UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

LABORATÓRIOS DE INFORMÁTICA

Animação utilizando *Inverse Kinematics*

Grupo 99

Autores:

Francisco Lira (A73909)
Joel Gama (A82202)
Paulo Barbosa (A82886)

22 de Julho de 2020

Resumo

O presente relatório surge no âmbito da Unidade Curricular de Laboratórios de Engenharia Informática. Nele apresentamos a abordagem adotada pelo grupo para responder ao tema selecionado, "Animação utilizando *Inverse Kinematics*". Para além da abordagem são mostrados os resultados obtidos e as limitações da solução.

Conteúdo

Introdução

A utilização de Computação Gráfica para animação tem sido um tema que se destaca cada vez mais com a evolução das tecnologias, técnicas e algoritmos de animação.

Desta forma, como todos os elementos do grupo frequentam o perfil de Computação Gráfica e animação foi um tema pelo qual todos mostraram igual interesse foi decidido realizar o projeto de LEI sobre de uma técnica específica de animação: Animação utilizando *Inverse Kinematics*.

A partir deste projeto foi possível criar um programa que faz a animação de um esqueleto e fazer a simulação da presença de objetos dos quais o esqueleto se desvia para não colidir. Nos próximos capítulos apresentamos o algoritmo responsável pelas operações de movimento e os resultados obtidos.

Por fim, são apresentadas as limitações do programa desenvolvido.

Caso de Estudo

O nosso caso de estudo consiste em realizar animações utilizando *Inverse Kinematics*, que permite calcular a posição dos ossos de um esqueleto sabendo apenas o ponto final (*target*).

Mas o que é *Inverse Kinematics*? *Inverse Kinematics* é um processo matemático que permite calcular os parâmetros variáveis das *joints*. Estes parâmetros são necessários para colocar o final de uma *chain* (conjunto de ossos ligados) numa determinada posição e orientação em relação ao início da mesma. Dados os parâmetros da *joint*, a posição e orientação do final da *chain*, por exemplo, a mão do personagem, os cálculos podem ser realizados diretamente usando fórmulas trigonométricas. Este processo é conhecido como *forward kinematics*.

Neste caso específico será utilizado o algoritmo *FABRIK: forward and backward reaching inverse kinematics*, uma variação do *Inverse Kinematics*. Este algoritmo destaca-se dos outros existentes por ser leve e por gerar posses mais realistas. Usando o *FABRIK* o problema de determinar as posições de cada *joint* foi reduzido a encontrar pontos numa linha, evitando assim cálculos complexos.

Estruturas de Dados

Depois de bem estudado o problema chegou a altura de tomar as decisões no que toca à estrutura de dados a ser usada para representar os esqueletos. Desse modo surgiram duas classes:

- *Bone*
- *Skeleton*

A classe Bone vai representar cada osso e a classe *Skeleton* vai representar um esqueleto ou parte de um esqueleto. Começando por explicar a classe mais simples, temos a *Skeleton* que é composta por os seguintes atributos:

```
std::vector<skeleton*> children;
float target[3];
Bone* me;
skeleton* parent;
```

Nesta classe temos um vetor de *skeleton's* que vão ser os filhos (continuação do esqueleto) do esqueleto presente. De seguida temos, ou não, um *target*. Este *target* é normalmente usado nos ossos que se encontram nas pontas do esqueleto e tem como finalidade controlar todo o algoritmo. Temos também toda a informação referente ao osso presente no *me* e por fim temos um apontador para o osso pai, caso este exista.

