ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ № 7

за курсом «Програмування»

студента групи ПЗ-24у-1

Кондрачука Олександра Вадимовича

кафедра математичного забезпечення ЕОМ, ДНУ

2024/2025 навч.р.

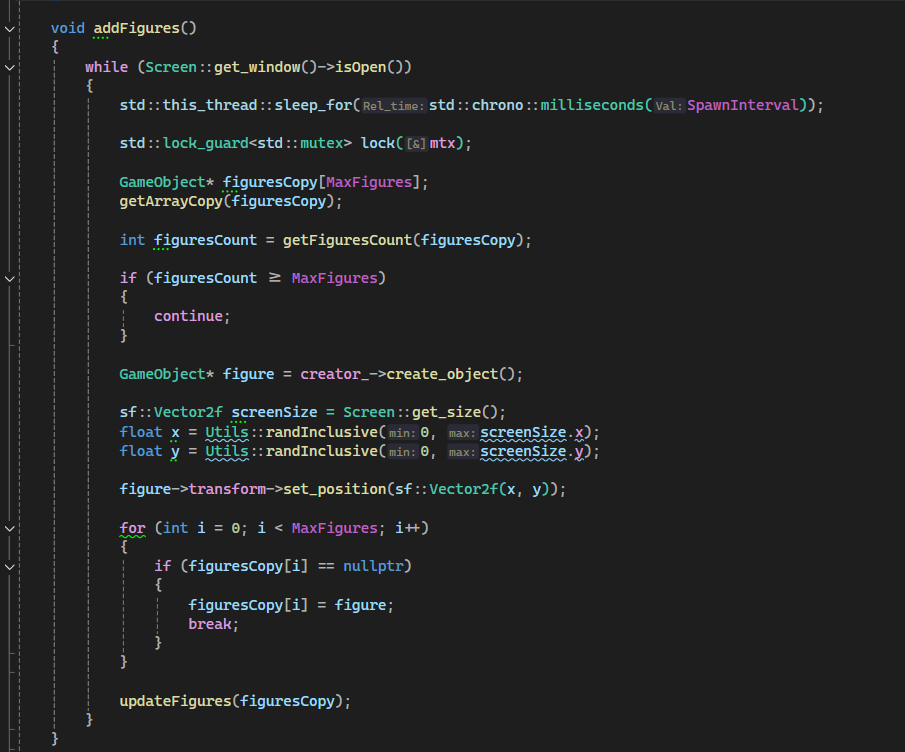
1. **Постановка завдання**

Взявши за основу класи геометричних фігур і графічний інтерфейс, які були розроблені в курсовій роботі, створіть програму, яка працює в багатопоточному режимі.

1. **Рішення**

Було створено клас RuntimeProcessor з 4 методами.

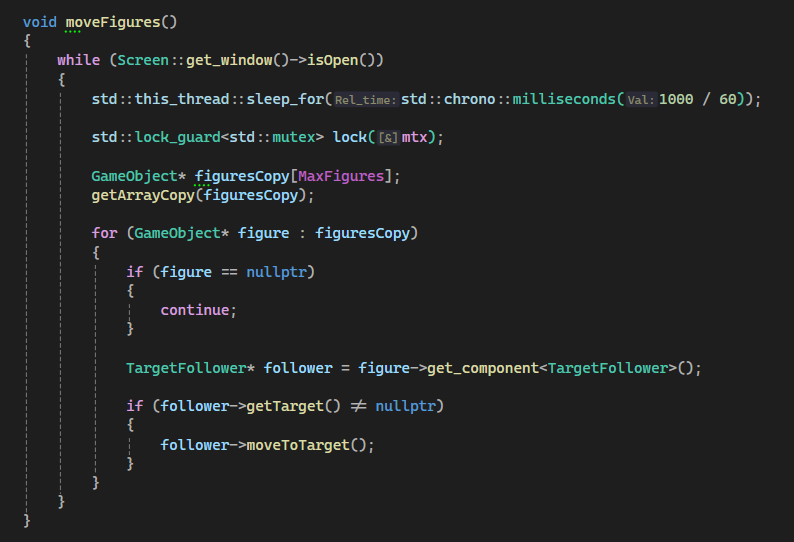
addFigures (): Цей метод створюж нову фігуру раз у 600 мілісекунд, якщо кількість фігур менше за 10 та додає нові фігури у список всії фігур. Цей метод працює в окремому потоці.

