RTGP PROJECT REPORT

Hasan Kuşpınar

04/07/2024

Overview

This project showcases a real-time graphics and physics simulation using OpenGL and the Bullet Physics Library. The application renders 3D models with OpenGL, processes user inputs to control a camera, and simulates physical interactions using the Bullet Physics engine. The environment includes several rooms, each with distinct gravitational settings, where the user can shoot projectiles and observe their behavior.

Libraries and Dependencies

GLFW: Manages window creation and user input.

GLAD: Loads OpenGL function pointers.

GLM: Handles matrix and vector math operations.

STB Image: Used for image loading.

Bullet Physics Library: Manages physics simulations.

Custom Utility Libraries: Handles shaders, model loading, camera movement, and physics.

Setup and Initialization

Window and OpenGL Context

The application initializes a window using GLFW, setting the appropriate OpenGL version and creating an OpenGL context. GLAD is then used to load OpenGL functions.

Shader and Model Loading

Shaders for rendering are loaded from external files, and models for objects such as cubes and spheres are imported.

User Interaction

Keyboard Inputs

WASD Keys: Control the camera's movement.

ESC Key: Closes the application.

L Key: Toggles between wireframe and filled polygon rendering modes.

SPACE Key: Shoots a projectile from the camera's position.

Mouse Inputs

The mouse movement updates the camera's view direction, allowing for free look navigation.

```
void apply_camera_movements()
{
    float originalY = camera.Position.y;
    GLboolean diagonal_movement = (keys[GLFW_KEY_W] ^ keys[GLFW_KEY_S]) && (keys[GLFW_KEY_A] ^ keys[GLFW_KEY_D]);
    camera.SetMovementCompensation(diagonal_movement);

    if(keys[GLFW_KEY_W])
        camera.ProcessKeyboard(FORWARD, deltaTime);
    if(keys[GLFW_KEY_S])
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    if(keys[GLFW_KEY_A])
        camera.ProcessKeyboard(LEFT, deltaTime);
    if(keys[GLFW_KEY_D])
        camera.ProcessKeyboard(RIGHT, deltaTime);

    camera.Position.y = originalY;
}
```

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);

    if(key == GLFW_KEY_L && action == GLFW_PRESS)
        wireframe=!wireframe;


    btVector3 impulse;
    glm::vec3 rot = glm::vec3(0.0f, 0.0f, 0.0f);
    glm::vec4 shoot;

    GLfloat shootInitialSpeed = 15.0f;

    btRigidBody* sphere;
    glm::mat4 unproject;

    if(key == GLFW_KEY_SPACE && action == GLFW_PRESS)
    {
```

Rendering Process

Camera and Projection

Camera Matrix: The view matrix is derived from the camera's position and orientation.

Projection Matrix: A perspective projection matrix is set up to simulate depth.

Object Rendering

Static Objects: Plane and room models are rendered at fixed positions with transformations applied. Rooms are based on the cube model from model class.

Dynamic Objects: Projectiles and other dynamic entities are rendered based on their current state in the physics simulation. Projectiles are based on the sphere model from model class.

Shader Usage

Custom shaders are used to apply lighting models to objects, making use of uniform variables to control properties like color, lighting, and material properties.

Physics Simulation

Room Definition

Rooms are defined with specific positions, sizes, gravity, and friction coefficients. These properties influence the behavior of objects within the room.

```
rooms.push_back(Room{glm::vec3(-10.0f, 0.0f, 0.0f), glm::vec3(5.0f, 5.0f, 5.0f), glm::vec3(0.0f, -30.8f, 0.0f), 0.5f});
rooms.push_back(Room{ glm::vec3(10.0f, 0.0f, 0.0f), glm::vec3(5.0f, 5.0f, 5.0f), glm::vec3(0.0f, -5.0f, 0.0f), 3.5f});
rooms.push_back(Room{ glm::vec3(0.0f, 0.0f, 10.0f), glm::vec3(5.0f, 5.0f, 5.0f), glm::vec3(0.0f, -15.0f, 0.0f), 1.5f});
rooms.push_back(Room{ glm::vec3(0.0f, 0.0f, -10.0f), glm::vec3(5.0f, 5.0f, 5.0f), glm::vec3(0.0f, -100.0f, 0.0f), 5.5f});
```

Dynamic Object Creation

Objects are added to the physics world, with dynamic properties such as mass, friction, and restitution. The simulation updates these objects' states over time.

