УНИВЕРЗИТЕТ „СВ. КИРИЛ И МЕТОДИЈ“ – СКОПЈЕ

ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И
КОМПЈУТЕРСКО ИНЖЕНЕРСТВО – СКОПЈЕ

НАСОКА: КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

СЕМИНАРСКА РАБОТА

ПРЕДМЕТ:

Компјутерска графика

ТЕМА:

Super Pang Game

ментор:

м-р Бобан Јоксимоски

кандидат:

Мартин Трајковски 195063

Ноември 2021

Содржина

1.Вовед

Оваа игра може да ја играат максимум два плеери кои се контролира со копчињата: UP, DOWN, LEFT, RIGHT (за поместување на првиот карактер), SPACE, X za за пукање и A, S, D, W (за поместување на вториот карактер), K, L за пукање, и исто така ако сакате да играте екран играта овозможува и опција за тоа при клик на копчето F11 ако не се сетира опцијата дека ке играат два плеери играта секако започнува со еден плеер, за да поминете еден левел треба да ги испукате сите топчиња кои ги содржи левелот, кога ке се заврши еден левел автоматски го лоадира наредниот се додека има левели во играта, тоа значи дека сте ги поминале сите левели и ви дава слика дека сте ја победиле играта.

2.Објаснување на главните компоненти

Играта се состои од коцки кои се рендерираат преку класата Sprite3DRenderer исто така има и сфери кои се рендерираат преку класата SphereRenderer, повеќето елементи се коцки, а топчињата и шестоаголниците се сфери исто така се користат текстури за да се претстават елементите во погледот.

1. ResourceManager класа

   Преку оваа класа се лоадираат сите потребни шејдери и текстури кои ни се потребни за оваа игра и се чуваат во мапи со методот GetShader се зема потребниот Shader но секако прво треба да се лоадира тој Shader преку методот LoadShader за да може да се искористи GetShader, истото важи и за GetTexture и LoadTexture.

```
#ifndef RESOURCE_MANAGER_H
#define RESOURCE_MANAGER_H

#include <map>
#include <string>

#include <glad/glad.h>

#include "Texture.h"
#include "Shader.h"

class ResourceManager
{
public:
    // resource storage
    static std::map<std::string, Shader> Shaders;
    static std::map<std::string, Texture2D> Textures;

    static Shader    LoadShader(const char *vShaderFile, const char *fShaderFile, const char *gShaderFile, std::string name);
    static Shader    GetShader(std::string name);
    static Texture2D LoadTexture(const char *file, bool alpha, std::string name);
    static Texture2D GetTexture(std::string name);
    static void      Clear();
private:
    ResourceManager() { }
    static Shader    loadShaderFromFile(const char *vShaderFile, const char *fShaderFile, const char *gShaderFile = nullptr);
    static Texture2D loadTextureFromFile(const char *file, bool alpha);
};

#endif
```

Слика 1: ResourceManager prototype

2. GameObject класа

Сите објекти во играта наследуваат од оваа класа која ги содржи сите примарни атрибути на еден објект подолу на сликата 1 можете да ги видите кои се тие атрибути, преку Texture атрибутот се лоадира текстурата која ке ја

```
#pragma once
#include <glm/glm.hpp>
#include <unordered_map>

#include "Texture.h"
#include "SpriteRenderer.h"
#include "Sprite3DRenderer.h"
#include "Utility.h"

class GameObject
{
protected:
    std::unordered_map<std::string, float> frame;
public:
    glm::vec3 Position, Size, Velocity, Color;
    float Rotation;
    bool IsSolid;
    bool Destroyed;
    Texture2D Texture;

    GameObject();
    virtual ~GameObject() = default;
    GameObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f));
    virtual Collision checkCollision(GameObject&);
    virtual void Draw(Sprite3DRenderer& renderer);
    virtual glm::vec3& Move(float dt, unsigned int window_width, unsigned int window_height);
    virtual void ProcessInput(float dt, unsigned int window_width, unsigned int window_height);
    virtual void Reset();
};
```

Слика 2: GameObject Prototype

користиме за овој елемент, isSolid покажува дали објектот е цврст или не е и може да биде Destroyed, секако ако објектот е цврст тогаш во овој објект неможе да се случи Destroyed = true, а другите атрибути се основни кои ја даваат позицијата, големината на објектот итн, исто така тука е важно frame, на пример ако сакаме да ја менуваме текстурата на карактерот по неколку frame-ови тогаш мора да имаме некоја променлива frame која кога ке достигне некоја вредност ние само ке ја поставиме текстурата на следната текстура, за таа цел тука користиме мапа бидејќи може да ни се потребни повеќе такви променливи, поважни методи кои ги содржи оваа класа се Draw за исцртување на коцката или сферата, checkCollision за проверка дали се случила колизија со некој друг објект во сцената и Move кој ја ажурира позицијата на елементот и го поместува.

3. Коцки

Коцките се претставени со класата Sprite3DRenderer која се користи за исцртување на коцките кој се наоѓаат во соодветниот левел и таа може да ја видите на слика 2 и имплементираните функции на слика 3 а функцијата initRenderData може да ја погледнете на крајот од овој документ. Во конструкторот на оваа класа се зема како аргумент Shader-от кој ке го користиме за исцртување на коцките и точките на коцката, а во DrawSprite елементот може да се поместува, ротира, и скалира, и на крај се bind-ова текстурата која ке ја користиме за исцртување на објектот, се bind-ова Vertex

Array Bufferot кој го генерираме во initRenderData и се цртаат триаголниците со glDrawArrays.

```
#pragma once
#include "Shader.h"
#include "Texture.h"

class Sprite3DRenderer
{
public:
    Sprite3DRenderer(Shader& shader);
    ~Sprite3DRenderer();

    void DrawSprite(Texture2D& texture, glm::vec3 position, glm::vec3 size, float rotate, glm::vec3 color);

private:
    Shader shader;
    unsigned int QuadVAO;

    void initRenderData();
};
```

Слика 3: Sprite3DRenderer Prototype

```
#include "Sprite3DRenderer.h"

Sprite3DRenderer::Sprite3DRenderer(Shader& shader) {
    this->shader = shader;
    this->initRenderData();
}

Sprite3DRenderer::~Sprite3DRenderer() {
    glDeleteVertexArrays(1, &this->QuadVAO);
}

void Sprite3DRenderer::DrawSprite(Texture2D& texture, glm::vec3 position, glm::vec3 size = glm::vec3(10.0f), float rotate = 0.0f, glm::vec3 color = glm::vec3(1.0f)) {
    this->shader.Use();
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, position);

    model = glm::translate(model, glm::vec3(size.x * 0.5f, size.y * 0.5f, 0.0f));
    model = glm::rotate(model, glm::radians(rotate), glm::vec3(0.0f, 0.0f, 1.0f));
    model = glm::translate(model, glm::vec3(size.x * -0.5f, size.y * -0.5f, 0.0f));

    model = glm::scale(model, size);

    this->shader.SetMatrix4("model", model);
    this->shader.SetVector3f("sprite3DColor", color);

    glActiveTexture(GL_TEXTURE0);
    texture.Bind();

    glBindVertexArray(this->QuadVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
}
```

