Imię i nazwisko Kamil Stokłosa Wojciech Rubacha Antoni Synowiec	**Kierunek** Informatyka Techniczna	**Rok studiów i grupa** 1 rok grupa 4
Data prezentacji 19.01.2026	**Nazwa projektu:** Nowoczesny zbijak	

1. Cel i opis projektu

Celem projektu było stworzenie dwuwymiarowej gry zręcznościowej dla dwóch graczy opartej na fizyce (physics-based), zrealizowanej w języku C++ przy wykorzystaniu biblioteki SFML. Głównym założeniem było opracowanie sprawnego systemu kolizji, który pozwala na realistyczne odbijanie się piłek od ścian poprzez odwracanie ich kierunku oraz na wzajemne oddziaływanie graczy i elementów otoczenia. Gra pozwala na lokalną rywalizację, której celem jest wyeliminowanie przeciwnika za pomocą piłek, przy jednoczesnej możliwości wykorzystania bonusów zmieniających parametry fizyczne, takie jak prędkość, wielkość obiektów czy tryb strzału. Nasz projekt zawiera także rozbudowane menu z obsługą nicków graczy i regulacją głośności muzyki. Cały proces tworzenia projektu był zarządzany przez system kontroli wersji Git na platformie GitHub, gdzie regularnie dodawaliśmy różne commity.

2. Zastosowane techniki programistyczne

- **Delta Time**
- **Programowanie Strukturalne**
- **Inheritance (Głównie struktur)**
- **Polimorfizm wizualny**
- **Automat wizualny (Finite – State machine)**

3. Wykorzystane biblioteki

- **SFML**

4. Wybrane funkcjonalności wraz z kodem i screenami z aplikacji przestawiającymi działanie tego fragmentu

A) Stworzyliśmy własny silnik fizyczny wewnątrz funkcji `_physics_process`. Obiekt posiada wektory prędkości (velocity) i tacia (accel, hard_accel itp) na podstawie których obicza swoje położenie. Zastosowaliśmy normalizację wektora kierunku, aby postać nie poruszała się szybciej po skosie.
Silnik fizyczny działa też używając parametru delta (odpowiadającego za Delta Time) .

```
void Object::_physics_process(float delta) {
```

```
float length = std::sqrt(direction.x * direction.x + direction.y * direction.y);
if (length != 0) {
direction.x /= length;
direction.y /= length;
last_direction = direction;
}
if (movable) {
velocity.y = direction.y * speed;
velocity.x = direction.x * speed;
sprite.move(velocity * delta);
hard_accel = -5 * max_speed;
}
else {
direction = { 0.f, 0.f };
speed = 0;
}
if (!max_speed_on) {
if (speed + used_accel * delta > 0.f) {
speed += used_accel * delta;
}
else {
speed = 0.f;
}
}
```

B) Stworzyliśmy też obsługę (8) kierunków dla gracza aby obracać sprite gracza w odpowiednim kierunku. System opera się na zmiennej direction na której bazuje też obliczanie velocity przez co można uniknąć błędów i obsługuje też kierunki ukośne.

```
void Object::rotations() {
if (abs(direction.x) > 0.9f) {
if (direction.x > 0) {
sprite.setRotation(sf::degrees(0.f));
}
if (direction.x < 0) {
sprite.setRotation(sf::degrees(180.f));
}
}
else if (abs(direction.y) > 0.9f) {
if (direction.y > 0) {
sprite.setRotation(sf::degrees(90.f));
}
if (direction.y < 0) {
sprite.setRotation(sf::degrees(270.f));
}
}
else if (abs(direction.y) > 0.3 and abs(direction.x) > 0.3) {
if (direction.y > 0) {
if (direction.x > 0) {
sprite.setRotation(sf::degrees(45.f));
}
if (direction.x < 0) {
```

```
sprite.setRotation(sf::degrees(135.f));
}
}
if (direction.y < 0) {
if (direction.x > 0) {
sprite.setRotation(sf::degrees(315.f));
}
if (direction.x < 0) {
sprite.setRotation(sf::degrees(225.f));
}
}
}
}
```

