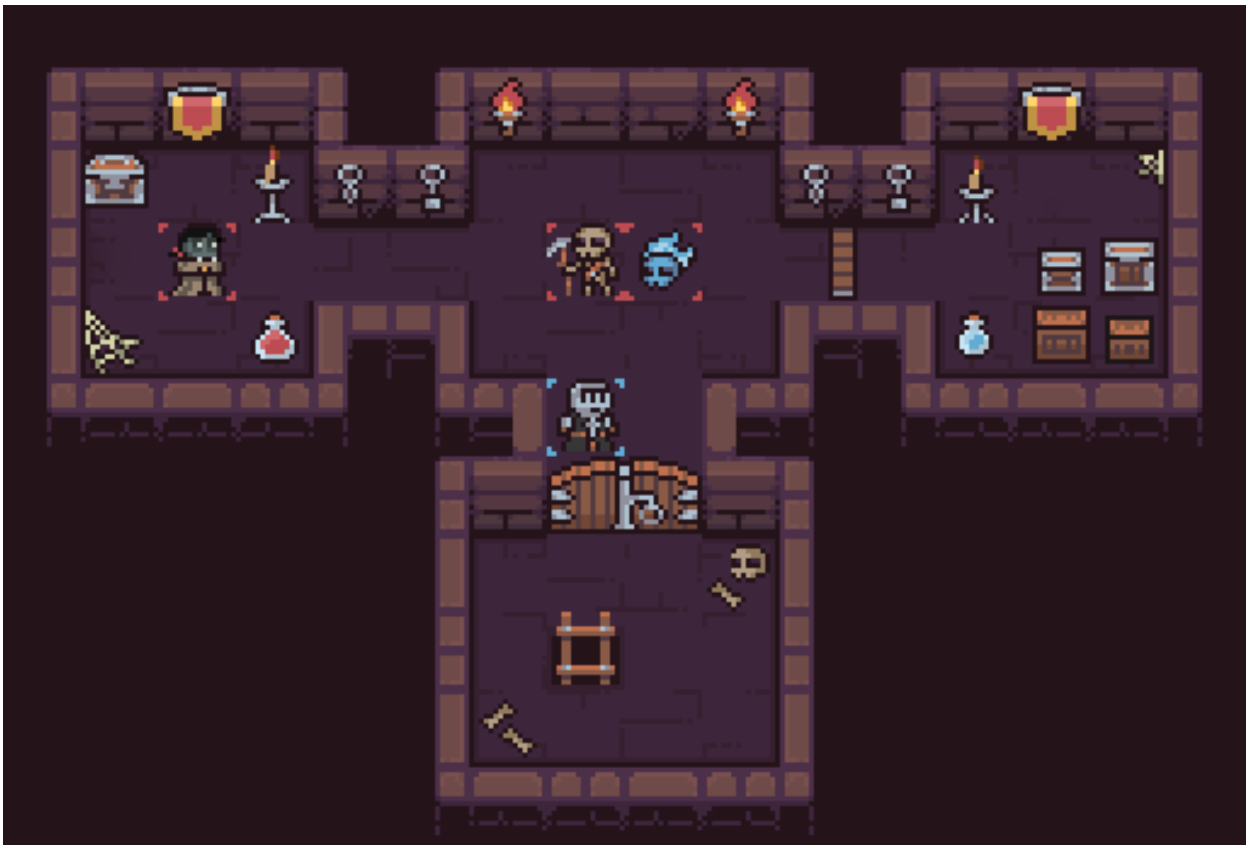
PROIECT DISCIPLINA POO

SwordStone: Implementarea conceptelor POO într-un joc 2D retro în C++ și SFML



Ecoboaea Denis - Mihai

14/06/2023

3123b - CALCULATOARE

INTRODUCERE

Proiectul SwordStone este un joc 2D retro de tip pixel art, dezvoltat în limbajul de programare C++ și utilizând biblioteca SFML. Jucătorul preia rolul unui cavaler într-o temniță plină de inamici, comori și provocări, în timp ce utilizează conceptele de programare orientată pe obiecte pentru a implementa animații, coliziuni și alte elemente de joc interactiv.

