

Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki

Programowanie Komputerów 4

AngryBlocks

autor	Dominik Uszok
prowadzący	dr hab. inż., prof. PŚ Roman Starosolski
rok akademicki	2020/2021
kierunek	informatyka
rodzaj studiów	SSI
semestr	4
termin laboratorium	wtorek, 13:45 – 15:15
sekcja	21
termin oddania spra	2021-05-17

1 Temat

Celem programu jest stworzenie gry dla 1 gracza, którego zadaniem jest rzucanie obiektem w taki sposób, żeby strącić wszystkie wrogie obiekty. Utrudnieniem jest obecność przeszkód w postaci blochków o różnych kształtach. Przeszkody można zniszczyć, ale występują one w 3 poziomach wytrzymałości: drewno, kamień, metal.

Gracz dysponuje 3 rzutami, gdy po wykorzystaniu 3 rzutów nie uda się strącić wszystkich wrogich obiektów, poziom jest przegrany. W grze dostępne będą dla gracza 3 rodzaje obiektów do rzucania: zwykły, odłamkowy i bombardujący. Rzucać będzie można, przeciągając myszą w przeciwną stronę do zamierzonego przez nas kierunku ruchu. Trajektoria lotu będzie pokazywana kilkunastoma kropkami.

W grze będzie 5 poziomów do ukończenia. Zniszczenia przeszkód, zniszczenia blochków przeciwnika i niewykorzystanie wszystkich obiektów do rzucania przyznają graczowi dodatkowe punkty. Punkty będą zapisywane dla każdej planszy z osobna i po uzyskaniu większej niż poprzednio ilości punktów, wynik będzie zastępowany.

2 Analiza tematu

W programie odbywa się symulacja fizycznego świata z grawitacją, obiektami i kolizjami, które odbywają się między symulowanymi obiektami. Obliczenia odpowiedzialne za tę symulację są realizowane przez bibliotekę Box2D. Jest to biblioteka, która nie ma możliwości graficznego przedstawienia tej symulacji, dlatego konieczne jest użycie innej biblioteki za to odpowiedzialnej.

W tym programie za graficzną reprezentację odpowiedzialna jest biblioteka SFML. Problemem połączenia obu bibliotek jest różnica w używanych przez nie jednostek.

Do wyrażenia miary odległości SFML używa pikseli a Box2D metrów. Do wyrażenia miary kąta SFML używa stopni a Box2D radianów. Połączenie obu bibliotek, żeby synchronicznie ze sobą działały, wymaga zastosowania konwersji jednostek. Funkcje odpowiedzialne za tą konwersję znajdują się w przestrzeni nazw *Converter*. Sam obiekt, który jest widoczny dla użytkownika (SFML) i którego fizyka jest symulowana (Box2D), jest realizowany przez klasę *SimulatedObject*.

Wygrana oznacza, że na mapie nie znajduje się żaden obiekt o rodzaju przeciwnik. Przegrana oznacza, że każdy z obiektów do rzucania został użyty i każdy z nich skończył się poruszać.

Rzucanie odbywa się poprzez obliczenie różnicy koordynatu początkowego obiektu do rzucania a koordynatu myszki w momencie puszczenia lewego przycisku myszy. W tym momencie następuje impulsywne nałożenie prędkości liniowej w centrum masy obiektu.

Trajektoria lotu w momencie trzymania lewego przycisku myszy realizowana jest wzorem:

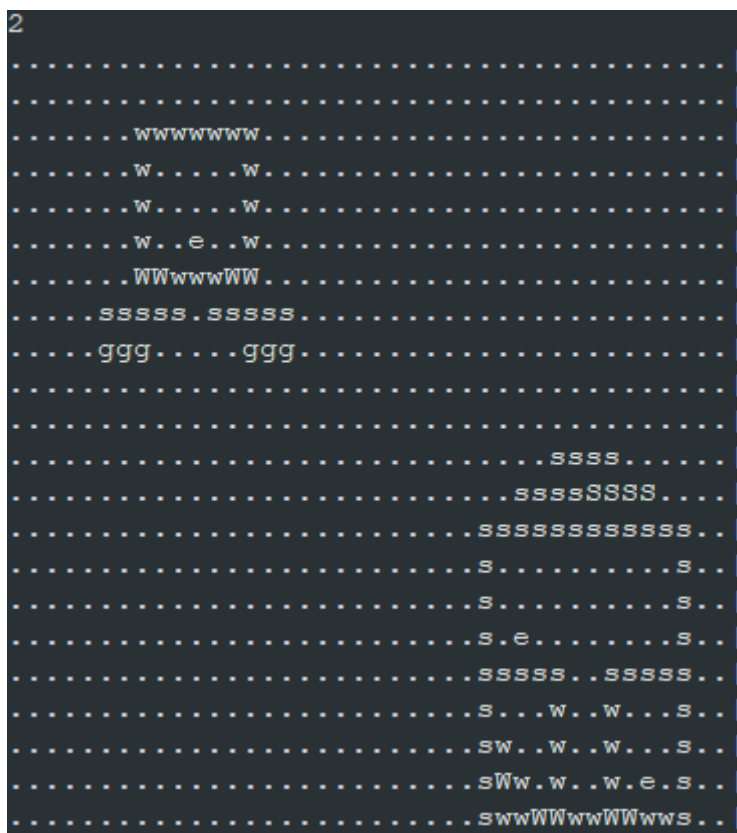
$$p(n) = p_0 + nv + \frac{(n^2 + n)a}{2}$$

Gdzie:

- **p(n)** - pozycja w skoku czasowym **n**
- **p₀** - pozycja początkowa (zerowy skok czasowy)
- **v** - początkowa prędkość na skok czasowy
- **a** - przyspieszenie ziemskie na skok czasu

Poziom jest tworzony, interpretując plik tekstowy lub binarny i zapisując projekt poziomu w przeznaczonych na to trójwymiarowej, statycznej tablicy znaków. Interpretacja polega na liniowym przeszukiwaniu pliku aż do jego końca i zapisie odczytanych znaków, jeżeli wczytana linia zawiera odpowiednią ilość znaków.