Слика 4: Sprite3DRenderer Methods

4. Сфери

За исцртување на сфери се користи класата SphereRenderer, при што треба да специфицирате колку сектори и стакови ќе има сферата за исцртување на коцка 18 стакови и 36 сектори, а за исцртување на шестоаголник 6 стакови и 12 сектори. Функцијата DrawSpehre е скоро иста со функцијата DrawSprite со разликата тоа што тука исцртуваме точки во TRIANGLE_STRIP мод. Методот initRenderData може да го погледнете на крајот од овој документ.

```
#pragma once
#include <vector>
#include <glm/glm.hpp>

#include "Shader.h"
#include "Texture.h"

class SphereRenderer
{
public:
    SphereRenderer(Shader& shader, int squares = 18, int layers = 36);
    ~SphereRenderer();

    void DrawSphere(Texture2D& texture, glm::vec3 position, glm::vec3 size, float rotate, glm::vec3 color);

private:
    Shader shader;
    unsigned int QuadVAO;
    int squares;
    int layers;

    void initRenderData();
};
```

Слика 5: SphereRenderer Prototype

```cpp
#include "SphereRenderer.h"

SphereRenderer::SphereRenderer(Shader& shader, int squares, int layers)
    :squares(squares), layers(layers), shader(shader)
{
    this->initRenderData();
}

SphereRenderer::~SphereRenderer() {
    glDeleteVertexArrays(1, &this->QuadVAO);
}

void SphereRenderer::DrawSphere(Texture2D& texture, glm::vec3 position, glm::vec3 size = glm::vec3(10.0f), float rotate = 0.0f, glm::vec3 color = glm::vec3(1.0f)) {
    this->shader.Use();
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, position);
    model = glm::rotate(model, glm::radians(rotate), glm::vec3(0.0f, 0.0f, 1.0f));
    model = glm::scale(model, size);

    this->shader.SetMatrix4("model", model);
    this->shader.SetVector3f("sprite3DColor", color);

    glActiveTexture(GL_TEXTURE0);
    texture.Bind();

    glBindVertexArray(this->QuadVAO);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 6 * squares * layers);
    glBindVertexArray(0);
}
```

Слика 6: SphereRenderer methods

5. Напагачи

За напагачите има посебна абстрактна класа AttackerObject која исто така наследува од GameObject но го override-ова методот Draw за да ги исцртува Сферите бидејќи сите напаѓачи се или топка или шестоаголник а тие ги цртаме преку класата SphereRenderer и тука се чуваат две дополнителни атрибути и тоа Radius и дали Напагачот е во состојба на пукање. Од оваа класа најважен метод е checkCollision кој проверува дали одреден објект од играта кој е коцка има колизија со елементот напагач.

```cpp
#pragma once
#include <glm/glm.hpp>

#include "GameObject.h"
#include "SphereRenderer.h"

class AttackerObject: public GameObject
{
public:
    float Radius;
    bool pop;

    AttackerObject();
    AttackerObject(glm::vec3 position, float radius, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f));
    AttackerObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f));

    void Draw(SphereRenderer& renderer);
    Collision checkCollision(GameObject&);
    virtual glm::vec3& Move(float dt, unsigned int window_width, unsigned int window_height) = 0;
    void Reset(glm::vec3 position, glm::vec3 velocity);
    virtual void Pop(float dt) = 0;
};
```

Слика 7: AttackerObject Prototype

```cpp
#include "AttackerObject.h"
#include "ResourceManager.h"

AttackerObject::AttackerObject(glm::vec3 position, float radius, Texture2D texture, glm::vec3 color, glm::vec3 velocity)
    :GameObject(position, glm::vec3(2 * radius, 2 * radius, 1.0f), texture, color, velocity), Radius(radius), pop(false) {}

AttackerObject::AttackerObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color, glm::vec3 velocity)
    :AttackerObject(position, size.x, texture, color, velocity) {}

AttackerObject::AttackerObject()
    :GameObject(), Radius(12.5f), pop(false) {}

void AttackerObject::Draw(SphereRenderer& renderer) {
    renderer.DrawSphere(this->Texture, this->Position, glm::vec3(this->Radius, this->Radius, 1.0f), this->Rotation, this->Color);
}

Collision AttackerObject::checkCollision(GameObject& obj) {
    glm::vec2 center(this->Position);
    // calculate AABB info (center, half-extents)
    glm::vec2 rectangle_half_extents(obj.Size.x / 2.0f, obj.Size.y / 2.0f);
    //glm::vec2 rectangle_half_extents(0.0f);
    glm::vec2 rectangle_center(obj.Position.x, obj.Position.y);

    glm::vec2 difference = center - rectangle_center;
    glm::vec2 clamped = glm::clamp(difference, -rectangle_half_extents, rectangle_half_extents);
    // add clamped value to AABB_center and we get the value of box closest to circle
    glm::vec2 closest = rectangle_center + clamped;

    difference = closest - center;

    if (glm::length(difference) <= this->Radius) {
        return Collision(true, VectorDirection(difference), difference);
    }
    else {
        return Collision();
    }
}


void AttackerObject::Reset(glm::vec3 position, glm::vec3 velocity) {
    this->Position = position;
    this->Velocity = velocity;
}
```

Слика 8: AttackerObject methods

6. BallObject и HexagonObject

Овие два објекти наследуваат од класата AttackerObject и двата методи ги override-оваат методите Move и Pop, во BallObject има атрибут Gravity кој служи за гравитација за да може топката да се движи надолу, а HexagonObject на неколку frame-ови ротира за да изгледа како да се врти објектот, методот Pop е исти и кај двата објекти и служи за откако топката ке биде погодена од некое оружје (WeaponObject) да му се менува текстурата за да изгледа како тој да експлодира.

```cpp
#pragma once
#include <glm/glm.hpp>

#include "AttackerObject.h"

class BallObject: public AttackerObject
{
public:
    glm::vec3 Gravity;

    BallObject();
    BallObject(glm::vec3 position, float radius, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f), glm::vec3 gravity = glm::vec3(0.0f, 1.4f, 0.0f));

    glm::vec3& Move(float dt, unsigned int window_width, unsigned int window_height);
    void Reset(glm::vec3 position, glm::vec3 velocity);
    void Pop(float dt);
};
```