Passando agora para a classe que representa cada osso, esta tem os seguintes atributos:

```
float start[3];
float end[3];
float size;

float original_vec[3];
float original_in[3];
float original_out[3];

float angle_in;
float angle_vector_in[3];

float angle_out;
float angle_vector_out[3];

bool fixed_in;
bool fixed_out;
```

Os primeiros três atributos são bastante simples e representam o inicio, fim e tamanho do osso. De seguida temos três variáveis usadas para guardar dados que são usados durante o algoritmo *FABRIK*. Estas variáveis vão guardar as posições originais de três vetores, visto que eles vão ser alterados ao longo do programa. Para terminar temos o *angle in*, *angle vector in* e *fixed in* (temos o mesmo para o *out*). O *angle in* é o ângulo máximo que um osso vai dobrar em relação ao *angle vector in* e o *fixed in* determina se o vetor *angle vector in* vai ser ou não atualizado sempre que o osso se move (por exemplo, caso se queira que um vetor esteja sempre perpendicular ao osso é necessário atualizar esse vetor para ele rodar com o osso, nesse caso o *fixed in* ficaria a *false*).

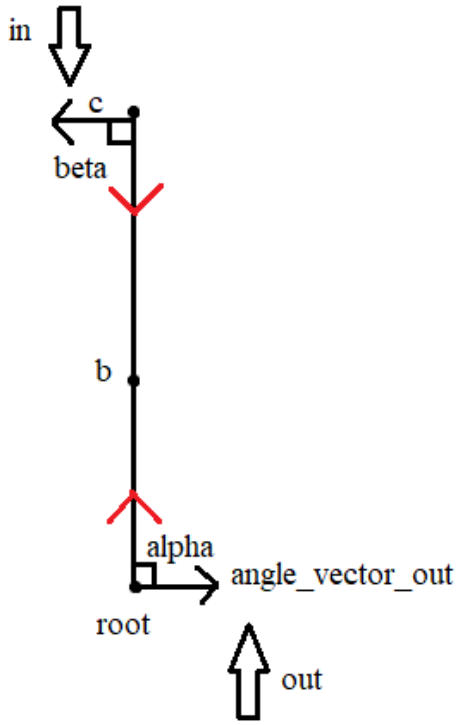

Figura 4.1: Exemplo dos atributos de um osso.

De modo a explicar melhor alguns dos atributos temos a figura 4.1 que mostra o *angle vector out* e *in* e o respetivo ângulo que formam com o ponto b. No caso do ângulo *alpha* ser maior que *angle out* o ponto b será rodado e o mesmo acontece para o ângulo beta.

4.1 Exemplo de Esqueleto

De seguida irá mostrar-se como é representado um pequeno esqueleto de forma a dar a entender melhor como é representada a nossa estrutura de dados.

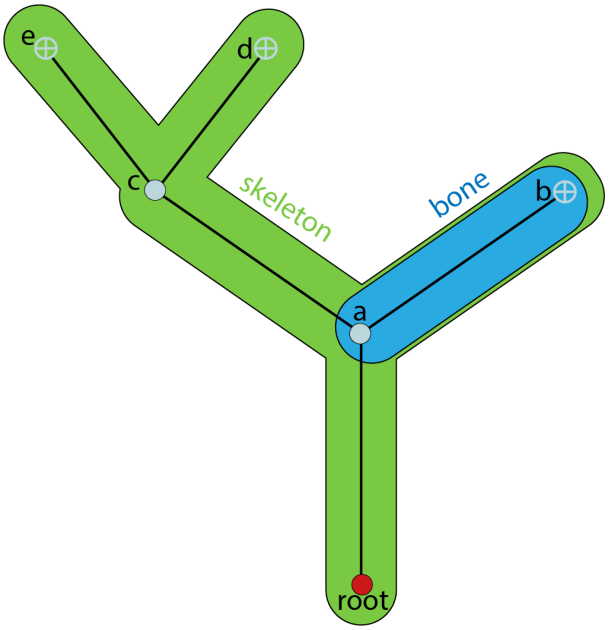

Figura 4.2: Exemplo de esqueleto.