Przykład projektu poziomu w pliku tekstowym:

**Gdzie:**

- **w** - przeszkoda drewniana
- **s** - przeszkoda kamienna
- **m** - przeszkoda metalowa
- **e** - przeciwnik
- **g** - podłoże

W trakcie działania programu można zapisać zinterpretowany plik tekstowy do pliku w postaci binarnej.

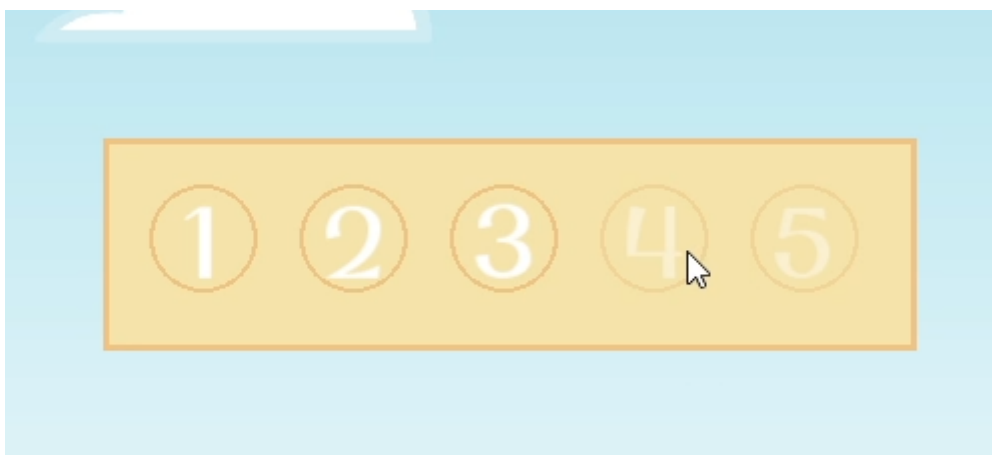
Na początku działania programu wczytywane są wcześniej zdobyte punkty na każdym z poziomów, a pod koniec programu są one zapisywane. Znow można podjąć decyzje zapisu/odczytu z pliku binarnego lub tekstowego, przy pomocy parametru metody.

3 Specyfikacja zewnętrzna

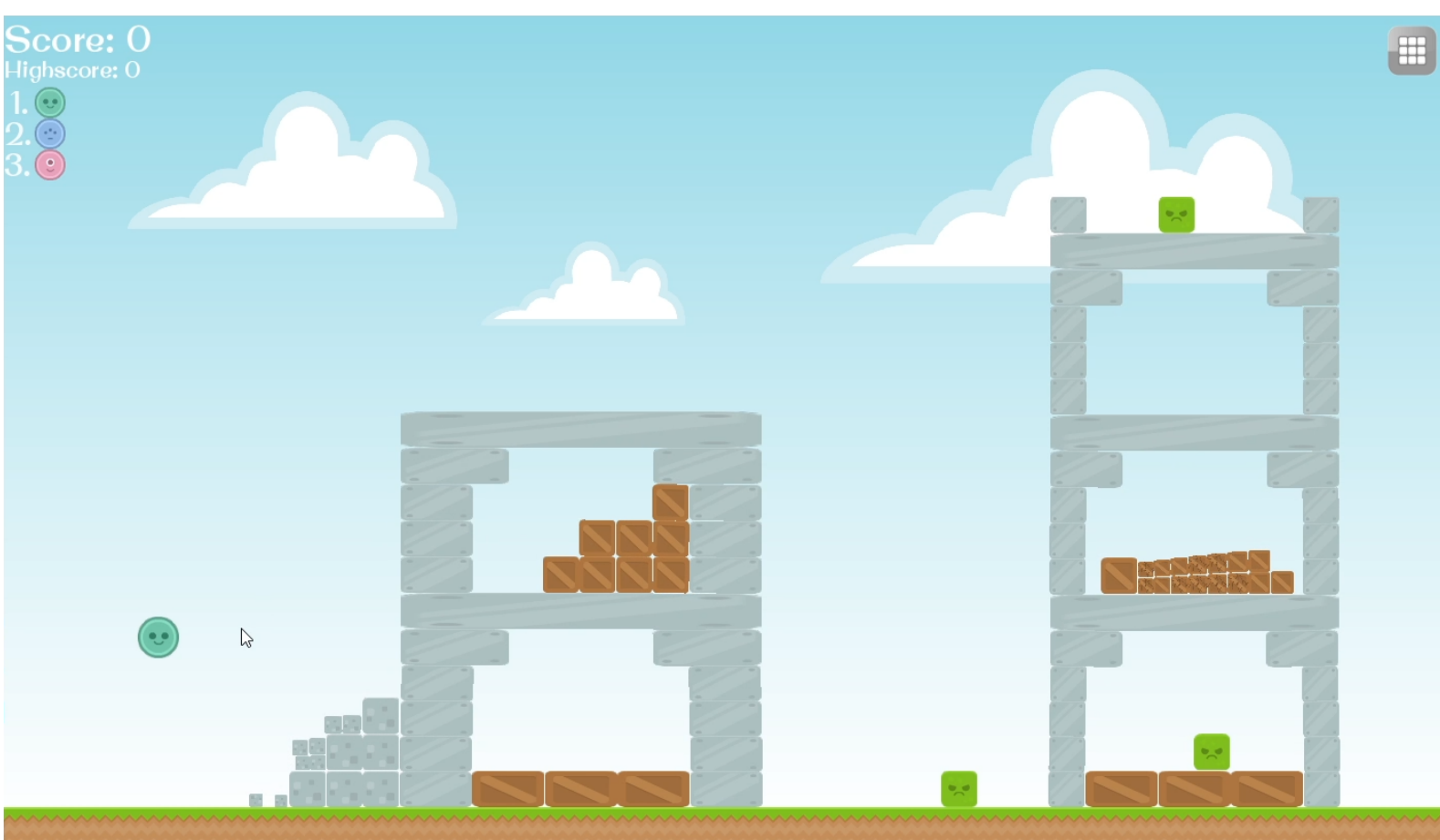
Po uruchomieniu programu użytkownikowi ukazuje się menu z wszystkimi dostępnymi mapami. Mapa jest dostępna do kliknięcia, jeżeli poprzednia mapa została pokonana.