C) Stworzyliśmy też obsługę imputów dla graczy, oraz funkcjonalności z tym związane takie na przykład jak: (Object Pooling) przy wystrzeliwaniu pocisków gracz przeszukuje listę pocisków i szuka tych dostępnych. Co pozwala przypisać daną ilość pocisków na początku gry.

```
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::C)) {
if (gracz.shot_cooldown <= 0.f) {
int ile_pociskow;
if (gracz.multishot == true) {
ile_pociskow = 2;
}
else {
ile_pociskow = 1;
}
int wystrzelone = 0;
for (int i = 0; i < max_bullets; i++) {
if (bullets[i].visible == false) {
bullets[i].visible = true;
...
}
```

D) Wykrywanie zderzeń w naszej grze opiera się na analizie prostokątnych obramowań obiektów. Wykorzystaliśmy do tego gotowe mechanizmy biblioteki SFML, wykorzystując przy tym 2 funkcje: **getGlobalBounds()** - służy

ona do pobrania aktualnego położenia i rozmiaru prostokątnej ramki otaczającej dany sprite, **findIntersection()** - służy ona do sprawdzania czy ramki dwóch obiektów nachodzą na siebie. Jeśli tak, zwraca ona precyzyjne informacje o rozmiarze obszaru ich styku - części wspólnej.

- Mechanika odbijania pocisków od ścian:

Najważniejszym elementem kolizji ze ścianą jest wykrycie, z której strony nastąpiło uderzenie. Dzięki wspomnianej funkcji findIntersection otrzymujemy prostokąt kolizji i porównujemy jego szerokość i wysokość. Jeśli size.x < size.y - zderzenie nastąpiło z boku ściany (oś X). Jeśli size.x>size.y - zderzenie nastąpiło z góry lub z dołu ściany (oś Y). W momencie wykrycia kolizji, odpowiednia składowa wektora kierunku jest mnożona przez wartość -1.0f. Operacja ta zmienia zwrot wektora na przeciwny, zachowując jego pęd. Aby pocisk nie utknął w ścianie zastosowaliśmy wypchanie go o wartość pole_kolizji->size oraz dodatkowy ruch wypychający go o 2 piksele w nowym kierunku.

```
auto pole_kolizji =
bullets[i].sprite.getGlobalBounds().findIntersection(walls[j].sprite.getGlobalBound
s());
    //zmienna przechowuje mały prostokąt, czyli wspolny obszar piłki i ściany gdy
zachodzi kolizja


    if (pole_kolizji) { //doszło do zderzenia
        if (pole_kolizji->size.x < pole_kolizji->size.y) { //uderzenie z boku
(węższe pole kolizji)
            bullets[i].direction.x *= -1.0f; // odwracanie kierunku poziomego
            //wypchnięcie pocisku poza ścianę o szerokość kolizji
            if (bullets[i].sprite.getPosition().x >
walls[j].sprite.getPosition().x) {
                bullets[i].sprite.move({ pole_kolizji->size.x, 0.0f }); //
przesuwanie piłki poza ścianę
            }
            else {
                bullets[i].sprite.move({ -pole_kolizji->size.x, 0.0f });
            }
        }
        else { //uderzenie z góry lub dołu
            bullets[i].direction.y *= -1.0f;
            //wypchnięcie w pionie o wysokość kolizji
            if (bullets[i].sprite.getPosition().y >
walls[j].sprite.getPosition().y) {
                bullets[i].sprite.move({ 0.0f , pole_kolizji->size.y }); //
przesuwanie piłki poza ścianę
            }
            else {
                bullets[i].sprite.move({ 0.0f , -pole_kolizji->size.y });
            }
        }
        bullets[i].sprite.move(bullets[i].direction * 2.0f);
    }
```

- Kolizja między dwoma graczami:

