Universidade do Minho

Escola de Engenharia

Computação Gráfica

Licenciatura em Engenharia Informática

Sistema Solar 3D

Grupo 35

10 de Março de 2023

	
A86817 Rafael Arêas (LEI)	A87893 Pedro Pinto (LEI)
	
A91639 Afonso Cruz (LCC)	a87951 Hélder Lopes (LCC)

Índice

Lista de Figuras

Introdução

Este relatório apresenta o projeto de um sistema solar em 3D implementado em C++ utilizando OpenGL e a biblioteca GLUT. O objetivo do projeto é criar uma simulação interativa do sistema solar, que permita ao usuário explorar o espaço e interagir com os corpos celestes.

O sistema solar é composto por oito planetas, incluindo Mercúrio, Vênus, Terra, Marte, Júpiter, Saturno, Urano e Netuno, bem como o Sol e outros corpos celestes, como luas e asteroides. A simulação inclui o movimento orbital dos planetas em torno do Sol, bem como a rotação dos planetas em torno de seus próprios eixos.

O projeto utiliza conceitos de computação gráfica, como modelagem 3D, texturização, iluminação e sombreamento para criar uma representação realista do sistema solar. Além disso, a interatividade é incorporada através do uso do controlo do teclado para permitir que o usuário explore o espaço e interaja com os planetas.

Este relatório descreve os principais componentes do projeto, incluindo a modelagem dos corpos celestes, a implementação do movimento orbital e de rotação, a iluminação e sombreamento, a interatividade do usuário e a arquitetura geral do sistema. Além disso, o relatório discute as principais dificuldades encontradas durante o desenvolvimento do projeto e as soluções adotadas para superá-las.

Ao final do relatório, são apresentadas as conclusões do projeto, incluindo os resultados obtidos e possíveis melhorias para trabalhos futuros.

Arquitetura do Sistema

No âmbito deste projeto de *Computação Gráfica* consiste em duas partes principais: o *Generator* e o *Engine*. Essa estrutura permite uma abordagem modular para o desenvolvimento de aplicações gráficas. O Generator facilita a criação de modelos personalizados, enquanto o Engine usa as informações de configuração para renderizar os modelos de maneira flexível e dinâmica. Ao longo deste relatório, exploraremos em detalhes as funcionalidades, processos e resultados alcançados por meio do *Engine* e do *Generator*.

Engine

A *Engine* é a aplicação do motor principal. Ela recebe um arquivo de configuração escrito em *XML*. Esse arquivo de configuração contém informações sobre como renderizar o modelo, como as coordenadas de textura, as normais, os materiais e outras propriedades relevantes. O Engine lê o arquivo de configuração e utiliza essas informações para renderizar o modelo de forma apropriada.

Camera

Definiu-se uma classe *Camera* que esta classe é usada para controlar a posição e a orientação de uma câmera virtual. Esta possui atributos para a posição, ponto de visualização, vetor de orientação vertical, campo de visão e outras propriedades relacionadas à câmera.

O construtor inicializa os atributos com valores padrão. Há métodos para obter e definir os valores dos atributos. Além disso, existem métodos para converter as coordenadas esféricas em cartesianas e para atualizar o valor do raio com base nas coordenadas cartesianas.

Light

Foi implementada uma classe *Light* que representa a fonte de luz. Esta classe possui atributos para posição, direção, ângulo de corte, tipo e métodos para configurar e obter os valores desses atributos.

O construtor padrão não faz nada, deixando os atributos da luz com valores padrão ou indefinidos.

Os métodos *setPos*, *setDir*, *setCutoff* e *setType* são usados para definir os valores da posição, direção, ângulo de corte e tipo da luz, respectivamente. Esses métodos recebem os valores como parâmetros e atribuem esses valores aos atributos correspondentes da classe.

Os métodos *getPos*, *getDir* e *getCutoff* retornam os valores dos atributos de posição, direção e ângulo de corte da luz, respectivamente.

Os atributos de posição e direção são armazenados em vetores de três elementos (x, y, z),

enquanto o ângulo de corte é armazenado como um valor de ponto flutuante. O tipo da luz é armazenado como um valor inteiro.

Model

Temos ainda uma classe *Model* que representa um modelo gráfico. Essa classe possui atributos para pontos, transformações, normais, coordenadas de textura, ID da textura e propriedades de material, além de métodos para configurar e obter esses atributos e para executar transformações e desenhar o modelo.

