Relatório do Projeto 1 INF1761 - Computação Gráfica

Integrantes do grupo:

2310822 - Eduardo Eugênio de Souza
2210508 - Gabriel Augusto Gedey Porto

O código para a biblioteca EnGene e suas dependências pode ser encontrado aqui: repositório do EnGene
O código para o trabalho em si está neste: repositório da matéria
O vídeo demonstrando também se encontra dentro do repositório e aqui está um link para o video no drive

Vamos à explicação alto nível do código:
A ideia principal para o interesse do projeto foi criar uma função que seja capaz de criar um corpo dentro do contêiner. Olhemos para ela a seguir:

```
void createPhysicsCircle(const glm::vec2& initialPosition, float radius,
shader::ShaderPtr shader, std::string container)
{
    // 1. Create the transform that will be shared between the scene node and the
physics body.
    auto circle_transform = transform::Transform::Make();

    // 2. Create the visual representation (the scene node).
    scene::graph()->buildAt(container)
    .addNode("Circle" + std::to_string(circle_count))
        .with<component::GeometryComponent>(earth)
        .with<component::ShaderComponent>(shader)
        .with<component::TextureComponent>(
            texture::Texture::Make("../assets/images/earth_from_space.jpg"), //
Using earth texture for all
            "tex",
            0
        )
        // This component holds the transform that the physics engine will update.
        .with<component::TransformComponent>(circle_transform)
        .with<component::TransformComponent>(
            transform::Transform::Make()
            ->translate(0,0,0.5)
            ->scale(radius,radius,radius),
            101 // priority
        );

    circle_count++;

    // 3. Create the physics body.
```

```
    // Pass the initial position, the shared transform, and the radius.
    auto physics_body = PhysicsBody::Make(initialPosition, circle_transform,
radius);

    // 4. Add the new body to the engine.
    physicsEngine->addBody(physics_body);
}
```

Esta função cria, abaixo do nó do container fornecido, um nó de Círculo texturizado (que é o earth que é inicializado em outro local do código de tal maneira que se reutiliza a geometria, tendo mais eficiência de código), garantindo que vá ser renderizado apropriadamente ao também forçar aquele nó a ter um shader que suporte textura (caso já seja o shader sendo utilizado antes do nó, a biblioteca é inteligente para não ficar se repetindo). A biblioteca também é inteligente para não carregar uma textura na GPU mais de uma vez caso aquela já tenha sido carregada anteriormente, assim evitando carregar uma nova textura a cada instância do círculo, principalmente durante a execução. A physicsEngine que criamos funciona de tal maneira que ela aplica o posicionamento do corpo a partir de um transform, que é encapsulado no PhysicsBody, o que por definição leva como hipótese o fato que a geometria deve estar centrada na origem das suas coordenadas locais. Decidimos também fazer desta maneira para que qualquer corpo do grafo de cena possa ser “ligado” à física simplesmente adicionando o componente de tal, e chamando o update da engine associada.

A seguir, mostraremos aqui a inicialização da cena estática, onde é construído o container, e os parâmetros da physicsEngine e da geometria da terra:

```
// creates the texture shader and configures its uniforms
        textured_shader = shader::Shader::Make(
            "../shaders/textured_vertex.glsl",
            "../shaders/textured_fragment.glsl"
        )
        ->configureUniform<glm::mat4>("M", transform::current)
        ->configureUniform<int>("tex", texture::getUnitProvider("tex"));

        // <<< 1. Initialize the Physics Engine
        // We define the simulation area to match the typical OpenGL normalized
device coordinates.
        physicsEngine = Engine::make(-1.0f, 1.0f, -1.0f, 1000.0f, glm::vec2(0.0f,
-2.0f));

        // <<< 2. Create the container for the circles
        scene::graph()->addNode("container")
            .with<component::GeometryComponent>(
                Quad::Make(-1,-1,1,1)
            )
            .with<component::ShaderComponent>(textured_shader)
            .with<component::TextureComponent>(
                texture::Texture::Make("../assets/images/starred-paint.jpg"),
                "tex",
                1
```

```
            );

        // <<< 3. Initialize the reused Circle Geometry
        earth = TexturedCircle::Make(
                0.0f, 0.0f,  // Center pos (local to the node)
                1.0f,        // Radius
                32,          // Segments
                0.5f, 0.5f,  // Texture scale/offset if needed
                0.45f
            );
```

Aqui a textura do container é um quadrado texturizado simples, da classe Quad. É relevante ressaltar que o shader obtém os valores para os uniforms de maneira dinâmica, chamando a função que for passada por referência (ou lambda). As classes de geometria texturizada não especificam de antemão qual a textura que será renderizada, apenas especificam as coordenadas de textura dos vértices. O container tem topo especificado como um valor muito alto para atender ao requisito do container não ter um topo, o que simplifica o código da Engine para que não precise verificar toda hora se alguma dimensão foi ou não setada para limitar a área de cálculo. Dito isto, vamos ao cerne do código desta, que está em duas funções: solveCollisions() e update(deltaTime).

