Politechnika Śląska

Wydział Informatyki, Elektroniki i Informatyki

Programowanie Komputerów

Projekt Arkanoid

autor	**Bartosz Kępa**
prowadzący	dr inż. Michał Piela
rok akademicki	2024/2025
kierunek informatyka	rodzaj studiów SSI semestr 4
termin laboratorium	Czwartek, 13:30 – 15:00
sekcja	5
termin oddania sprawozdania	2025-06-12

Opis programu

Arkanoid to klasyczna gra zręcznościowa 2D, w której gracz steruje paletką poruszającą się w poziomie na dole ekranu (800x600 pikseli), odbijając piłkę w celu niszczenia kolorowych bloków w górnej części planszy. Celem gry jest zniszczenie wszystkich możliwych do zniszczenia bloków na poziomie, unikając wypadnięcia piłki poza dolną krawędź ekranu, co skutkuje utratą życia (domyślnie 3 życia). Gra oferuje różnorodne mechaniki, w tym:

- **Typy bloków**: Zwykłe (niszczone po 1 uderzeniu), niezniszczalne (odbijają piłkę), bonusowe (uwalniają power-upy).
- **Bonusy**: Rozszerzenie paletki, dodatkowe piłki (multi-ball), przyspieszenie piłki.
- **Poziomy**: Wczytywane z plików assets/levelX.txt, z różnymi układami bloków i parametrami (szerokość paletki, prędkość piłki).
- **Wyniki**: Zapisywane w plikach players/nickname.txt i assets/highscores.txt, z rankingiem najlepszych graczy.

Rozgrywka rozpoczyna się od menu głównego, oferującego opcje: rozpoczęcie gry, wybór poziomu, przeglądanie rankingu lub wyjście. Po wybraniu gry gracz wpisuje pseudonim (walidowany: 1–10 znaków, zaczynający się od wielkiej litery, alfanumeryczny), który jest zapisywany z wynikiem. Gra przechodzi do wybranego poziomu, gdzie gracz steruje paletką (klawisze Left/Right), a piłka odbija się od ścian, paletki i bloków. Klawisz Escape aktywuje pauzę, a po ukończeniu poziomu lub utracie żyć wyświetla się ekran końca gry lub ekran ukończenia poziomu. Projekt zaimplementowano w C++20 z biblioteką SFML, stosując moduły, wzorce projektowe (State, Fabryka, Strategia) oraz nowoczesne techniki, takie jak std::ranges i std::filesystem.

Specyfikacja zewnętrzna

Ekran menu głównego

Menu główne jest pierwszym ekranem gry, z tłem assets/background2.png i czcionką Arial. W centrum znajdują się przyciski tekstowe:

- **Start Game**: Podświetlany na zielono, przechodzi do ekranu wpisywania pseudonimu z poziomem 1.
- **Select Level**: Otwiera podmenu wyboru poziomu (1 do maxLevel, określonego przez liczbę plików levelX.txt).
- **High Scores**: Wyświetla ranking (do 4 wyników: pseudonim, punkty).
- **Exit**: Czerwony, zamyka grę.

W podmenu wyboru poziomu przyciski „Level X" i „Back" umożliwiają wybór poziomu lub powrót. Muzyka tła (assets/background.mp4) odtwarzana jest automatycznie.

Rys.1. Ekran menu głównego

Rys.2. Szkic ekranu menu głównego

Ekran wpisywania pseudonimu

Ekran (assets/background2.png) zawiera pole tekstowe z migającym kursorem, gdzie gracz wpisuje pseudonim (1–10 znaków, pierwsza litera wielka, tylko alfanumeryczne). Walidacja przez std::regex wyświetla błędy (czerwony tekst), np. „Only letters and numbers allowed!”. Po naciśnięciu Enter pseudonim jest sprawdzany pod kątem unikalności w players/nickname.txt. Poprawny pseudonim inicjuje rozgrywkę na wybranym poziomie.

