Relatório de Desenvolvimento

Desenvolvimento do Jogo 3D Wormhole Runner em WebGL Puro

Introdução

Este projeto consiste no desenvolvimento de um jogo 3D utilizando somente WebGL, sem o uso de bibliotecas de alto nível como Three.js ou Babylon.js. O objetivo principal foi aplicar, de forma prática, os conceitos ensinados na UC Computação Gráfica, incluindo pipeline gráfico, transformações geométricas, projeção perspectiva, múltiplas câmeras, iluminação, texturização e interação em tempo real.

O jogo desenvolvido apresenta uma nave espacial que percorre um túnel tridimensional, ou buraco de minhoca, enfrentando asteroides que surgem à frente e avançam em direção ao jogador. O jogador deve desviar e destruir esses obstáculos utilizando tiros, acumulando pontos e avançando por níveis até atingir a condição de vitória ou perder todas as vidas.

Estrutura geral do projeto

O projeto foi organizado de forma modular, separando responsabilidades em arquivos distintos. O arquivo main.js atua como núcleo da aplicação, sendo responsável pela inicialização do WebGL, controle do loop principal, gerenciamento do estado do jogo, entrada do usuário, colisões, progressão de níveis e renderização da cena.

Os arquivos asteroid.js, laser.js e tunnel.js são responsáveis pela geração de **g**eometrias procedurais, enquanto loader.js implementa o carregamento e processamento de um modelo externo no formato OBJ, que contém os parâmetros para o ‘plot’ da nave e dos asteróides. Os shaders GLSL estão definidos em shaders.js, as texturas procedurais em textures.js, e a interface gráfica, como HUD e telas de estado, em style.css e index.html.

Essa separação facilita tanto a leitura quanto a manutenção do código, além de deixar clara a distinção entre lógica de jogo, geração de geometria e renderização gráfica.

Inicialização do WebGL e pipeline gráfico

A aplicação inicia criando explicitamente um **contexto WebGL** a partir de um elemento <canvas> no HTML, caracterizando o uso de WebGL puro:

```
const canvas = document.getElementById("glcanvas");
const gl = canvas.getContext("webgl");
if (!gl) {
  alert("WebGL não suportado");
  return;
}
```

Após a criação do contexto, o projeto compila os shaders e configura o pipeline gráfico. O programa de shader é criado a partir dos códigos definidos em shaders.js, que incluem um vertex shader e um fragment shader.

O vertex shader é responsável por aplicar as transformações geométricas e calcular a iluminação básica, enquanto o fragment shader realiza a aplicação das texturas e efeitos visuais, como o “flash” vermelho quando um objeto sofre dano.

A renderização segue o a ordem:

1. Transformação do objeto (Model Matrix)
2. Transformação da câmera (View Matrix)
3. Projeção perspectiva (Projection Matrix)

Projeção perspectiva e câmera

A projeção utilizada no jogo é perspectiva, dando profundidade visual e sensação tridimensional. Ela é criada com a função mat4.perspective, com campo de visão de 45 graus e planos de recorte configurados adequadamente, conforme abaixo:

```
const fieldOfView = 45 * Math.PI / 180;
const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
const projectionMatrix = mat4.create();
mat4.perspective(projectionMatrix, fieldOfView, aspect, 0.1, 200.0);
```

O sistema de câmera suporta múltiplas posições, controladas pela variável cameraMode. Ao pressionar a tecla C, o jogador alterna entre dois modos:

- Uma câmera fixa, posicionada atrás da nave.
- Uma câmera que acompanha a posição da nave, criando uma sensação de perseguição.

Essa alternância é implementada por meio de dois usos distintos da função mat4.lookAt, aplicados dinamicamente durante o loop de renderização.

Personagens e objetos do jogo

Nave do jogador

A nave é o personagem principal do jogo e é representada por um modelo 3D externo no formato OBJ (nave.obj). O carregamento desse modelo é feito pela função loadObjFile, localizada em loader.js.

Durante o carregamento, o arquivo OBJ é lido linha por linha, extraindo vértices e faces. Para cada face, o código calcula a normal geométrica utilizando produto vetorial, o que permite a correta aplicação da iluminação:

```
let nx = uy * vz - uz * vy;
let ny = uz * vx - ux * vz;
let nz = ux * vy - uy * vx;
```

Além disso, são geradas coordenadas de textura (UV) por mapeamento planar, permitindo que a nave receba uma textura procedural posteriormente.