Niedostępne mapy są przezroczyste i nie podświetlają się po najechaniu na nie myszką.



Po kliknięciu, na ekranie wyświetla się odpowiedni poziom z dostępnymi obiektami do rzucania dostosowanymi do trudności poziomu. Trudność poziomu zależy od ilości i rodzaju przeszkód (drewno - łatwe, kamień - średni, metal - ciężki), ilości przeciwników i ich umieszczenia.



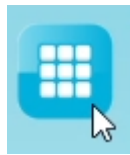
W lewym górnym rogu ekranu znajduje się ilość punktów zdobytych w aktualnej próbie, pod nią jest maksymalna ilość punktów, jaką udało się zdobyć na tym poziomie.

Score: 6050
Highscore: 7900

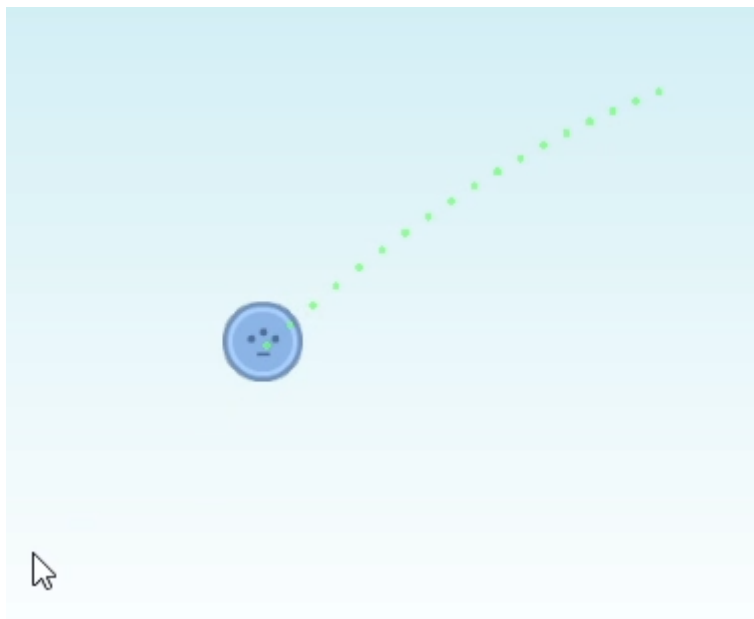
Pod punktami znajdują się dostępne rzuty wraz ze skrótem klawiszowym do ich wyboru. Użyte obiekty są transparentne i niemożliwe do ponownego wyboru.



W prawym górnym rogu znajduje się przycisk manu map, który przenosi do tego samego menu, które wyświetla się po starcie programu. W menu map znajduje się ten sam przycisk, który przenosi z powrotem do aktualnego poziomu.



Gdy użytkownik zdecydował się na jeden z typów obiektu do rzucania, może rozpocząć celowanie przez przytrzymanie lewego przycisku myszy.



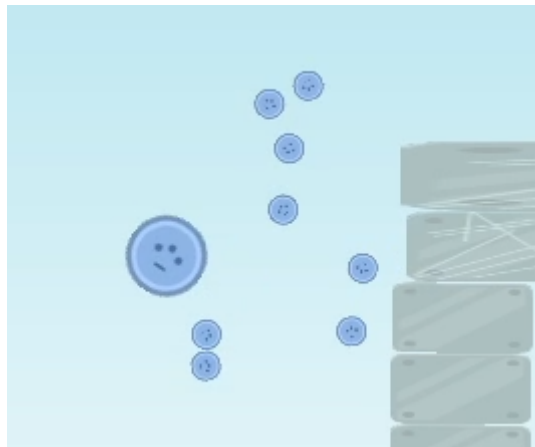
Puszczenie przycisku powoduje wyrzut obiektu.