O construtor sobrecarregado recebe um vetor de transformações e atribui esse vetor ao atributo *Transforms* do modelo.

Os métodos *getPoints*, *getTransformations*, *getVBO*, *getNormals*, *getTextureCoords* e *getTexture* são usados para obter os valores dos atributos correspondentes do modelo.

O método *executeTransformations* percorre as transformações do modelo e aplica cada uma delas usando o valor de *frames* por segundo (fps). Além disso, chama a função *glVertexPointer* com os parâmetros apropriados.

O método *draw* desenha o modelo na cena. Ele faz uso da função *glBindTexture* para vincular a textura especificada pelo ID à textura atualmente ativa. Se o ID da textura for zero, são definidas as propriedades de material (como cor difusa, ambiente, especular e emissiva) e brilho usando as funções *glMaterialfv* e *glMaterialf*. Em seguida, o método aplica as transformações do modelo usando o valor de fps e chama *glDrawArrays* para desenhar os triângulos do modelo.

Os métodos *setTexture*, *setDiffuse*, *setAmbient*, *setSpecular*, *setEmissive* e *setShininess* são utilizados para configurar os valores dos atributos correspondentes do modelo.

O método *freeMem* limpa os vetores de pontos, normais e coordenadas de textura.

O método *setPointsSize* define o tamanho dos pontos.

O método *ReadFile* lê um arquivo contendo informações sobre o modelo. Ele lê as coordenadas dos pontos, normais e coordenadas de textura do arquivo e armazena-os nos vetores apropriados. O arquivo deve seguir um formato específico, com linhas separadas para triângulos, normais e texturas.

Reader

Foi criada uma classe Reader para fazer o parsing do ficheiro XML e guardar as informações em variáveis.

A função *groupParser* é responsável por percorrer os elementos *transform*, *translate* e *scale* do *XML* e criar objetos *Transformation* com as informações de rotação, translação e escala. Em seguida, ela adiciona essas transformações a um vetor *TransVec*.

A função *readxml* carrega um arquivo *XML* usando a biblioteca *tinyxml2* e analisa as informações contidas nele. Ela lê as configurações de janela, câmera e grupo do arquivo *XML*

e chama a função *groupParser* para processar o elemento *group*. Além disso, ela cria buffers OpenGL usando a função *glGenBuffers* para armazenar os dados dos modelos.

A função *draw* percorre os modelos e realiza as chamadas OpenGL necessárias para renderizá-los na tela.

As demais funções são acessores (getters e setters) para obter e configurar informações como tamanho da janela, câmera, modelos, etc.

A função *loadTexture* carrega uma imagem de textura usando a biblioteca *DevIL* (Image Library) e cria um identificador de textura OpenGL para a mesma. A imagem é carregada usando *ilLoadImage* e convertida para o formato *RGBA*. Em seguida, é gerado um identificador de textura com *glGenTextures* e a imagem é associada a ele usando *glTexImage2D*. Por fim, são definidos os parâmetros de filtragem e repetição da textura.

Em resumo, esta classe lê um arquivo *XML* que contém informações sobre modelos 3D, as suas transformações, texturas e cores, e utiliza OpenGL para renderizar esses modelos com as configurações especificadas no arquivo *XML*.

Transform

Foi criada a classe *Transform* para tratar das transformações geométricas que vão ocorrer neste projeto.

A classe possui vários construtores para inicializar os atributos da transformação, como tipo (t), ângulo (angle), coordenadas de translação (x, y, z) e coordenadas de escala (x, y, z). Ela também possui métodos *getter* e *setter* para acessar e modificar esses atributos.

O método *apply*(float fps) é o principal método da classe e é responsável por aplicar a transformação. Dependendo do tipo da transformação (representado pelo atributo type), diferentes operações OpenGL são executadas. Se o tipo for 0, é feito um deslocamento (glTranslatef) ou é desenhada uma curva utilizando a função *getGlobalCatmullRomPoint*. Se o tipo for 1, é feita uma rotação *glRotatef*. Caso contrário, é feita uma escala *glScalef*.

Além disso, a classe possui outros métodos auxiliares, como *getGlobalCatmullRomPoint*, *getCatmullRomPoint*, *multMatrixVector*, *cross*, *normalize* e *buildRotMatrix*, que são utilizados para realizar cálculos necessários durante a aplicação das transformações.

No geral, a classe *Transformation* permite realizar diferentes tipos de transformações em um sistema gráfico 3D, como translação, rotação e escala, utilizando as funções OpenGL apropriadas.