Слика 9: BallObject Prototype

```cpp
#pragma once
#include "AttackerObject.h"

class HexagonObject : public AttackerObject
{
public:
    HexagonObject();
    HexagonObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f, 1.0f, 0.0f), glm::vec3 velocity = glm::vec3(0.0f));

    glm::vec3& Move(float dt, unsigned int window_width, unsigned int window_height);
    void Reset(glm::vec3 position, glm::vec3 velocity);
    void Pop(float dt);
};
```

Слика 10: HexagonObject Prototype

```cpp
#include "BallObject.h"
#include "ResourceManager.h"

BallObject::BallObject(glm::vec3 position, float radius, Texture2D texture, glm::vec3 color, glm::vec3 velocity, glm::vec3 gravity)
    :AttackerObject(position, radius, texture, color, velocity), Gravity(gravity)
{
    this->frame["Pop"] = 0.0f;
}

BallObject::BallObject()
    :AttackerObject(), Gravity(0.0f, 1.4f, 0.0f)
{
    this->frame["Pop"] = 0.0f;
}

unsigned int PopTextureBall = 1;

void BallObject::Pop(float dt) {
    if (this->pop) {
        if (PopTextureBall < 5) {
            if (frameCount(dt, this->frame["Pop"], 0.01f)) {
                this->Texture = ResourceManager::GetTexture("ball-pop-" + std::to_string(PopTextureBall));
                ++PopTextureBall;
            }
        }
        else {
            this->Destroyed = true;
            this->pop = false;
            PopTextureBall = 1;
        }
    }
}

glm::vec3& BallObject::Move(float dt, unsigned int windowWidth, unsigned int windowHeight) {

    this->Position += this->Velocity * dt;
    this->Velocity += this->Gravity;

    if (this->Position.x + this->Size.x / 2.0f > windowWidth || this->Position.x < this->Size.x / 2.0f) {
        this->Velocity.x = -this->Velocity.x;
    }
    if (this->Position.y <= this->Size.y / 2.0f) {
        this->Velocity.y = -this->Velocity.y;
        this->Position.y = this->Size.y / 2.0f;
    }

    if (this->Position.y - this->Size.y / 2.0f >= windowHeight - this->Size.y) {
        this->Velocity.y = -this->Velocity.y;
        this->Position.y = windowHeight - this->Size.y / 2.0f;
        this->Gravity = glm::vec3(0.0f, 1.6f, 0.0f);
    }
    return this->Position;
}

void BallObject::Reset(glm::vec3 position, glm::vec3 velocity) {
    this->Position = position;
    this->Velocity = velocity;
}
```

Слика 12: BallObject methods

```cpp
#pragma once
#include "HexagonObject.h"
#include "ResourceManager.h"

HexagonObject::HexagonObject()
    :AttackerObject() {
    this->frame["Rotate"] = this->frame["Pop"] = 0.0f;
}

HexagonObject::HexagonObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color, glm::vec3 velocity)
    : AttackerObject(position, size.x / 2.0f, texture, color, velocity)
{
    this->frame["Rotate"] = this->frame["Pop"] = 0.0f;
}

unsigned int HexagonTexture = 1;

glm::vec3& HexagonObject::Move(float dt, unsigned int windowWidth, unsigned int windowHeight) {
    if (frameCount(dt, this->frame["Rotate"], 0.1f)) {
        if (this->Rotation <= 360.0f) {
            this->Rotation += 10.0f;
        }
        else {
            this->Rotation = 0.0f;
        }
    }

    this->Position += this->Velocity * dt;

    if (this->Position.x + this->Size.x / 2.0f > windowWidth || this->Position.x < this->Size.x / 2.0f) {
        this->Velocity.x = -this->Velocity.x;
    }
    if (this->Position.y <= this->Size.y / 2.0f) {
        this->Velocity.y = -this->Velocity.y;
        this->Position.y = this->Size.y / 2.0f;
    }

    if (this->Position.y - this->Size.y / 2.0f >= windowHeight - this->Size.y) {
        this->Velocity.y = -this->Velocity.y;
        this->Position.y = windowHeight - this->Size.y / 2.0f;
    }
    return this->Position;
}

void HexagonObject::Reset(glm::vec3 position, glm::vec3 velocity) {
    this->Position = position;
    this->Velocity = velocity;
}

unsigned int PopTextureHexagon = 1;

void HexagonObject::Pop(float dt) {
    if (this->pop) {
        if (PopTextureHexagon < 5) {
            if (frameCount(dt, this->frame["Pop"], 0.005f)) {
                this->Texture = ResourceManager::GetTexture("ball-pop-" + std::to_string(PopTextureHexagon));
                ++PopTextureHexagon;
            }
        }
        else {
            this->Destroyed = true;
            this->pop = false;
            PopTextureHexagon = 1;
        }
    }
}
```

Слика 11: HexagonObject methods

7. PlayerObject класа

Оваа е класа која се користи за сите карактери во играта, има методи

```
class PlayerMovement {
public:
    unsigned short int UP, DOWN, LEFT, RIGHT, SHOOT1, SHOOT2;
    PlayerMovement(unsigned short int UP = GLFW_KEY_UP,
                   unsigned short int DOWN = GLFW_KEY_DOWN,
                   unsigned short int LEFT = GLFW_KEY_LEFT,
                   unsigned short int RIGHT = GLFW_KEY_RIGHT,
                   unsigned short int SHOOT1 = GLFW_KEY_SPACE,
                   unsigned short int SHOOT2 = GLFW_KEY_X)
        :UP(UP), DOWN(DOWN), LEFT(LEFT), RIGHT(RIGHT), SHOOT1(SHOOT1), SHOOT2(SHOOT2) {}
};

class PlayerObject: public GameObject
{
public:
    std::unordered_map<std::string, bool> CollisionWith;
    std::list<WeaponObject*> Weapons;
    unsigned int Lives;
    bool Alive;

    PlayerObject();
    ~PlayerObject();
    PlayerObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f), int playerNumber = 1);
    void Reset(glm::vec3 position, glm::vec3 velocity);
    void ProcessInput(float dt, unsigned int window_width, unsigned int window_height);
    void Shoot();

    bool isAlive();
    bool PlayerAttackerCollision(GameObject&);
    void ResetWeapons();
    glm::vec3& Move(float dt, unsigned int windowWidth, unsigned int windowHeight);
private:
    PlayerMovement playerMovement;
    unsigned short int UpTexture, WalkTexture;
};
```

Слика 13: PlayerObject Prototype

кои му овозможуваат на играчот да пука, да го поместува карактерот итн. Player објектот во себе содржи Weapons кој играчот може да ги користи и кој може да се менуваат ако е земен PowerUpObject кој ги менува соодветните Weapons. processInput само проверува дали одредено копче од тастатурата е кликнато ако е да направи нешто, во зависност од тоа кое копче е притиснато и важи само за PlayerObject. Има и класа PlayerMovement која служи за да му каже на PlayerObject кои копчина ќе ѓи користиме за поместување и правење нешто со карактерот.