****

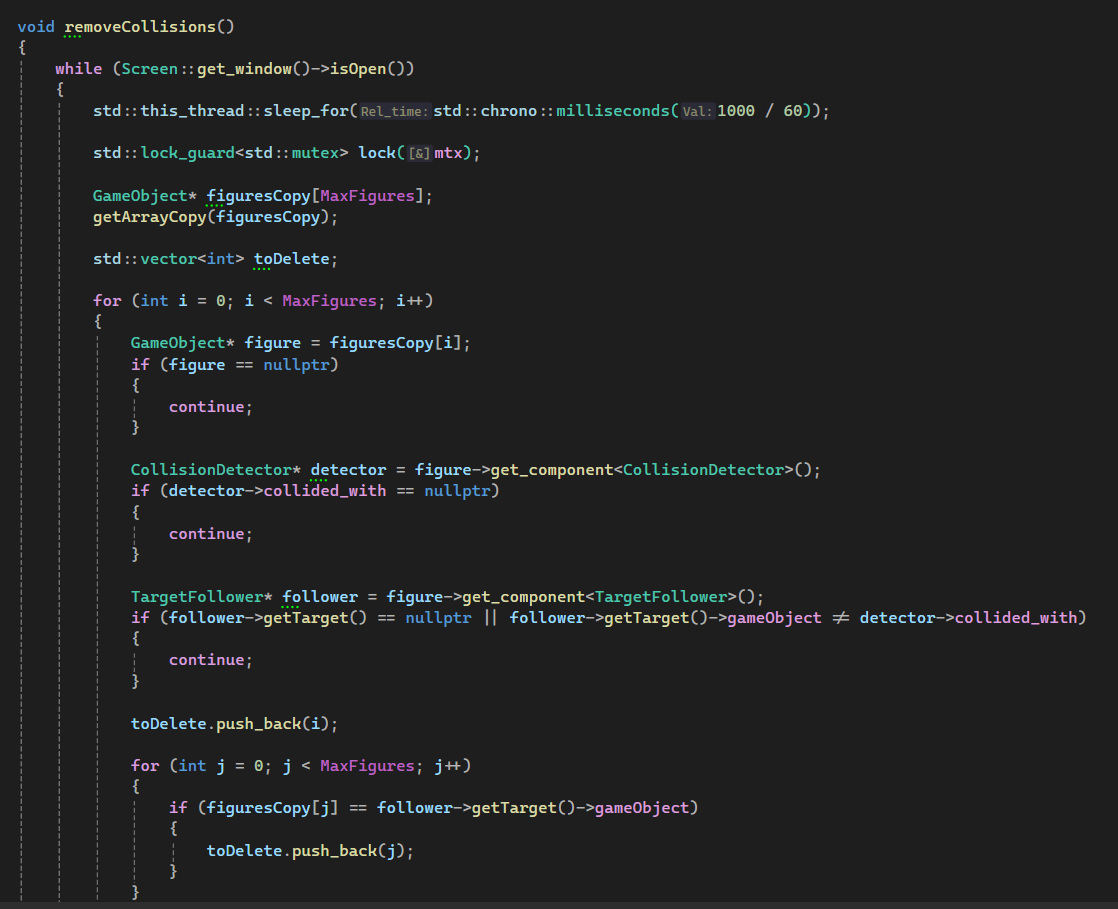
assignTargets(): Цей метод проходиться по кожній фігурі та якщо у неї немає цілі, то задається нова ціль.

****

moveFigures(): Цей метод відповідає за рух фігур. Він проходить по кожній фігурі та викликає метод руху у ній.

