12.11.2023

Advanced Games Technology Milestone 2

Report

Emin Bora Akdeniz

CITY UNIVERSITY
LONDON

Assets

Audio

- DST-StardustMemory.mp3, Accessed: 03/10/2023 - http://www.nosoapradio.us/
- DST-Roadway.mp3, Accessed: 03/10/2023 - http://www.nosoapradio.us/
- submit_button_click.wav, Accessed: 03/10/2023 – freesound.org
- bounce.wav, Accessed: 03/10/2023 - freesound.org
- crash.vaw, Accessed: 01/12/2023 - freesound.org
- pick_up.vaw, Accessed:01/12/2023 - freesound.org

Models

- AGT_Template and the provided documents from Moodle.
- Player ship, Accessed: 21/11/2023 - https://free3d.com/3d-models/
- Enemy ship, Accessed 21/11/2023 - https://free3d.com/3d-models/
- Pick up/ star model, 01/12/2023 - https://free3d.com/3d-models/

Shaders

- text_2D.glsl, mesh_shader.glsl

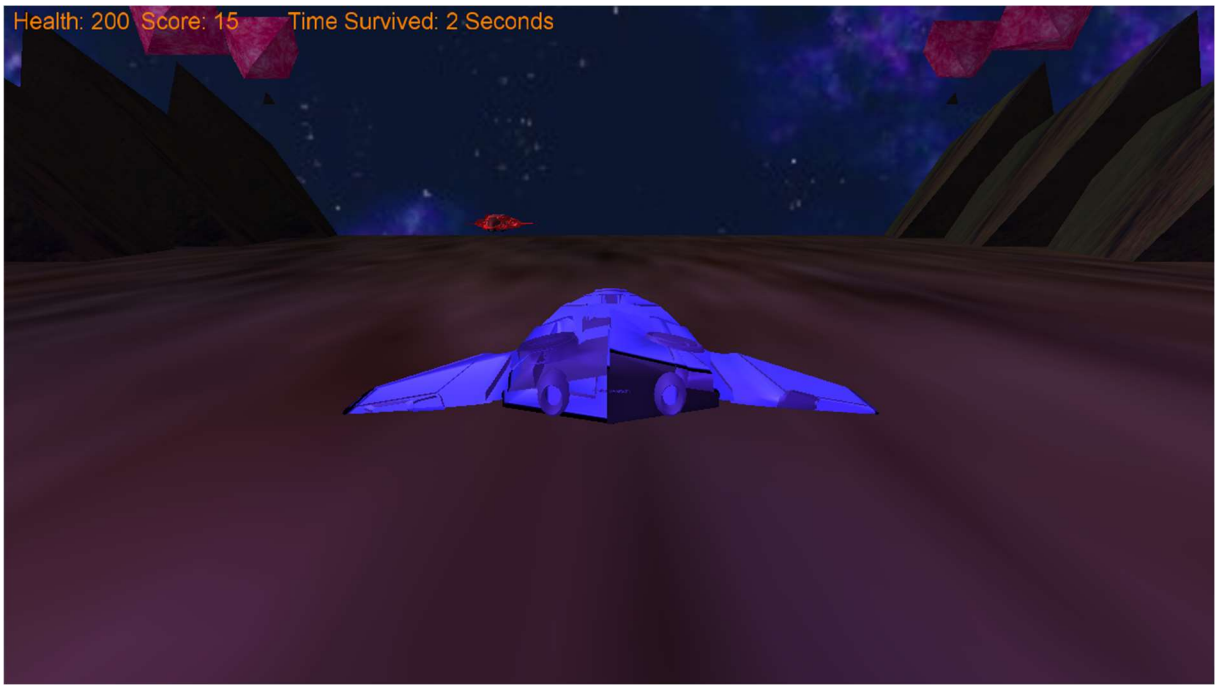
Textures

- Intro Screen Base, Accessed, 10/11/2023 - https://image3.uhdpaper.com/wallpaper-hd/digital-art-retrowave-spaceship-sci-fi-uhdpaper.com-hd-80.jpg
- Skybox, Accessed, 10/11/2023 - http://www.vwall.it/wp-content/plugins/canvasio3dpro/inc/resource/cubeMaps/
- Pink Slime Texture, Accessed, 12/11/2023 - https://www.istockphoto.com/photos/slime-texture