Como podemos observar pela figura 4.2 o esqueleto tem inicio no *root* e contem toda a árvore. Vai agora descrever-se o conteudo do primeiro apontador para o skeleton para dar uma ideia de como estão os dados:

- *target=Null*
- *parent=Null*
- *children = [skeleton right , skeleton left]*
- *Bone* me*
  - *start = (0,0,0)*
  - *end = (0,1,0)*
  - *size = 1*
  - *...*

Toda a árvore vai seguir uma estrutura parecida à descrita anteriormente. A única diferença é que nas folhas estas vão ter o target definido com algum valor.

Algoritmo

5.1 *FABRIK: a new heuristic IK solution*

Vai se falar agora do algoritmo responsável pela posição de todos os ossos do esqueleto. Primeiramente será abordado o algoritmo na sua forma mais básica e de seguida vai estender-se esse para esqueletos mais complexos, ou seja, com vários *end effectors* (*end effector* é a ponta de um esqueleto como se pode observar na figura 5.1) e até restrições.

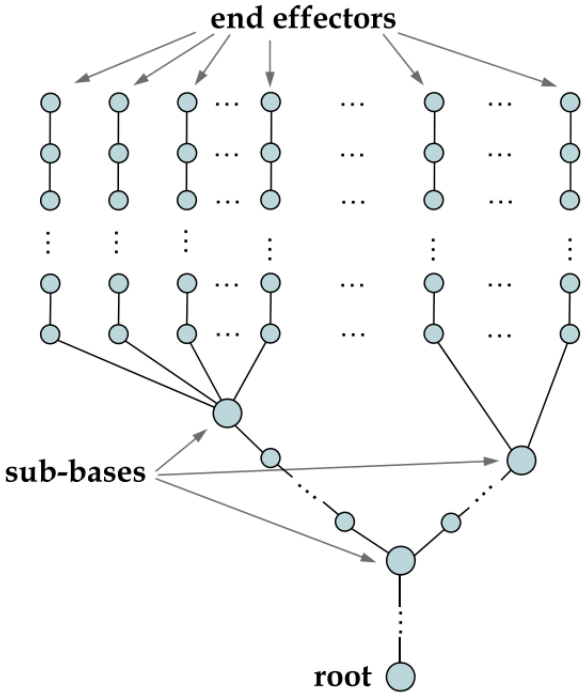

Figura 5.1: Exemplo de esqueleto.

Este algoritmo está dividido em duas grandes partes:

- *Forward reaching*, ou como preferimos chamar, *In* ou *Inward*. Nesta etapa o algoritmo vai correr o esqueleto do seu *end effector* até à raiz do mesmo. Primeiramente começa por colocar o *end effector* na posição do *target* (figura 5.2 a). De seguida vai descendo na árvore e para cada osso vai traçar uma reta desse mesmo osso até ao seguinte, posicionando posteriormente o osso onde se encontra a uma distancia correta do osso que estava a sua frente (fazendo assim com que cada osso mantenha o seu tamanho - figura 5.2 b,

c e d). No final desta etapa a *root* vai ser deslocada e cabe à segunda parte do algoritmo corrigir essa deslocação.