Do wyboru są 3 typy obiektów do rzucania:

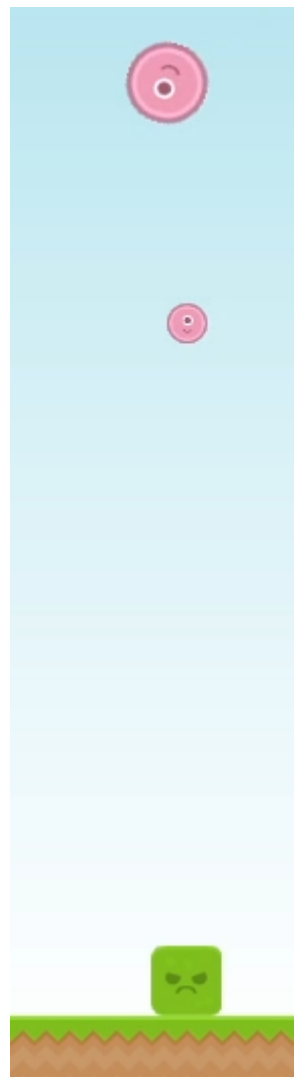
- **Zwyczajny** - brak specjalnych efektów



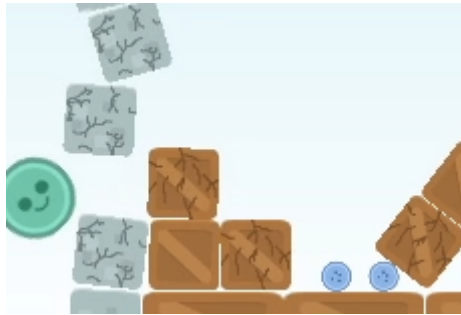
- **Odłamkowy** - po kolizji tworzy 8 odłamków



- **Bombardujący** - po wciśnięciu lewego przycisku myszy tworzy bombę (maksymalnie 3)



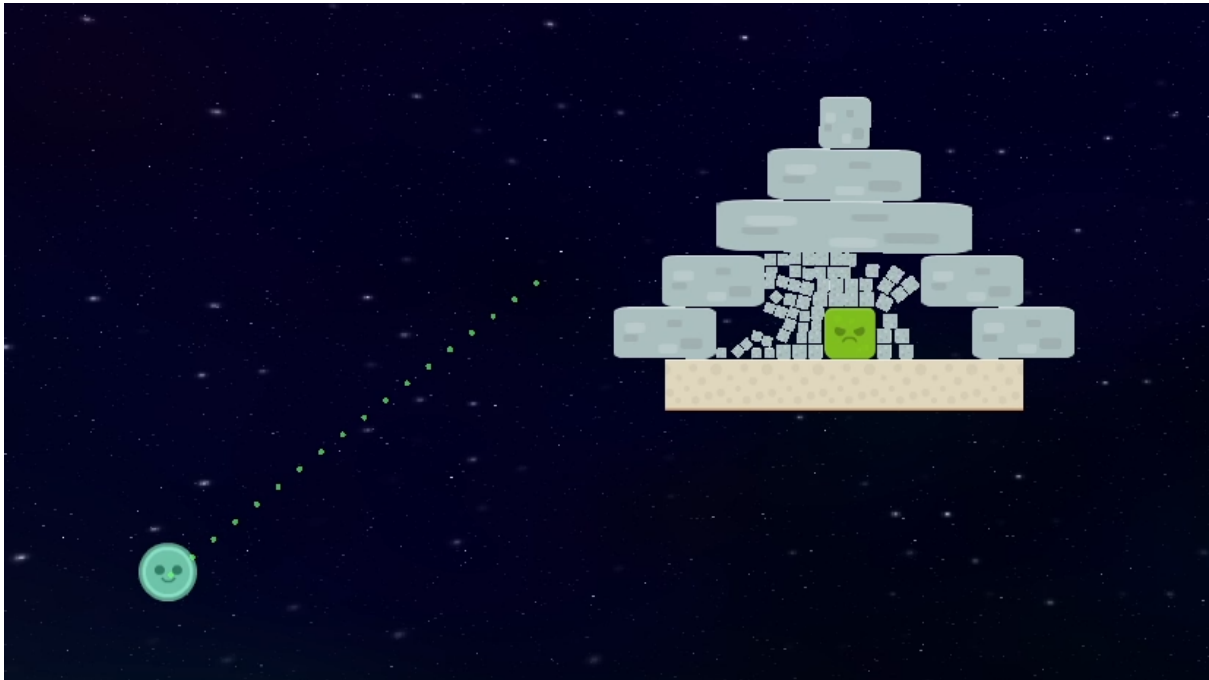
Przyjmowanie obrazów przez przedmioty jest zobrazowane w taki sposób:



Po przegraniu lub wygraniu wyświetlają się odpowiednie okna, które pozwalają na powtórzenie poziomu lub przejście do następnego w przypadku wygranej.



Ostatnie 2 poziomy odbywają się na księżycu o obniżonym przyspieszeniu grawitacyjnym.



Przydatne skróty klawiszowe dla użytkownika:

- **R** - restart aktualnego poziomu
- **1-3** - wybór obiektu do rzucania
- **M** - menu map
- **Esc** - wyjście z programu

4 Specyfikacja wewnętrzna

Program został zrealizowany zgodnie z paradygmatem strukturalnym. W programie rozdzielono interfejs (komunikację z użytkownikiem) od logiki aplikacji (symulacji fizyki).

4.1 Klasy

1) **Screen**

Odpowiada za stworzenie okna, w którym wyświetlany będzie program, za komunikację z użytkownikiem, operacje na plikach i koordynacje stanów gry (poziom, wygrana przegrana, menu map).

Zawiera w sobie obiekty klasy *Button* do stworzenia przycisków i wskaźnik na obiekt klasy *Level*.