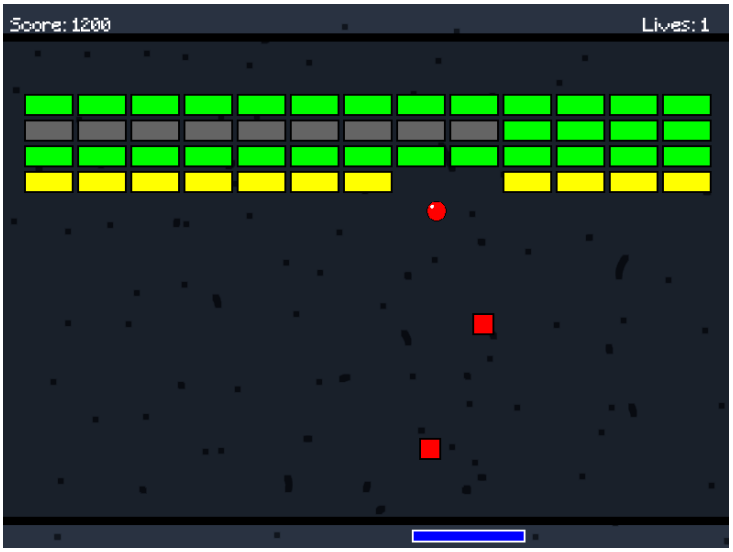
Rys.3. Ekran wpisywania pseudonimu

Ekran gry

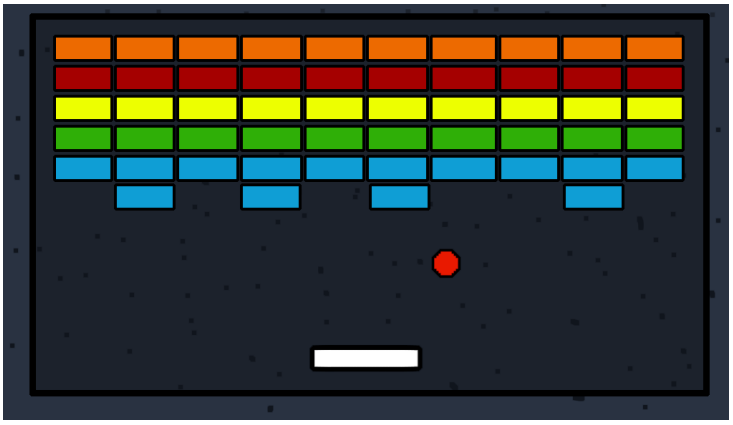
Ekran rozgrywki dzieli się na:

- **Plansza**: Tło assets/background.png, paletka (niebieski prostokąt), piłka (sprite assets/ball.png, 24x24 piksele), bloki (50x20 pikseli: zwykłe – różne kolory, niezniszczalne – szare, bonusowe – żółte).
- **Interfejs**: Wynik („Score: X”, lewy górny róg), życia („Lives: X”, prawy górny róg), czcionka Minecraft, 20 pt.
- **Bonusy**: Spadające prostokąty (20x20 pikseli): cyjan – rozszerzenie paletki, magenta – multi-ball, czerwony – przyspieszenie piłki.

Gracz steruje paletką klawiszami Left/Right, a Escape aktywuje pauzę. Zniszczenie bloku bonusowego generuje losowy bonus, który znika po dotarciu do dolnej krawędzi ($y > 600$).



Rys.4. Ekran gry



Rys.5. Szkic ekranu gry

Ekran pauzy

Wciśnięcie Escape przyciemnia ekran nakładką (czarna, 50% przezroczystości) i wyświetla:

- Tekst „Paused” (Minecraft.ttf, 50 pt).
- Przyciski: „Resume” (wznawia grę), „Restart” (resetuje poziom), „Main Menu”, „Exit” (zamyka grę).

Przyciski podświetlają się na zielono (oprócz „Exit” – czerwony) po najechaniu myszą.

Rys.6. Ekran pauzy

Rys.7. Szkic ekranu pauzy

Ekran ukończenia poziomu

Po zniszczeniu wszystkich bloków możliwych do zniszczenia wyświetla się nakładka z tekstem „Level X Complete!” (Minecraft.ttf, 50 pt) i przyciskami:

- **Next Level**: Dostępny, jeśli istnieje kolejny poziom, przechodzi do levelX+1.txt.
- **Restart**, **Main Menu**, **Exit**: Jak w pauzie.

Wynik jest zapisywany w HighScoreManager.