Prezentare și motivare:

Alegerea temei proiectului SwordStone a fost determinată de dorința de a explora și aprofunda conceptele de programare orientată pe obiecte într-un context practic și captivant. Prin dezvoltarea acestui joc 2D, am dorit să pun în aplicare principii fundamentale ale POO, cum ar fi încapsularea și moștenirea, pentru a crea o experiență de joc dinamică și plină de viață.

Implementarea unui joc retro în stil pixel art a adus oportunități interesante de proiectare și dezvoltare, iar utilizarea bibliotecii SFML mi-a oferit un mediu prietenos și robust pentru gestionarea elementelor grafice, sunetelor și interacțiunilor jucătorului. Prin intermediul acestui proiect, mi-am propus să învăț și să aplic bune practici de programare, structurare a codului și gestionare a resurselor într-un mediu de dezvoltare complex.

Pe parcursul documentației, voi explora detaliat modul în care am folosit conceptele de programare orientată pe obiecte în implementarea funcționalităților jocului, voi descrie elementele de animație și coliziune, precum și diversele unelte și tehnici utilizate pentru a facilita crearea acestui joc captivant. Prin documentarea și explicarea procesului de dezvoltare, îmi propun să ofer cititorilor o înțelegere clară a modului în care POO poate fi aplicată într-un context real și să-i inspire să exploreze și ei potențialul acestei paradigme în propriile lor proiecte.

În continuare, voi explora detaliile implementării și voi oferi exemple practice pentru a sublinia avantajele și importanța programării orientate pe obiecte în dezvoltarea jocurilor.

CUPRINS

Elemente de joc

AnimatedObject.hpp

Clasa `AnimatedObject` definește un obiect animat care aduce viață jocului. Aceasta utilizează o colecție de cadre de animație (texturi) pentru a afișa animații fluide. Obiectul animat poate fi actualizat și afișat pe fereastra de joc, iar poziția și scalarea acestuia pot fi configurate.

CollisionElement.hpp

Clasa `CollisionElement` gestionează elementele de coliziune statice din joc. Aceasta oferă metode pentru adăugarea de elemente de coliziune sub forma unor dreptunghiuri și pentru verificarea coliziunii cu alte dreptunghiuri. Elementele de coliziune pot fi desenate pe fereastra de joc pentru a oferi o experiență vizuală corespunzătoare.

DrawnButton.hpp

Clasa `DrawnButton` adaugă interactivitate la joc prin furnizarea unui buton desenat pe fereastra de joc. Butonul poate fi poziționat și personalizat cu texturi diferite pentru stările de inactivitate, supraînălțare și apăsare. În plus, acesta poate fi configurat cu o acțiune personalizată care se declanșează atunci când butonul este apăsat. Actualizarea și desenarea butonului se fac cu ușurință.

DrawnImage.hpp

Clasa `DrawnImage` permite afișarea de imagini pe fereastra de joc. Imaginea poate fi poziționată și redimensionată în funcție de preferințe. Afișarea imaginii se realizează cu ajutorul bibliotecii SFML și oferă un aspect vizual plăcut jocului.

DrawnText.hpp

Clasa `DrawnText` gestionează afișarea de texte pe fereastra de joc. Aceasta permite configurarea fontului, a dimensiunii, a culorii și a poziției textului. Textul poate fi personalizat în funcție de necesități și poate fi desenat pe fereastra de joc cu ușurință.

FontLoader.hpp

Clasa `FontLoader` se ocupă de încărcarea fonturilor folosite în joc. Aceasta permite încărcarea fonturilor dintr-un director specificat și oferă o metodă pentru a accesa lista de fonturi încărcate. Astfel, textele afișate în joc vor avea un aspect coerent și atractiv.

MusicPlayer.hpp

Clasa `MusicPlayer` oferă funcționalitatea de redare a muzicii în joc. Aceasta permite selectarea și redarea fișierelor audio specifice și asigură posibilitatea de a opri redarea la cerere. Muzica ambientală și efectele sonore adaugă o atmosferă captivantă și imersivă jocului.

