game Explosion: A History from PONG to Playstation and Beyond*. [S.l.]: ABC-CLIO, 2008. Citado na página 8.

ZACKARIASSON, P.; WILSON, T. *The Video Game Industry: Formation, Present State, and Future*. [S.l.]: New York: Routledge, 2012. Citado 2 vezes nas páginas 8 e 9.