Rys.8. Ekran ukończenia poziomu

Ekran końca gry

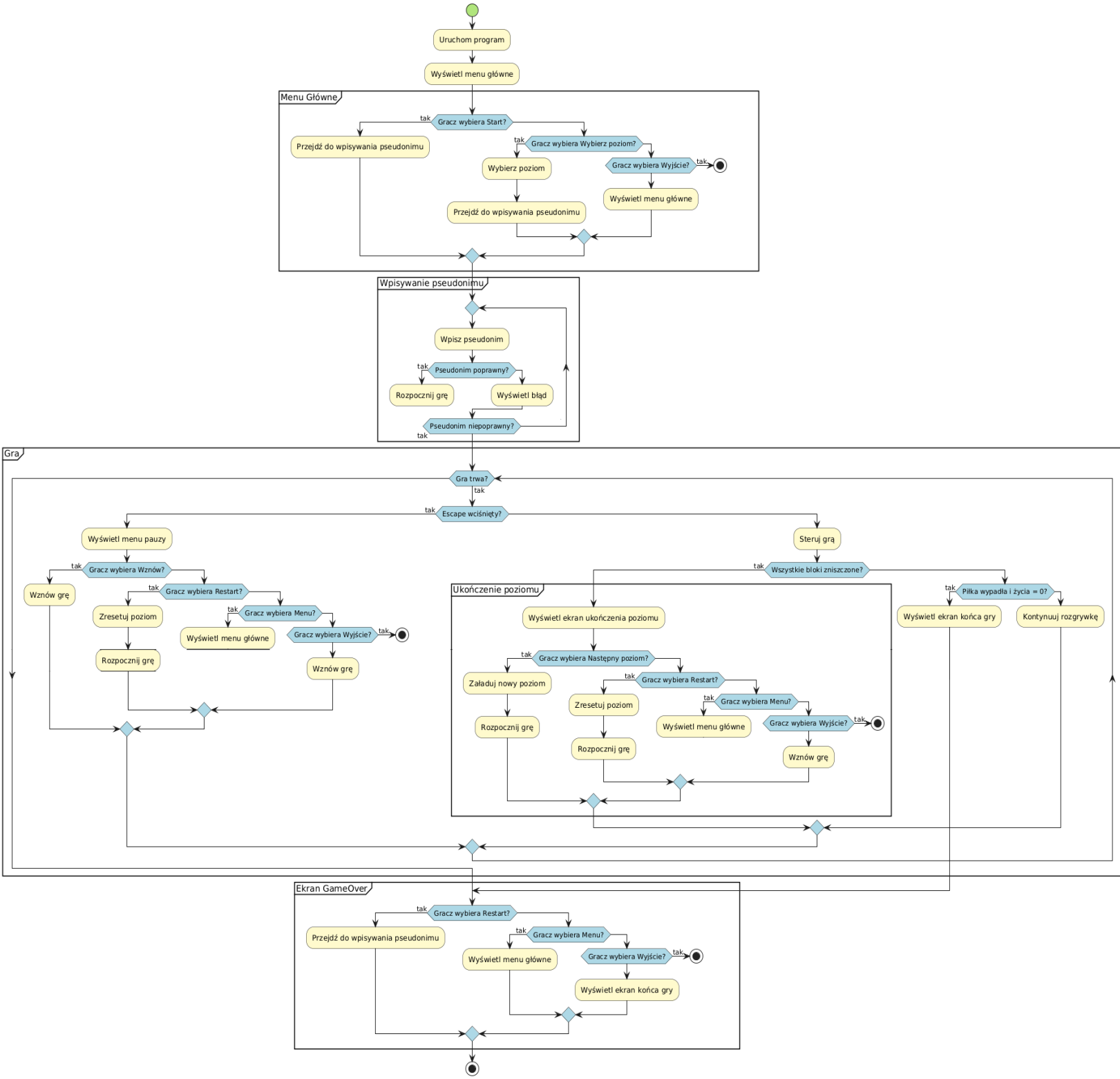
Po utracie wszystkich żyć ekran (assets/background2.png) wyświetla:

- „Game Over” (czerwony, Minecraft, 50 pt).
- „Your Score: X” (biały, 30 pt).
- Ranking (do 3 wyników: pseudonim, punkty).
- Przyciski: „Restart” (nowa gra od pseudonimu), „Main Menu”, „Exit”.

Wynik zapisywany jest w players/nickname.txt i assets/highscores.txt.