ParallaxBackground.hpp

Clasa `ParallaxBackground` implementează un fundal paralax în joc, adăugând un efect de adâncime și mișcare la fereastra de joc. Acesta utilizează un set de imagini stratificate pentru a crea iluzia de paralax și poate fi actualizat în funcție de poziția mouse-ului. Adăugarea unui fundal paralax oferă o experiență vizuală impresionantă și atrăgătoare.

Player.hpp

Clasa `Player` definește caracteristicile și comportamentul jucătorului în joc. Aceasta gestionează mișcarea jucătorului în toate direcțiile, atacul, actualizarea și desenarea pe fereastra de joc. De asemenea, se ocupă de detecția și gestionarea coliziunilor cu alte obiecte și inamici. Jucătorul are un nivel de viață (HP) care poate fi modificat și actualizat în funcție de evenimentele jocului. Designul animațiilor și al caracterului jucătorului oferă o experiență captivantă și interactivă.

TextureLoader.hpp

Clasa `TextureLoader` se ocupă de încărcarea texturilor folosite în joc. Aceasta permite încărcarea texturilor dintr-un director specificat și oferă o metodă pentru a accesa lista de texturi încărcate. Texturile sunt utilizate pentru a da viață jocului prin adăugarea de grafică de calitate și detaliu.

Implementarea conceptelor de Programare Orientată pe Obiecte (POO)

Implementarea jocului se bazează pe principiile și conceptele cheie ale Programării Orientate pe Obiecte (POO). POO este o paradigmă de programare care se concentrează pe organizarea și structurarea codului în jurul obiectelor, care sunt instanțe ale claselor. Această abordare oferă modularitate, reutilizabilitate și flexibilitate în dezvoltarea jocului.

Clase și obiecte

În cadrul jocului, conceptul central al POO este reprezentat de utilizarea claselor și obiectelor. Fiecare element sau entitate în joc este reprezentat printr-o clasă care definește caracteristicile și comportamentul acestuia. De exemplu, clasa Player definește jucătorul, clasa AnimatedObject reprezintă obiectele animate, iar clasa CollisionElement gestionează elementele de coliziune. Obiectele sunt instanțe ale claselor respective și interacționează între ele în cadrul jocului.

Abstracție și încapsulare

Prin intermediul POO, conceptele de abstracție și încapsulare sunt aplicate în dezvoltarea jocului. Clasele sunt utilizate pentru a defini și implementa abstracțiile, oferind o interfață simplificată și clară către funcționalitățile respective. Încapsularea asigură că detaliile interne ale claselor sunt ascunse și că interacțiunea cu acestea se face prin metodele publice definite în clasă. Aceasta promovează modularitatea și reduce dependențele dintre componente.

Moștenire și polimorfism

Prin utilizarea moștenirii și polimorfismului, jocul permite crearea de entități specifice cu caracteristici comune și comportamente suplimentare. De exemplu, clasa Player poate moșteni caracteristicile generice ale clasei AnimatedObject, adăugând în plus metode și atribute specifice jucătorului. Astfel, se reduce duplicarea de cod și se asigură o structură coerentă a claselor în cadrul jocului.

Asociere și agregare

În implementarea jocului, conceptele de asociere și agregare sunt utilizate pentru a defini relațiile dintre diferite obiecte și entități. De exemplu, clasa Player poate avea o asociere cu clasa CollisionElement pentru a detecta coliziunile cu alte elemente din

joc. De asemenea, clasa ParallaxBackground utilizează agregarea pentru a gestiona colecția de imagini stratificate pentru fundalul paralax.

Aceste concepte cheie ale POO contribuie la structurarea și organizarea codului în jocul nostru, asigurând modularitatea, flexibilitatea și reutilizabilitatea componentelor. Prin aplicarea acestor principii și concepte, dezvoltarea și întreținerea jocului SwordStone devin mai eficiente și mai ușor de gestionat.