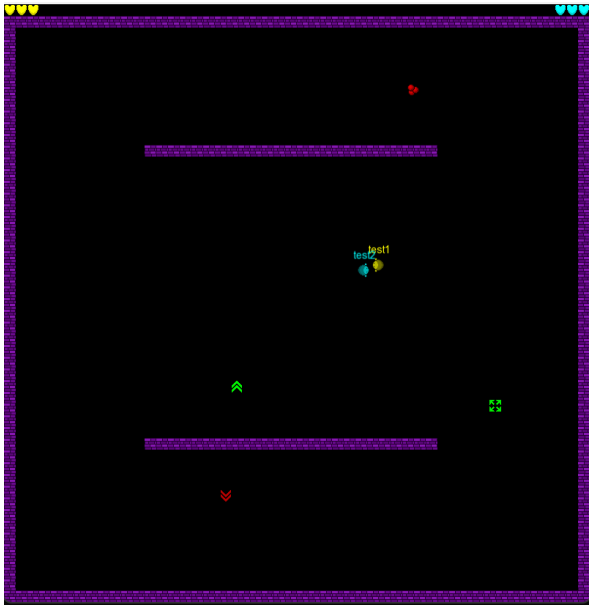
W przypadku zderzenia dwóch ruchomych obiektów, system musi zadbać o ich wzajemne rozdzielenie, aby postacie się nie przenikały. Funkcja findIntersection pozwala wyznaczyć obszar wspólny obu graczy, następnie jego wymiary są porównywane (size.x i size.y) i następuje określenie płaszczyzny zderzenia. W przeciwieństwie do kolizji ze ścianą, oba obiekty są wypychane jednocześnie. Każdy z graczy zostaje przesunięty o połowę głębokości zderzenia (size/2.0f) w przeciwnym kierunku. Dodatkowo, w momencie zderzenia prędkość w osi kolizji jest zerowana (velocity=0), co zapobiega nienaturalnemu przenikaniu się obiektów przy ciągłym napieraniu na siebie:

```
//kolizja między 1 a 2 graczem
//sprawdzanie czy gracze na siebie nachodzą, jeśli tak to tworzy się ich część wspólna
auto pole_zderzenia = gracz.sprite.getGlobalBounds().findIntersection(gracz2.sprite.getGlobalBounds());
if (pole_zderzenia) {//gracze się zderzyli
    if (pole_zderzenia->size.x < pole_zderzenia->size.y) {//jeśli prostokąt kolizji jest węższy niż wyższy to zderzyli się poziomo
        //kolizja pozioma
        float kierunek;
        if (gracz.sprite.getPosition().x < gracz2.sprite.getPosition().x) {
            kierunek = -1.0f; //gracz 1 jest po lewej, więc wypycha w lewo
        }
        else {
            kierunek = 1.0f; //gracz 1 jest z prawej więc wypycha w prawo
        }
        gracz.sprite.move({ (pole_zderzenia->size.x / 2.0f) * kierunek,0.0f });//głębokość zderzenie dzielona na pół i przesunięcie obu w przeciwnych kierunkach
        gracz2.sprite.move({ (pole_zderzenia->size.x / 2.0f) * -kierunek,0.0f });

        gracz.velocity.x = 0;//zerowanie prędkości w osi poziomej czyli tej w której jest zderzenie w tym przypadku, prędkość y bez zmian
        gracz2.velocity.x = 0;
    }
    else {
        //kolizja pionowa
        float kierunek;
        if (gracz.sprite.getPosition().y < gracz2.sprite.getPosition().y) {
            kierunek = -1.0f; //gracz 1 jest nad graczem 2, więc wypycha w górę
        }
        else {
            kierunek = 1.0f; //gracz 1 jest pod graczem 2, więc wypycha w dół
        }
        gracz.sprite.move({ 0.0f, (pole_zderzenia->size.y / 2.0f) * kierunek });
        gracz2.sprite.move({ 0.0f, (pole_zderzenia->size.y / 2.0f) * -kierunek });

        gracz.velocity.y = 0;
        gracz2.velocity.y = 0;
```

```
    }
}
```

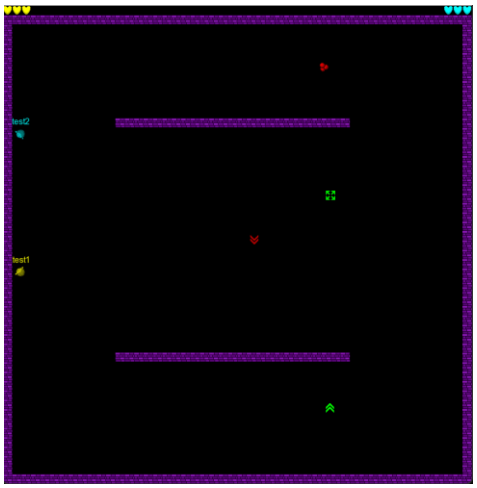


- Ślizganie się graczy po ścianach:

Najważniejszym elementem kolizji gracza ze ścianą jest umożliwienie płynnego poruszania się wzdłuż przeszkody, zamiast blokowania całego ruchu. Dzięki funkcji findIntersection obliczamy pole przecięcia sprite'a gracza ze ścianą (pole_g1) i na podstawie porównania wymiaru tego pola określamy rodzaj kolizji. W zależności od tego, po której stronie ściany znajduje się gracz, obliczany jest kierunek wypchnięcia. Program modyfikuje także wektor prędkości: prędkość w osi zderzenia jest zerowana, natomiast prędkość w osi równoległej do ściany jest mnożona przez współczynnik 0.8f, co symuluje tarcie o ścianę, które spowalnia postać przy kontakcie z przeszkodą:

```
//ślizganie graczy na ścianach
for (int j = 0; j < walls.size(); j++) {
    //kolizje dla 1 gracza
    auto pole_g1 =
gracz.sprite.getGlobalBounds().findIntersection(walls[j].sprite.getGlobalBounds());
    if (pole_g1) {
        if (pole_g1->size.x < pole_g1->size.y) {//kolizja pozioma
            float kierunek;
            if (gracz.sprite.getPosition().x < walls[j].sprite.getPosition().x) {
                kierunek = -1.0f; // wypchany w lewo, gdyż gracz jest po lewej
stronie ściany
            }
            else {
                kierunek = 1.0f;
            }
            gracz.sprite.move({ pole_g1->size.x * kierunek, 0.0f });
            gracz.velocity.x = 0; //tylko bieg góra/dół
            gracz.velocity.y *= 0.8f; //tarcie (spowolnienie)
        }
```

```
        else {//kolizja pionowa
            float kierunek;
            if (gracz.sprite.getPosition().y < walls[j].sprite.getPosition().y) {
                kierunek = -1.0f; // wypchany w górę, gdyż gracz jest nad ścianą
            }
            else {
                kierunek = 1.0f;
            }
            gracz.sprite.move({ 0.0f, pole_g1->size.y * kierunek });
            gracz.velocity.y = 0; //tylko bieg prawo/lewo
            gracz.velocity.x *= 0.8f; //tarcie (spowolnienie)
        }
    }
```



- Kolizja pocisku z graczami

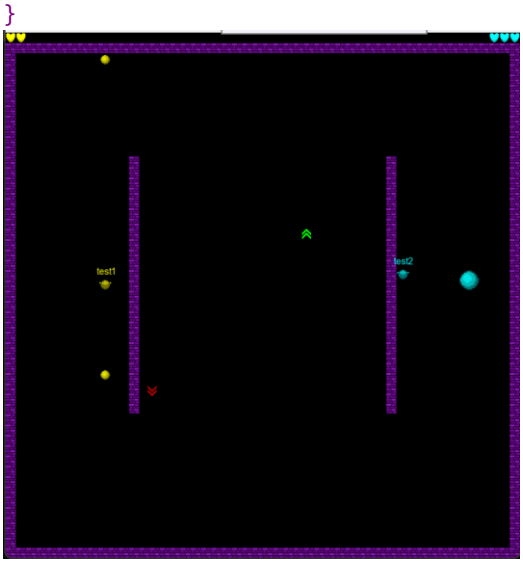
Mechanizm ten odpowiada za kolizję między lecącą piłką, a graczem. Kluczowym zabezpieczeniem jest tutaj weryfikacja przynależności do drużyny, dzięki czemu pocisk nie reaguje na kontakt z graczem, który go wystrzelił. W momencie trafienia następuje odjęcie jednego życia u gracza, a sama piłka zostaje od razu ukryta. Dzięki temu pocisk znika z pola widzenia i nie uczestniczy już w dalszej fizyce gry:

```
// kolizja z graczami

if (bullets[i].sprawdzKolizje(gracz) and bullets[i].team != gracz.team) {
    gracz.health -= 1;
    bullets[i].visible = false;
}
if (bullets[i].sprawdzKolizje(gracz2) and bullets[i].team != gracz2.team) {
    gracz2.health -= 1;
    bullets[i].visible = false;
}
```