```
void updateBulletPhysics() {
    int num_cobjs = bulletSimulation.dynamicsWorld->getNumCollisionObjects();
    for (int i = 0; i < num_cobjs; i++) {
        btCollisionObject* obj = bulletSimulation.dynamicsWorld->getCollisionObjectArray()[i];
        btRigidBody* body = btRigidBody::upcast(obj);
        if (body && body->getMass() == 1.0f) {
            btTransform transform;
            body->getMotionState()->getWorldTransform(transform);
            glm::vec3 position = glm::vec3(transform.getOrigin().getX(), transform.getOrigin().getY(), transform.getOrigin().getZ());

            Room* currentRoom = getCurrentRoom(position);
            if (currentRoom) {
                btVector3 roomGravity = btVector3(currentRoom->gravity.x, currentRoom->gravity.y, currentRoom->gravity.z);
                body->setGravity(roomGravity);
            }
        }
    }
}
```

Gravity and Friction Handling

Objects are assigned different gravitational forces and frictions based on the room they are in. This is achieved through functions that check an object's position relative to room boundaries and apply the corresponding physics.

```
bool isInsideRoom(const glm::vec3& position, const Room& room)
{
    glm::vec3 minBounds = room.position - room.size / 2.0f;
    glm::vec3 maxBounds = room.position + room.size / 2.0f;
    return (position.x >= minBounds.x && position.x <= maxBounds.x &&
    position.y >= minBounds.y && position.y <= maxBounds.y &&
    position.z >= minBounds.z && position.z <= maxBounds.z);
}
Room* getCurrentRoom(glm::vec3 position)
{
    for (Room& room : rooms)
    {
        if (isInsideRoom(position, room))
        {
            return &room;
        }
    }
    return nullptr;
}
```

Shooting Mechanism

When the user presses SPACE, a new projectile is created and propelled from the camera's position. The initial velocity and the physics effects are determined by the cursor's position on the screen and the room that it is fired in.

```
if(key == GLFW_KEY_SPACE && action == GLFW_PRESS)
{

    Room* firingRoom = getCurrentRoom(glm::vec3(camera.Position.x, camera.Position.y, camera.Position.z));

    if (firingRoom) {
        sphere = bulletSimulation.createRigidBody(SPHERE, camera.Position, sphere_size, rot, 1.0f, firingRoom->friction, 0.3f);
        sphere->setGravity(btVector3(firingRoom->gravity.x, firingRoom->gravity.y, firingRoom->gravity.z));
    } else {
        sphere = bulletSimulation.createRigidBody(SPHERE, camera.Position, sphere_size, rot, 1.0f, 0.3f, 0.3f);
    }
    shoot.x = (cursorX/screenWidth) * 2.0f - 1.0f;
    shoot.y = -(cursorY/screenHeight) * 2.0f + 1.0f;

    shoot.z = 1.0f;

    shoot.w = 1.0f;
    unproject = glm::inverse(projection * view);


    shoot = glm::normalize(unproject * shoot) * shootInitialSpeed;

    impulse = btVector3(shoot.x, shoot.y, shoot.z);
    sphere->applyCentralImpulse(impulse);
}
```

Performance Monitoring

The application tracks and displays the frame rate (FPS) to monitor performance and ensure smooth operation. Since the graphics are not that enhanced the FPS is always over 60 as expected.

```
double currentTime = glfwGetTime();
frameCount++;
if ( currentTime - lastTime >= 1.0 ){
    double fps = frameCount/(lastTime-currentTime);
    string newTitle = "RTGP Project - " + to_string(fps) + "FPS";
    glfwSetWindowTitle(window, newTitle.c_str());
    frameCount = 0;
    lastTime += 1.0;
}
```

Conclusion

This project successfully integrates real-time 3D rendering with dynamic physics simulation. The inclusion of multiple rooms with varying gravitational forces makes the simulation more interesting since different physics effects can be experienced. Future enhancements could include more advanced collision detection, additional interactive elements, and improved visual effects.