Implementation

Skybox and Terrain

For the cube map, I first implemented an alien planet with a stary night that had no seams, but I was not satisfied with the result since it did not provide me with the atmosphere I was aiming for. So, I decided to make everywhere the top part of the cube map to obtain a “you are in space” energy.



Menu System

To create an intro screen and a gameover screen that well suits my game, I found a retro space background and I edited it on Photoshop to add the necessary details.

In order to load this texture into the game, I first created a class called "quad" that covered the player's screen by forming a mesh the size of the game window. Next, I created a class called "menusystem" that allowed me to specify the file path of the texture to load and make it active by adjusting the transparency of the image. Both of the above-mentioned classes are taken from the lecture material of week 5 and altered to fit my program.

```
#pragma once
#include <engine.h>

class quad;

class menusystem
{
public:
    menusystem(const std::string& path, float max_time, float width, float height);
    ~menusystem();

    void on_update(const engine::timestep& time_step);
    void cross_fade(const engine::timestep& time_step);
    void on_render(engine::ref<engine::shader> shader);
    void activate();
    void stop();

    static engine::ref<menusystem> create(const std::string& path, float max_time, float width, float height);

private:
    bool s_active;

    float m_timer;
    float m_max_time;

    engine::ref<engine::texture_2d> m_texture;
    float m_transparency;
    engine::ref<quad> m_quad;
};
```

I came up with texture using the previously stated "create" function in the "example_layer" class (which is now called “theGame”). I then used a "key_pressed" event in the theGame’s "on_event" method to set it as transparent after the player pressed the space key during the intro screen and as activated once the game was launched.

Primitives

I decided to include an octahedron, pyramid, and sphere as my primitives; I had to map out the forms, establish the vertices, and determine which vertices made up which faces of the shape in order to comprehend how to depict each one.

I made game objects for both tetrahedron and sphere, which was already included in the template. I also set the octahedron, tetrahedron and sphere location, scale, texture, and bounding shape. After that, I used the "on_render" method to submit each item to the textured lighting shader so that it would appear in the game window and have the right amount of illumination. These primitives are used to set up the environment.

```
// Sphere
engine::ref<engine::sphere> sphere_shape = engine::sphere::create(10, 20, 0.5f);
engine::game_object_properties sphere_props;
sphere_props.position = { -5.f, 2.5f, -20.f };
sphere_props.meshes = { sphere_shape->mesh() };
sphere_props.type = 1;
sphere_props.bounding_shape = glm::vec3(0.5f);
sphere_props.restitution = 0.92f;
sphere_props.mass = 0.000001f;
m_ball = engine::game_object::create(sphere_props);
```

```
// Tetrahedron
std::vector<glm::vec3> tetrahedron_vertices;
tetrahedron_vertices.push_back(glm::vec3(0.f, 10.f, 0.f));      //0
tetrahedron_vertices.push_back(glm::vec3(0.f, 0.f, 10.f));      //1
tetrahedron_vertices.push_back(glm::vec3(-10.f, 0.f, -10.f));   //2
tetrahedron_vertices.push_back(glm::vec3(10.f, 0.f, -10.f));    //3

engine::ref<engine::tetrahedron> tetrahedron_shape = engine::tetrahedron::create(tetrahedron_vertices);
engine::game_object_properties tetrahedron_props;
tetrahedron_props.position = { 0.f, 2.5f, -20.f };
tetrahedron_props.meshes = { tetrahedron_shape->mesh() };
std::vector<engine::ref<engine::texture_2d>> tetrahedron_textures = { engine::texture_2d::create("assets/textures/stone.bmp", false) };
tetrahedron_props.textures = tetrahedron_textures;
tetrahedron_props.scale = glm::vec3(0.2);
m_tetrahedron = engine::game_object::create(tetrahedron_props);
```

```
// Octahedron
std::vector<glm::vec3> octahedron_vertices;
octahedron_vertices.push_back(glm::vec3(0.f, 10.f, 0.f));       //0
octahedron_vertices.push_back(glm::vec3(-10.f, 0.f, -10.f));    //1
octahedron_vertices.push_back(glm::vec3(10.f, 0.f, -10.f));     //2
octahedron_vertices.push_back(glm::vec3(10.f, 0.f, 10.f));      //3
octahedron_vertices.push_back(glm::vec3(-10.f, 0.f, 10.f));     //4
octahedron_vertices.push_back(glm::vec3(0.f, -10.f, 0.f));      //5

engine::ref<engine::octahedron> octahedron_shape = engine::octahedron::create(octahedron_vertices);
engine::game_object_properties octahedron_props;
octahedron_props.position = { 5.f, 2.5f, -20.f };
octahedron_props.meshes = { octahedron_shape->mesh() };
std::vector<engine::ref<engine::texture_2d>> octahedron_textures = { engine::texture_2d::create("assets/textures/PurpleSlime.bmp", false) };
octahedron_props.textures = octahedron_textures;
octahedron_props.scale = glm::vec3(0.2);
m_octahedron = engine::game_object::create(octahedron_props);
```