E) W grze zaimplementowaliśmy cztery rodzaje bonusów, które po zebraniu przez gracza tymczasowo modyfikują parametry obiektów. Wykorzystaliśmy bonusy z licznikami czasu, aby efekty automatycznie wygasały.

```
void Object::apply_buffs(int rodzaj) {
    if (!buffable) {
        return;
    }
    switch (rodzaj) {
    case 0:
        speed_buff_time = 3.f;
        max_speed += 100;
        speedy = true;
        break;
    case 1:
        slow_buff_time = 3.f;
        max_speed -= 100;
        slowy = true;
        break;
    case 2:
        big_ball_time = 5.f;
        big_ball = true;
        break;
    case 3:
        multishot_time = 10.f;
        multishot = true;
        break;
    }
}
```



Bonusy wracają na mapę po 10 sekundach od zebrania. Zastosowaliśmy algorytm sprawdzający czy nowa pozycja nie koliduje ze ścianami. Przed testem kolizji wymuszamy visible=true, aby funkcja mogłą poprawnie odczytać granice obiektu.

```
//sprawdzanie każdego bonusu
if (speedup.visible == false) {
    timer_speedup += delta;
    if (timer_speedup >= respawn_time_limit) {
        bool kolizja = true;
        while (kolizja) {
            speedup.sprite.setPosition({ (float)(rand() % 600 + 100),
(float)(rand() % 600 + 100) });
```

```
                kolizja = false; //zakładam że jest ok
                speedup.visible = true;
                //sprawdzanie czy dotyka ściany
                for (int i = 0; i < walls.size(); i++) {
                    if (speedup.sprawdzKolizje(walls[i])) {
                        kolizja = true; //trafiło na ścianę
                        break;
                    }
                }
            }
            timer_speedup = 0.f;
        }
    }
```

Przełączanie stanów gry

Dodaliśmy definicje stanów, która pozwala programowi „widzieć” w jakiej fazie obecnie się znajduje

```
enum class GameState //enum zarzadza stanami gry w tym samym oknie
{
MENU,
GAME,
END_SCREEN,
NAME_INPUT
};
```

Użycie enum class zamiast zwykłych liczb zwiększa czytelność kodu i zapobiega błędom (np. przypisaniu nieistniejącego stanu).

Każdy stan odpowiada za inną logikę , głowną logike gry (GAME), przemieszczanie się po (MENU), ekran po zakończeniu gry END_SCREEN, możliwość wpisywania nicków graczy (NAME_INPUT).

```
if (menu.shouldStartGame()) {
// resertuje stan gry przed nowym startem
//Jeśli menu mówi, że gra ma wystartować, upewniamy się, że stan to GAME
resetujGre(gracz, gracz2, bullets, zycia, walls, seed);
menu.resetFlags();
state = GameState::GAME;
```

```
//sprawdzanie konca gry
if (gracz.health <= 0 || gracz2.health <= 0) {
int winner = (gracz.health > 0) ? 1 : 2;

menu.setEndScreen(winner); // Ustawia nagłówek i zmienia nazwy przycisków
```

```
state = GameState::END_SCREEN;
}
```

Przełączanie odbywa się poprzez zmianę wartości zmiennej state.

- **Z Menu do Gry:** Następuje, gdy metoda shouldStartGame() zwróci true (po naciśnięciu Enter na opcji "Graj").


- **Z Gry do Ekranu Końcowego:** Następuje w momencie wykrycia warunku zwycięstwa (spadek punktów życia jednego z graczy do zera).


System stanów decydując o tym co użytkownik widzi na ekranie zapobiega nakładaniu się widoku gry na menu.