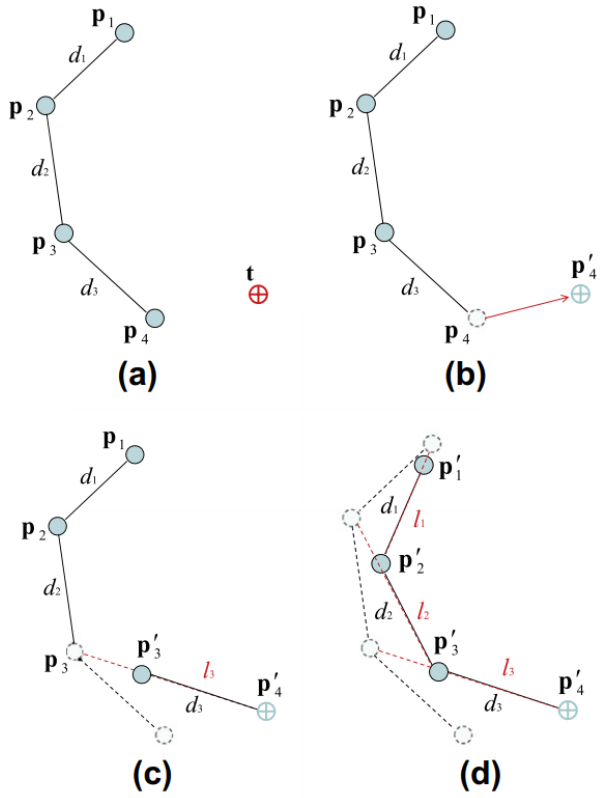

Figura 5.2: Exemplo da parte *inward* do algoritmo.

- *Backward reaching*, ou como preferimos chamar, *Out* ou *Outward*. Nesta etapa a primeira coisa a fazer será colocar a *root* na sua posição original, visto que a *root* do esqueleto nunca se vai mover e de seguida é corrido todo o esqueleto até chegar ao *end effector*. Para cada osso (além da *root*) é traçada uma reta do mesmo até ao próximo e é ajustada a posição do osso seguinte com base no tamanho (como podemos observar na figura 5.3).

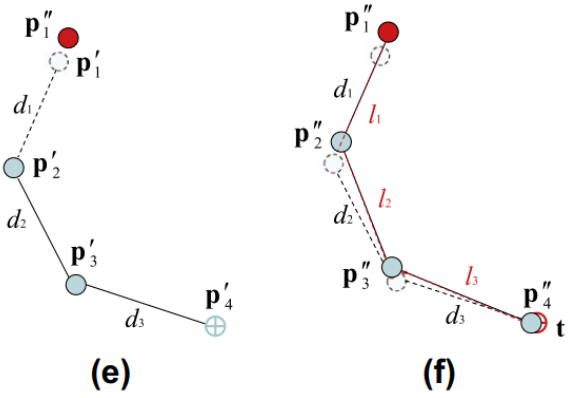

Figura 5.3: Exemplo da parte *outward* do algoritmo.

5.2 Múltiplos *end effectors*

De uma maneira bastante intuitiva, este algoritmo pode ser alargado a esqueletos com vários *end effectors*. Para tal é necessário aplicar a primeira parte do algoritmo partindo de cada *end effector* até ser atingida uma sub-base. Quando todos os *end effectors* estiverem processados cada sub-base vai ter várias posições (cada uma delas resultante de cada filho). É necessário pegar em todas essas posições e calcular o centroide das mesmas e tomar a posição da sub-base como a do centroide (como se pode observar na figura 5.4). Caso existam mais sub-bases o mesmo processo é aplicado até se chegar ao ramo principal da árvore. Por fim, aplica-se o algoritmo normal ao ramo principal da árvore.

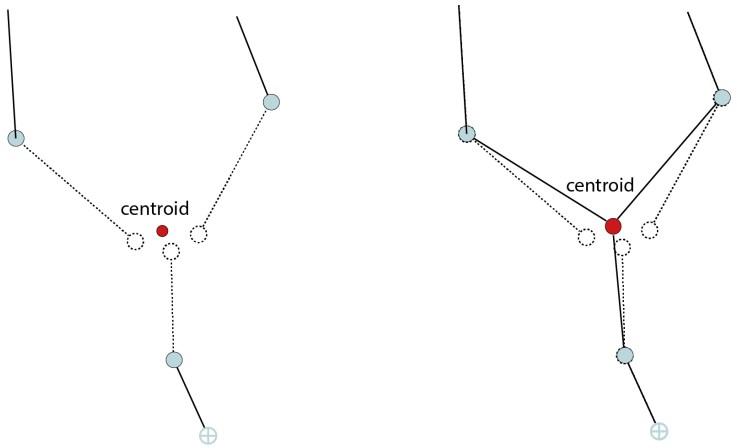

Figura 5.4: Exemplo da parte *inward* do algoritmo para um esqueleto com dois *end effectors*.