8. Оружје

Имаме класа WeaponObject од која наследуваат две класи а тоа се ArrowObject и PowerArrowObject, Using атрибутот служи за дали карактерот кој може да го користи ова оружје е во состојба Using за да неможе да го искористи повеќе од еднаш се дури го користи еднаш. Понатаму ArrowObject не додава некоја дополнителна функционалност, а во PowerArrowObject имаме Stuck аттрибут. Stuck се користи кога оружјето ќе направи колизија со некој BlockObject или стигне до крај на екранот тогаш овој објект мора да се прикаже уште неколку frame-ови и тука се сетира Stuck откако ке поминат одреден број frame-ови Stuck доаѓа на false и ова оружје може да се користи пак.

```cpp
#pragma once
#include <glm/glm.hpp>

#include "GameObject.h"

class WeaponObject: public GameObject
{
public:
    bool Using;

    WeaponObject();
    WeaponObject(GameObject& Player, Texture2D texture, glm::vec3 velocity);
    WeaponObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f));
    void UseWeapon();
    virtual glm::vec3& Move(float dt, unsigned int windowWidth, unsigned int windowHeight) = 0;
    virtual void Reset(GameObject* Player);
};

class ArrowObject: public WeaponObject
{
public:
    ArrowObject();
    ArrowObject(GameObject& Player, glm::vec3 velocity);
    ArrowObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f));
    glm::vec3& Move(float dt, unsigned int windowWidth, unsigned int windowHeight);
};

class PowerArrowObject : public WeaponObject {
public:
    bool Stuck;

    PowerArrowObject();
    PowerArrowObject(GameObject& Player, glm::vec3 velocity);
    PowerArrowObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f));
    glm::vec3& Move(float dt, unsigned int windowWidth, unsigned int windowHeight);
    void Reset(GameObject* Player);
};
```

Слика 14: WeaponObjects Prototypes

9. PowerUp

За PowerUpObject се чува тип за кој powerUpObject се работи и во зависност од овој тип методот Activate прави нешто на пример го менува оружјето на карактерот.

```cpp
#pragma once
#include "GameObject.h"
#include "PlayerObject.h"
#include "WeaponObject.h"

class PowerUpObject : public GameObject
{
public:
    std::string Type;

    PowerUpObject();
    PowerUpObject(glm::vec3 position, glm::vec3 size, Texture2D texture, glm::vec3 color = glm::vec3(1.0f), glm::vec3 velocity = glm::vec3(0.0f));
    PowerUpObject(GameObject& spawnedFrom, Texture2D texture, std::string type);
    glm::vec3& Move(float dt, unsigned int windowWidth, unsigned int windowHeight);
    void Activate(PlayerObject* Player);
};
```

Слика 15: PowerUpObject Prototype

10. Други објекти во играта

Понатаму има други објекти во играта како што се BlockObject кои нема дополнителни методи само се користи како Block за именување и LadderObject кои исто така не нуди никаква дополнителна функционалност и се користи за исцртување на скали во играта.

11. Game класа

На крај стигнавме до класата Game која ја содржи целата логика на играта и различни методи, Init се користи за иницијализација на сите потребни ресурси како shader-и текстури, плеери итн. LoadFiles служи за лоадирање на фајлови како слики, левели, менито итн. ProcessInput служи за event кога ќе се кликне некое копче, Render за исцртување на сите објекти во играта како што се, текстовите, коцките, сферите итн. Reset е за ресетирање на играта, ResetPlayers е метод кој ја ресетира позицијата на сите плеери во играта, shouldGeneratPowerUp самото име кажува дали да генерира PowerUpObject, една од поважните е DoCollisions, во овој метод се проверуваат колизии дали се случула колизија со Attacker и Player, дали со Attacker и Weapon итн, понатаму ке го објасниме методот DoCollisions поопширно.

```
#ifndef GAME_H
#define GAME_H

#include <vector>
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include "ft2build.h"
#include FT_FREETYPE_H

#include <GameLevel.h>
#include <GameMenu.h>

enum GameState {
    GAME_PAUSE,
    GAME_ACTIVE,
    GAME_MENU,
    GAME_WIN,
    GAME_OVER,
    GAME_SLEEP
};

class Game
{
public:
    GameMenu* Menu;
    GameState State;
    static bool Keys[1024];
    static bool KeysProcessed[1024];
    unsigned int Width, Height;
    std::vector<GameLevel*> Levels;
    unsigned int Level;

    Game(GLFWwindow* currentWindow, unsigned int width, unsigned int height);
    ~Game();

    // initialize game state (load all shaders/textures/levels)

    void Init();
    void LoadFiles();
    void ProcessInput(float dt);
    void Update(float dt);
    void Render();
    void DoCollisions();
    void Reset();
    void ResetPlayers(bool withPlayerLives = false);
private:
    GLFWwindow* currentWindow;
    void ShouldGeneratePowerUp(GameObject& object);
};

#endif
```

Слика 16: Game Prototype

12. GameLevel класа

Во оваа класа се чуваат сите објекти кои ги има во левелот и тоа се Напаѓачите како низа од AttackerObject, сите останати објекти како BlockObject и LadderObject, и во посебена низа се чуваат PowerUpObject за да може полесно да се користат сите тие објекти понатаму во кодот, Load методот се користи за да се прочита левелот од file и потоа Load во себе го користи Init методот преку кој прочитаните податоци се сместуваат во соодветните низи на објекти во зависност од тоа за каков објект се работи, дали е напагач или друг објект како Block или Ladder, isCompleted проверува дали сите напаѓачи од низата Attackers се уништени за да може да одиме на наредниот левел, а методот Reset се користи за да го ресетираме левелот каков што бил во почетната состојба.

```
#include <vector>

#include "Sprite3DRenderer.h"
#include "BallObject.h"
#include "PowerUpObject.h"
#include "BlockObject.h"
#include "HexagonObject.h"
#include "LadderObject.h"


class Data {
public:
    std::string type;
    glm::vec3 pos;
    glm::vec3 size;
    bool solid;
    Data(std::string type, glm::vec3 pos, glm::vec3 size, bool solid)
        :type(type), pos(pos), size(size), solid(solid) {}
};

class GameLevel
{
public:
    char* file;
    std::vector<GameObject*> Objects;
    std::vector<AttackerObject*> Attackers;
    std::vector<PowerUpObject*> PowerUps;

    GameLevel(const char* file);
    ~GameLevel();

    void Load(unsigned int windowWidth, unsigned int windowHeight);
    void Draw(Sprite3DRenderer& renderer);
    bool isCompleted();
    void Reset();
public:
    void Init(std::vector<Data> Data, unsigned int levelWidth, unsigned int levelHeight);
};
```