Segue o código:

```
    void solveCollisions()
    {
        for (size_t i = 0; i < bodies.size(); i++)
        {
            for (size_t j = i + 1; j < bodies.size(); j++)
            {
                glm::vec2 posA = bodies[i]->getPosition();
                glm::vec2 posB = bodies[j]->getPosition();
                float distance = glm::length(posA - posB);
                float minDistance = bodies[i]->getRadius() +
bodies[j]->getRadius();

                if (distance < minDistance && distance > 0.0f)
                {
                    glm::vec2 collisionNormal = glm::normalize(posB - posA);
                    glm::vec2 correction = collisionNormal * (minDistance -
distance) * 0.5f;

                    if (rigid) bodies[i]->moveRigid(-correction);
                    else bodies[i]->move(-correction);

                    if (rigid) bodies[j]->moveRigid(correction);
                    else bodies[j]->move(correction);
                }
            }
        }
    }
    void update(float deltaTime)
    {
```

```
        for (int i = 0; i < numSubsteps; i++)
        {
            float substepDelta = deltaTime / static_cast<float>(numSubsteps);
            for (auto &body : bodies)
            {
                body->accelerate(gravity);
                body->calculateNextPosition(substepDelta);
            }
            for (int k = 0; k < solverSteps; k++) {
                solveCollisions();
                for (auto &body : bodies)
                {
                    constrainToArea(areaMin.x, areaMax.x, areaMin.y, areaMax.y,
body);
                }
            }
        }
        for (auto &body : bodies) {
            body->update();
        }
    }
```

Então, o solveCollisions() é a função que faz a parte mais importante de resolver as colisões usando a distância entre os centros, e checando se é menor do que a soma dos raios. Se for menor, então ela dá um empurrão nos dois corpos na direção da normal entre eles. Segundo, o update é chamado a cada 1/60 segundos (ou o mais próximo disso quanto possível, e também é desacoplado com o framerate) e ele subdivide o cálculo do que deve acontecer para o intervalo em 10 partições (substeps). Para cada substep ele primeiro faz o cálculo do movimento, e depois resolve as constraints, que são colisões e paredes. A resolução de constraints também pode ser iterada sobre para se aproximar de uma solução mais correta, mas mesmo com apenas uma iteração (o que foi usado para o projeto) as colisões são bem resolvidas. Algo que fizemos como uma otimização foi separar o cálculo da posição correta com a atualização da posição visual, que só é chamada logo antes de terminar o update da engine.

Finalmente, segue o código fonte da main e da física da simulação:

proj1.cpp:

```
#include <EnGene.h>
#include <core/scene.h>
#include <core/scene_node_builder.h>
#include <other_genes/textured_shapes/textured_circle.h>
#include <other_genes/textured_shapes/quad.h>
#include <other_genes/basic_input_handler.h>
#include <gl_base/error.h>
#include <gl_base/shader.h>
#include <components/all.h>
```

```
// <<< Include physics engine files
#include "physics/engine.h"
#include "physics/physicsBody.h"

#include <random> // For generating random positions

#define BACKGROUND_COLOR 0.05f, 0.05f, 0.1f

// <<< Declare the physics engine pointer so it can be accessed by on_init and
on_update
EnginePtr physicsEngine;
int circle_count = 0;
TexturedCirclePtr earth;

// Function to create a circle with a physics body
void createPhysicsCircle(const glm::vec2& initialPosition, float radius,
shader::ShaderPtr shader, std::string container)
{
    // 1. Create the transform that will be shared between the scene node and the
physics body.
    auto circle_transform = transform::Transform::Make();

    // 2. Create the visual representation (the scene node).
    scene::graph()->buildAt(container)
    .addNode("Circle" + std::to_string(circle_count))
        .with<component::GeometryComponent>(earth)
        .with<component::ShaderComponent>(shader)
        .with<component::TextureComponent>(
            texture::Texture::Make("../assets/images/earth_from_space.jpg"), //
Using earth texture for all
            "tex",
            0
        )
        // This component holds the transform that the physics engine will update.
        .with<component::TransformComponent>(circle_transform)
        .with<component::TransformComponent>(
            transform::Transform::Make()
            ->translate(0,0,0.5)
            ->scale(radius,radius,radius),
            101
        );

    circle_count++;

    // 3. Create the physics body.
    // Pass the initial position, the shared transform, and the radius.
    auto physics_body = PhysicsBody::Make(initialPosition, circle_transform,
radius);

    // 4. Add the new body to the engine.
```

```
    physicsEngine->addBody(physics_body);
}


int main() {

    // Setup for random positions
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> distr(-0.7f, 0.7f);
    std::uniform_real_distribution<> radii(0.03f, 0.3f);
    int initialNumberOfCircles = 30;


    shader::ShaderPtr textured_shader;

    auto on_init = [&](engene::EnGene& app) {
        // configures the uniforms from the base shader.
        // app.getBaseShader()->configureUniform<glm::mat4>("M",
transform::current);

        // creates the texture shader and configures its uniforms
        textured_shader = shader::Shader::Make(
            "../shaders/textured_vertex.glsl",
            "../shaders/textured_fragment.glsl"
        )
        ->configureUniform<glm::mat4>("M", transform::current)
        ->configureUniform<int>("tex", texture::getUnitProvider("tex"));

        // <<< 1. Initialize the Physics Engine
        // We define the simulation area to match the typical OpenGL normalized
device coordinates.
        physicsEngine = Engine::make(-1.0f, 1.0f, -1.0f, 1000.0f, glm::vec2(0.0f,
-2.0f));

        // <<< 2. Create the container for the circles
        scene::graph()->addNode("container")
            .with<component::GeometryComponent>(
                Quad::Make(-1,-1,1,1)
            )
            .with<component::ShaderComponent>(textured_shader)
            .with<component::TextureComponent>(
                texture::Texture::Make("../assets/images/starred-paint.jpg"),
                "tex",
                1
            );

        // <<< 3. Initialize the reused Circle Geometry
        earth = TexturedCircle::Make(
                0.0f, 0.0f,  // Center pos (local to the node)
```

```
                1.0f,         // Radius
                32,           // Segments
                0.5f, 0.5f,   // Texture scale/offset if needed
                0.45f
            );
    };

    double time_passed = 0;

    // This function handles the fixed-timestep simulation logic.
    auto on_fixed_update = [&](double fixed_timestep) {
        // Update the physics engine.
        // This will calculate new positions based on gravity and collisions.
        // Because we linked the transforms, the visual objects will move
automatically.

        if (circle_count < initialNumberOfCircles)
        {
            time_passed += fixed_timestep;
            if (time_passed > 1) {
                glm::vec2 pos(distr(gen), 0.8f); // Random initial position
                createPhysicsCircle(pos, radii(gen), textured_shader, "container");
                time_passed = 0;
            }
        }

        if(physicsEngine)
        {
            physicsEngine->update(static_cast<float>(fixed_timestep));
        }

    };

    // This function handles all rendering.
    auto on_render = [&](double alpha) {
        // The 'alpha' parameter can be used for smooth interpolation between
physics states
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Render the scene graph with the updated positions.
        scene::graph()->draw();

        Error::Check("render");
    };

    try {
        engene::EnGeneConfig config;
        config.width = 800;
        config.height = 800;
        config.title = "Physics Engine Demo";
```

```
        config.clearColor[0] = 0.05f;
        config.clearColor[1] = 0.05f;
        config.clearColor[2] = 0.1f;
        config.clearColor[3] = 1.0f;
        // config.base_vertex_shader_source = "../shaders/vertex.glsl";
        // config.base_fragment_shader_source = "../shaders/fragment.glsl";

        auto* handler = new input::InputHandler();

handler->registerCallback<input::InputType::MOUSE_BUTTON>([&](MOUSE_BUTTON_HANDLER_
ARGS) {
            if (action == GLFW_PRESS) {
                double xpos, ypos;
                glfwGetCursorPos(window, &xpos, &ypos);

                int fb_w, fb_h;
                glfwGetFramebufferSize(window, &fb_w, &fb_h);

                // Convert screen coordinates (pixels) to Normalized Device
Coordinates (NDC) [-1, 1]
                float x_ndc = ((float)xpos / (float)fb_w) * 2.0f - 1.0f;
                float y_ndc = (1.0f - ((float)ypos / (float)fb_h)) * 2.0f - 1.0f;

                glm::vec2 pos(x_ndc,y_ndc);
                createPhysicsCircle(pos, radii(gen), textured_shader, "container");
            }
        });

        bool m_wireframe_mode = false;

        handler->registerCallback<input::InputType::KEY>([&](KEY_HANDLER_ARGS){

            if (key == GLFW_KEY_Q && action == GLFW_PRESS) {
                glfwSetWindowShouldClose(window, GLFW_TRUE);
            }
            // Toggle wireframe mode
            else if (key == GLFW_KEY_T && action == GLFW_PRESS) {
                m_wireframe_mode = !m_wireframe_mode;
                if (m_wireframe_mode) {
                    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
                } else {
                    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                }
                std::cout << "Wireframe mode " << (m_wireframe_mode ? "ON" : "OFF")
<< std::endl;
            }
            else if (key == GLFW_KEY_C && action == GLFW_PRESS) {
                scene::graph()->clearGraph();
                physicsEngine->clearBodies();
            }
```

```
        });

        engene::EnGene app(
            on_init,
            on_fixed_update,
            on_render,
            config,
            handler
        );

        app.run();

    } catch (const std::runtime_error& e) {
        std::cerr << "An error occurred: " << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```

physics/engine.h:

```
#pragma once
#include "physicsBody.h"
#include <glm/glm.hpp>

class Engine;
using EnginePtr = std::shared_ptr<Engine>;

class Engine
{
private:
    glm::vec2 gravity = glm::vec2(0.0f, -9.81f);
    glm::vec2 areaMin = glm::vec2(-1.0f, -1.0f);
    glm::vec2 areaMax = glm::vec2(1.0f, 1.0f);
    int numSubsteps;
    int solverSteps;
    bool rigid;
    std::vector<PhysicsBodyPtr> bodies;

    Engine(int substeps, int solverSteps = 1, bool rigid = true) :
numSubsteps(substeps), solverSteps(solverSteps), rigid(rigid) {}

    void constrainToArea(float minX, float maxX, float minY, float maxY,
PhysicsBodyPtr body)
    {
        glm::vec2 pos = body->getPosition();
        float radius = body->getRadius();
        glm::vec2 correction(0.0f, 0.0f);

        if (pos.x - radius < minX)
```

```
        {
            correction.x = (minX + radius) - pos.x;
        }
        else if (pos.x + radius > maxX)
        {
            correction.x = (maxX - radius) - pos.x;
        }
        if (pos.y - radius < minY)
        {
            correction.y = (minY + radius) - pos.y;
        }
        else if (pos.y + radius > maxY)
        {
            correction.y = (maxY - radius) - pos.y;
        }
        if (rigid) body->moveRigid(correction);
        else body->move(correction);
    }


    void solveCollisions()
    {
        for (size_t i = 0; i < bodies.size(); i++)
        {
            for (size_t j = i + 1; j < bodies.size(); j++)
            {
                glm::vec2 posA = bodies[i]->getPosition();
                glm::vec2 posB = bodies[j]->getPosition();
                float distance = glm::length(posA - posB);
                float minDistance = bodies[i]->getRadius() +
bodies[j]->getRadius();

                if (distance < minDistance && distance > 0.0f)
                {
                    glm::vec2 collisionNormal = glm::normalize(posB - posA);
                    glm::vec2 correction = collisionNormal * (minDistance -
distance) * 0.5f;

                    if (rigid) bodies[i]->moveRigid(-correction);
                    else bodies[i]->move(-correction);

                    if (rigid) bodies[j]->moveRigid(correction);
                    else bodies[j]->move(correction);
                }
            }
        }
    }
public:

    static EnginePtr make(int substeps = 5, int solverSteps = 1)
```

```
    {
        return EnginePtr (new Engine(substeps, solverSteps));
    }

    static EnginePtr make(float gravityX, float gravityY, int substeps = 5, int
solverSteps = 1)
    {
        EnginePtr engine = Engine::make(substeps, solverSteps);
        engine->setGravity(glm::vec2(gravityX, gravityY));
        return engine;
    }

    static EnginePtr make(glm::vec2 gravity, int substeps = 5, int solverSteps = 1)
    {
        EnginePtr engine = Engine::make(substeps, solverSteps);
        engine->setGravity(gravity);
        return engine;
    }

