Documentație Proiect OpenGL

Simulator Fotbal

Student: Pântea Marius-Nicușor

Grupa: 30232

2. Prezentarea temei

Proiectul reprezintă un simulator de fotbal interactiv dezvoltat utilizând tehnologii moderne de grafică 3D. Implementarea folosește OpenGL ca API grafic principal, împreună cu următoarele biblioteci:

- GLFW pentru managementul ferestrei și input
- GLM pentru operații matematice și transformări
- stb_image pentru încărcarea texturilor
- tinyobjloader pentru încărcarea modelelor 3D

Obiectivele principale ale proiectului includ:

1. **Vizualizare avansată a scenei**:
    - Multiple moduri de vizualizare (solid, wireframe, point, smooth)
    - Controlul camerei cu 6 grade de libertate
    - Sistem de scalare și rotație a obiectelor
2. **Iluminare complexă**:
    - Iluminare direcțională globală
    - Spot lights pentru reflectoare
    - Umbre dinamice folosind shadow mapping
    - Implementare PBR (Physically Based Rendering)
3. **Efecte speciale**:
    - Sistem de particule pentru simularea ploii
    - Sistem de ceață volumetrică
    - Efecte de smooth shading
4. **Interactivitate**:
    - Control complet al camerei
    - Animații pentru minge și portar

- Detecția coliziunilor
- Mod de prezentare automată

3. Scenariul

3.1. Descrierea scenei și a obiectelor

3.1.1. Structura scenei

Scena este organizată ierarhic, constând din:

1. **Stadion**:
    - Model 3D complex încărcat din "models/arena/scenes/arena.obj"
    - Texturi pentru teren, tribune și alte elemente
    - Materiale PBR pentru realism sporit
    - Scară globală ajustabilă (0.1f * scaleFactorScene)
2. **Minge**:
    - Model 3D independent ("models/ball/ball.obj")
    - Sistem de fizică simplificat pentru mișcare
    - Materiale speciale pentru reflexie și strălucire
    - Scară: 0.2f * scaleFactorScene
3. **Portar**:
    - Model 3D animat ("models/goalkeeper/goalkeeper.obj")
    - Sistem de animație pentru plonjon
    - Detecție de coliziuni cu mingea
    - Scară: 7.0f * scaleFactorScene
4. **Sistem de iluminare**:

// Iluminare direcțională

glm::vec3 lightDir = glm::vec3(-0.5f, 1.0f, -0.5f);

glm::vec3 lightColor = glm::vec3(1.0f, 0.95f, 0.8f);

```
// Spot light pentru reflector
glm::vec3 spotLightPosition = glm::vec3(0.0f, 10.0f, 0.0f);
glm::vec3 spotLightDirection = glm::vec3(0.0f, -1.0f, 0.0f);
float spotLightCutOff = glm::cos(glm::radians(12.5f));
```

3.1.2. Sisteme de particule și efecte

1. **Sistem de ploaie**:

```
struct RainDrop {
    glm::vec3 position;
    float speed;
    float length;
};
const int MAX_RAINDROPS = 1000;
```

2. **Sistem de ceață**:

```
// Parametri ceață
float density = 0.007f;
float gradient = 1.5f;
vec3 fogColor = vec3(0.5f, 0.6f, 0.7f);
```

3.2. Funcționalități

3.2.1. Controlul camerei

Camera implementează următoarele funcționalități:

```
void Camera::move(MOVE_DIRECTION direction, float speed) {
    switch (direction) {
        case MOVE_FORWARD:
            cameraPosition += cameraFrontDirection * speed;
            break;
        case MOVE_BACKWARD:
```

```
            cameraPosition -= cameraFrontDirection * speed;
            break;
        // ...
    }}
void Camera::rotate(float pitch, float yaw) {
    glm::vec3 direction;
    direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    direction.y = sin(glm::radians(pitch));
    direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    cameraFrontDirection = glm::normalize(direction);
}
```

3.2.2. Animații și interacțiuni

1. **Animație minge**:

```
void updateBallPosition() {
    if (!isAnimating) return;
    float currentTime = glfwGetTime();
    float deltaTime = currentTime - lastFrameTime;
    ballVerticalSpeed += GRAVITY * deltaTime;
    ballPosition.y += ballVerticalSpeed * deltaTime;
    if (ballPosition.y <= 0.2f) {
        ballPosition.y = 0.2f;
        ballVerticalSpeed = -ballVerticalSpeed * BOUNCE_FACTOR;
    }
}
```

2. **Animație portar**:

```
void updateGoalkeeper() {
    const float DIVE_HEIGHT = 2.5f;
    const float DIVE_DISTANCE = 3.0f;
    const float DIVE_DURATION = 1.0f;

    if (isDiving) {
        diveProgress += 0.016f;
        float t = diveProgress / DIVE_DURATION;
        goalkeeperPosition.y = DIVE_HEIGHT * sin(t * 3.14159f);
        goalkeeperPosition.x = diveDirection * DIVE_DISTANCE * t;
    }
}
```