Слика 17: GameLevel Prototype

- Левели

  За секој левел најпрво се чува ширината и висината на екранот на кој сакаме да ги мапираме објектите во левелот на соодветниот екран понатаму во наредните редови првата колона е за каков објект се работи, втората и третата за позицијата на објектот и трета и четрвта за големина, така што ако е топка тогаш има само една вредност за радиусот а ако е некој друг објект има две вредности

```
1280 720
BALL 115.0 115.0 10.0
BALL 115.0 115.0 20.0
BLOCK 200.0 200.0 100.0 40.0
BLOCK 880.0 200.0 100.0 40.0
BLOCKPOWERUP 480.0 200.0 100.0 40.0
HEXAGON 90.0 90.0 60.0 60.0
LADDER 1260.0 700.0 40.0 40.0
LADDER 1260.0 660.0 40.0 40.0
LADDER 1260.0 620.0 40.0 40.0
LADDER 1260.0 580.0 40.0 40.0
LADDER 1260.0 540.0 40.0 40.0
LADDER 1260.0 500.0 40.0 40.0
LADDER 1260.0 460.0 40.0 40.0
LADDER 1260.0 420.0 40.0 40.0
BLOCK 1090.0 420.0 300.0 40.0
BLOCK 890.0 420.0 100.0 40.0
```

Слика 18: Level file creation

13. GameMenu класа

Оваа класа се користи за генерирање на менито со соодветните опции преку file, и за исцртување на менито. Selected е атрибут кој кажува која опција од менито е селектирана. Исто како и левелите, менито се чува во file, во кој се чуваат сите опции за менито, findOptionByName се користи за да најдеме соодветна опција од менито преку нејзиното име и да ја земеме нејзината вредност Value атрибутот

```
class GameMenu
{
public:
    std::string Path;
    Option* Selected;

    GameMenu(std::string path);
    ~GameMenu();

    Option* findOptionByName(std::string optionName);
    void Load(unsigned int windowWidth, unsigned int windowHeight);
    void Draw(TextRenderer& Renderer);
private:
    void loadMenuFromFile(unsigned int windowWidth, unsigned int windowHeight, std::vector<Option*>& Options, Option* Parent);
    Option* GameMenu::findOptionByNameR(std::string optionName, std::vector<Option*> Options);
    void DestroyRecursive(std::vector<Option*>& Options);
};
```

Слика 19: GameMenu Prototype

- game.menu file

```
GAME START
SETTINGS
 SETTING 1
 SETTING 2
  PLAYERS:COUNTER(1,2)
  SETTING 2-2
 SETTING 3
  SETTING 3-1
EXIT
```

Слика 20: game.menu file creation

Овој file се креира така што се пишуваат имињата на опциите и ако има одредена вредност после името се додава ":{пример COUNTER(1, 2) или некој друг тип} со празните места се знае на која опција припаѓаат додатните опции.

- Option

Во Option се чуваат податоци за името на опцијата Value ако постои други опции кој се подопции на оваа опција во низа, Selected за кој елемент од низата Options е селектиран и ParentOption за да знаеме на кој Option да се вратиме ако сакаме да одиме назад на претходната опција, има еден статички метод кој само парсира Option од std::string како што е во game.menu file-от. Во зависност од тоа дали постои Value на оваа Опција, може да биде или CounterValue или нешто друго така што

```
class Option {
public:
    Option* ParentOption;
    std::string Name;
    OptionValue* Value;
    glm::vec2 Position;
    float FontSize;
    std::vector<Option*> Options;
    unsigned int Selected;

    Option(std::string Value, glm::vec2 Position = glm::vec2(0.0f), float FontSize = 12.0f, OptionValue* value = nullptr);
    ~Option();

    static Option* ParseOption(glm::vec2 position, std::string strOption);
    void Draw(TextRenderer& Renderer);
};
```

Слика 21: Option Prototype

```cpp
class OptionValue {
public:
    OptionValue() = default;
    virtual void Action() = 0;
    virtual std::string toString() = 0;
};

class OptionWithoutValue: public OptionValue {
    void Action() {
        return;
    }
    virtual std::string toString() {
        return "";
    }
};

 template<class T>
class OptionWithValue : public OptionValue {
protected:
    T value;
public:
    OptionWithValue(int value = 0) :value(value) {}
    virtual void Action() = 0;
    virtual std::string toString() = 0;
    virtual T& getValue() {
        return this->value;
    }
};

class CounterValue : public OptionWithValue<int> {
    unsigned short int from, to;
public:
    CounterValue(int from, int to) : OptionWithValue(from), from(from), to(to) {}
    void Action();
    std::string toString() {
        return std::to_string(this->getValue());
    }
};
```

Слика 22: OptionValues Prototype

OptionValue е генерализирана класа која има метод toString за да можеме да го земеме елементот value за понатаму да го исцртаме во Draw од Option класата.

3. Колизии

```cpp
bool collisionX = this->Position.x + this->Size.x / 2.0f >= obj.Position.x - obj.Size.x / 2.0f &&
                  obj.Position.x + obj.Size.x / 2.0f >= this->Position.x - this->Size.x / 2.0f;
bool collisionY = this->Position.y + this->Size.y / 2.0f >= obj.Position.y - obj.Size.y / 2.0f &&
                  obj.Position.y + obj.Size.y / 2.0f >= this->Position.y - this->Size.y / 2.0f;
bool collision = collisionX && collisionY;

return Collision(collision);
```

Слика 23: AABB collision

```cpp
glm::vec2 center(this->Position);
// calculate AABB info (center, half-extents)
glm::vec2 rectangle_half_extents(obj.Size.x / 2.0f, obj.Size.y / 2.0f);
//glm::vec2 rectangle_half_extents(0.0f);
glm::vec2 rectangle_center(obj.Position.x, obj.Position.y);

glm::vec2 difference = center - rectangle_center;
glm::vec2 clamped = glm::clamp(difference, -rectangle_half_extents, rectangle_half_extents);
// add clamped value to AABB_center and we get the value of box closest to circle
glm::vec2 closest = rectangle_center + clamped;

difference = closest - center;

if (glm::length(difference) <= this->Radius) {
    return Collision(true, VectorDirection(difference), difference);
}
else {
    return Collision();
}
```

