AKADEMIA GÓRNICZO-HUTNICZA

Dokumentacja do projektu

Silnik do gier w terminalu z odwzorowaną grą TETRIS

z przedmiotu

Języki programowania obiektowego

Elektronika i Telekomunikacja (stacjonarnie), III rok

Adrian Bik

Grupa VI, Czwartek 15:00

prowadzący: Rafał Frączek

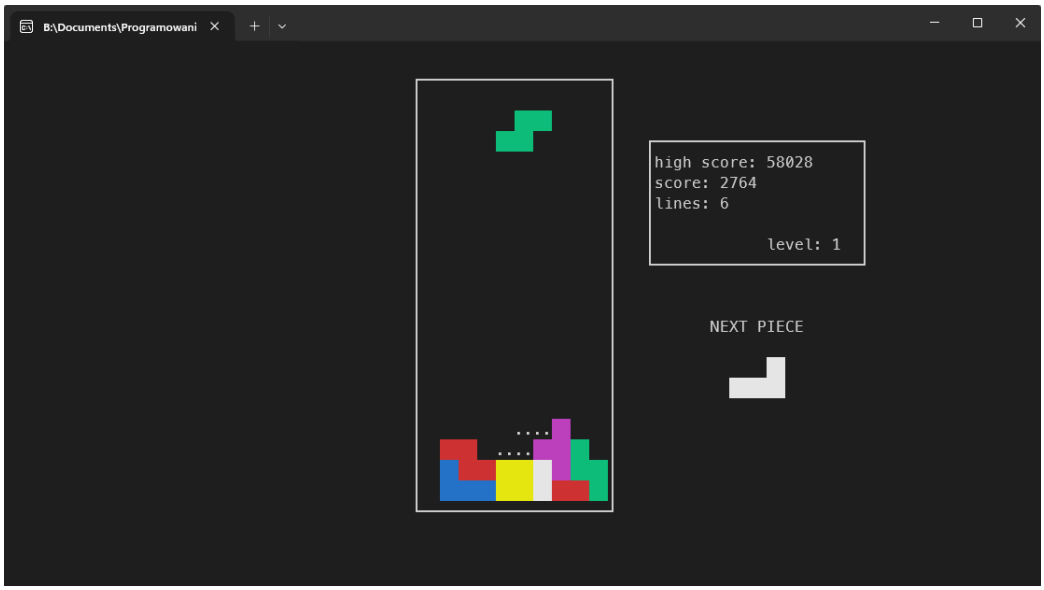
22.01.2026

1. Opis projektu

Silnik do tworzenia gier – modularny system pozwalający na wymianę fragmentów odpowiedzialnych za input gracza, logikę gry oraz generowanie prostych efektów dźwiękowych. Jest podzielony na część logiczną, która przechowuje i przetwarza dane na temat każdego elementu gry oraz część wizualną (renderer), która dba o wyświetlanie ich na ekranie. Posiada on również system odczytu/zapisu i osobny wątek pilnujący czasu. Projekt obejmuje również stworzenie w tym silniku własnego odwzorowania gry TETRIS z dbałością o szczegóły. /

A game engine for creating video-games – a modular system that allows swapping components responsible for player input, game logic, and the generation of simple sound effects. It is divided into a logical part, which stores and handles data about every game element, and a visual part (renderer), which handles displaying them on the screen. It also includes a read/write system and a separate thread responsible for time management. The project also involves creating a custom implementation of the game TETRIS within this engine, with attention to detail.

Rysunek 1 - Wygląd menu głównego gry



Rysunek 2: Wygląd okna gry

Instrukcja użytkownika

(^, <, >, v – odpowiadają strzałkom na klawiaturze)

a) Menu Główne / menu ustawień

Po włączeniu programu jest widoczne menu główne / menu ustawień. Podczas gdy w nich przebywamy:

^ / W – góra

v / S – dół

< / A – dekrementacja wartości

> / D – inkrementacja wartości

[enter] – wybór / toggle zaznaczonego elementu

[esc] – wyjście z programu / cofnięcie do menu głównego

Gracz ma do wyboru trzy opcje:

1. Start Game - rozpoczęcie gry
2. Settings – wejście do menu ustawień, tam gracz może je edytować, zostają one automatycznie zapisywane po wyjściu z niego za pomocą opcji Back lub klawisza [esc]
3. Exit – wyjście z gry

Dostępne ustawienia gry:

1. **ASCII mode** – tryb w którym zamiast kolorów, do renderowania są wykorzystywane znaki ASCII.
2. **Flashing effects** – włączca / wyłącza efekty z migającym obrazem, które u niektórych graczy mogą wywołać epilepsję.
3. **Randomness** – pozwala wybrać model losowości, którego gra używa do losowania nowych figur:

   I) **TGM3** (z gry TETRIS: The Grand Master 3) - Bardziej sprawiedliwy, oferuje ciekawszą rozgrywkę.