4. Detalii de implementare

4.1. Funcții și algoritmi

4.1.1. Soluții posibile

1. **Sistemul de particule pentru ploaie**:

a) Implementare shader-based:

```
// Vertex Shader
void main() {
    gl_Position = projection * view * model * vec4(position, 1.0);
    gl_PointSize = 6.0;
}

// Fragment Shader
void main() {
    vec2 circCoord = 2.0 * gl_PointCoord - 1.0;
```

```
    float circle = dot(circCoord, circCoord);
    if (circle > 1.0) discard;
    FragColor = vec4(0.7, 0.7, 1.0, 0.5);
}
```

b) Sistem de particule CPU-based:

```
struct Particle {
    glm::vec3 position;
    glm::vec3 velocity;
    float life;
};
```

2. **Shadow Mapping**:

a) Basic shadow mapping:

```
void renderShadowMap() {
    glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
    glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
}
```

b) PCF shadow mapping:

```
float ShadowCalculation(vec4 fragPosLightSpace, float bias) {
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    for(int x = -1; x <= 1; ++x) {
        for(int y = -1; y <= 1; ++y) {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
```

```
    }
  }
  shadow /= 9.0;
  return shadow; }
```

4.1.2. Motivarea abordării alese

1. **Sistem de particule pentru ploaie**:
    - S-a ales implementarea shader-based pentru performanță superioară
    - Permite manipularea unui număr mare de particule (1000+)
    - Reduce încărcarea CPU-ului
2. **Shadow mapping**:
    - Implementare PCF pentru umbre mai moi și mai realiste
    - Compromis bun între calitate și performanță
3. **Fizica mingii**:
    - Sistem simplificat bazat pe ecuații de mișcare
    - Suficient de realist pentru scopul aplicației
    - Performanță bună

4.2. Modelul grafic

4.2.1. Pipeline-ul de rendering

1. **Shadow Pass**:

```
depthMapShader.useShaderProgram();
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
```

2. **Main Rendering Pass**:

```
glViewport(0, 0, width, height);
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
myBasicShader.useShaderProgram();
```

4.2.2. Shading și materiale

1. **PBR Shader**:

```
// Fragment shader
uniform sampler2D metallicTexture;
uniform sampler2D roughnessTexture;
uniform bool usePBR = false;

// PBR calculations
float metallic = usePBR ? texture(metallicTexture, fTexCoords).r : 1.0;
float roughness = usePBR ? texture(roughnessTexture, fTexCoords).r : 0.5;
```

2. **Spot Light**:

```
vec3 calculateSpotLight(vec3 normal, vec3 viewDir, vec3 fragPos) {
    float theta = dot(lightToFrag, -spotLightDirView);
    float epsilon = spotLightCutOff - spotLightOuterCutOff;
    float intensity = clamp((theta - spotLightOuterCutOff) / epsilon, 0.0, 1.0);
}
```

4.3. Structuri de date

1. **Mesh**:

```
struct Vertex {
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
```

```
};

struct Texture {
    GLuint id;
    std::string type;
    std::string path;
};

class Mesh {
    std::vector<Vertex> vertices;
    std::vector<GLuint> indices;
    std::vector<Texture> textures;
    Buffers buffers;
};
```

2. **Model3D**:

```
class Model3D {
private:
    std::vector<gps::Mesh> meshes;
    std::vector<gps::Texture> loadedTextures;

    void ReadOBJ(std::string fileName, std::string basePath);
    gps::Texture LoadTexture(std::string path, std::string type);
};
```

5. Prezentarea interfeței grafice utilizator / manual de utilizare

5.1. Controlul camerei

- **W**: Deplasare înainte
- **S**: Deplasare înapoi
- **A**: Deplasare stânga
- **D**: Deplasare dreapta
- **Mouse**: Rotire cameră
- **Scroll**: Zoom in/out

5.2. Controlul scenei

- **B**: Animație minge (săritură)
- **V**: Șut spre portar
- **R**: Activare/dezactivare ploaie
- **L**: Activare/dezactivare ceață
- **P**: Schimbare mod de vizualizare
- **O**: Activare/dezactivare mod prezentare
- **ESC**: Ieșire din aplicație

5.3. Controlul iluminării

- **1,2**: Ajustare poziție X spot light
- **3,4**: Ajustare poziție Y spot light

6. Concluzii și dezvoltări ulterioare

6.1. Realizări

1. Implementarea cu succes a tuturor cerințelor de bază
2. Sistem de iluminare complex cu PBR

3. Efecte speciale functionale (ploaie, ceață)
4. Interactivitate ridicată

6.2. Limitări actuale

1. Fizică simplificată pentru minge
2. Animații limitate pentru portar
3. Lipsa efectelor sonore

6.3. Dezvoltări ulterioare posibile

1. Îmbunătățiri grafice:
    - Implementare SSAO (Screen Space Ambient Occlusion)
    - Bloom și HDR
    - Motion blur pentru mișcări rapide
2. Gameplay:
    - Sistem de scor
    - Mai mulți jucători
    - AI pentru portar
3. Tehnic:
    - Optimizare rendering pipeline
    - Implementare deferred rendering
    - Îmbunătățire sistem de coliziuni

7. Referințe

1. OpenGL Programming Guide (Red Book)
2. Learn OpenGL - https://learnopengl.com/
3. GLFW Documentation - https://www.glfw.org/docs/latest/
4. GLM Documentation - https://glm.g-truc.net/0.9.9/index.html

5. Real-Time Rendering, Fourth