Слика 24: AABB Sphere and Cube

За декетција на колизија се користат два различни методи и тоа: Колзија помегу круг и квадрат (слика 23) и колизија помеѓу квадрат и квадрат (слика 24) и овие методи се користат во doCollisions во Game класата за детекција на колизија за сите објекти кои постојат во играта.

```
//COLLISION POWERUPS AND WEAPONS
for (auto& object : this->Levels[this->Level]->PowerUps) {
    for (auto& Player : Players) {
        if (!Player->isAlive()) {
            continue;
        }
        for (auto& Weapon : Player->Weapons) {
            if (!Weapon->Using) {
                continue;
            }
            if (object->checkCollision(*Weapon).collision) {
                object->Destroyed = true;
                Weapon->Reset(Player);
                object->Activate(Player);
                break;
            }
        }
    }
}

//COLLISION POWERUPS AND PLAYER
for (auto& object : this->Levels[this->Level]->PowerUps) {
    for (auto& Player : Players) {
        if (!Player->isAlive()) {
            continue;
        }
        Collision PlayerPowerUpCollision = object->checkCollision(*Player);
        if (PlayerPowerUpCollision.collision) {
            object->Destroyed = true;
            object->Activate(Player);
        }
    }
}
```

Слика 25: Collision PowerUps and Weapons

Прво за колизија помеѓу PowerUp и Weapon, ако има колизија тогаш во зависност на тоа кој Player го користел оружјето се користи Activate методот од PowerUp објектот на соодветниот карактер и исто ако каректерот е на иста позиција како и PowerUp објектот тогаш се користи Activate методот на истиот начин.

```
//COLLISION OBJECTS AND WEAPONS
for (auto& object : this->Levels[this->Level]->Objects) {
    if (object->Destroyed || dynamic_cast<LadderObject*>(object)) {
        continue;
    }
    for (auto& Player : Players) {
        if (!Player->isAlive()) {
            continue;
        }
        for (auto& Weapon : Player->Weapons) {
            if (!Weapon->Using) {
                continue;
            }
            if (object->checkCollision(*Weapon).collision) {
                if (object->IsSolid) {
                    if (dynamic_cast<PowerArrowObject*>(Weapon)) {
                        dynamic_cast<PowerArrowObject*>(Weapon)->Stuck = true;
                        break;
                    }
                    Weapon->Reset(Player);
                    break;
                }
                SoundEngine->play2D("../resources/audio/solid.wav");
                object->Destroyed = true;
                this->ShouldGeneratePowerUp(*object);
                Weapon->Reset(Player);
            }
        }
    }
}
```

Слика 26: Collision Objects and Weapons

За колизија помеѓу останати објекти и оружје ако објектот е цврст тогаш тој неможе да се уништи ако не тогаш се поставува Destryoed = false на тој објект, а ако користиме PowerArrow и објектот е цврст тогаш Stuck на PoweArrow ке биде true.

```
//COLLISION ATTACKERS AND OBJECTS
for (auto& object : this->Levels[this->Level]->Attackers) {
    if (object->Destroyed || object->pop) {
        continue;
    }
    for (auto& obj : this->Levels[this->Level]->Objects) {
        if (obj->Destroyed || dynamic_cast<LadderObject*>(obj))
            continue;

        Collision& collisionAttackerObj = object->checkCollision(*obj);
        if (collisionAttackerObj.collision) {
            if (collisionAttackerObj.direction == LEFT || collisionAttackerObj.direction == RIGHT)
            {
                object->Velocity.x = -object->Velocity.x;
                float penetration = object->Radius - std::abs(collisionAttackerObj.difference.x);
                if (collisionAttackerObj.direction == LEFT)
                    object->Position.x += penetration;
                else
                    object->Position.x -= penetration;
            }
            else {
                object->Velocity.y = -object->Velocity.y;
                float penetration = object->Radius - std::abs(collisionAttackerObj.difference.y);
                if (collisionAttackerObj.direction == UP)
                    object->Position.y -= penetration;
                else
                    object->Position.y += penetration;
            }
        }
    }
}
```

Колизија помеѓу напагач и друг објект без објектот скала, проверува дали има колизија ако има ја зема насоката во која удрил напаѓачот и ја намалува/зголемува позицијата за радиусот-{колку топката влегла внатре во другиот објект}.

Слика 27: Collision Attackers and Objects

За колизија помеѓу напагач и оружје ако топката е погодена се генерира нова топка со половина од радиусот на претходната топка и се додава на низата од Напаѓачи.

```
//COLLISION ATTACKERS AND WEAPONS
for (int i = 0; i < this->Levels[this->Level]->Attackers.size(); ++i) {
    auto& object = this->Levels[this->Level]->Attackers[i];
    if (object->Destroyed || object->pop) {
        continue;
    }
    for (auto& Player : Players) {
        if (!Player->isAlive()) {
            continue;
        }
        for (auto& Weapon : Player->Weapons) {
            if (!Weapon->Using) {
                continue;
            }

            if (object->checkCollision(*Weapon).collision && !object->IsSolid) {
                SoundEngine->play2D("../resources/audio/ball-pop.mp3");
                object->pop = true;
                Weapon->Using = false;

                this->ShouldGeneratePowerUp(*object);

                //Create new balls half the size of the collision ball
                if (object && object->Size.x / 4.0f > 12.0f) {
                    object->pop = true;
                    glm::vec3 Position = object->Position;
                    float radius = object->Size.x / 4.0f;

                    AttackerObject* a = nullptr;
                    AttackerObject* b = nullptr;
                    if (dynamic_cast<HexagonObject*>(object)) {
                        a = new HexagonObject(Position, glm::vec3(radius * 2.0f, radius * 2.0f, 1.0f), ResourceManager::GetTexture("hexagon-1"), object->Color, glm::vec3(130.0f, 190.0f, 0.0f));
                        b = new HexagonObject(Position, glm::vec3(radius * 2.0f, radius * 2.0f, 1.0f), ResourceManager::GetTexture("hexagon-1"), object->Color, glm::vec3(130.0f, 190.0f, 0.0f));
                        a->Velocity = -a->Velocity;
                        b->Velocity.y = -b->Velocity.y;
                    }
                    else if (dynamic_cast<BallObject*>(object)) {
                        a = new BallObject(Position, radius, ResourceManager::GetTexture("ball"), object->Color, glm::vec3(130.0f, 190.0f, 0.0f));
                        b = new BallObject(Position, radius, ResourceManager::GetTexture("ball"), object->Color, glm::vec3(130.0f, 190.0f, 0.0f));
                        a->Velocity = -a->Velocity;
                        b->Velocity.y = -b->Velocity.y;
                    }

                    this->Levels[this->Level]->Attackers.push_back(a);
                    this->Levels[this->Level]->Attackers.push_back(b);
                    break;
                }
            }
        }
    }
}
```