Istotne pola:

- **window** - reprezentuje okno, w którym przedstawiana jest gra
- **event** - odczytuje input użytkownika
- **game_state** - określa stan gry (poziom, wygrana przegrana, menu map)
- **level_icon** - tablica wskaźników na przycisk do wczytania danego poziomu

- **level_design** - tablica trójwymiarowa jest miejscem zapisywania z pliku i wczytywania designu poziomów
- **active_level** - wskaźnik na aktualny poziom
- **scores** - tablica z punktami dla każdego poziomu

Istotne metody:

- **loadChosenLevel** - wczytuje poziom z *level_design* na wskaźnik *active_level*
- **renderTrajectory** - tworzy trajektorie lotu podczas celowania rzutem
- **updateObjects** - iteruje przez wszystkie obiekty, żeby zaktualizować ich pozycje na ekranie na podstawie ich oddziaływań z poziomem
- **handleLevelEvents** - pobiera informacje od użytkownika w trakcie trwania poziomu
- **handleMapMenuEvents** - pobiera informacje od użytkownika w trakcie bycia w menu map
- **interpretLevelLayout** - interpretuje informacje z pliku tekstowego lub binarnego o designie poziomu i zapisuje je w *level_design*.
- **loadScore** - wczytuje punkty poziomów z pliku tekstowego lub binarnego
- **saveScore** - zapisuje punkty poziomów do pliku tekstowego lub binarnego
- **createWindow** - tworzy okno gry i zawiera w sobie pętle trwającą do momentu zakończenia pracy programu, w której odbywają się najważniejsze części programu

2) Button

Odpowiada za stworzenie interaktywnego przycisku, który wykrywa pozycje kursora myszy na ekranie i potrafi poinformować o kliknięciu.

Znajduje się w klasie *Screen*.

Istotne pola:

- **button_type** - rodzaj przycisku (okrągły, prostokątny, oparty o teksturę)
- **text** - napis na przycisku
- **allow_to_click** - zmienna boolowska, określa czy przycisk można kliknąć
- **size_x** - szerokość przycisku
- **size_y** - wysokość przycisku
- **radius** - promień przycisku

Istotne metody:

- **isHovered** - zwraca true jeżeli kursor myszy znajduje się na przycisku
- **isClicked** - zwraca true jeżeli przycisk został kliknięty, działa tylko jeżeli *allow_to_click* = true.
- **setButtonPosition** - ustawia pozycje przycisku i jego napisu
- **drawTo** - wyświetla przycisk na ekranie

3) Level

Odpowiada za tworzenie i przechowywanie w sobie wszystkich obiektów poziomu, których fizyka jest symulowana, za interakcje między tymi elementami, wykrywanie kolizji między nimi i niszczenie ich.

Zawiera w sobie obiekty klasy *SimulatedObject* w tym: *Obstacle*, *Ground*, *Enemy*, *Throwable* i ich rodzaje. Zawiera w sobie pojedynczą instancję klasy *ContactListener*. Występuje w klasie *Screen*.

Istotne pola:

- **score** - ilość punktów aktualnie zdobytych na poziomie
- **throwable_number** - stopień ulepszenia obiektów do rzucania
- **level_number** - liczba jednoznacznie identyfikująca poziom
- **level_type** - typ poziomu: normalny lub księżycowy
- **world** - wskaźnik na obiekt biblioteki Box2D, odpowiadający za symulację obiektów
- **contact_listener_instance** - obiekt biblioteki Box2D, odpowiadający za wykrywanie kolizji
- **objects** - wektor biblioteki STL zawierający wskaźniki obiektów *SimulatedObject*, w której znajdują się wszystkie obiekty poziomu
- **active_throwable** - aktualnie wybrany obiekt do rzucania
- **throwables** - tablica wskaźników wszystkich obiektów do rzucania

Istotne metody:

- **createWholeGround** - tworzy podłoże na samym dole poziomu i na jego całej szerokości
- **createGround** - tworzy pojedynczy blok podłoża i umieszcza pointer stworzonego obiektu w wektorze *objects*
- **createObstacle** - tworzy pojedynczy blok przeszkody i umieszcza pointer stworzonego obiektu w wektorze *objects*
- **createEnemy** - tworzy pojedynczy blok przeciwnika i umieszcza pointer stworzonego obiektu w wektorze *objects*
- **createThrowables** - tworzy obiekty gotowe do rzutu gdy zostaną aktywowane
- **makeLevel** - pozwala stworzyć obiekty w dowolnych rozmiarach i dowolnych pozycjach omijając tworzenie poziomu z interpretacji tablicy *level_design*
- **destroyFlaggedObjects** - niszczy obiekt, którego HP jest mniejsze lub równe 0 przyznając punkty równe jego maksymalnego HP * 10
- **isLevelWon** - iteruje poprzez wszystkie obiekty ze złożonością obliczeniową $O(n)$ i sprawdza, czy istnieje obiekt typu *Enemy*
- **isLevelLost** - sprawdza czy wszystkie obiekty do rzucania zostały użyte i skończyły się poruszać
- **addPointsForUnusedThrows** - przyznaje punkty za każdy z nieużytych obiektów do rzucania

4) **ContactListener**

Odpowiada za wykrywanie kolizji i identyfikacji, które ciała w nich uczestniczą. Klasa ta zwraca informacje o kolizji tylko wtedy gdy 2 ciała się stykają, dzięki temu nie ma konieczności iterowania przez każdy obiekt i sprawdzania, czy jego granice stykają się z granicami każdego innego obiektu na poziomie. Miałoby to złożoność obliczeniową $O(n^2)$.