Em alguns casos os tamanhos dos ossos não irão estar corretos nas bifurcações do esqueleto, esses erros irão ser corrigidos pela segunda parte do algoritmo.

Na segunda parte do algoritmo este é começado normalmente a partir da *root* do esqueleto. Quando é atingida uma bifurcação é aplicado o algoritmo normal a cada um dos filhos até se chegar aos *end effectors*. Este passo vai corrigir qualquer falha de tamanho criada pelo primeiro passo (como podemos observar na figura 5.5).

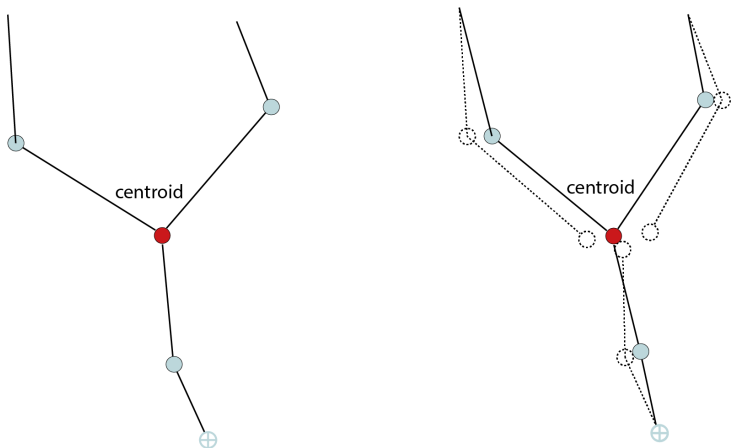

Figura 5.5: Exemplo da parte *outward* do algoritmo para um esqueleto com dois *end effectors*.

Muito resumidamente, para este pequeno exemplo é feito o a parte *inward* para o ramo da esquerda seguido pelo ramo da direita, depois é calculado o centroide com base nas duas posições dadas por cada braço e para terminar a primeira parte é feito o algoritmo normal (*inward*) para o ramo principal.

No que toca à segunda parte é feito o algoritmo normal ao ramo principal e quando se encontra a bifurcação é feito o algoritmo para o ramo direito e de seguida para o ramo esquerdo (a ordem de qual se faz primeiro não é relevante).

5.2.1 Dificuldades

Uma das grandes dificuldades em implementar o algoritmo foi perceber como realmente se fazia para modelos com múltiplos *end effectors*. Primeiramente tentou-se fazer o algoritmo completo para cada sub ramo, ou seja, começando por cada *end effector* era aplicado o algoritmo, *in* e *out*, a cada ramo dos *end effectors*. De seguida era aplicado, da mesma maneira, *in* e *out*, o algoritmo aos ramos da camada inferior até se chegar à *root* da árvore. Esta maneira de resolver não dava resultados corretos.

Depois de várias leituras ao artigo [1] foi possível perceber a maneira correta de implementar o algoritmo.

5.3 Restrições

De forma a tornar o algoritmo mais realista e permitir uma maior variedade de modelos implementou-se restrições de ângulos ao mesmo. Com isto é possível definir um modelo onde cada osso não possa dobrar mais do que 90 graus. Esse ângulo pode ser relativo à direção do osso (figura 5.6) ou relativo a um vetor específico (parecido com a figura 5.6 mas o vetor aponta em outra direção).

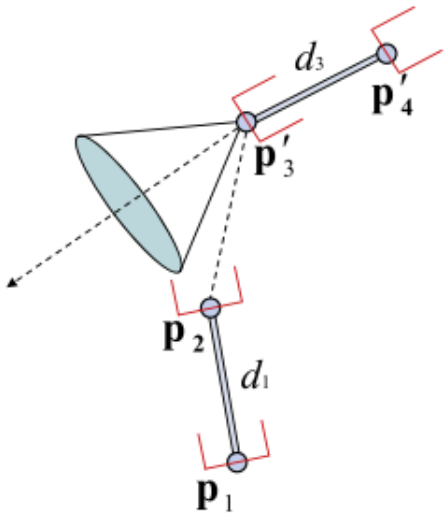

Figura 5.6: Exemplo de uma restrição relativo à direcção do osso.

