Criação de um Visualizador BVH: Estruturas Articuladas e Movimentação

Paulo Eduardo Carvalho Mansano[*], Thomaz Abrantes de Oliveira Martinelli Silva[†]
Escola Politécnica — PUCRS

24 de março de 2025

Resumo

Este trabalho apresenta o desenvolvimento de um visualizador BVH, utilizando OpenGL para processar dados hierárquicos e sequenciais de arquivos no formato BVH. A solução implementa funções para leitura da hierarquia esquelética, extração de movimentos e aplicação dinâmica de transformações, armazenando os dados em estruturas otimizadas. A modularidade do código garante flexibilidade para manipular movimentos em tempo real, contribuindo para aplicações de animação e captura de movimento. Desafios superados incluem a construção da hierarquia do esqueleto e a alocação dinâmica eficiente. Os resultados destacam a importância de ferramentas robustas para a visualização de movimentos em contextos interativos e gráficos.

Palavras-chave

Visualizador BVH, OpenGL, Arquivos BVH, Hierarquias Articuladas, Visualização de Movimentos, Programação de Baixo Nível.

1 Introdução

No escopo da computação gráfica e da animação digital tornou-se uma prática indispensável o uso de captura de movimento para a criação de animações realistas e interativas. Dessa forma, técnicas como a hierarquização permitem representar movimentos articulados como os dos seres humanos precisamente. Tal prática é fundamental para a possibilidade de se desenvolver boas aplicações de movimentos e animações em videogames, filmes, realidade virtual e simulações médicas, por exemplo. Sendo assim, o formato BVH (BioVision Hierarchy) destaca-se como uma solução muito utilizada, por sua capacidade de armazenar tanto a estrutura hierárquica de um esqueleto quanto os dados de movimento associados a cada segmento.

Partindo desse cenário, no contexto da disciplina de Programação de Baixo Nível, presente no curso de Ciência da Computação, da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), foi proposta a implementação de um algoritmo voltado para a leitura, interpretação e manipulação de arquivos no formato BVH. O programa implementado realiza a leitura dos blocos de hierarquia e movimento, construindo uma estrutura de dados que representa a hierarquia e os valores de transformação de cada frame. A hierarquia esquelética contém informações como offsets, canais de transformação (posição e rotação) e relações de parentesco entre os segmentos do esqueleto. Já

[*]p.mansano@edu.pucrs.br
[†]thomaz.abrantes@edu.pucrs.br

os dados de movimento são armazenados em uma matriz dinâmica, permitindo a aplicação de movimentos específicos aos nós do esqueleto.

Este relatório detalha as etapas de implementação, as decisões técnicas envolvidas, os desafios encontrados e os resultados alcançados, além das conclusões que foram obtidas após o término do trabalho. Assim, o trabalho reforça a importância de ferramentas robustas e modulares para o processamento de dados no campo da animação e captura de movimento, contribuindo para avanços em aplicações gráficas e interativas.