Audio

After I collected all of the tracks and audio samples from the internet, I set an intro scene track that starts playing as soon as the game launches. When the player presses “Space Button”, a button click sound plays, the game launches and the in-game theme starts playing instead of the intro track.

Furthermore, when player connects with a star model; which is the model I used for pick ups; the pick up sound is played. Also, whenever player ship gets in contact with an enemy ship, the crash sound is played.

```
// Initialise audio and play background music
m_audio_manager = engine::audio_manager::instance();
m_audio_manager->init();
m_audio_manager->load_sound("assets/audio/submit_button_click.wav", engine::sound_type::event, "space_click");      // Royalty free sound from freesound.org
m_audio_manager->load_sound("assets/audio/crash.wav", engine::sound_type::event, "crash");                          // Royalty free sound from freesound.org
m_audio_manager->load_sound("assets/audio/bounce.wav", engine::sound_type::spatialised, "bounce");                  // Royalty free sound from freesound.org
m_audio_manager->load_sound("assets/audio/pick_up.wav", engine::sound_type::event, "pickup");                       // Royalty free sound from freesound.org
m_audio_manager->load_sound("assets/audio/DST-Roadway.mp3", engine::sound_type::track, "theme");                    // Royalty free music from http://www.nosoapradio.us/
m_audio_manager->load_sound("assets/audio/DST-StardustMemory.mp3", engine::sound_type::track, "intro");             // Royalty free music from http://www.nosoapradio.us/
m_audio_manager->play("intro");
```

HUD

For the heads-up display functionality, I added simple health and score and time survived using text shader, Health decreases on collisions with enemy ships, the score increases with the “tick” variable defined in code, and time survived increases as the ship stays alive.

Camera Motion Technique

I had to make a camera motion to match my infinite runner-style game, which tracked the player ship continuously at a set distance. The "update_camera" method in the player class is where I build up a view matrix that accepts the camera's coordinates, which are set to be slightly above and behind the player's coordinates, as well as the camera's fixed distance in front of the player's coordinates. Because the camera is using the player's coordinates as parameters, this allowed the camera to be synchronized with every movement made by the player.

```
void player::update_camera(engine::perspective_camera& camera)
{
    //set up the camera boom and the camera position
    float A = 1.5f;
    float B = 4.f;
    float C = 8.f;

    glm::vec3 cam_pos = m_object->position() - glm::normalize(m_object->forward()) * B;
    cam_pos.y += A;

    glm::vec3 cam_look_at = m_object->position() + glm::normalize(m_object->forward()) * C;
    cam_look_at.y = 0.f;
    camera.set_view_matrix(cam_pos, cam_look_at);
}
```

Mesh-based Objects

The mesh-based objects that are used in the game are as follows:

- Player Ship
- Enemy Ship
- Star

There are multiple instances of enemy ship which is the red one, and multiple pick ups which are the stars. There is only one blue ship and it belongs to the player.