   II) **pure** - Prawdziwy rozkład równomierny, w którym każda figura jest tak samo prawdopodobna.

4. **Show future piece** – wyświetla z boku ekranu, jaka figura będzie następna. Nie jest to oficjalna mechanika TETRIS, ale pozwala ona na bardziej złożone strategie i urozmaica ona grę.

5. **Start Level** – Pozwala wybrać początkowy poziom.

b) Gameplay

Podczas gry sterowanie wygląda następująco:

< / A – przesunięcie figury o jedno pole w lewo

> / D – przesunięcie figury o jedno pole w prawo

v / S – soft drop (przyspieszenie opadania w dół o jedno pole)

^ / W – hard drop (zatwierdzenie położenia, natychmiastowe postawienie figury)

Q / J – obrót figury zgodnie z kierunkiem ruchu wskazówek zegara

E / K – obrót figury w kierunku przeciwnym do ruchu wskazówek zegara

[esc] – poddanie gry

Zasady gry:

Celem gry Tetris jest zdobywanie punktów poprzez układanie spadających figur (tetrominów) w taki sposób, aby wypełniały one całe poziome linie planszy. Wypełnione linie są usuwane, a gracz otrzymuje za to punkty. Możliwe jest usunięcie kilku linii jednocześnie. Gra trwa do momentu, gdy nie ma już miejsca na wprowadzenie nowego klocka.

Przebieg rozgrywki:

Na planszy pojawia się figura, która automatycznie opada w dół pod wpływem grawitacji. Gracz może sterować figurą w trakcie jego spadania, przesuwając go, obracając lub przyspieszając jego opadanie. Po zetknięciu się tetromina z innymi elementami planszy lub z jej dolną krawędzią, zostaje on zablokowany i wchodzi w skład planszy. Następnie na planszy pojawia się kolejna figura.

Poziomy i prędkość gry:

Gra posiada system poziomów, który zwiększa się po usunięciu określonej liczby linii. Każdy kolejny poziom powoduje zwiększenie prędkości opadania klocków, co znacząco podnosi poziom trudności. Maksymalny poziom jest ograniczony.

*Ilość linii wymagana do przejścia do kolejnego poziomu = Obecny poziom * 10*

Punktacja:

Punkty przyznawane są za:

I) Przyspieszone opadanie klocków - im większa odległość upadku, tym więcej punktów.

Soft drop: *+1 * obecny poziom*

Hard drop: *+2 * obecny poziom*

II) Czyszczenie linii - liczba punktów zależy od:

- ilości linii usuniętych jednocześnie - *kolejno 100pkt, 400pkt, 900pkt, 2000pkt * obecny poziom*
- Perfect Clear - jeśli po usunięciu linii plansza jest całkowicie pusta, zdobyte punkty są mnożone dziesięciokrotnie.

Element podglądowy (ghost piece):

Gra wyświetla tzw. ghost piece - półprzezroczysty cień aktualnego tetromina. Pokazuje on miejsce, w którym figura wyląduje po jej zatwierdzeniu.

Zakończenie gry (Game Over):

Gra kończy się, gdy:

I) Nowa figura wystaje częściowo poza górną krawędź planszy.

II) Figura nie może zostać wprowadzona, ponieważ górne rzędy planszy są już zajęte.

III) Tetromino podglądowe nachodzi na aktywną figurę.

Po spełnieniu jednego z powyższych warunków wyświetlany jest stan Game Over, a rozgrywka zostaje zakończona.

2. Kompilacja

Do kompilacji projektu należy wpierw zainstalować bibliotekę PDCurses (dostępną za pomocą VCPKG) a następnie skompilować go z użyciem MSVC dla języka C++ 20. Biblioteka json jest załączona w plikach źródłowych.

Program zadziała tylko w systemie Windows.