Dziedziczy z *b2ContactListener* - klasy biblioteki Box2D. Jej instancja jest obecna w klasie *Level*.

Istotne pola:

- **object_a** - obiekt A, który wchodzi w kolizje
- **object_b** - obiekt B, który wchodzi w kolizje
- **damage** - ilość obrażeń przyjmowanych przez obiekt podczas kolizji

Istotne metody:

- **BeginContact** - metoda wywoływana w momencie wykrycia kolizji
- **calculateContactDamage** - oblicza obrażanie zadane obiektowi będące sumą prędkości liniowej obu ciał na płaszczyźnie x i y
- **applyDamage** - odejmuje HP przeszkodzie lub przeciwnikowi na podstawie wcześniej obliczonych obrażeń
- **checkIfClusterContact** - sprawdza czy obiekt, który uległ kolizji jest typu Cluster

5) **SimulatedObject**

Odpowiada za połączenie dwóch bibliotek zewnętrznych: SFML (grafika) i Box2D (fizyka). Wykonuje odpowiednie konwersje jednostek (stopnie-radiany i piksele-metry) z użyciem przestrzeni nazw **Converter** zawierającej funkcje:

- `pixelsToMeters`
- `metersToPixels`
- `degToRad`
- `radToDeg`

oraz stałe:

- `PIXELS_PER_METER = 32.0`
- `PI = 3.14159265358979323846`

Konwersja jest konieczna, żeby grafiki obiektów pokazywały się w tym miejscu na ekranie, na który wskazują obliczenia Box2D.

Jest to klasa abstrakcyjna z czysto wirtualną metodą `createObject`, z której dziedziczą wszystkie symulowane obiekty występujące na poziomie. Jej klasami potomnymi są `Obstacle`, `Ground`, `Enemy` i `Throwable`. Kolekcja wszystkich symulowanych obiektów znajduje się w klasie `Level` w jej wektorze wskaźników na `SimulatedObject`.

Istotne pola:

- **x** - położenie obiektu na płaszczyźnie x
- **y** - położenie obiektu na płaszczyźnie y
- **hp** - HP obiektu
- **block_size_x** - szerokość bloku
- **block_size_y** - wysokość bloku
- **radius** - promień obiektu
- **graphics** - grafika obiektu
- **physics** - fizyczne właściwości obiektu
- **object_type** - typ obiektu: przeszkoda, podłoże, przeciwnik, obiekt do rzucania

Istotne metody:

- **createObject** - metoda czysto wirtualna
- **initializePosition** - ustawia świat, w którym znajduje się obiekt i jego położenie
- **updatePosition** - ustawia odpowiednie położenie i rotację grafiki obiektu, na podstawie jego interakcji z innymi obiektami w świecie
- **deleteObject** - usuwa obiekt ze świata
- **setAsDamaged** - ustawia teksturę obiektu na wersję poniszczoną gdy jego HP spadnie poniżej 50%

6) **Obstacle**

Reprezentuje przeszkodę - dynamiczny (możliwy do poruszenia) obiekt, który może ulec zniszczeniu. Występuje w 3 typach: drewno, kamień, metal. Typ przeszkody wpływa na maksymalne HP, teksturę i gęstość obiektu.

Dziedziczy z klasy *SimulatedObject*.

Istotne pola:

- **obstacle_type** - typ przeszkody

Istotne metody:

- **createObject** - tworzy przeszkodę, która jest gotowa do symulacji fizyki (określa jej parametry po stworzeniu: pozycję, rozmiar, gęstość i tarcie)
- **setType** - ustawia typ przeszkody

7) **Ground**

Reprezentuje podłoże - statyczny (niemożliwy do poruszenia) obiekt, który nie może ulec zniszczeniu. Występuje w 2 typach: zwykły i księżycowy. Typ podłoża zależy od rodzaju poziomu.

Dziedziczy z klasy *SimulatedObject*.

Istotne metody:

- **createObject** - tworzy podłoże, które jest gotowe do symulacji fizyki (określa jego parametry po stworzeniu: pozycję, rozmiar i gęstość)
- Zawiera dwa konstruktory: jeden oparty o teksturę - w którym wybierany jest typ podłoża, drugi oparty o kolor (tworzy jednolity blok)

8) **Enemy**

Reprezentuje przeciwnika - dynamiczny (możliwy do poruszenia) obiekt, który może ulec zniszczeniu. Wszystkie obiekty tego typu muszą zostać zniszczone, żeby obiekt typu *Level* mógł podjąć decyzję o zwycięstwie.

Dziedziczy z klasy *SimulatedObject*.

Istotne metody:

- **createObject** - tworzy przeciwnika, który jest gotowy do symulacji fizyki (określa jego parametry po stworzeniu: pozycję, rozmiar, gęstość i tarcie)

9) **Throwable**

Do bazy, jaką stanowi klasa *SimulatedObject* klasa *Throwable* wprowadza swoje metody i pola, które umożliwią celowanie i wystrzelenie obiektu oraz sprawdzanie, czy obiekt wciąż się porusza. Zawiera 3 stany:

- Oczekujący (Idle) - wyświetlana jest grafika obiektu i oczekuje na kliknięcie, aby wyświetlić trajektorię
- Celujący (Aiming) - wyświetlana jest trajektoria lotu
- Użyty (Used) - obiekt został użyty i jego fizyka jest symulowana

Stan obiektu (użyty) i stwierdzenie nieporuszania się umożliwia poziomowi podjęcie decyzji o przegranej.