Слика 28: Collision Attackers and Weapons

```
//COLLISION OBJECTS AND PLAYER
for (auto& Player : Players) {
    if (!Player->isAlive()) {
        continue;
    }
    Player->CollisionWith["ladder"] = Player->CollisionWith["block"] = false;
    for (auto& object : this->Levels[this->Level]->Objects) {
        if (object->Destroyed) {
            continue;
        }
        Collision& collisionObjectPlayer = object->checkCollision(*Player);
        if (collisionObjectPlayer.collision) {
            if (dynamic_cast<BlockObject*>(object)) {
                Player->CollisionWith["block"] = true;
            }
            else if (dynamic_cast<LadderObject*>(object)) {
                Player->CollisionWith["ladder"] = true;
            }
```

Слика 29: Collision Objects and Player

За колизија помеѓу објект и карактер, за тоа дали може да се качува по скалата и дали ке остане на друг BlockObject кој не е Destroyed или не.

И последно за колизијата помеѓу напагач и карактер ако напаѓачот удрил во карактерот тогаш на карактерот му се одзема живот и се сетира Alive атрибутот на false затоа што ако и се проверува дали има некој карактер кој е жив, ако нема таков тогаш се ресетира левелот и се стартува од ново, ако пак карактерите се без животи тогаш завршува играта.

```
//COLLISION ATTACKERS AND PLAYERS
for (auto& object : this->Levels[this->Level]->Attackers) {
    if (object->Destroyed || object->pop) {
        continue;
    }
    for (auto& Player : Players) {
        if (!Player->isAlive()) {
            continue;
        }
        if (Player->PlayerAttackerCollision(*object)) {
            if (!wasSleeping) {
                this->State = GAME_SLEEP;
                sleepForFrames = 1.0f;
            }
            else {
                --Player->Lives;
                Player->Alive = false;
                Player->ResetWeapons();
                wasSleeping = false;

                if (!anyPlayerAlive()) {
                    this->Levels[this->Level]->Reset();
                    this->ResetPlayers();
                    if (playersHaveLives()) {
                        SoundEngine->stopAllSounds();
                        SoundEngine->play2D("../resources/audio/stage1.mp3", true);
                        break;
                    }
                    SoundEngine->stopAllSounds();
                    SoundEngine->play2D("../resources/audio/game-over.mp3");
                    break;
                }
            }
        }
    }
}
```

Слика 30: Collision Attackers and Players

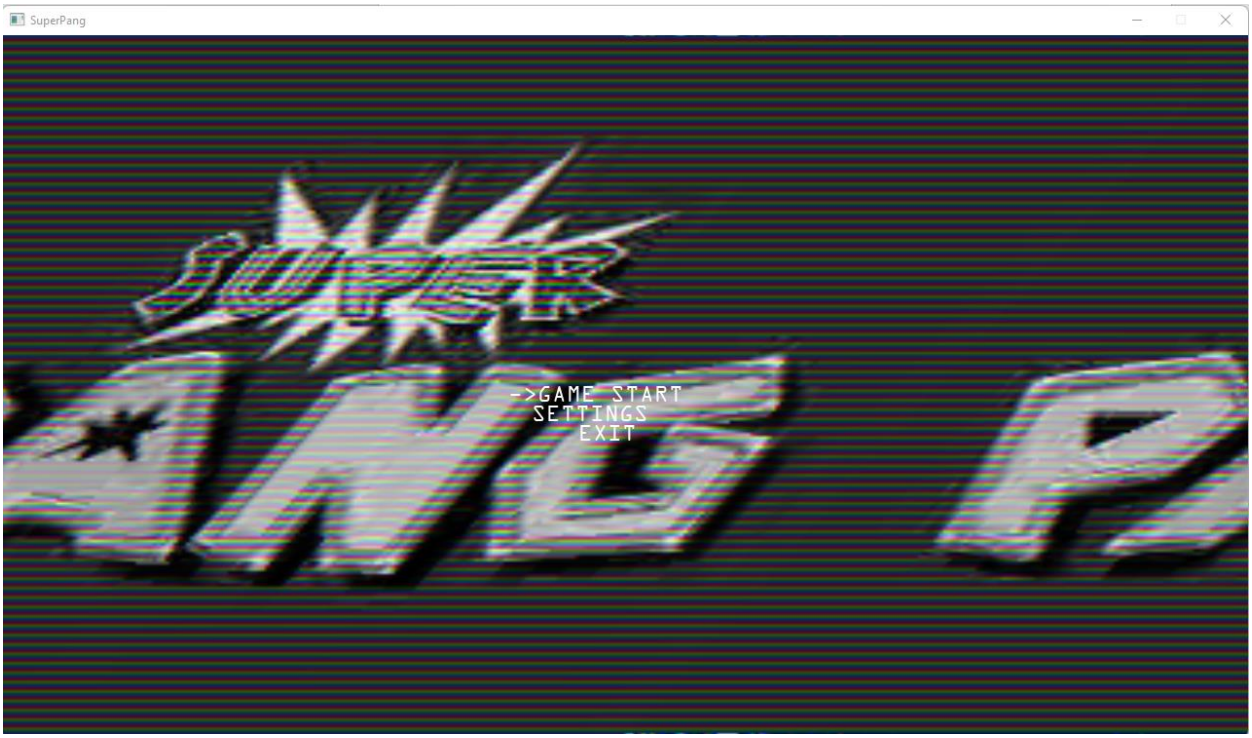
4. Главна програма

Во главната програма најпрво се вчитуваат библиотеките, различните шејдери, менито, потоа се поставуваат униформните променливи во шејдерите и се иницијализираат карактери, потребните Renderers, и инстанта од SoundEngine за користење на звук, се повикува методор Game::Render за да се исцртаат потребните објекти на екранот.