****

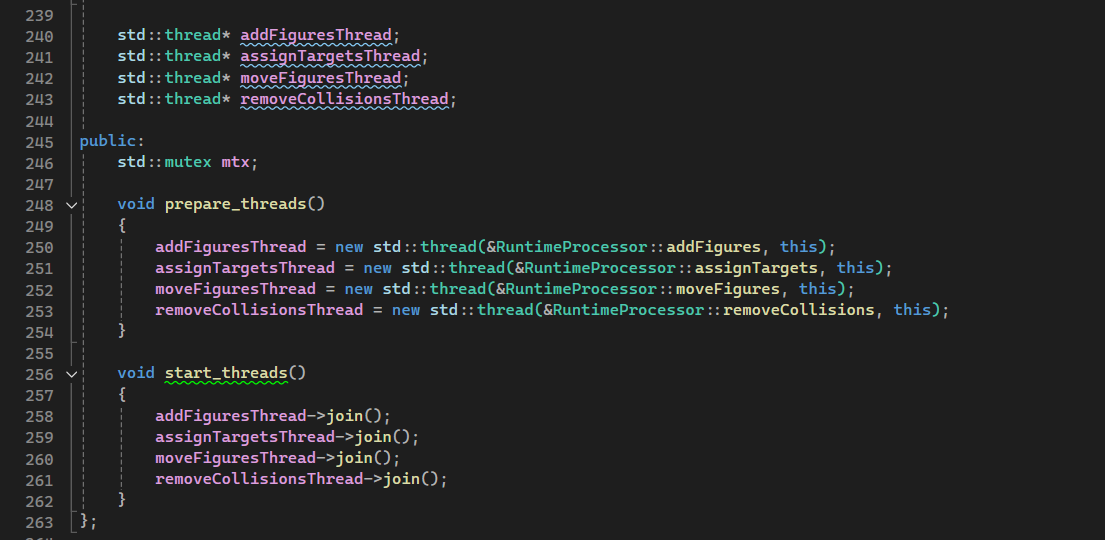
removeCollisions(): Цей метод перевіряє чи зіштовхувались фігури одна за одною і якщо колізія була з ціллю однієї з фігур, то вони видаляються.

****

Також були використані потоки та мутекси:

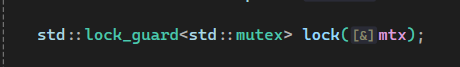
Потоки:

У функціях addFigures (), assignTargets(), moveFigures(), та removeCollisions() використовуються окремі потоки для виконання різних функцій гри паралельно. Наприклад, функція addFigures () додає нові фігури та цілі у гру, виконуючи свою роботу у власному потоці, тоді як функція assignTargets() прив'язує цілі до фігур у іншому потоці. Це дозволяє грі працювати ефективно та уникнути блокування головного потоку.



Мутекси:

Мутекс mtx використовується для синхронізації доступу до спільних даних, таких як вектори фігур, з різних потоків. Наприклад, перед доступом до вектору createdFigures, потоки блокують мутекс за допомогою std::lock\_guard<std::mutex> lock(mtx);. Це забезпечує безпечний доступ до даних та уникнення гонки за ресурсами.



1. **Лістинги:**

#pragma once

#include <mutex>

#include <thread>

#include "CollisionDetector.h"

#include "FiguresCreator.h"

#include "MonoBehaviour.h"

#include "Screen.h"

#include "TargetFollower.h"

#define SpawnInterval 600

#define MaxFigures 10

class RuntimeProcessor : public MonoBehaviour

{

FiguresCreator\* creator\_;

GameObject\* createdFigures\_[MaxFigures];

void awake() override

{

for (int i = 0; i < MaxFigures; i++)

{

createdFigures\_[i] = nullptr;

}

creator\_ = gameObject->get\_component<FiguresCreator>();

}

int getFiguresCount(GameObject\*\* figures)

{

int amount = 0;

for (int i = 0; i < MaxFigures; i++)

{

if (figures[i] != nullptr)

{

amount++;

}

}

return amount;

}

void getArrayCopy(GameObject\* destination[MaxFigures]) {

for (int i = 0; i < MaxFigures; ++i) {

destination[i] = createdFigures\_[i];

}

}

void updateFigures(GameObject\* updateArray[MaxFigures])

{

for (int i = 0; i < MaxFigures; ++i) {

createdFigures\_[i] = updateArray[i];

}

}

void addFigures()

{

while (Screen::get\_window()->isOpen())

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(SpawnInterval));

std::lock\_guard<std::mutex> lock(mtx);

GameObject\* figuresCopy[MaxFigures];

getArrayCopy(figuresCopy);

int figuresCount = getFiguresCount(figuresCopy);

if (figuresCount >= MaxFigures)

{

continue;

}

GameObject\* figure = creator\_->create\_object();

sf::Vector2f screenSize = Screen::get\_size();

float x = Utils::randInclusive(0, screenSize.x);

float y = Utils::randInclusive(0, screenSize.y);

figure->transform->set\_position(sf::Vector2f(x, y));

for (int i = 0; i < MaxFigures; i++)

{

if (figuresCopy[i] == nullptr)

{

figuresCopy[i] = figure;

break;

}

}

updateFigures(figuresCopy);

}

}

void assignTargets()

{

while (Screen::get\_window()->isOpen())

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000 / 60));

std::lock\_guard<std::mutex> lock(mtx);

GameObject\* figuresCopy[MaxFigures];

getArrayCopy(figuresCopy);

for (auto figure : figuresCopy)