Rys.9. Ekran końca gry

Schemat działania program



Rys.10 Schemat działania programu.

Specyfikacja wewnętrzna

Klasy

1. **GameState** (abstrakcyjna, game_state.ixx)
   - **Rola**: Interfejs dla stanów gry (menu, rozgrywka, pauza, koniec gry, pseudonim).
   - **Powiązania**: Używana przez GameStateManager. Pochodne: MenuState, PlayingState, GameOverState, NicknameInputState.
   - **Pola**: Brak, tylko czysto wirtualne metody.
   - **Metody**:
     - handleEvents(sf::RenderWindow&, sf::Event&): Obsługuje zdarzenia (klawisze, mysz).
     - update(float deltaTime): Aktualizuje stan.
     - draw(sf::RenderWindow&): Rysuje stan.
2. **GameStateManager** (game_state_manager.cpp)
   - **Rola**: Koordynuje stany gry, przechowuje pseudonim i poziom.
   - **Powiązania**: Współpracuje z GameState i jego pochodnymi.
   - **Pola**:
     - std::unique_ptr<GameState> currentState: Aktualny stan.
     - std::string currentNickname: Pseudonim gracza.
     - int currentLevel: Numer poziomu.
     - sf::RenderWindow& window: Okno gry (w kodzie implikowane).
   - **Metody**:
     - setState<T, Args...>: Szablonowa metoda tworząca nowy stan.
     - setNickname(const std::string&): Ustawia pseudonim.
     - setcurrentLevel(const int&): Ustawia poziom.
     - handleEvents, update, draw: Przekazują akcje do currentState.
3. **GameObject** (abstrakcyjna, game_object.ixx)
   - **Rola**: Interfejs dla obiektów gry (paletka, piłka, bloki, bonusy).

- **Powiązania**: Używana przez GameObjectManager. Pochodne: Paddle, Ball, Block, Bonus.
- **Pola**: Brak, tylko czysto wirtualne metody.
- **Metody**:
    - update(float deltaTime), draw(sf::RenderWindow&), getBounds(), isDestroyed().

4. **GameObjectManager** (game_object_manager.cpp)
    - **Rola**: Zarządza obiektami gry, wczytuje poziomy, aktualizuje i rysuje.
    - **Powiązania**: Współpracuje z CollisionSystem, PlayingState, Config.
    - **Pola**:
        - std::vector<std::unique_ptr<GameObject>> objects: Lista obiektów.
        - int score: Wynik gracza.
    - **Metody**:
        - add(std::unique_ptr<GameObject>): Dodaje obiekt.
        - reset(float paddleWidth, float ballSpeed, const std::vector<std::string>&): Inicjuje poziom.
        - getPaddle(), getBall(), getBlocks(), getBonuses(), getBalls(): Zwracają obiekty.
        - allDestructibleBlocksDestroyed(): Sprawdza zniszczenie bloków.
5. **PlayingState** (playing_state.cpp)
    - **Rola**: Zarządza rozgrywką, pauzą, ukończeniem poziomu i kolizjami.
    - **Powiązania**: Używa GameObjectManager, CollisionSystem, AudioManager, HighScoreManager.
    - **Pola**:
        - GameObjectManager gameObjects: Obiekty gry.
        - CollisionSystem collisionSystem: System kolizji.
        - int lives: Liczba żyć (domyślnie 3).
        - int currentLevel, maxLevel: Poziom i liczba poziomów.
        - sf::Text scoreText, livesText, pauseText, levelCompleteText, ...: Interfejs.

    - LevelConfig config: Konfiguracja poziomu.
  - **Metody**:
    - handleEvents: Obsługuje klawisze (Escape) i mysz (pauza, ukończenie poziomu).
    - update: Aktualizuje grę, sprawdza kolizje i warunki końca.
    - draw: Rysuje planszę, interfejs, pauzę lub ekran ukończenia.
6. **Paddle** (paddle.cpp)
  - **Rola**: Paletka sterowana przez gracza.
  - **Powiązania**: Interaguje z Ball i Bonus.
  - **Pola**:
    - sf::RectangleShape shape: Grafika (niebieski prostokąt).
    - float defaultWidth, extendTimer: Szerokość i czas bonusu.
    - sf::Vector2f velocity: Prędkość (dla kolizji).
    - bool isExtended: Stan rozszerzenia.
  - **Metody**:
    - update: Przesuwa paletkę (klawisze Left/Right, prędkość 400 pikseli/s).
    - extend(float factor): Rozszerza paletkę (10 s).
    - resetWidth: Przywraca domyślną szerokość.
7. **Ball** (ball.cpp)
  - **Rola**: Piłka odbijająca się od obiektów.
  - **Powiązania**: Interaguje z Paddle, Block, CollisionSystem.
  - **Pola**:
    - sf::Sprite sprite: Grafika (assets/ball.png).
    - sf::Vector2f velocity: Prędkość.
    - float speed: Wartość prędkości.
    - bool outOfBounds: Czy wypadła poza ekran.
  - **Metody**:

- update: Aktualizuje pozycję, sprawdza kolizje z krawędziami.
- bounceFromPaddle: Odbija od paletki z kątem zależnym od miejsca trafienia.
- bounceFromBlock: Odbija od bloku na podstawie strony kolizji.
- normalizeVelocity: Utrzymuje stałą prędkość.
- increaseSpeed(float factor): Zwiększa prędkość (np. 1.2x).

8. **Block** (abstrakcyjna, block.cpp)
   - **Rola**: Bazowa klasa dla bloków.
   - **Powiązania**: Interaguje z Ball, CollisionSystem.
   - **Pola**:
     - sf::RectangleShape shape: Grafika.
     - bool destroyed: Stan zniszczenia.
   - **Metody**:
     - hit(): Reakcja na trafienie (wirtualna).
9. **NormalBlock, IndestructibleBlock, BonusBlock** (block.cpp)
   - **Rola**: Specjalizacje bloków.
   - **Pola** (NormalBlock):
     - int hitsRemaining: Liczba trafień (1).
   - **Metody**:
     - hit: NormalBlock niszczy się po 1 trafieniu, IndestructibleBlock ignoruje, BonusBlock niszczy się i generuje bonus.
     - isBonusBlock (BonusBlock): Zwraca true.
10. **Bonus** (bonus.cpp)
    - **Rola**: Bonusy spadające po zniszczeniu BonusBlock.
    - **Powiązania**: Interaguje z Paddle przez CollisionSystem.
    - **Pola**:
      - BonusType type: Enum (ExtendPaddle, MultiBall, SpeedUp).
      - sf::RectangleShape shape: Grafika (kolory: cyjan, magenta, czerwony).

- bool collected: Czy zebrany.

- **Metody**:
  - update: Przesuwa bonus w dół (100 pikseli/s).
  - collect: Aktywuje efekt (rozszerzenie, nowa piłka, przyspieszenie).

