Computação Gráfica - Projeto 1

Lucas Toscano Pimentel Appolinário Cerqueira - 2110695

A ideia central está na criação de nós em uma hierarquia onde cada um deles representa um astro no sistema solar, permitindo uma organização clara e eficiente dos objetos na cena. Cada nó não apenas encapsula as propriedades de transformação do astro, como posição, rotação e escala, mas também carrega as texturas que definem sua aparência visual. As animações dos planetas são geradas aplicando transformações a esses nós, o que possibilita simular movimentos orbitais realistas ao redor do Sol. Essa abordagem hierárquica facilita o controle e a manipulação dos objetos, uma vez que as transformações aplicadas ao nó pai (o Sol) afetam todos os nós filhos (os planetas), promovendo uma interação dinâmica e coesa no ambiente tridimensional.

As alterações feitas no código base e as classes criadas estão listadas abaixo.

Classe MovePointer

A classe MovePointer é responsável por controlar a movimentação dos planetas no mini sistema solar, aplicando transformações de rotação e translação. Inicialmente, ela recebe um objeto de transformação e os parâmetros de velocidade de translação, raio de translação e velocidade de rotação durante sua construção. O método Update é chamado a cada quadro, onde o ângulo de translação e rotação é atualizado com base no tempo decorrido. A nova posição dos planetas é então calculada utilizando coordenadas polares, e as transformações são aplicadas ao objeto de transformação.

```
virtual void Update(float dt)
{
  // Update the trans_angle of rotation based on the trans_speed
  trans_angle += dt * trans_speed;
  rot_angle += dt * rotation_speed;

  // Ensure the trans_angle stays within [0, 360) degrees
  if (trans_angle > 360.0f)
    trans_angle -= 360.0f;

  if(rot_angle > 360.0f)
    rot_angle -= 360.0f;

  // Calculate new position using circular motion
  float x = tans_radius * cos(glm::radians(trans_angle));
  float y = tans_radius * sin(glm::radians(trans_angle));


  // Reset transformation and apply translation for circular motion
  m_trf->LoadIdentity();
  m_trf->Translate(x, y, 0);
  m_trf->Rotate(rot_angle, 0, 0, 1);
}
;
```

Função initialize

Na função initialize, foram feitas várias alterações em relação ao código base. Primeiramente, a câmera é configurada para definir os limites de visualização do sistema solar. Em seguida, um fundo é criado com uma textura que simula o espaço, posicionado para servir como pano de fundo dos astros, os quais são instanciados com suas respectivas transformações e texturas, e organizados em uma hierarquia de nós. A implementação de shaders é adicionado e, por fim, a cena é montada, combinando todos os nós e suas transformações, e as engines que controlam a movimentação dos planetas são integradas, assegurando que as animações ocorram de forma fluida e dinâmica durante a simulação do mini sistema solar.

```
// Earth
auto trfEarth = Transform::Make();
trfEarth->Translate(3,3,0);
auto texEarth = Texture::Make("face","./images/earth.jpg");
auto earth = Node::Make(trfEarth,{Color::Make(0.0f,0.5f,0.8f),texEarth},{Disk::Make(nslice, radius)});
```

```
// Add nodes to the scene
earth->AddNode(moon);
sun->AddNode(earth);
sun->AddNode(venus);
```

Função resize

Além de atualizar o viewport, a função recalcula a proporção de tela com as novas dimensões da tela. Esse resultado é utilizado para ajustar a câmera de forma a manter a visualização adequada, isso é feito criando uma nova câmera com os limites apropriados, garantindo que a cena seja exibida corretamente em diferentes formatos de janela.

```
static void resize(GLFWwindow* win, int width, int height)
{
    // Update the viewport to match the new window size
    glViewport(0, 0, width, height);

    // Store the new screen dimensions
    screenWidth = width;
    screenHeight = height;

    // Adjust the camera to match the new aspect ratio
    float aspect = (float)width / (float)height;
    camera = Camera2D::Make(-aspect * 10.0f, aspect * 10.0f, -10.0f, 10.0f);
}
```

Classe Disk

Na classe Disk, o suporte a textura foi implementado usando o método generateDiskTexCoord para calcular as coordenadas de textura e armazenando elas em um buffer próprio.