Coliziuni

Coliziunile reprezintă o parte crucială a jocului meu, deoarece asigură interacțiunea corectă între diferitele entități. Pentru gestionarea coliziunilor, am implementat o clasă numită `CollisionElement`, care se bazează pe detecția și rezolvarea coliziunilor.

Pentru a asigura existența unei singure instanțe a clasei `CollisionElement`, am utilizat un design pattern numit Singleton. Astfel, am implementat metoda `getInstance()` care returnează instanța unică a clasei:

```
CollisionElement& CollisionElement::getInstance()
{
    if (instance == nullptr) {
        instance = new CollisionElement();
    }
    return *instance;
}
```

Am creat și metoda `addElement()` pentru a adăuga elemente de coliziune la vectorul `collisionElements`. Această metodă primește un obiect de tip `sf::FloatRect` care reprezintă zona de coliziune și îl adaugă la vector:

```
void CollisionElement::addElement(const sf::FloatRect& rect)
{
    collisionElements.push_back(rect);
}
```

Pentru a verifica coliziunile între două obiecte, am implementat metoda `checkCollision()`. Această metodă parcurge fiecare obiect din `collisionElements` și verifică dacă se produce o intersecție cu `otherRect`:

```
bool CollisionElement::checkCollision(const sf::FloatRect& otherRect)
const
{
    for (const auto& rect : collisionElements) {
        if (rect.intersects(otherRect)) {
            return true;
        }
    }
    return false;
}
```

De asemenea, am creat metoda `draw()` pentru a afișa obiectele de coliziune pe fereastră. Pentru fiecare element din `collisionElements`, desenăm un dreptunghi roșu utilizând clasa `sf::RectangleShape`:

```
void CollisionElement::draw(sf::RenderWindow& window) const
{
    sf::RectangleShape shape;
    shape.setFillColor(sf::Color::Red);

    for (const auto& rect : collisionElements) {
        shape.setSize(sf::Vector2f(rect.width, rect.height));
        shape.setPosition(rect.left, rect.top);
        window.draw(shape);
    }
}
```

Animații

Animațiile adaugă jocului meu un nivel suplimentar de realism și interactivitate. Am implementat animațiile utilizând o abordare bazată pe schimbarea texturilor și gestionarea stărilor de animație.

Pentru a crea obiecte animate în jocul meu, am implementat clasa `AnimatedObject`. Această clasă utilizează un vector de texturi și un sprite pentru a reda animațiile. Constructorul clasei primește calea către texturile separate ale obiectului animat și le încarcă în vectorul `animationFrames`:

```
AnimatedObject::AnimatedObject(const std::string& texturePath)
{
    for (int i = 1; i <= 4; i++)
    {
        sf::Texture* texture = new sf::Texture();
        if (!texture->loadFromFile(texturePath + std::to_string(i) +
".png"))
        {
            // Gestionează eroarea în cazul în care apare o problemă la
încărcarea texturii
        }
        animationFrames.push_back(texture);
    }

    sprite.setTexture(*animationFrames[0]);

    // Alte inițializări
    currentFrame = 0;
    frameCount = 4;
}
```

Pentru a reda animațiile, clasa `AnimatedObject` utilizează o metodă `update()` care calculează timpul scurs de la ultimul frame și trece la următorul frame dacă a trecut suficient timp. Aceasta se realizează prin verificarea valorii returnate de `elapsed.asMilliseconds()` și compararea cu o valoare de prag (de exemplu, 200 milisecunde):

```
void AnimatedObject::update()
{
    sf::Time elapsed = frameClock.getElapsedTime();

    if (elapsed.asMilliseconds() >= 200)
    {
        currentFrame = (currentFrame + 1) % frameCount;
        sprite.setTexture(*animationFrames[currentFrame]);

        frameClock.restart();
    }
}
```

De asemenea, am implementat metoda `render()` pentru a desena obiectul animat pe fereastră utilizând `sf::RenderWindow::draw()`:

```
void AnimatedObject::render(sf::RenderWindow& window) const
{
    window.draw(sprite);
}
```