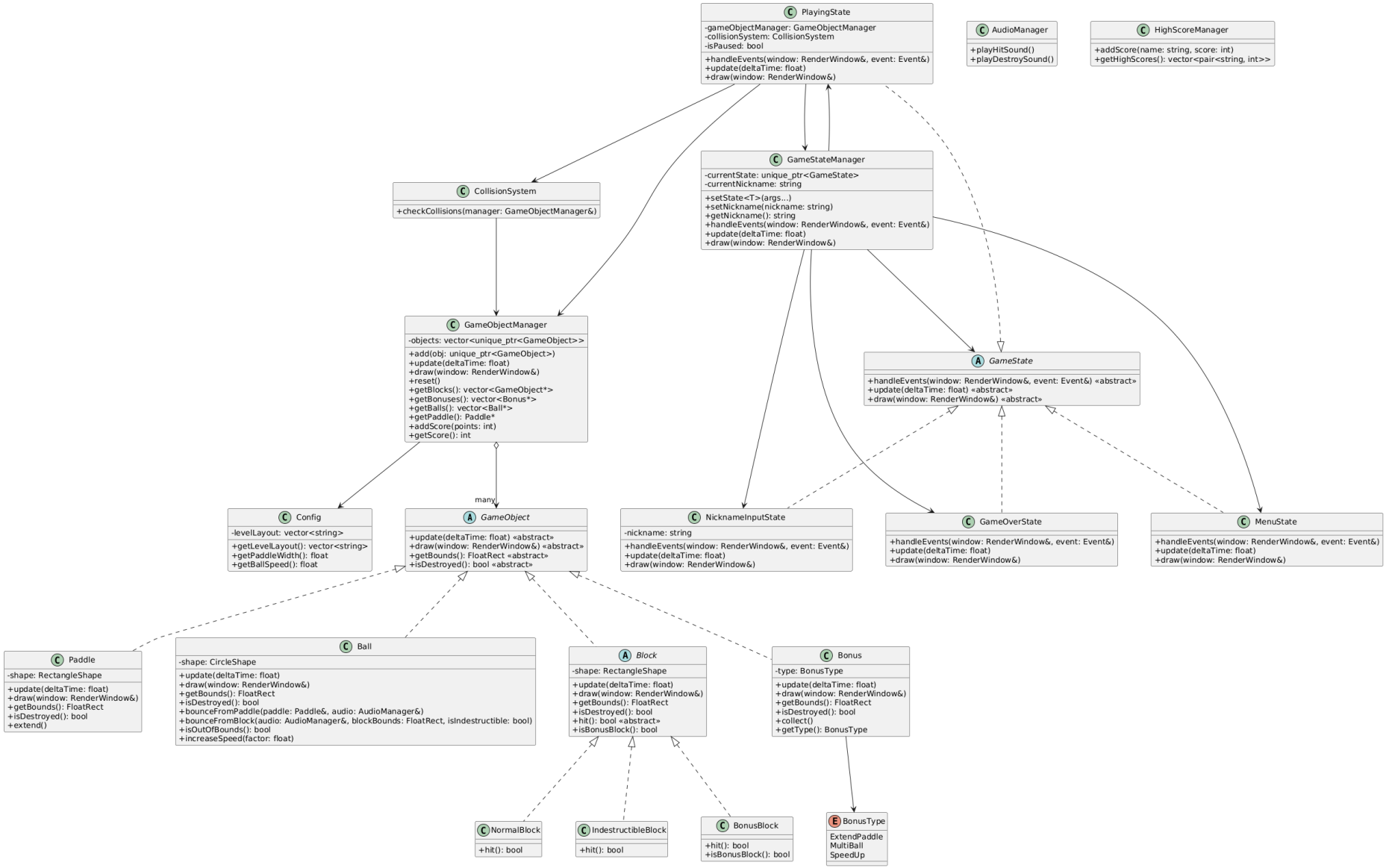
11. **CollisionSystem** (collision_system.cpp)
    - **Rola**: Wykrywa i obsługuje kolizje.
    - **Powiązania**: Używa GameObjectManager, AudioManager.
    - **Pola**:
      - AudioManager& audioManager: Dla dźwięków.
    - **Metody**:
      - checkCollisions: Sprawdza kolizje piłka–paletka, piłka–blok, bonus–paletka.
12. **HighScoreManager** (highscore.cpp)
    - **Rola**: Zarządza wynikami, zapisuje w players/nickname.txt i assets/highscores.txt.
    - **Pola**:
      - std::vector<std::pair<std::string, int>> highScores: Ranking (max 10).
      - std::string highScoreFile, playersDir: Ścieżki.
    - **Metody**:
      - addScore: Dodaje wynik, aktualizuje ranking.
      - isNicknameUnique: Sprawdza unikalność pseudonimu.
      - load/saveHighScores: Wczytuje/zapisuje wyniki.
      - addPlayerScore: Zapisuje wynik gracza z numerem poziomu.
13. **AudioManager** (audio.cpp)
    - **Rola**: Odtwarza muzykę i dźwięki.
    - **Pola**:
      - sf::Music backgroundMusic: assets/background.wav.
      - sf::Sound hitSound, destroySound: assets/hit.wav, assets/destroy.wav.

- **Metody**:
    - playBackgroundMusic, playHitSound, playDestroySound.

14. **Config** (config.cpp)
    - **Rola**: Wczytuje konfigurację poziomu.
    - **Pola**:
        - LevelConfig: Struktura z paddleWidth, ballSpeed, std::vector<std::string> levelLayout.
    - **Metody**:
        - loadLevelConfig: Parsuje plik levelX.txt z regex.
15. **MenuState** (menu_state.cpp)
    - **Rola**: Menu główne i wybór poziomu.
    - **Pola**:
        - sf::Text startText, selectLevelText, highscoreText, exitText: Przyciski menu.
        - std::vector<sf::Text> levelTexts: Przyciski poziomów.
        - bool isSelectingLevel: Tryb wyboru poziomu.
        - int maxLevel: Liczba poziomów.
    - **Metody**:
        - handleEvents: Obsługuje mysz.
        - updateHighscoreText: Aktualizuje ranking.
16. **GameOverState** (game_over_state.cpp)
    - **Rola**: Ekran końca gry.
    - **Pola**:
        - sf::Text gameOverText, scoreText, highscoreText, restartText, ...: Interfejs.
        - int finalScore: Końcowy wynik.
    - **Metody**:
        - updateHighscoreText: Aktualizuje ranking.
17. **NicknameInputState** (nickname_input_state.cpp)