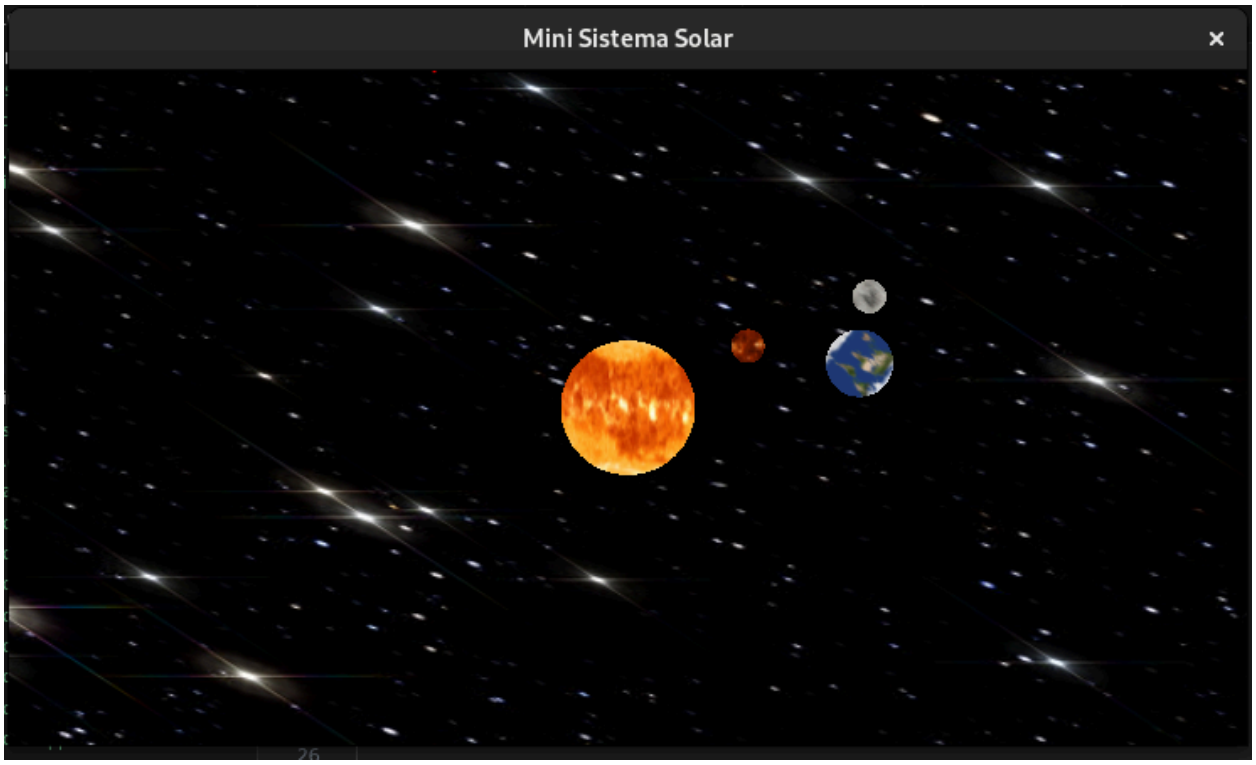
```cpp
// Generate texture coordinates for the disk
float* Disk::generateDiskTexCoord() {
    // Allocate memory for the texture coordinates
    float* texCoords = new float[(m_nslice + 2) * 2]; // u, v

    // The center of the disk
    texCoords[0] = 0.5f; // u
    texCoords[1] = 0.5f; // v

    // The angle increment between each slice
    float angleIncrement = 2.0f * glm::pi<float>() / m_nslice;

    // Loop to create texture coordinates for the perimeter of the disk
    for (int i = 0; i < m_nslice; ++i) {
        // Calculate the angle for the current slice
        float angle = i * angleIncrement;

        // Texture coordinates based on polar coordinates, mapping to [0, 1] range
        float u = 0.5f + 0.5f * cos(angle);
        float v = 0.5f + 0.5f * sin(angle);

        texCoords[(i + 1) * 2]     = u; // u
        texCoords[(i + 1) * 2 + 1] = v; // v
    }

    // Add the first perimeter vertex's texture coordinates again to close the fan
    texCoords[(m_nslice + 1) * 2]     = texCoords[2];  // u
    texCoords[(m_nslice + 1) * 2 + 1] = texCoords[3];  // v

    return texCoords;
}
```

Classe square

No construtor, as coordenadas dos vértices são especificadas de forma a posicionar o quadrado em relação ao ponto de origem, com vértices localizados nos cantos inferior esquerdo, inferior direito, superior esquerdo e superior direito. Simultaneamente, as coordenadas de textura são mapeadas a esses vértices. Essas informações são armazenadas em buffers de forma similar ao disk.

```cpp
Square::Square(float x, float y) : m_x(x), m_y(y) {
    float coord[] = {
        -x / 2.0f, -y / 2.0f,  // Bottom left
         x / 2.0f, -y / 2.0f,  // Bottom right
        -x / 2.0f,  y / 2.0f,  // Top left
         x / 2.0f,  y / 2.0f   // Top right
    };
    float texcoord[] = {
        0.0f, 0.0f,  // Bottom left
        1.0f, 0.0f,  // Bottom right
        0.0f, 1.0f,  // Top left
        1.0f, 1.0f   // Top right
    };
```

Resultado



Os resultados obtidos na simulação do mini sistema solar foram satisfatórios, com todos os requisitos implementados funcionando conforme o esperado. A criação dos astros foi realizada com sucesso, incluindo a representação do Sol, planetas e luas, todos interagindo de maneira dinâmica. A implementação das animações permite que os astros rotacionem em torno de seus próprios eixos, além de orbitarem em torno de outros astros. As texturas aplicadas a cada corpo celeste proporcionam um visual tipico de cada astro, enquanto o fundo espacial foi adequadamente representado.