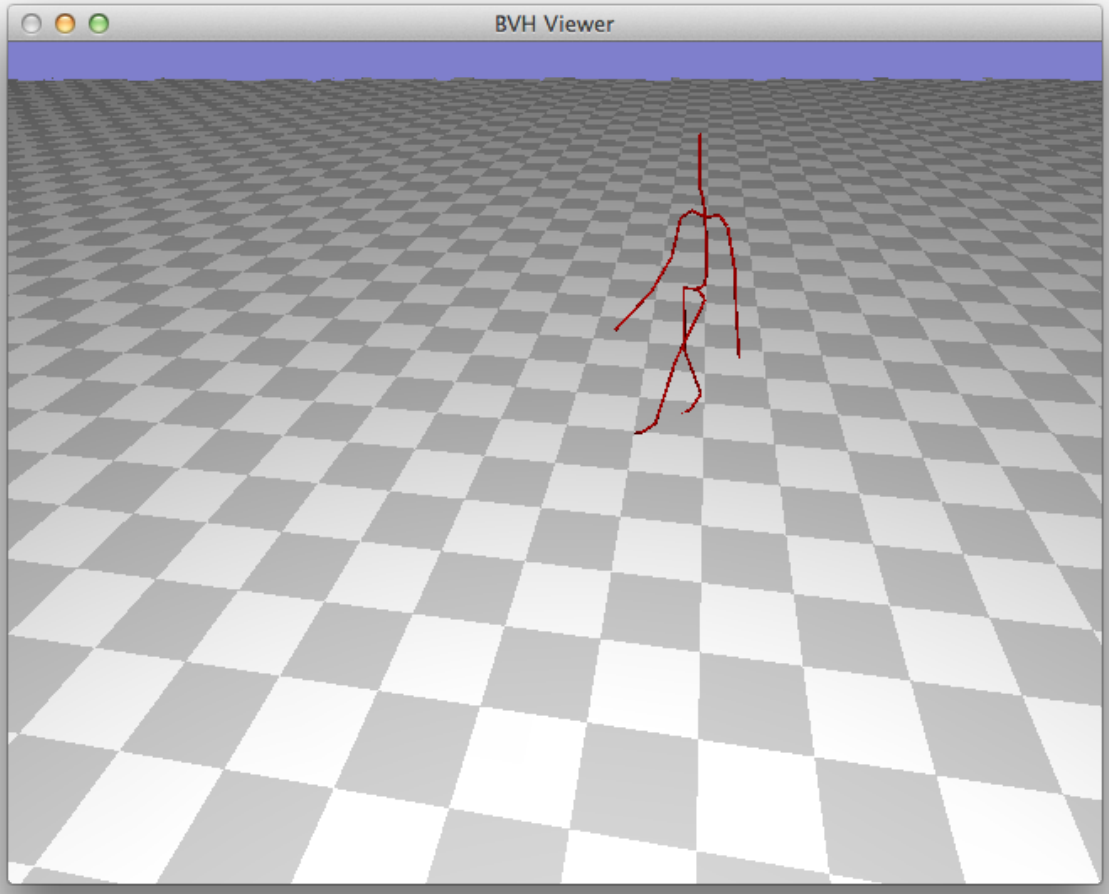

Figura 1: Imagem ilustrativa do esqueleto realizando um movimento.

2 Desenvolvimento do Algoritmo

2.1 Contextualização

O desenvolvimento deste algoritmo tem como objetivo a manipulação eficiente de arquivos BVH, garantindo a interpretação da hierarquia esquelética e dos dados de movimento. Para isso, foram implementadas funções específicas para lidar com as particularidades do formato, como a leitura da hierarquia esquelética, a extração dos dados de movimento e a liberação de memória alocada dinamicamente. Esta seção detalha a estrutura e o funcionamento dessas funções, incluindo ajustes realizados para aprimorar sua funcionalidade.

2.2 Estrutura e Funcionamento

A implementação do código foi dividida em funções, cada uma delas responsável por cuidar de uma parte específica do processo de manipulação dos arquivos BVH. A seguir serão explicadas cada uma das funções.

2.2.1 Função parseHierarchy

A função parseHierarchy é responsável pela leitura e pela construção da hierarquia esquelética do arquivo BVH. Ela interpreta cada segmento do esqueleto, armazenando informações como "offsets", canais de transformação e relações de parentesco. Abaixo é possível visualizar o trecho de código referente à função parseHierarchy.

```
Node* parseHierarchy(FILE* file) {
    char line[MAX_LINE_LENGTH];
    Node* root = NULL;
    Node* currentNode = NULL;

    while (fgets(line, sizeof(line), file)) {
        trimString(line); // Remove espacos em branco ao redor
        if (strncmp(line, "ROOT", 4) == 0 || strncmp(line, "JOINT", 5) == 0) {
            Node* newNode = createNode(line);
            if (currentNode != NULL) {
                addChildNode(currentNode, newNode);
            } else {
                root = newNode;
            }
            currentNode = newNode;
        } else if (strcmp(line, "}") == 0) {
            currentNode = currentNode->parent; // Volta para o no pai
        } else if (strncmp(line, "OFFSET", 6) == 0) {
            setOffsets(currentNode, line);
        } else if (strncmp(line, "CHANNELS", 8) == 0) {
            setChannels(currentNode, line);
        }
    }return root;
}
```

Essa função garante que a hierarquia seja construída de maneira estruturada, facilitando o acesso e a manipulação de nós posteriormente.