- **Rola**: Wpisywanie pseudonimu.
- **Pola**:
  - std::string nickname: Pseudonim.
  - sf::Text promptText, nicknameText, errorText: Interfejs.
  - int startLevel: Poziom startowy.
- **Metody**:
  - handleEvents: Przetwarza wpisywanie i Enter.
  - update: Aktualizuje tekst pseudonimu.

Diagram hierarchii klas



Struktury danych

- std::vector<std::unique_ptr<GameObject>> (GameObjectManager): Przechowuje obiekty gry, z bezpiecznym zarządzaniem pamięcią.
- std::vector<std::pair<std::string, int>> (HighScoreManager): Ranking wyników (pseudonim, punkty), max 10 wpisów.
- LevelConfig (Config): Struktura z paddleWidth (float), ballSpeed (float), std::vector<std::string> levelLayout (układ bloków: „1" – zwykły, „2" – niezniszczalny, „3" – bonusowy).
- sf::Vector2f (Ball, Paddle): Prędkość i pozycja.
- std::vector<sf::Text> (MenuState): Przyciski wyboru poziomu.

Algorytmy

- **Wykrywanie kolizji** (CollisionSystem):
  - Metoda przecięcia prostokątów (sf::FloatRect::intersects).
  - Dla piłka–blok: Oblicza minimalne przecięcie (lewo, prawo, góra, dół) i odwraca odpowiednią składową prędkości:

$$\text{velocity}_x = -\text{velocity}_x \quad \text{(kolizja boczna)}$$

$$\text{velocity}_y = -\text{velocity}_y \quad \text{(kolizja pionowa)}$$

  - Dla piłka–paletka: Kąt odbicia zależy od miejsca trafienia:

$$\text{velocity}_x = (\text{hitRatio} - 0.5) \cdot \text{speed} \cdot 1.5$$

    gdzie hitRatio to pozycja trafienia na paletce (0–1).

- **Normalizacja prędkości** (Ball):

$$\text{velocity} = \text{speed} \cdot \frac{\text{velocity}}{\sqrt{\text{velocity}_x^2 + \text{velocity}_y^2}}$$

- **Parsowanie poziomu** (Config):
- **Sortowanie wyników** (HighScoreManager):

```
std::sort(highScores.begin(), highScores.end(), [](const auto&a, const auto&b) {return a.second > b.second; })
```

- **Losowe generowanie bonusów** (CollisionSystem):

```
std::uniform_int_distribution<> dis(0, 2); BonusType type = static_cast<BonusType>(dis(gen));
```

Wykorzystane techniki obiektowe

- **Hermetyzacja**: Prywatne pola w Ball, Paddle, Block, dostępne przez metody (np. Ball::getBounds).
- **Dziedziczenie**: NormalBlock, IndestructibleBlock, BonusBlock dziedziczą po Block; stany gry po GameState.
- **Polimorfizm**: Wirtualne metody hit w Block, update/draw w GameObject i GameState.
- **Wzorzec State**: GameStateManager przełącza stany gry.
- **Wzorzec Fabryki**: GameObjectManager::add tworzy obiekty dynamicznie.
- **Wzorzec Strategii**: Bonus::collect implementuje różne efekty w zależności od BonusType.
- **Zarządzanie pamięcią**: std::unique_ptr w GameObjectManager i GameStateManager.

Użyte biblioteki zewnętrzne

SFML:

- sf::RenderWindow: Okno gry (800x600, 120 FPS).
- sf::RectangleShape, sf::Sprite: Grafika paletki, bloków, piłki, bonusów.
- sf::Text, sf::Font: Interfejs (Minecraft, assets/arial.ttf).
- sf::Music, sf::Sound: Dźwięki (assets/*.wav).
- sf::Event: Wejście (klawisze, mysz).