Generator

O *Generator* é responsável por criar os arquivos de modelo. Ele é onde recebemos parâmetros, como o tipo de primitiva gráfica e outros parâmetros necessários para a criação do modelo. Além disso, o *Generator* também recebe um arquivo de destino onde os vértices serão armazenados. Com base nos parâmetros fornecidos, o *Generator* gera os vértices correspondentes à primitiva gráfica desejada e os armazena no arquivo de destino.

A classe *fileGenerator* é responsável pela geração de arquivos de modelo 3D com base em parâmetros fornecidos pelo usuário. Ele inclui diferentes formas geométricas, como *plano*, *esfera*, *caixa*, *cone*, *patch de Bezier* e *toro*.

Ao executar o programa, o usuário deve fornecer o nome da forma desejada e os parâmetros necessários. O código verifica o tipo de forma e a quantidade correta de argumentos fornecidos.

Para cada forma, o programa gera os vértices correspondentes e armazena as informações em um arquivo. Esses arquivos são salvos em uma pasta específica chamada *GeneratedFiles*.

As formas suportadas e seus respectivos argumentos são:

1. *Plane*: Requer a quantidade de subdivisões em cada eixo.
2. *Esfera*: requer o raio, o número de fatias e o número de pilhas.
3. *Caixa*: requer as dimensões da caixa e o número de divisões por aresta.
4. *Cone*: requer o raio da base, a altura, o número de fatias e o número de pilhas.
5. *Patch de Bezier*: requer um arquivo de entrada contendo informações sobre os pontos de controle e um valor de tesselação.
6. *Toro*: requer o raio interno, o raio externo, o número de fatias e o número de camadas.

Após a geração dos arquivos de modelo, eles são salvos com o nome fornecido pelo usuário na pasta *GeneratedFiles*.

Para cada um dos formas suportadas e seus respectivos argumentos, foram desenvolvidas classes dedicadas a tratar desses dados de forma organizada e eficiente. Cada classe possui métodos e atributos específicos para lidar com as propriedades individuais de cada forma geométrica. Através da utilização dessas classes, foi possível implementar algoritmos especializados para a geração dos vértices correspondentes a cada forma.

Plano

Para a classe *Plane*, definida no arquivo "Plane.hpp", corresponde a implementação da geração de um plano no projeto.

A função *generate* da classe *Plane* recebe dois parâmetros: o comprimento do plano (length) e o número de divisões (divisions). Esses parâmetros determinam as dimensões e a subdivisão do plano.

Dentro da função, é definido um vetor normal (normal) que representa a orientação do plano. Em seguida, é calculado o tamanho de cada subdivisão (divisionSize) com base no comprimento e no número de divisões informados. O raio (r) do plano também é calculado.

Os laços de repetição são utilizados para percorrer cada subdivisão do plano. São criados pontos (p1, p2, p3, p4) que formam os triângulos que compõem o plano. A função *addPoint* adiciona os pontos ao plano, a função *addNormal* adiciona a normal correspondente e a função *addTextureP* adiciona as coordenadas de textura (texturas no plano) associadas a cada ponto.

Ao final da execução da função, o objeto *Plane* contém todos os vértices, normais e coordenadas de textura necessárias para definir o plano desejado.

Essa implementação permite gerar um plano retangular subdividido em uma malha de triângulos, possibilitando sua renderização e manipulação posteriormente na aplicação gráfica.

Esfera

Na implementação da classe *Sphere*, implementa a geração de uma esfera no projeto.

A função "generate"da classe "Sphere"recebe três parâmetros: o raio da esfera (radius), o número de fatias (slices) e o número de pilhas (stacks). Esses parâmetros determinam as dimensões e a subdivisão da esfera.

Dentro da função, são calculados os ângulos necessários para a criação dos pontos da esfera. O ângulo entre as pilhas é determinado por "angleStacks"e o ângulo entre as fatias é determinado por "angleSlices".

Em seguida, são utilizados laços de repetição para percorrer cada pilha e fatia da esfera. São criados pontos (v1, v2, v3, v4, v5, v6) que formam os triângulos que compõem a esfera. A função "addPoint"adiciona os pontos à esfera, a função "addNormal"adiciona a normal correspondente e a função "addTextureP"adiciona as coordenadas de textura associadas a cada ponto.