Pentru a poziționa și scala obiectul animat, am creat metodele `setPosition()` și `setScale()`, care se bazează pe metodele corespunzătoare ale obiectului `sf::Sprite`:

```
void AnimatedObject::setPosition(float x, float y)
{
    sprite.setPosition(x, y);
}

void AnimatedObject::setPosition(const sf::Vector2f& position)
{
    sprite.setPosition(position);
}

void AnimatedObject::setScale(float scaleX, float scaleY)
{
    sprite.setScale(scaleX, scaleY);
}
```

```
void AnimatedObject::setScale(const sf::Vector2f& scale)
{
    sprite.setScale(scale);
}
```

Utilizarea CMake

Pentru gestionarea proiectului și a dependențelor, am utilizat CMake, un sistem de generare a fișierelor de proiect în diferite medii de dezvoltare. CMake ne-a permis să configurăm și să construim proiectul într-un mod flexibil și portabil. În această secțiune, vom prezenta modul în care am utilizat CMake în proiect.

Configurarea versiunii CMake

Am specificat versiunea minimă necesară a CMake în fișierul `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.21)
```

Aceasta asigură faptul că proiectul este construit folosind o versiune compatibilă de CMake.

Declararea și obținerea bibliotecii SFML

Pentru a utiliza biblioteca SFML în proiect, am folosit funcționalitatea `FetchContent` din CMake. Am declarat biblioteca SFML și am specificat sursa și versiunea de pe GitHub:

```
include(FetchContent)
FetchContent_Declare(SFML
    GIT_REPOSITORY https://github.com/SFML/SFML.git
    GIT_TAG 2.6.x)
FetchContent_MakeAvailable(SFML)
```

Apoi, am obținut biblioteca SFML și am făcut-o disponibilă în proiect.

Configurarea surselor și directorului de ieșire

Am folosit comanda `FILE(GLOB ...)` pentru a lista toate fișierele sursă din directorul `source` și subdirectoarele sale:

```
FILE(GLOB SRCFILES source/*.cpp source/scenes/*.cpp
source/scenes/utils/*.cpp)
```

Aceasta ne-a permis să colectăm automat toate fișierele sursă în proiect, fără a fi nevoie să le enumerăm manual.

Am setat apoi directorul de ieșire pentru fișierul executabil folosind comanda `SET(EXECUTABLE_OUTPUT_PATH ...)`:

```
SET(EXECUTABLE_OUTPUT_PATH ${CMAKE_SOURCE_DIR}/bin)
```

Crearea executabilului și legarea bibliotecilor

Am adăugat un executabil numit `game` și i-am atribuit toate fișierele sursă:

```
ADD_EXECUTABLE(game ${SRCFILES})
```

Am folosit apoi comanda `target_link_libraries` pentru a lega bibliotecile SFML necesare:

```
target_link_libraries(game PRIVATE sfml-graphics sfml-audio sfml-window
sfml-system)
```

Aceasta asigură faptul că executabilul are acces la funcționalitățile oferite de SFML.

Setarea standardului C++

Am specificat standardul C++ utilizat în proiect folosind comanda

`target_compile_features`:

```
target_compile_features(game PRIVATE cxx_std_17)
```

Aceasta a asigurat că compilatorul utilizează standardul C++17.

Instalarea executabilului

Am inclus comanda `install` pentru a instala executabilul în sistem:

```
install(TARGETS game)
```

Aceasta permite utilizatorilor să instaleze jocul pe propriul sistem.

Bibliografie

Informatii:

- https://www.sfml-dev.org/learn.php
- https://www.geeksforgeeks.org/implementation-of-singleton-class-in-cpp/
- https://www.geeksforgeeks.org/sfml-graphics-library-quick-tutorial/
- https://www.w3schools.com/cpp/cpp_classes.asp

Assets:

- https://0x72.itch.io/dungeontileset-ii
- https://0x72.itch.io/16x16-dungeon-tileset
- https://pixel-poem.itch.io/dungeon-assetpuck
- https://xdeviruchi.itch.io/8-bit-fantasy-adventure-music-pack?download