C++ Standard Library:

- std::vector, std::unique_ptr: Zarządzanie obiektami.
- std::filesystem: Pliki wyników i poziomów.
- std::ranges: Filtrowanie w GameObjectManager i CollisionSystem.
- std::regex: Walidacja pseudonimów.
- std::random: Losowanie bonusów.

Zaawansowane techniki programowania

- **Moduły C++20**: Każdy plik (ball.cpp, paddle.cpp, itd.) definiuje moduł (np. module ball;), eksportując interfejsy.
- **Zakresy** (std::ranges):

```
objects | std::ranges::views::filter([](const auto&obj) {return !obj->isDestroyed() &&obj->getBounds().top < 600; })
```

  w GameObjectManager::draw i CollisionSystem.

- **Regex** (std::regex):

  ^[A-Z][A-Za-z0-9]{0,9}$

  w NicknameInputState dla pseudonimów.

- **System plików** (std::filesystem): Tworzenie katalogu players/, wczytywanie levelX.txt i nickname.txt.
- **Szablony**: GameStateManager::setState<T, Args...> dla dynamicznych stanów.
- **Losowość**: std::mt19937 i std::uniform_int_distribution dla bonusów.

Przepływ gry

Gra startuje w MenuState, gdzie gracz wybiera „Start Game" (poziom 1), „Select Level" (1–maxLevel) lub „Exit". Wybór gry prowadzi do NicknameInputState, gdzie pseudonim jest walidowany i zapisywany. Następnie PlayingState wczytuje poziom z levelX.txt, inicjuje paletkę, piłkę i bloki. Gracz steruje paletką, a CollisionSystem zarządza kolizjami. Escape aktywuje pauzę z opcjami „Resume", „Restart", „Main Menu", „Exit". Zniszczenie wszystkich bloków przełącza na ekran ukończenia poziomu z „Next Level" (jeśli dostępny). Utrata żyć prowadzi do GameOverState, zapisując wynik i oferując restart lub powrót do menu.

Testowanie i uruchamianie

- **Środowisko testowe**: Visual Studio 2022, C++20, SFML 2.6.1, Windows 11
- **Testy jednostkowe**:
  - **Walidacja pseudonimów**: Sprawdzono regex ^[A-Z][A-Za-z0-9]{0,9}$ i unikalność w NicknameInputState.
  - **Kolizje**: Przetestowano odbicia piłki od paletki (kąty), bloków (minimalne przecięcie) i krawędzi.
  - **Bonusy**: Zweryfikowano rozszerzenie paletki (10 s), multi-ball (nowa piłka z assets/bonus_ball.png), przyspieszenie (1.2x).
  - **Wyniki**: Sprawdzono zapis w players/nickname.txt (format „Level X: Y") i highscores.txt, sortowanie rankingu.

  - **Poziomy**: Przetestowano wczytywanie levelX.txt z różnymi układami i parametrami.

- **Wykryte błędy**: Błąd wczytywania pustych wierszy w levelX.txt – poprawiono w loadLevelConfig.
  Błąd kolizji podczas ruchu paletki przy bokach paletki (piłka wchodziła w bok paletki) – poprawiono system kolizji w Collisionsystem.

- **Uruchamianie**: Wymaga SFML 2.6.1, C++20, zasobów (assets/*.png, *.wav, *.ttf) w katalogu projektu.

Uwagi i wnioski

- **Zalety**:
  - Modułowa architektura (C++20 modules) ułatwiła organizację kodu.
  - std::filesystem zapewniło niezawodne zarządzanie plikami.
  - Wzorce projektowe (State, Fabryka) umożliwiły łatwą rozbudowę.
- **Perspektywy rozwoju**:
  - Dodanie smugi za piłką.
  - Cząsteczki przy zniszczeniu bloków.
  - Nowe bonusy, np. magnes lub spowolnienie piłki.
  - Sterowanie myszą dla paletki.
- **Wnioski**: Projekt umożliwił praktyczne zastosowanie C++20, SFML i wzorców projektowych. Znacząco ułatwiło to rozbudowę i utrzymanie kodu. Gra działa stabilnie, a jej mechanika jest intuicyjna dla użytkowników. W przyszłości można rozważyć dodanie dodatkowych funkcjonalności.