Além disso, a função "generateTextures"é utilizada para gerar as coordenadas de textura para cada ponto da esfera, com base nos ângulos alpha e beta. Essas coordenadas são adicionadas à esfera juntamente com os pontos e normais correspondentes.

Ao final da execução da função, o objeto "Sphere"contém todos os vértices, normais e coordenadas de textura necessárias para definir a esfera desejada.

Essa implementação permite gerar uma esfera 3D subdividida em uma malha de triângulos, possibilitando sua renderização e manipulação posteriormente na aplicação gráfica.

Caixa

A classe *Box* é uma implementação da geração de uma caixa tridimensional. A caixa é gerada a partir de um tamanho especificado e dividida em um número de divisões. O objetivo é criar as faces da caixa, adicionando pontos, normais e coordenadas de textura para cada face.

O código utiliza um laço duplo para percorrer todas as divisões e gerar as faces da caixa. A cada iteração do laço, são calculados os pontos que definem os vértices das faces da caixa. Em seguida, esses pontos são adicionados à estrutura de dados da caixa juntamente com as normais correspondentes e as coordenadas de textura.

As faces são geradas separadamente para a face inferior, face superior, face esquerda, face direita, face frontal e face traseira da caixa. Os pontos são adicionados à estrutura de dados da caixa usando o método *addPoint*, as normais são adicionadas usando o método *addNormal* e as coordenadas de textura são geradas e adicionadas usando o método *addTextureP*.

As coordenadas de textura são geradas pela função *generateTextures*, que recebe três

parâmetros: *a*, *b* e *max*. Essa função calcula as coordenadas de textura para um ponto específico na caixa com base em sua posição relativa e no tamanho máximo da caixa.

Cone

A classe *Cone* é responsável em gerar um cone tridimensional. Ele utiliza pontos, texturas e normais para construir o cone.

A função *generate* recebe como parâmetros o tamanho do cone, sua altura, o número de fatias (slices) e o número de camadas (stacks). Dentro da função, são calculadas as distâncias entre as camadas do cone, os tamanhos das camadas e os ângulos entre as fatias. Em seguida, há um loop que percorre as camadas do cone, e dentro desse loop, há outro loop que percorre as fatias. Para cada combinação de camada e fatia, são calculadas as coordenadas e os ângulos dos pontos do cone. É chamada a função *generateTextures* para gerar as coordenadas de textura para cada ponto do cone. Também são calculadas as normais dos pontos do cone. Em seguida, os pontos, as coordenadas de textura e as normais são adicionados ao objeto do cone. No final do loop, é verificado se estamos na última camada do cone. Se sim, é adicionado um ponto especial na parte inferior do cone.

A função *generateTextures* é responsável por calcular as coordenadas de textura para cada ponto. Ela utiliza as coordenadas x, y, z e a altura do ponto para realizar o cálculo.

Patch de Bezier

Na classe *Bezier* implementa um patch de Bezier tridimensional. Ele lê um arquivo de entrada que contém informações sobre os pontos do patch e os índices dos pontos que formam cada patch.

A função *parseBezier* recebe o nome do arquivo de entrada e realiza a leitura do arquivo. Ela extrai o número de patches, os índices dos pontos que formam cada patch e o número de pontos. Os pontos são armazenados em um vetor. Em seguida, a função itera sobre os patches e cria objetos *Shape* para cada um deles. Para cada patch, os pontos correspondentes aos índices são adicionados ao objeto *Shape*.

A função *bezierPoints* recebe como parâmetro a quantidade de divisões desejada para a tesselação do patch. Ela itera sobre os patches e, para cada patch, realiza a tesselação criando os pontos resultantes da avaliação da função de *Bezier* para diferentes valores de parâmetros u e v. Os pontos são adicionados ao objeto do patch de *Bezier*.

A função *bezierPatch* recebe os valores *u* e *v* e um objeto *Shape* representando um patch. Ela realiza a avaliação da função de *Bezier* para determinar um ponto no patch com base nos valores dos parâmetros *u* e *v*.

A função *bValue* recebe os valores *u* e quatro pontos de controle *p0, p1, p2, p3*. Ela calcula o valor da função de *Bezier* para os pontos de controle e o valor do parâmetro *u*.

No final, o patch de *Bezier* é composto pelos pontos resultantes da tesselação, que são adicionados ao objeto do patch de *Bezier* junto com as normais e as coordenadas de textura.