2.2.2 Função parseMotion

A função parseMotion extrai os dados de movimento do arquivo BVH e os organiza em uma matriz dinâmica, onde cada linha representa um frame e cada coluna corresponde a um canal de movimento. Abaixo encontra-se o código referente ao método parseMotion.

```
float** parseMotion(FILE* file, int numFrames, int numChannels) {
    float** motionData = allocateMatrix(numFrames, numChannels);
    for (int i = 0; i < numFrames; i++) {
        for (int j = 0; j < numChannels; j++) {
            fscanf(file, "%f", &motionData[i][j]);
        }
    }return motionData;
}
```

Essa abordagem otimiza o armazenamento dos dados de movimento e permite uma integração fácil com as outras partes do sistema.

2.2.3 Função freeMotionData

A função freeMotionData libera a memória alocada dinamicamente para a matriz de dados de movimento, prevenindo vazamentos de memória (quando a memória alocada dinamicamente por um programa não é liberada corretamente após o seu uso). A seguir, encontra-se o código da função freeMotionData.

```
void freeMotionData(float** motionData, int numFrames) {
    for (int i = 0; i < numFrames; i++) {
        free(motionData[i]);
    }
    free(motionData);
}
```

Essa prática garante que o programa gerencie recursos de memória.

2.2.4 Função trimString

A função trimString remove espaços em branco desnecessários no início e no final das linhas, facilitando o processamento das informações do arquivo BVH, evitando erros de interpretação causados por caracteres indesejados. Abaixo, pode ser visto o código referente ao código do método trimString.

```
void trimString(char* str) {
    char* end;
    while (isspace((unsigned char)*str)) str++; // Remove espacos iniciais
    end = str + strlen(str) - 1;
    while (end > str && isspace((unsigned char)*end)) end--; // Remove espacos
       finais
    *(end + 1) = '\0';
}
```

2.2.5 Ajustes na Função apply e na Declaração de data

A função apply foi ajustada para integrar os dados de movimento diretamente com os nós da hierarquia esquelética, permitindo a aplicação dinâmica dos movimentos em tempo de execução. Ademais, a declaração de data foi modificada para armazenar os valores de movimento de forma mais eficiente. Essa modificação melhora a flexibilidade da função, possibilitando a reutilização dos dados em múltiplos frames sem reprocessamentos desnecessários. A seguir está o código do método apply.

```
void apply(Node* node, float** data, int frame, int numChannels) {
    if (node == NULL) return;
    int offset = node->channelOffset;
    for (int i = 0; i < node->numChannels; i++) {
        node->channelValues[i] = data[frame][offset + i];
    }
    for (int i = 0; i < node->numChildren; i++) {
        apply(node->children[i], data, frame, numChannels);
    }
}
```

2.2.6 Funcionamento Geral do Código

Como já foi falado anteriormente, o código desenvolvido tem como objetivo processar dados de movimento, ajustando e manipulando as informações para simular o movimento de um esqueleto. O processo começa com a leitura dos dados, que são então analisados e convertidos para uma estrutura mais adequada ao processamento, utilizando a função parseMotion. Essa função interpreta os dados - como posição - e os organiza para que possam ser manipulados de forma eficiente. Em seguida, a função apply é responsável por aplicar transformações nos dados de movimento, ajustando atributos como posição e velocidade com base em cálculos definidos.

Seguidamente, a função freeMotionData é chamada para liberar a memória alocada, evitando vazamentos de memória, um aspecto importante para garantir a eficiência do código, especialmente quando se está lidando com grandes volumes de dados. Ao longo do desenvolvimento, o código foi modularizado, ou seja, dividido em funções pequenas e coesas, permitindo fácil manutenção e compreensão, garantindo que o código esteja flexível e possa ser adaptado a novas funcionalidades ou requisitos no futuro.