Lighting

At first, I implemented a spotlight that followed the player from above and appeared as an orb of light. To supply the scene with a enough quantity of light so that shade may be seen on the appropriate sides of objects within the scene, I developed a directional light using the engine's "DirectionalLight," which is pre-set and modified ambient and diffuse intensity values. To add some more details while satisfying the criteria for the course, point lights are added to some coordinates that can be seen in the following code snippets. To ensure that the light was uniform throughout the scene, I then sent the directional light to the mesh material, animated mesh, and mesh lighting shaders.

```
// Directional light
m_directionalLight.Color = glm::vec3(1.0f, 1.0f, 1.0f);
m_directionalLight.AmbientIntensity = 0.25f;
m_directionalLight.DiffuseIntensity = 0.6f;
m_directionalLight.Direction = glm::normalize(glm::vec3(1.0f, -1.0f, 0.0f));

// Spot light
m_spotLight.Color = glm::vec3(7.0f, 3.0f, 10.0f);
m_spotLight.AmbientIntensity = 1.f;
m_spotLight.DiffuseIntensity = 0.6f;
m_spotLight.Position = glm::vec3(0.f, 5.f, 0.f);
m_spotLight.Direction = glm::normalize(glm::vec3(0.0f, -1.0f, -1.0f));
m_spotLight.Cutoff = 0.5f;
m_spotLight.Attenuation.Constant = 1.0f;
m_spotLight.Attenuation.Linear = 0.1f;
m_spotLight.Attenuation.Exp = 0.01f;

// Point Light
m_pointLight.Color = glm::vec3(1.0f, 1.0f, 1.0f);
m_pointLight.AmbientIntensity = 0.2f;
m_pointLight.DiffuseIntensity = 0.6f;
m_pointLight.Position = glm::vec3(0.f, 1.f, 0.f);
```

After having light sources initialized in the constructor, I made the spotlight follow the player with a simple function that will be called in the “on_update” part of the code.

```
void theGame::updateSpotlightPosition()
{
    // Update the position of the spotlight to follow the ship
    m_spotLight.Position = m_ship->position() + glm::vec3(0.0f, 10.0f, 20.0f);
}
```

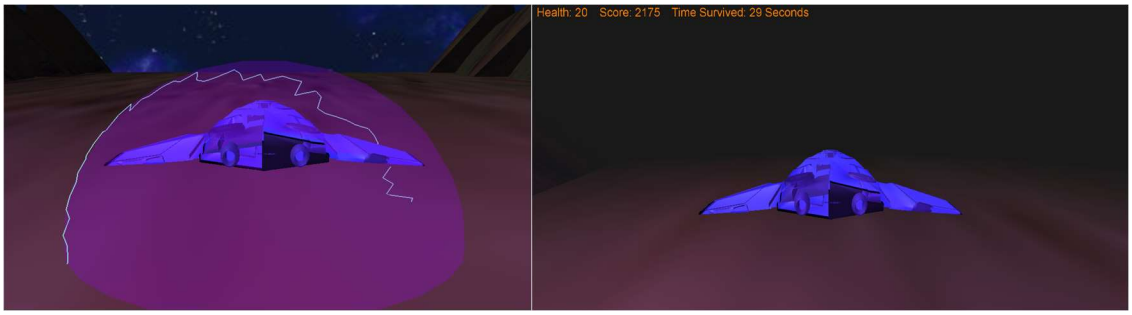
Special Effects

For the special effects, I added three of them. Namely; an alpha-sphere, cross_fade, that will be acting like a shield and fog.