Asteroides

Os asteroides são gerados de forma procedural, sem uso de modelos externos. A função createAsteroidData, em asteroid.js, cria uma esfera de baixa resolução utilizando divisões por latitude e longitude. Cada triângulo possui normais próprias, resultando em um visual facetado, adequado ao estilo do jogo.

Os asteroides são instanciados durante o jogo pela função spawnObstacle, conforme abaixo:

```
function spawnObstacle() {
  const x = (Math.random() * 6) - 3;
  const y = (Math.random() * 6) - 3;
  obstacles.push({
    position: [x, y, -100.0],
    hp: 5,
    hitFlash: 0
  });
}
```

Eles nascem longe no eixo Z (-100) e avançam em direção à nave ao longo do tempo, criando a sensação de aproximação.

Movimento e sensação de profundidade

O avanço dos asteroides ocorre dentro da função updateGame, responsável pela atualização do estado do jogo a cada frame:

```
obs.position[2] += 15.0 * deltaTime;
```

Esse deslocamento no eixo Z, combinado com a projeção perspectiva, faz com que os asteroides aparentem crescer à medida que se aproximam, mesmo sem alteração explícita da escala.

Laser (tiros)

O laser é uma geometria simples, gerada proceduralmente em laser.js, representada por um paralelepípedo fino. Cada tiro nasce na posição atual da nave e se desloca no sentido negativo do eixo Z:

```
shots.push({
  position: [shipPosition[0], shipPosition[1], shipPosition[2]]
});
```

Os tiros possuem tempo de vida limitado e são removidos quando saem da área visível ou colidem com um asteroide.

Túnel (cenário)

O túnel é o cenário principal do jogo e também é gerado proceduralmente. Ele consiste em um cilindro segmentado ao longo do eixo Z, com normais voltadas para o interior, simulando um ambiente fechado.

A textura do túnel se repete ao longo do comprimento, reforçando a sensação de movimento contínuo e velocidade.

Iluminação e shaders

A iluminação é implementada no vertex shader e consiste em:

- Uma luz ambiente.
- Duas luzes direcionais com direções e cores diferentes.

O shader também possui um uniforme (uUseLighting) que permite ativar ou desativar a iluminação por objeto. Isso é usado, por exemplo, para manter o túnel e os lasers com aparência mais “neon”, enquanto a nave e os asteroides recebem iluminação completa.

Além disso, o shader implementa um efeito de flash vermelho quando um objeto sofre dano, controlado por um uniforme (uFlash) enviado pelo JavaScript.

Aplicação das Texturas procedurais

Todas as texturas do jogo são procedurais, criadas dinamicamente utilizando um canvas 2D no arquivo textures.js. São geradas texturas distintas para:

- Nave
- Asteroides
- Túnel
- Laser

Após a criação, essas imagens são convertidas em WebGLTexture, com mipmaps e repetição habilitados. Essa abordagem elimina dependência de arquivos externos.

Jogabilidade, colisões e progressão

A jogabilidade é controlada pela função updateGame, que centraliza:

- Movimento da nave (teclado WASD ou setas).
- Controle de tiro com cooldown.
- Spawn de asteroides por intervalo de tempo.
- Detecção de colisões baseada em distância euclidiana.
- Atualização de pontuação e vidas.

Ao atingir uma pontuação alvo (data pela variável levelTargetScore), o jogo entra em estado de level up, pausando a ação e exibindo uma tela intermediária. A cada cinco níveis, o jogador recebe um bônus de vida.

O jogo termina em vitória ao ultrapassar o nível final, ou em derrota quando o HP chega a zero. Em ambos os casos, o jogador pode reiniciar a partida pressionando Enter.

Conclusão

O projeto demonstra, de forma prática, a aplicação dos principais conceitos de Computação Gráfica em tempo real, incluindo pipeline gráfico, projeção perspectiva, múltiplas câmeras, iluminação, texturas, geometria procedural e interação.

A decisão de utilizar WebGL puro exigiu a implementação manual de todas as etapas do pipeline, reforçando o entendimento técnico do funcionamento interno de motores gráficos. O resultado é um jogo funcional, visualmente consistente e alinhado aos requisitos propostos, servindo como uma aplicação completa dos conteúdos estudados na disciplina.