Ademais, foram feitos ajustes na função apply e nas declarações de variáveis para otimizar o desempenho e garantir a precisão dos cálculos. Assim, o código se mostrou eficiente em termos de tempo de execução e uso de memória, assegurando que os dados de movimento fossem processados corretamente.

3 Exemplos Práticos de Funcionamento

Primeiramente, de modo a compilar o código, clica-se as teclas Ctrl+Shift+B no Visual Studio Code. Seguidamente, após ter compilado o programa, utiliza-se algum comando relativo a um dos arquivos BVH, tal como: `"./bvhviewer bvh\Male2_G15_RoundhouseBodyRight.bvh"`. Assim, após inserir o comando citado, por exemplo, pressiona-se a tecla "Right Arrow (→)"para realizar o movimento do esqueleto.

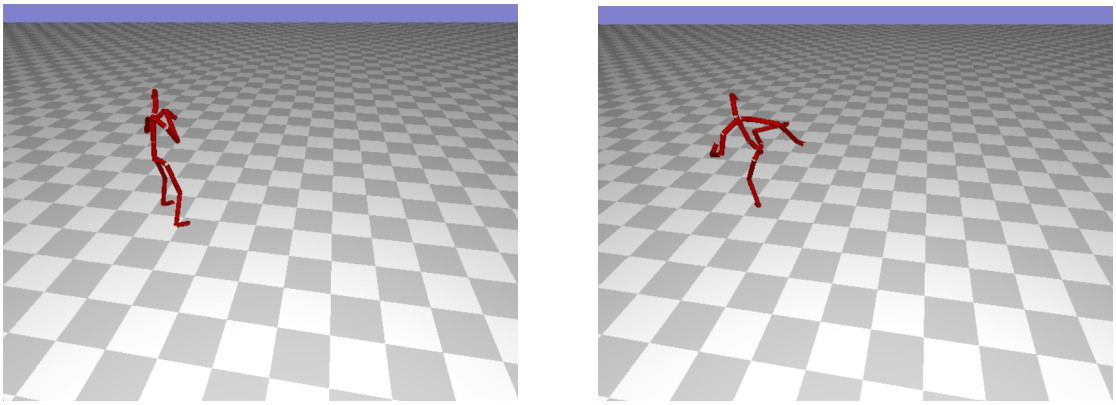

Figura 2: Representação do esqueleto realizando o movimento.

4 Conclusões

O desenvolvimento deste trabalho proporcionou uma ótima experiência na manipulação de dados hierárquicos e sequenciais em arquivos ".bvh", além da implementação de estruturas algorítmicas para

processamento e visualização de movimentos corporais. A importância desse aprendizado está na aplicação direta de conceitos de estrutura de dados, alocação dinâmica e na compreensão de formatos específicos, habilidades valiosas no contexto de computação gráfica, animação e processamento de dados complexos.

Apesar do sucesso no desenvolvimento do programa, foram enfrentadas algumas dificuldades as quais devem ser destacadas. Inicialmente, ocorreram problemas para compilar e executar o código, entretanto, através da demonstração do professor, foi possível superar esse obstáculo. Ademais, outro desafio foi compreender e implementar a hierarquia do esqueleto. Foi decidido simplificar a abordagem ao substituir um código mais otimizado e complexo por uma solução baseada em condicionais (ifs e elses). Embora essa decisão tenha limitado a flexibilidade do código diante das mudanças no formato do arquivo, ela foi fundamental para viabilizar o progresso no trabalho.

Por fim, a implementação dinâmica de uma matriz para armazenar os pontos do esqueleto revelou-se mais desafiadora do que o previsto. Apesar do sucesso ao resolver o problema, o código ainda apresenta uma limitação: o esqueleto inicia na posição padrão do arquivo, só assumindo a posição correta a partir do primeiro frame processado. Mesmo com essas dificuldades, foi possível cumprir os requisitos do trabalho, reforçando a importância de adaptação e de soluções práticas frente a problemas complexos.