3. Pliki źródłowe

Projekt składa się z następujących plików źródłowych:

- *Main.cpp* – Plik będący punktem wejścia programu.
- *GameEngine.h*, *GameEngine.cpp* – deklaracja oraz implementacja klasy `GameEngine`.
- *GameRenderer.h*, *GameRenderer.cpp* – deklaracja oraz implementacja klasy `GameRenderer`.
- *TimeManager.h*, *TimeManager.cpp* – deklaracja oraz implementacja klasy `TimeManager`.
- *InputManager.h*, *InputManager.cpp* – deklaracja oraz implementacja klasy `InputManager`.
- *WindowManager.h*, *WindowManager.cpp* – deklaracja oraz implementacja klasy `WindowManager`.
- *SaveSystem.h*, *SaveSystem.cpp* – deklaracja oraz implementacja funkcji `loadState()` i `saveState()` do odczytu i zapisu danych programu do pliku.
- *Tetromino.h*, *Tetromino.cpp* – deklaracja oraz implementacja klasy `Tetromino`.
- *SoundFX.h*, *SoundFX.cpp* – deklaracja oraz implementacja funkcji do generowania efektów dźwiękowych.
- *GameLogic.cpp* – implementacja zasad gry TETRIS.
- *UIElements.cpp* – implementacja funkcji `makeX()`, `initX()`, `refreshX()` oraz `xLogic()` składających się na wygląd, kształt i zachowanie poszczególnych okien interfejsu.
- *Events.h* – deklaracja enum `Event` wykorzystywanego do komunikacji podsystemów z silnikiem gry asynchronicznie.
- *Settings.h* – deklaracja klasy `GameSettings`.
- *Constants.h* – deklaracja stałych preprocesora, które definiują właściwości gry oraz przydatnych makr.
- *GameState.h* – deklaracja i implementacja klasy `GameState` przechowującej stan silnika oraz gry.
- *CursesWinCompat.h* – deklaracja pomocniczego pliku nagłówkowego rozwiązującego konflikty bibliotek `curses.h` oraz `windows.h` (jest załączany w programie w zamian za `curses.h`)

W projekcie wykorzystano następujące dodatkowe biblioteki:

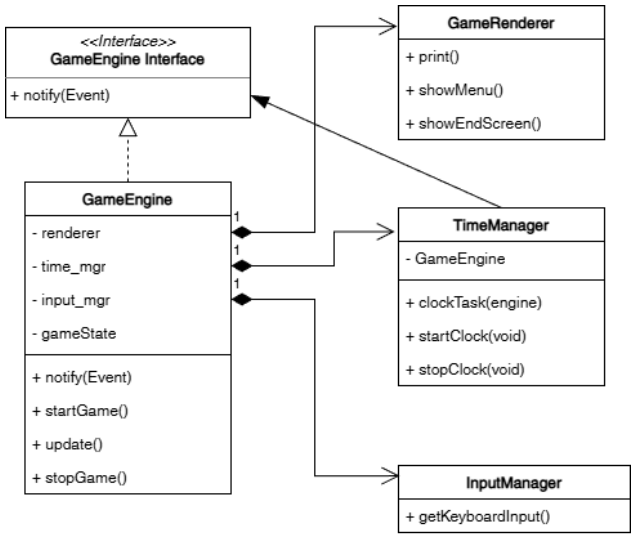
- PDCurses – Port legendarnej biblioteki curses.h do zarządzania oknami w programach z interfejsem tekstowym na system operacyjny Windows.

  Strona internetowa: https://pdcurses.org/ https://github.com/wmcbrine/PDCurses

- json – biblioteka do parsowania plików .json.

  Strona internetowa: https://github.com/nlohmann/json

4. Opis klas



Rysunek 3: Diagram UML zależności między najważniejszymi klasami

W projekcie utworzono następujące klasy:

- `GameEngine` – klasa odpowiedzialna za przechowywanie odniesień do stanu gry i wszystkich modułów programu oraz zarządzanie ich wspólnymi zależnościami.
  - `void setTimeManager(unique_ptr<TimeManager> ptr)` – używana w procesie inicjalizacji silnika gry.
  - `void setGameRenderer(unique_ptr<GameRenderer> ptr)` – używana w procesie inicjalizacji silnika gry.
  - `void setInputManager(unique_ptr<InputManager> ptr)` – używana w procesie inicjalizacji silnika gry.
  - `void notify(const Event& event)` – używana do przekazywania asynchronicznych `Event`ów pomiędzy podsystemami.
  - `void update()` – blok kodu wykonywanego synchronicznie. Jest sercem programu, steruje wywoływaniem najważniejszych funkcji. Sam wykonuje się po otrzymaniu przez silnik `Event`u o kolejnym tiku zegara od `TimeManager`a co ustaloną ilość milisekund.
  - `void startEngine()` – rozpoczyna pracę silnika.
  - `void stopEngine()` – zakańcza pracę silnika.
  - `void gameLogic(const int& k_input)` – definiuje zachowanie gry TETRIS.
  - `void menuLogic(const int& k_input)` – definiuje zachowanie menu głównego.
  - `void settingsLogic(const int& k_input)` - definiuje zachowanie menu ustawień.
  - `void gameOver()` – Zakańcza grę, porównuje osiągnięty wynik z dotychczasowym rekordem i odświeża interfejs użytkownika.
  - `void restartGame()` – resetuje silnik i rozpoczyna grę od nowa.
  - `uint8_t TGM3_randomizer()` – zasady losowości z gry TGM3.
  - `uint8_t pure_randomizer()` – zasady losowości z równomiernym rozkładem.