1. Функционалност

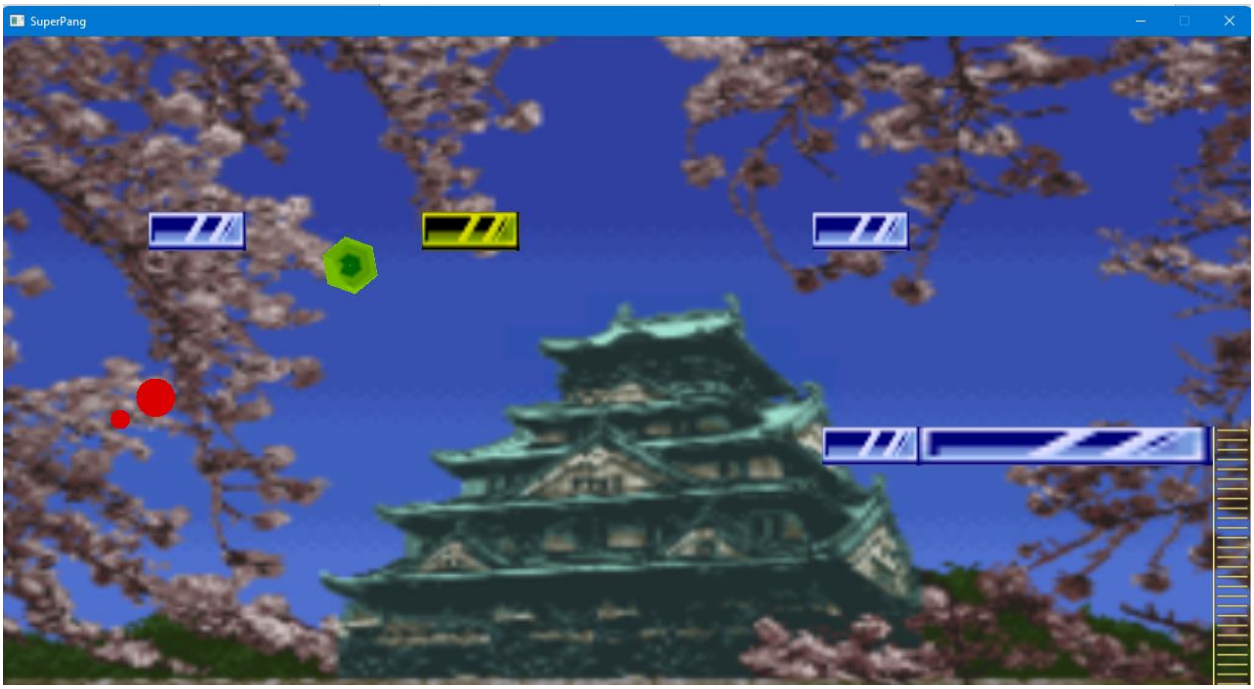
Како што беше кажано играта се состои од почетен екран со детектирање на притиснато копче на GAME START се започнува со првото ниво, со EXIT се затвора

апликацијата, со притискање на F11 се прави, и за сетирање на колку карактери ќе играат се прави со селектирање на опцијата SETTINGS -> SETTING 2 и тука има

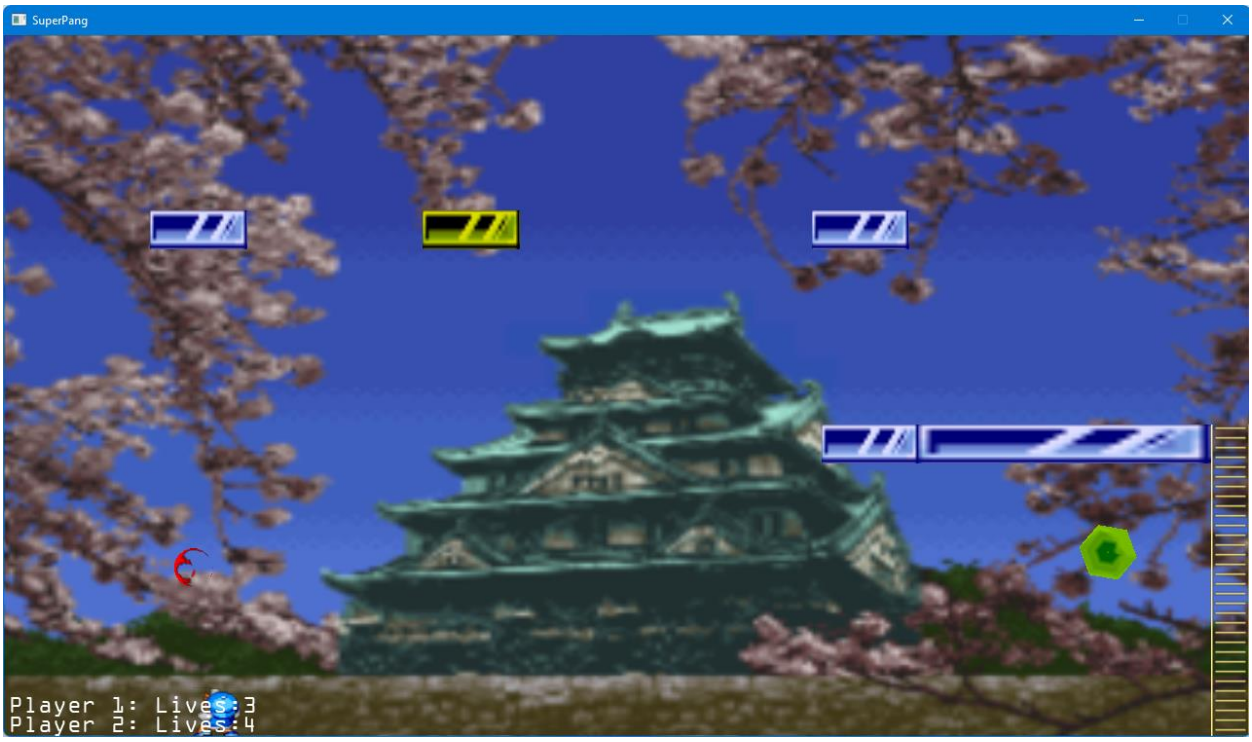


опција Players, со притискање на BACKSPACE се враќа назад на ParentOption, за контрола на карактерот се користат Up, Down, Left, Right, Space и X.

Слика 32: In-game menu



Слика 31: In-game started game look



Слика 33: In-game Ball popping

4.Заклучок

Играта мора да содржи некој примитивен 3D модел преку кој ке се рендерираат објектите кои ни се потребни во играта, исто така преку текстура може да се направи топка од коцка што е многу интересно

5.Искористена литература

[1] Learn OpenGL tutorial - https://learnopengl.com/

[2]OpenGL3DTutorial - https://youtube.com/playlist?list=PL6xSOsbVA1eYSZTKBxnoXYboy7wc4yg-Z