Ta część kodu wykonuje się gdy użytkownik przebywa w interfejsie użytkownika , funkcja *window.clear()* czyści poprzednią klatkę

```
if (state == GameState::MENU || state == GameState::END_SCREEN) {
 //..tutaj obsluga zdrazen
```

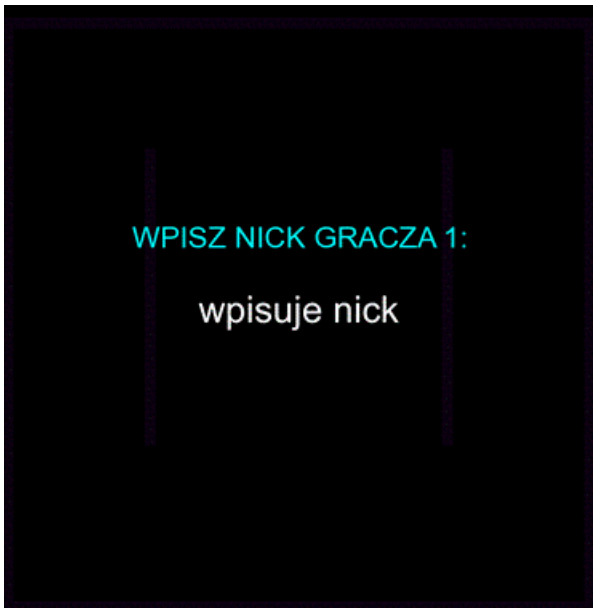
```
if (menu.shouldExit()) window.close();

window.clear();
menu.draw();
window.display();

continue; //przeskakuje reszte petli(logike gry)
}
```

Ten fragment kodu odpowiada za przełączanie się między trybem menu a trybem wpisywania nazw graczy. Jest to kluczowe dla separacji logiki interfejsu.

```
if (selected == 0) {
    // nicki wpisujesie tylko przy pierwszym odpaleniu gryy
    if (!namesEnteredOnce) {
        enteringNames = true;
        currentStep = 1;
        currentInput.clear();
        inputDisplay.setString("");
    }
    else {
        startGame = true; // Jeśli już wpisane, po prostu startuj
    }
}
```



5. **Link do repozytorium na GitHubie**
   https://github.com/Kapionek/projekt_podstawy_informatyki
6. **Instrukcja uruchomienia (README)**

Dope-Dodge - Physics-based Game

Dwuwymiarowa gra zręcznościowa dla dwóch graczy oparta na fizyce (physics-based) zrealizowana w języku C++ przy wykorzystaniu biblioteki SFML.

Wymagania systemowe

Do poprawnego skompilowania i uruchomienia gry potrzeba:

- **Kompilator C++**
- **Zainstalowana biblioteka SFML**(wersja 3.0 lub nowsza)
- **Narzędzia Git**

Instalacja i uruchomienie

1. Pobranie projektu

Sklonuj repozytorium na swój dysk lokalny:

git clone https://github.com/Kapionek/projekt_podstawy_informatyki.git

cd projekt_podstawy_informatyki

2. Kompilacja gry

Jeśli używasz kompilatora g++ i masz zainstalowaną bibliotekę SFML:

g++ *.cpp -o Zbijak -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio

3. Uruchomienie gry

Gdy kompilacja się skończy wpisz: ./Zbijak

Sterowanie

Gra przeznaczona jest dla dwóch osób grających na jednej klawiaturze:

- **Gracz 1:**

Ruch: WSAD, Dash: E, Strzał: C

- **Gracz 2:**

Ruch: IKJL, Dash: O, Strzał: .

Funkcje gry:

- **Fizyka:** realistyczne odbijanie piłek od ścian i innych obiektów
- **Mechanika ruchu:** płynne przyśpiszanie postaci oraz brak przenikania przez ściany
- **Powerupy:** Speedup - zwieksza prędkość gracza, Slowdown - spowalnia przeciwnika, Big Ball - zwiększa rozmiar wystrzeliwanej piłki, Multishot - pozwala na wystrzelenie dwóch piłek naraz
- **Personalizacja:** możliwość wpisywania nicków graczy przed rozpoczęciem rozgrywki
- **Audio:** dynamiczna muzyka z możliwością regulacji głośności w menu

Autorzy:

- Kamil Stokłosa Niżyński
- Antoni Synowiec
- Wojciech Rubacha

Credits/Licencje

- **Muzyka:** Cyber Forge by **Psychronic** ("https://pixabay.com/music/video-games-cyber-forge-394066/")
- **Grafika:** Wszystkie elementy graficzne są autorstwa: **Kamil Stokłosa Niżyński**
- **Grafika/Licencja:** Powyższe elementy graficzne są udostępniane na podstawie: **Creative Commons Attribution 4.0 International (CC BY 4.0)**