- `Tetromino` – klasa odpowiedzialna za reguły poruszania się po planszy i definiująca wyglad wszystkich wariantów figur.
  - `void reset(uint8_t random_id)` – Resetuje klocek tetromino, przygotowując go do ponownego spadania
  - `void rotateR(span<const char> board)` - Obraca tetromino zgodnie z ruchem wskazówek zegara.
  - `void rotateL(span<const char> board)` - Obraca tetromino przeciwnie do ruchu wskazówek zegara.
  - `void moveR(span<const char> board)` - Przesuwa tetromino o jedno pole w prawo.
  - `void moveL(span<const char> board)` - Przesuwa tetromino o jedno pole w lewo.
  - `void ghost_drop(span<const char> board)` - Natychmiast opuszcza tetromino aż do ostatniej poprawnej pozycji. Używane wyłącznie do pozycjonowania podglądu figury.
  - `void soft_drop(span<char> board)` - Opuszcza tetromino o jeden blok i zwiększa wartość fall_dist.
  - `void hard_drop(span<char> board)` - Inicjuje sekwencję opadania. Opuszcza ona tetromino aż do ostatniej poprawnej pozycji, krok po kroku (tick po ticku).
  - `const char realize_piece(const int8_t x, const int8_t y)` - zwraca pole o podanych współrzędnych x,y w lokalnym układzie odniesienia tetromino. Stosuje odpowiednią transformację orientacji figury.
  - `const uint8_t get_piece_id()` – zwraca identyfikator tetromino.
  - `const uint8_t get_rotation()` - zwraca obecną orientację tetromino.
  - `void set_piece_id(const uint8_t id)` – modyfikuje identyfikator tetromino.
  - `void set_rotation(const uint8_t rot)` - modyfikuje orientację tetromino.
  - `void set_xpos(const int8_t pos)` - modyfikuje położenie tetromino w poziomie.
  - `void set_ypos(const int8_t pos)` - modyfikuje położenie tetromino w pionie.

- `WindowManager` – klasa, która zarządza całym ekosystemem PDCurses. Inicjalizuje i deinicjalizuje obiekty WINDOW oraz przechowuje do nich wskaźniki.
  - `void showBorder(const int& win_id)` – Wyświetla obramowanie danego okna bez czyszczenia jego zawartości.
  - `void clearBorder(const int& win_id)` – Czyści obramowanie danego okna bez czyszczenia jego zawartości.
  - `void clearContents(const int& win_id)` – Czyści zawartość danego okna bez usuwania jego obramowania.
  - `void clearWindow(const int& win_id)` – Czyści zarówno obramowanie, jak i zawartość danego okna.
  - `WINDOW* getWindow(const int& win_id)` – Zwraca wskaźnik do wybranego obiektu.

- `GameRenderer` – klasa odpowiedzialna za wyświetlanie danych silnika i elementów interfejsu na ekranie.

  Opisy funkcji typu refreshX() oraz initX() zostały pominięte, ponieważ zostaną opisane poniżej.

  - `void renderColorFrame()` – Renderuje zawartość GAME_WIN na podstawie aktualnego GameState przy użyciu kolorowych bloków
  - `void renderASCIIFrame()` - Renderuje zawartość GAME_WIN na podstawie aktualnego GameState przy użyciu grafiki ASCII
  - `void flashEffect()` – Wywołuje efekt migania ekranu, jeśli efekty migania są włączone
  - `void lineClearEffect(vector<uint8_t> lines, uint16_t score)` - Wyświetla efekt czyszczenia linii
  - `void windowPrint(const int& win_id, const string& str)` - Dopisuje wiadomość na końcu danego okna i wyświetla ją
  - `void windowPrintAtPos(const int& win_id, const int& x, const int& y, const string& str)` - Wyświetla wiadomość w danej pozycji w określonym oknie
  - `void windowReset(const int& win_id)` – Czyści zawartość danego okna
  - `void errPrint(const string& str)` – Wyświetla błąd w oknie ERR_WIN
  - `void showTitleScreen()` - Renderuje tytuł gry w ASCII
  - `void showEndScreen(const GameState& state)` – Renderuje ekran końcowy na podstawie GameState

- `TimeManager` – klasa, która uruchamia i zarządza wątkami odpowiedzialnymi za odmierzanie czasu. Wątki te generują powiadomienia wyzwalające określone akcje w GameEngine.
  - `void clockTask(std::stop_token stopToken, GameEngine* engine)` – działa w oddzielnym wątku i pilnuje czasu. Wysyła `Event::CLK` do silnika gry raz na `GAME_TICK` milisekund.
  - `void TimeManager::startClock()` – tworzy i uruchamia wątek zegara.
  - `void TimeManager::stopClock()` – zakańcza wątek zegara.