No caso se ser dado o vetor pelo utilizador é possível também escolher se a posição do vetor dado é relativa ao osso, ou seja, se o vetor tiver a 90 graus do osso mesmo que o osso se mova esse vetor será actualizado para permanecer a 90 graus, ou pode escolher uma posição fixa, ou seja, mesmo que o osso se mova o vetor vai manter sempre o seu valor.

De forma a respeitar as restrições é verificado o grau que cada novo ponto forma com o vetor dado e caso o ângulo seja maior que o permitido é encontrado o ponto mais próximo dentro do cone permitido.

Isto é feito da seguinte maneira:

- 1 - Vetor v é o vector de p2 até p'3 (p2-p'3)
- 2 - Vetor v2 é o vector dado pelo utilizador ou é o vetor do osso (p'3-p'4)
- 3 - x=dot(v,v2)
- 4 - angulo=arccos(x)
- 5 - Caso o ângulo seja maior que o permitido é rodado o vetor v2 na direcção do vetor v alpha graus (este alpha vai ser o maior grau permitido pelo osso).
- 6 - Sendo v3 o vetor rotado, então o novo ponto p'2= p'3+v3*tamanho do osso

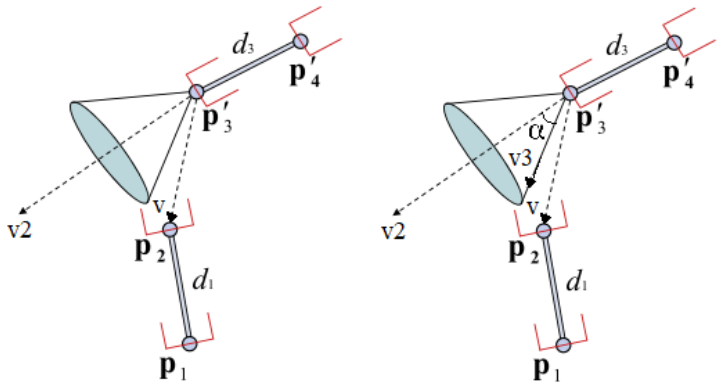

Figura 5.7: Exemplo da aplicação de uma restrição.

Com este tipo de restrições é possível, por exemplo, ter o modelo de um braço dando o vetor utilizado para medir os ângulos como um vetor perpendicular ao braço fazendo assim que o braço dobre apenas numa das direções.

5.4 Colisões

De modo a tornar o algoritmo mais realista implementou-se também colisões simples com uma esfera. Para tal é calculada a distancia do novo ponto até ao centro da esfera, caso essa distancia seja menor que o raio o ponto será projetado para a superfície da esfera. De seguida é criado um vetor do último ponto até ao ponto projetado e o novo ponto corrigido será ponto antigo + vetor * tamanho do osso (como se pode observar na figura 5.8).

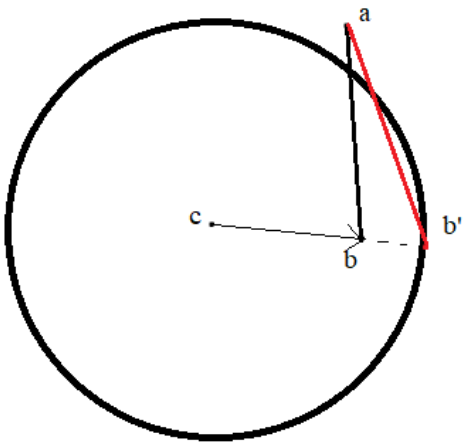

Figura 5.8: Exemplo de colisão.