To begin with, the alpha-sphere is used as a sphere that will be following the player ship. It is set to be off as default. Whenever a player ship collides with a pickup, this sphere get activated and when the time on it runs out, it disappears.

Following that, the cross-fade is implemented to give the player the feeling of taking damage. Whenever the player ship collides with an enemy ship, the cross-fade comes into play rendering a half-transparent red screen.

And last of the special effects, fog. The player ship begins with a health pool of 200. Whenever player health falls below 100. The fog is displayed to cut vision and make surviving harder for the player.



None of the above-mentioned effects rely on any external files except for cross-fade, since it needed a texture to display when triggered. The majority of the effects are based on manipulating the shaders.

Artificial intelligence (AI)

AI that is used in the game is implemented by the modification of the “enemy” class that is provided in the lab solutions.

```
void enemy::on_update(const engine::timestep& time_step, const glm::vec3& player_position)
{
    float distance_to_player = glm::distance(m_object->position(), player_position);

    // check which state is the enemy in, then execute the matching behaviour
    if (m_state == state::idle)
    {
        patrol(time_step);

        // check whether the condition has been met to switch to the on_guard state
        if (distance_to_player < m_detection_radius)
            m_state = state::on_guard;
    }
    else if (m_state == state::on_guard)
    {
        face_player(time_step, player_position);

        // check whether the condition has been met to switch back to idle state
        if (distance_to_player > m_detection_radius)
            m_state = state::idle;
        // check whether the condition has been met to switch to the chasing state
        else if (distance_to_player < m_trigger_radius)
            m_state = state::chasing;
    }
    else if (m_state == state::chasing)
    {
        m_speed = m_speed * 1.3f;
        chase_player(time_step, player_position);

        // check whether the condition has been met to switch back to idle state
        if (distance_to_player > m_detection_radius)
            m_state = state::idle;

        else if (distance_to_player < 8.f)
        {
            m_state = state::crashed;
        }
    }
    // check whether the condition has been met and set the flag variable for rendering the model
    else if (m_state == state::crashed)
    {
        s_render = false;
    }
```

The enemy class revolves around a simple state of machine logic.

The player enters the detection radius Player enters the aggro radius

IDLE -> ON GUARD -> CHASING

Connects with player / passes player

-> CRASHED

Gameplay Elements

For the gameplay elements, I decided to take a different approach to make things as interesting as possible for a simple game. If the player connects with a star, which is the model for a pickup, he/she will be granted a lighting shield.

There are multiple uses of a simple shield. These elements are touched on earlier in the document but not explored. First of all, if the player gets the lightning shield, he/she is granted the ability to bump into enemy ships without taking any damage. To make this more encouraging, the player will be granted extra points whenever he/she destroys an enemy ship with a shield on.

```
if (m_enemy_box_1.collision(m_player_box))
{
    if (!m_shield_sphere_pickup->get_s_active())
    {
        s_invincible = false;
        if (!s_invincible)
        {

            //m_player.object()->set_position(pos);
            health -= 10;
            m_cross_fade->activate();
            m_audio_manager->play("crash");
            enemy::setState_crashed(m_enemy_1);
        }
    }
    else
    {
        score += 1000;
    }
```

The second benefit of the shield is vision. Even though the player is granted a clear vision, this is not the case when he/she falls below %50 health. Since it is really hard to survive with little to no vision, there should be a counterplay. Whenever a player collects a shield pick up, even if he/she is below %50 health, they will be granted with vision for the duration of the shield.

```
if (m_shield_pickup_box_0.collision(m_player_box))
{
    m_fog = false;
    m_score += 1000;
    m_shield_sphere_pickup->activate(1.f, m_player.getPosition());
    s_invincible = true;
    m_audio_manager->play("pickup");
}
```

And the most boring one out of all, there is a timer that counts the number of seconds they stay alive.

If player dies, they get a red screen followed up by a game over screen, if they manage to survive the whole level, they will be seeing a game over screen directly.