- `InputManager` – Klasa odpowiedzialna za pozyskiwanie inputu gracza z klawiatury.
  - `int getKeyboardInput()` – Nieblokujący input. Zwraca identyfikator klawisza który został wciśnięty przez gracza
  - `int waitForAnyKey()` – Blokujący input. Zwraca identyfikator klawisza który został wciśnięty przez gracza

- `GameState` – klasa przechowująca obecny stan calej gry.
  - `void reset()` – Resetuje stan gry aby przygotować go do nowej rundy.

- `GameSettings` – klasa przechowująca ustawienia gry, które są zczytywane / wczytywane do pliku.

5. Zasoby

W projekcie wykorzystywane są następujące pliki zasobów:

- `save_state.json` – plik zawierający ustawienia oraz historyczny rekord gry:
  - kolejnosć linii dowolna, plik musi jedynie mieć strukturę zgodną z formatem .json
  - hi_score – punktowy rekord gry (wartosć całkowita)
  - ascii_mode – wartosć prawda/ fałsz
  - flash_on_clear - wartosć prawda/ fałsz
  - show_future - wartosć prawda/ fałsz
  - pure_randomness - wartosć prawda/ fałsz
  - start_level - wartosć całkowita

6. Dalszy rozwój i ulepszenia

- Zwiększenie odrębności funkcji xLogic() od jakichkolwiek klas
- Stworzenie generycznej klasy okna, od której wszystkie inne mogłyby dziedziczyć
- Lepszy system układania okien względem siebie (np. z adaptacją do kształtu okna terminala)
- Oddelegowanie odtwarzanie efektów dźwiękowych do osobnego wątku
- Dodanie większej ilości efektów dźwiękowych i rodzajów dźwięku do odtworzenia
- Funkcje graficzne do renderowania rastrowego (poza terminalem)

7. Inne

Ten silnik do gier oferuje programiście gotowe systemy do logiki synchronicznej oraz asynchronicznej, pokazywania czegokolwiek na ekranie, pobierania inputów od gracza oraz wydawania dźwięków poprzez głośniki. Jest zaprojektowany tak, aby łatwo było edytować i rozszerzać funkcjonalność wykonywanych w nim gier.

Struktura gry:

Logika gry mieści się w pliku GameLogic.cpp

Programista ma do dyspozycji domyślnie ustawiony ErrorWindow, GameWindow oraz InputWindow, które może on używać do swoich celów.

Dodawanie nowych okien:

Dla każdego okna zaleca się zdefiniowanie kilku funkcji, które powinny być zdefiniowane w pliku `Uielements.cpp`. Są to `makeX()`, `refreshX()` oraz (opcjonalnie) `xLogic() i initX()`, gdzie `x` to nazwa okna.

`makeX()`: Należy do `WindowManager`a. Definiuje położenie i rozmiar okna oraz go tworzy.

`initX()`: Należy do `GameRenderer`a. Definiuje sposób w jaki okno jest inicjalizowane gdy zajdzie taka potrzeba (jak np. po zmianie stanu z gameplayu spowrotem do menu głównego, przed wpisaniem czegokolwiek do menu)

`refreshX()`: Należy do `GameRenderer`a. Definiuje wygląd okna i sposób wyświetlania zawartych w nim danych na ekranie.

`xLogic()`: Należy do `GameEngine`. Ma dostęp do stanu gry i silnika. Pozwala zdefiniować zachowanie okna i sprawić że jest ono interaktywne.

Następnie nowo zdefiniowane funkcje należy zadeklarować w odpowiadających im klasach.

Dodawanie nowych wydarzeń:

Wydarzenia są zupełnie do dyspozycji programisty. Można je wykorzystać w celu wykonywania zadań pomiędzy wywołaniami funkcji `GameEngine::update()`. Są zdefiniowane w pliku `Events.h`, wystarczy je tam dodawać.

Dodawanie nowych efektów dźwiękowych:

Można dodawać efekty dźwiękowe do plików `SoundFX.h / SoundFX.cpp`.