Esta solução não é perfeita pois existe a possibilidade do novo ponto gerado ficar dentro da esfera, mas considerou-se que é uma aproximação aceitável.

Resultados

Depois de implementado o algoritmo resta testar o mesmo e discutir os resultados.

Para tal criou-se um simples esqueleto humano como se pode observar na figura 6.1. Nesse modelo podem se observar os vetores usados para limitar os ângulos das restrições.

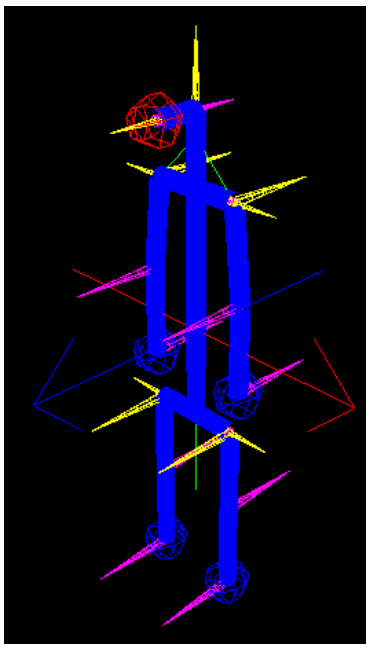

Figura 6.1: Modelo simples do esqueleto.

Utilizando este modelo e os *targets* já definidos é possível criar uma animação como pode ser visto nos vídeos [2], [3] e [4].

Limitações

Dado como concluída a implementação do algoritmo resta observar com olhar critico todo o trabalho realizado.

Devido à maneira como o algoritmo foi implementada torna-se muito difícil simular modelos mais complexos. Devido também ao sistema de vetores usado para controlar os ângulos de movimento, por vezes, é bastante complicado visualizar quais os vetores corretos a escolher tanto para a etapa *In* como a *Out*.

Para além da complexidade para criar os modelos, o algoritmo nem sempre tem o comportamento esperado. Em alguns casos bastante estranhos o esqueleto não tem uma posição fixa e a mesma é alterada a cada frame, fazendo com que o esqueleto trema bastante. Algo parecido acontece as vezes com a função que atualiza os vetores das restrições. A mesma altera o vetor todos os frames mesmo que o esqueleto esteja parado. Assim, o esqueleto vai se movimentando visto que o ângulo permitido vai se movendo com o vetor. Isto acontece devido a algum erro não identificado na função de atualização dos vetores. Pelo que foi possível perceber a função tem um bom comportamento para vetores alinhados com os eixos, o problema parece surgir quando utilizamos vetores inclinados não sabemos porquê.

Conclusão e Trabalho Futuro

Analisando o resultado final verificamos que o principal objetivo do projeto foi alcançado. Aplicando *Inverse Kinematics* foi possível fazer a animação de um modelo de um esqueleto, assim como a simulação de objetos.

Porém, existiram alguns pontos que poderão ser melhorados numa próxima fase. Como a correção das limitações apresentadas anteriormente. Utilizar formas diferentes para os ângulos, para além de cones, de forma a conseguir outro tipo de movimentos no esqueleto. E adicionar pontos de atração.

Por fim, é retirada uma apreciação bastante positiva em relação ao projeto desenvolvido. Pois permitiu desenvolver um programa que pode ser bastante útil na área de simulação de movimentos ou até na área de robótica.

Bibliografia

[1] Andreas Aristidou, Joan Lasenby
FABRIK: A fast, iterative solver for the Inverse Kinematics problem.
Graphical Models 73 (2011) 243–260

[2] Skeleton with head movement
`https://www.youtube.com/watch?v=r9CTLcO1WZA`

[3] Skeleton with constraints
`https://www.youtube.com/watch?v=wxxkO8V1aS4`

[4] Skeleton with collisions and constraints
`https://www.youtube.com/watch?v=yw6u2oegDfE`