{

if (figure == nullptr)

{

continue;

}

TargetFollower\* follower = figure->get\_component<TargetFollower>();

if (follower == nullptr)

{

continue;

}

if (follower->getTarget() == nullptr)

{

int targetIndex = Utils::randInclusive(0, MaxFigures - 1);

GameObject\* target = figuresCopy[targetIndex];

if (target == nullptr || target == figure)

{

continue;

}

follower->setTarget(target->transform);

}

}

}

}

void moveFigures()

{

while (Screen::get\_window()->isOpen())

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000 / 60));

std::lock\_guard<std::mutex> lock(mtx);

GameObject\* figuresCopy[MaxFigures];

getArrayCopy(figuresCopy);

for (GameObject\* figure : figuresCopy)

{

if (figure == nullptr)

{

continue;

}

TargetFollower\* follower = figure->get\_component<TargetFollower>();

if (follower->getTarget() != nullptr)

{

follower->moveToTarget();

}

}

}

}

void removeCollisions()

{

while (Screen::get\_window()->isOpen())

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000 / 60));

std::lock\_guard<std::mutex> lock(mtx);

GameObject\* figuresCopy[MaxFigures];

getArrayCopy(figuresCopy);

std::vector<int> toDelete;

for (int i = 0; i < MaxFigures; i++)

{

GameObject\* figure = figuresCopy[i];

if (figure == nullptr)

{

continue;

}

CollisionDetector\* detector = figure->get\_component<CollisionDetector>();

if (detector->collided\_with == nullptr)

{

continue;

}

TargetFollower\* follower = figure->get\_component<TargetFollower>();

if (follower->getTarget() == nullptr || follower->getTarget()->gameObject != detector->collided\_with)

{

continue;

}

toDelete.push\_back(i);

for (int j = 0; j < MaxFigures; j++)

{

if (figuresCopy[j] == follower->getTarget()->gameObject)

{

toDelete.push\_back(j);

}

}

}

for (int to\_delete : toDelete)

{

GameObject\* figure = figuresCopy[to\_delete];

if (figure == nullptr)

{

continue;

}

for (GameObject\* follower : figuresCopy)

{

if (follower == nullptr)

{

continue;

}

TargetFollower\* component = follower->get\_component<TargetFollower>();

if (component->getTarget() != nullptr && component->getTarget()->gameObject == figure)

{

component->setTarget(nullptr);

}

}

GameObject::objectsContainer\_->destroy(figure);

figuresCopy[to\_delete] = nullptr;

}

updateFigures(figuresCopy);

}

}

std::thread\* addFiguresThread;

std::thread\* assignTargetsThread;

std::thread\* moveFiguresThread;

std::thread\* removeCollisionsThread;

public:

std::mutex mtx;

void prepare\_threads()

{

addFiguresThread = new std::thread(&RuntimeProcessor::addFigures, this);

assignTargetsThread = new std::thread(&RuntimeProcessor::assignTargets, this);

moveFiguresThread = new std::thread(&RuntimeProcessor::moveFigures, this);

removeCollisionsThread = new std::thread(&RuntimeProcessor::removeCollisions, this);

}

void start\_threads()

{

addFiguresThread->join();

assignTargetsThread->join();

moveFiguresThread->join();

removeCollisionsThread->join();

}

};

#include <iostream>

#include <SFML/Graphics.hpp>

#include "FiguresCreator.h"

#include "MonoBehaviour.h"

#include "GameObjectsContainer.h"

#include "Input.h"

#include "RuntimeProcessor.h"

#include "Screen.h"

#include "Time.h"

GameObject\* add\_objects(GameObjectsContainer\* container)

{

GameObject\* mainObject = new GameObject();

mainObject->add\_component(new FiguresCreator());

mainObject->add\_component(new RuntimeProcessor());

container->instantiate(mainObject);

return mainObject;

}

int main()

{

sf::RenderWindow window(sf::VideoMode(800, 800), "SFML works!");

sf::Clock clock;

Input input(&window);

Screen screen(&window);

Time time(clock);

GameObjectsContainer container;

GameObject object(&container);

GameObject\* mainObject = add\_objects(&container);

RuntimeProcessor\* processor = mainObject->get\_component<RuntimeProcessor>();

processor->prepare\_threads();

while (window.isOpen())

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000 / 120));

std::lock\_guard<std::mutex> lock(processor->mtx);

screen.clear();

time.restart();

container.update(&window);

screen.display();

}

processor->start\_threads();

return 0;

}