    static EnginePtr make(glm::vec2 minArea, glm::vec2 maxArea, glm::vec2 gravity =
glm::vec2(0.0f, -9.81f), int substeps = 5, int solverSteps = 1)
    {
        EnginePtr engine = Engine::make(substeps, solverSteps);
        engine->setArea(minArea, maxArea);
        engine->setGravity(gravity);
        return engine;
    }
    static EnginePtr make(float minX, float maxX, float minY, float maxY, glm::vec2
gravity = glm::vec2(0.0f, -9.81f), int substeps = 5, int solverSteps = 1)
    {
        EnginePtr engine = Engine::make(substeps, solverSteps);
        engine->setArea(minX, maxX, minY, maxY);
        engine->setGravity(gravity);
        return engine;
    }

    void update(float deltaTime)
    {
        for (int i = 0; i < numSubsteps; i++)
        {
            float substepDelta = deltaTime / static_cast<float>(numSubsteps);
            for (auto &body : bodies)
            {
                body->accelerate(gravity);
                body->calculateNextPosition(substepDelta);
            }
            for (int k = 0; k < solverSteps; k++) {
                solveCollisions();
                for (auto &body : bodies)
                {
```

```
                    constrainToArea(areaMin.x, areaMax.x, areaMin.y, areaMax.y,
body);
                }
            }
        }
        for (auto &body : bodies) {
            body->update();
        }
    }

    void multiplyGravity(float factor)
    {
        gravity *= factor;
    }

    void setGravity(glm::vec2 gravity)
    {
        this->gravity = gravity;
    }

    void addBody(PhysicsBodyPtr body)
    {
        bodies.push_back(body);
    }

    void clearBodies()
    {
        bodies.clear();
    }

    void setArea(float minX, float maxX, float minY, float maxY)
    {
        areaMin = glm::vec2(minX, minY);
        areaMax = glm::vec2(maxX, maxY);
    }

    void setArea(glm::vec2 min, glm::vec2 max)
    {
        areaMin = min;
        areaMax = max;
    }
};
```

physics/physicsBody.h:

```
#pragma once
#include <glm/glm.hpp>
#include <memory>
#include <gl_base/transform.h>

class PhysicsBody;
```

```
using PhysicsBodyPtr = std::shared_ptr<PhysicsBody>;

class PhysicsBody
{
private:
    glm::vec2 positionOld;
    glm::vec2 positionCurrent;
    glm::vec2 acceleration;
    float radius;
    transform::TransformPtr nodeTransform;

    PhysicsBody(const glm::vec2 &oldPosition, const glm::vec2 &initialPosition,
transform::TransformPtr nodeTransform, float radius)
        : positionOld(oldPosition), positionCurrent(initialPosition),
acceleration(0.0f, 0.0f), radius(radius), nodeTransform(nodeTransform) {}

public:
    static PhysicsBodyPtr Make(const glm::vec2 &initialPosition,
transform::TransformPtr nodeTransform, float radius = 1.0f)
    {
        return PhysicsBodyPtr(new PhysicsBody(initialPosition, initialPosition,
nodeTransform, radius));
    }

    static PhysicsBodyPtr Make(const glm::vec2 &oldPosition, const glm::vec2
&initialPosition, transform::TransformPtr nodeTransform, float radius = 1.0f)
    {
        return PhysicsBodyPtr(new PhysicsBody(oldPosition, initialPosition,
nodeTransform, radius));
    }

    void setNodeTransform(transform::TransformPtr t)
    {
        nodeTransform = t;
    }

    void calculateNextPosition(float deltaTime)
    {
        glm::vec2 velocity = positionCurrent - positionOld;
        positionOld = positionCurrent;
        positionCurrent += velocity + acceleration * deltaTime * deltaTime;
        acceleration = glm::vec2(0.0f, 0.0f);

        // Atualiza o transform do nó, se existir
        if (nodeTransform)
        {
            nodeTransform->setTranslate(positionCurrent.x, positionCurrent.y,
0.0f);
        }
    }
```

```
    void accelerate(const glm::vec2 &accel)
    {
        acceleration += accel;
    }
    glm::vec2 getPosition() const
    {
        return positionCurrent;
    }
    float getRadius() const
    {
        return radius;
    }
    void setRadius(float radius)
    {
        this->radius = radius;
    }
    void move(const glm::vec2 &newPosition)
    {
        positionCurrent += newPosition;
        positionOld += newPosition;
    }
    void moveRigid(const glm::vec2 &newPosition)
    {
        positionCurrent += newPosition;
        // positionOld += newPosition;
    }
    void update() {
        if (nodeTransform)
        {
            nodeTransform->setTranslate(positionCurrent.x, positionCurrent.y,
0.0f);
        }
    }
};
```