Toro

A classe *Torus* implementa a geração de um *toro* (também conhecido como anel) tridimensional. O *toro* é gerado com base nos parâmetros de raio interno *inRadius*, raio externo *outRadius*, número de fatias *slices* e número de camadas *layers*.

A função *generate* recebe os parâmetros do *toro* e itera sobre as camadas e fatias para criar os pontos, normais e coordenadas de textura do *toro*.

Para cada camada e fatia, são calculados os ângulos *alpha*, *nextAlpha*, *beta* e *nextBeta*, com base nos quais os pontos são calculados usando equações paramétricas. Os pontos são adicionados ao objeto do *toro* junto com as normais e as coordenadas de textura.

O cálculo dos pontos leva em consideração as fórmulas matemáticas do *toro*, onde as coordenadas x, y e z são calculadas com base nos ângulos *alpha*, *beta*, *nextAlpha* e *nextBeta*, além dos raios interno e externo.

O *toro* resultante é formado por uma malha de triângulos, onde cada triângulo é formado por três pontos. Esses pontos são adicionados ao objeto do *toro*, juntamente com as normais e as coordenadas de textura.

No final, o objeto do *toro* contém os pontos, as normais e as coordenadas de textura que representam o toro gerado.

Geometric Transforms

Nesta parte do projeto, o objetivo foi criar cenários hierárquicos usando transformações geométricas. Um cenário é definido como uma árvore, em que cada nó contém um conjunto de transformações geométricas (translação, rotação e escala) e, opcionalmente, um conjunto de modelos. Todos esses dados são extraídos de um arquivo *XML*.

Curves, Cubic Surfaces and VBOs

Nesta parte do projeto, foram realizadas diversas atividades relacionadas ao desenvolvimento do sistema. No módulo Generator, foi implementada a capacidade de criar um novo tipo de modelo com base em *Patches de Bezier*. O *Generator*, já acima citado, recebe como parâmetros um arquivo contendo os pontos de controle de *Bezier* e o nível de tesselação necessário. O resultado é um arquivo que contém uma lista de pontos para desenhar a superfície.

No módulo *Engine*, houve uma extensão nos elementos de transformação e rotação. Agora, é possível definir uma curva cúbica de *Catmull-Rom* com base em um conjunto de pontos fornecidos, juntamente com o tempo necessário para percorrer toda a curva. Essa transformação permite a realização de animações. Além disso, foram adicionados campos de "alinhamento"para especificar se o objeto deve se alinhar com a curva. Os modelos podem ter uma transformação dependente do tempo ou uma transformação estática, como nas fases anteriores. No caso da rotação, o ângulo pode ser substituído pelo tempo necessário para uma rotação completa de 360 graus em torno do eixo especificado.

Também foi necessário atualizar o modo de desenho dos modelos, substituindo o modo imediato usado nas fases anteriores pelo uso de *VBOs* (Vertex Buffer Objects).

No geral, foram realizadas diversas implementações e atualizações nos módulos Generator e Engine para aprimorar a capacidade de criar e animar modelos *3d* utilizando curvas de *Bezier*, *tesselação* e *VBOs*.

Normals and Texture Coordinates

Nesta etapa, mudamos o aplicativo generator para gerar coordenadas de textura e normais para cada vértice.

Na engine 3D, criamos funcionalidades de iluminação e texturização, para além de ler e aplicar as normais e coordenadas de textura dos arquivos de modelo. Para isso foi criado um novo objeto chamado *Ligth*.

Durante o processo de renderização, as coordenadas de textura são utilizadas para mapear texturas nos modelos, permitindo a aplicação de detalhes visuais. As normais são essenciais para calcular a iluminação correta nos objetos, garantindo uma aparência mais realista e tridimensional.

Testes Fase 4

Foram desenvolvidos testes a partir do arquivos *Xml* fornecidos:

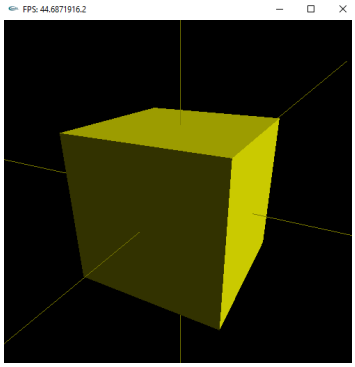

Figura 1: teste xml 1

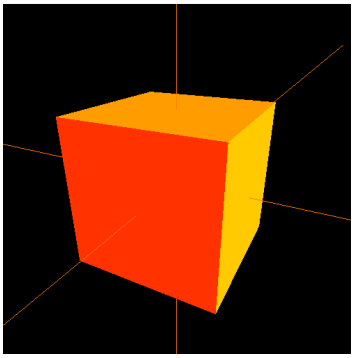

Figura 2: teste xml 2

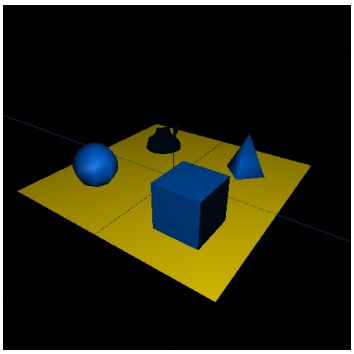

Figura 3: teste xml 3

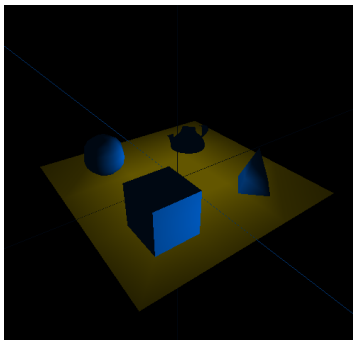

Figura 4: teste xml 4

Figura 5: teste xml 5

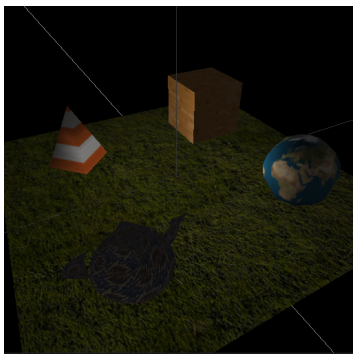

Figura 6: teste xml 6

Sistema Solar

Teste do sistema solar desenvolvido a partir do *XML* criado pelo grupo:

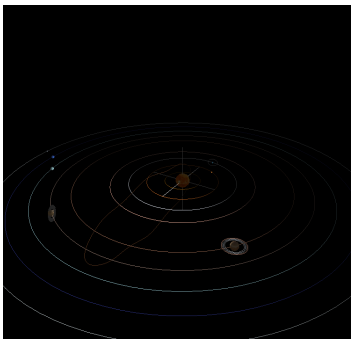

Figura 7: teste Sistema Solar

Conclusão

Em conclusão, o projeto de desenvolvimento do sistema solar utilizando OpenGL e GLUT foi uma experiência empolgante e enriquecedora no campo da computação gráfica. Durante o processo, pudemos aplicar conceitos fundamentais da disciplina, como transformações geométricas, iluminação e texturização, para criar uma representação visualmente impressionante do nosso sistema solar.

Ao longo do projeto, aprendemos a utilizar as ferramentas e bibliotecas fornecidas pelo OpenGL e GLUT para construir objetos 3D, aplicar materiais realistas, simular movimentos orbitais e rotacionais dos planetas, e até mesmo incorporar efeitos especiais, como a simulação de cometas.

Além disso, o projeto nos permitiu explorar conceitos mais avançados, como mapeamento de texturas em esferas para representar as superfícies planetárias e o uso de sombreamento para obter efeitos de iluminação mais complexos. Essas técnicas nos ajudaram a aprimorar a aparência visual do nosso sistema solar, tornando-o mais realista e imersivo.

Ao concluir este projeto, adquirimos habilidades práticas e conhecimentos teóricos que serão úteis em futuros projetos relacionados à computação gráfica. Além disso, obtivemos uma compreensão mais profunda dos desafios e complexidades envolvidos na criação de cenas 3D interativas.

Em suma, o projeto do sistema solar foi uma oportunidade emocionante para aplicar os conceitos e técnicas aprendidos na disciplina de computação gráfica. Ele nos permitiu explorar o potencial do OpenGL e GLUT na criação de ambientes virtuais imersivos e visualmente atraentes. Estamos orgulhosos do resultado alcançado e confiantes de que essas habilidades e conhecimentos adquiridos serão valiosos para nossa trajetória acadêmica e profissional na área de computação gráfica.

Bibliografia

OpenGL Programming Guide, Version 2, ARB, Addison Wesley

Interactive Computer Graphics, Edward Angel (2nd, 3rd or 4th edition), Addison Wesley

OpenGL Super Bible, Richard Wright (3rd edition), Sams publishing

YouTube: course on Computer Graphics - Kenneth Joy, University of California Davis

Geoemtric Transformations Songho