Dziedziczy z klasy *SimulatedObject*.

Istotne pola:

- **state** - stan obiektu do rzucania
- **linear_velocity** - prędkość liniowa

- **max_velocity** - maksymalna prędkość liniowa przy wystrzeleniu
- **thr_start_x** - pozycja startowa na płaszczyźnie x
- **thr_start_y** - pozycja startowa na płaszczyźnie y
- **is_moving** - określa czy obiekt jest wciąż w ruchu

Istotne metody:

- **createObject** - tworzy obiekt, który jest gotowy do symulacji fizyki (określa jego parametry po stworzeniu: pozycję, rozmiar, gęstość, tarcie, sprężystość, tłumienie kątowe)
- **getTrajectoryX** i **getTrajectoryY** - zwraca przyszłą lokację obiektu na podstawie startowej prędkości po wystrzeleniu i grawitacji
- **launch** - tworzy obiekt, który został rzucony z prędkością liniową określoną pozycją myszy
- **maximumVelocityLimitation** - ustala jaka jest maksymalna prędkość liniowa przy wyrzuceniu obiektu
- **checkIfMoving** - ustawia obiekt jako nieruszający się jeżeli suma jego prędkości na płaszczyźnie x i y nie przekracza określonej wartości

10) **DefaultThrowable**

Nie zawiera żadnych specjalnych efektów. Dziedziczy z klasy *Throwable*.

W konstruktorze obiektu klasy *DefaultThrowable* ustalony zostaje typ obiektu, jego lokacja, tekstura i rozmiar.

11) **Cluster**

Tworzy 8 odłamków po kolizji obiektu. Stworzone odłamki są wstrzeliwane z prędkością liniową w 4 różnych kierunkach.

Dziedziczy z klasy *Throwable*.

Istotne pola:

- **collided** - określa czy obiekt brał już udział w kolizji (tylko pierwsza kolizja powinna stworzyć odłamki)

W konstruktorze obiektu klasy *Cluster* ustalony zostaje typ obiektu, jego lokacja, tekstura, rozmiar oraz to czy obiekt nie jest odłamkiem. Jest to konieczne, ponieważ odłamki nie mogą tworzyć kolejnych odłamków.

12) **Bombarding**

Tworzy maksymalnie 3 bomby od obiektu typu *Bombarding*. Wystrzelenie bomby jest kontrolowane przez użytkownika lewym przyciskiem myszy w trakcie gdy obiekt typu *Bombarding* został użyty, ale jeszcze nie skończył się poruszać. Bomba jest wystrzeliwana z prędkością liniową skierowaną w dół.

Dziedziczy z klasy *Throwable*.

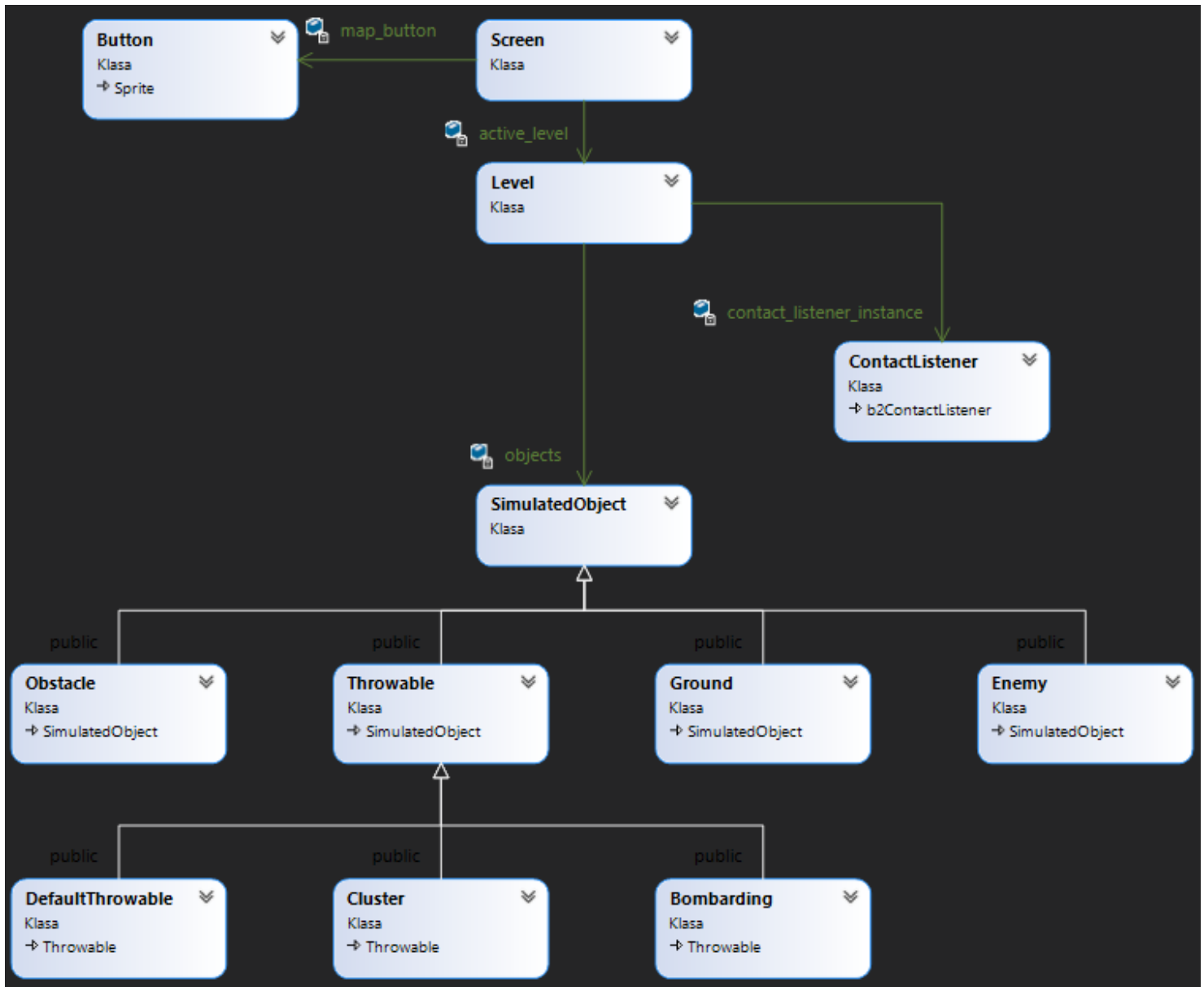
Istotne pola:

- **adds_used** - ilość stworzonych dodatkowych obiektów bombardujących (określa czy można stworzyć kolejną bombę z tego obiektu)

4.2 Szczegółowy opis klas i metod

Szczegółowy opis klas i metod zawarty w załączniku.

4.3 Diagram hierarchii klas



Dziedziczenie oznaczone jest białą strzałką a powiązania zieloną.

4.4 Istotne struktury danych i algorytmy

Istotnymi strukturami danych programu są:

- Statyczna tablica trójwymiarowa [5x22x41] znaków klasy **Screen** - wczytana z pliku tekstowego lub binarnego, określa design poziomów - pierwszy wymiar oznacza numer poziomu, drugi wysokość a trzeci szerokość poziomu. Wczytanie designu z pliku i zapisanie w tej tablicy umożliwia tworzenie poziomu bez konieczności ponownego korzystania z dysku - plik jest otwierany tylko na początku programu.

- Wektor biblioteki STL klasy *Level* - wektor będący tablicą dynamiczną nadaje się na kontener zmiennej ilości tworzonych i usuwanych obiektów na poziomie. Usuwanie obiektu z wektora odbywa się przy użyciu iteratora. Sąsiednie umieszczenie komórek pamięci przyspiesza liniowe przeszukiwanie tablicy, ponieważ usprawnia to wykorzystanie pamięci cache.

Istotne algorytmy:

- Liniowe przeszukiwanie wektora *objects* - musi odbyć się przy wykonywaniu wielu operacji:
 - zaktualizowaniu pozycji poruszających się obiektów
 - zniszczeniu obiektów, które mają HP mniejsze lub równe 0
 - wykrycia obecności lub braku obiektu typu *Enemy*
 Algorytm ma złożoność obliczeniową $O(n)$.

4.5 Wykorzystane techniki obiektowe

W projekcie wykorzystane zostało dziedziczenie z klasy abstrakcyjnej *SimulatedObject* z czysto wirtualną funkcją *createObject* umożliwiającą wykorzystanie polimorfizmu.

W przestrzeni nazw *Converter* zostało wykorzystane programowanie generyczne (wzorce), aby funkcje mogły przyjmować różne typy danych.

W używanych przeze mnie klasach użyłem enkapsulacji szczegółów implementacji np. łączenia biblioteki graficznej z fizyczną w klasie *SimulatedObject*.

Do przechowywania i przeszukiwania symulowanych obiektów poziomu wykorzystane są kontenery i iteratory STL.

W programie został użyty mechanizm wyjątków.

4.6 Ogólny schemat działania programu

W funkcji głównej tworzona jest instancja *main_screen* klasy *Screen*. Konstruktor tego obiektu wczytuje tekstury, czcionkę, napisy i wyskakujące okna. Następnie ładowane są punkty z pliku binarnego metodą *loadScore* i design poziomów z pliku binarnego lub tekstowego (zależnie który zostanie znaleziony) z użyciem metody *loadLevelsLayout*. Jeżeli nie uda się wczytać punktów z pliku zostanie wypisany komunikat "Unable to open text file" lub "Unable to open binary file".

Następnie tworzone jest okno programu w metodzie *createWindow*. Znajduje się tam pętla, która trwa, dopóki okno jest otwarte. Zawiera ona w sobie instrukcje warunkowe sprawdzające aktualny stan gry - menu map lub poziom.

Gdy stanem gry jest menu map, renderowane jest okno, w którym są przyciski do załadowania poziomu. Po kliknięciu przycisku odpowiadającemu danemu poziomowi tworzony jest obiekt klasy *Level* i wczytywany jego design i highscore na podstawie numeru klikniętego przycisku a wskaźnik na wczytany poziom znajduje się w zmiennej *active_level*.

Gdy stanem gry jest poziom, renderowane są wszystkie obiekty poziomu oraz napisy z punktacją, sprawdzane są warunki spełnienia wygranej lub przegranej, aby móc przejść do stanu gry wygranej lub przegranej. W stanie poziom odbywają się również wszystkie operacje konieczne do przeprowadzenia gry takie jak - aktualizowanie pozycji obiektów, niszczenie ich, tworzenie obiektów do rzucania i przyznawanie punktów.

W tej pętli znajduje się również kolejna pętla sprawdzająca, czy wczytywany jest jakiś input/event od użytkownika taki jak np. kliknięcie lewego przycisku myszy, czy klawisza Esc, po którym zamknięte zostaje okno programu, zapisanie wyników metodą `saveScore` i zakończenie pracy programu.

5 Testowanie i uruchamianie

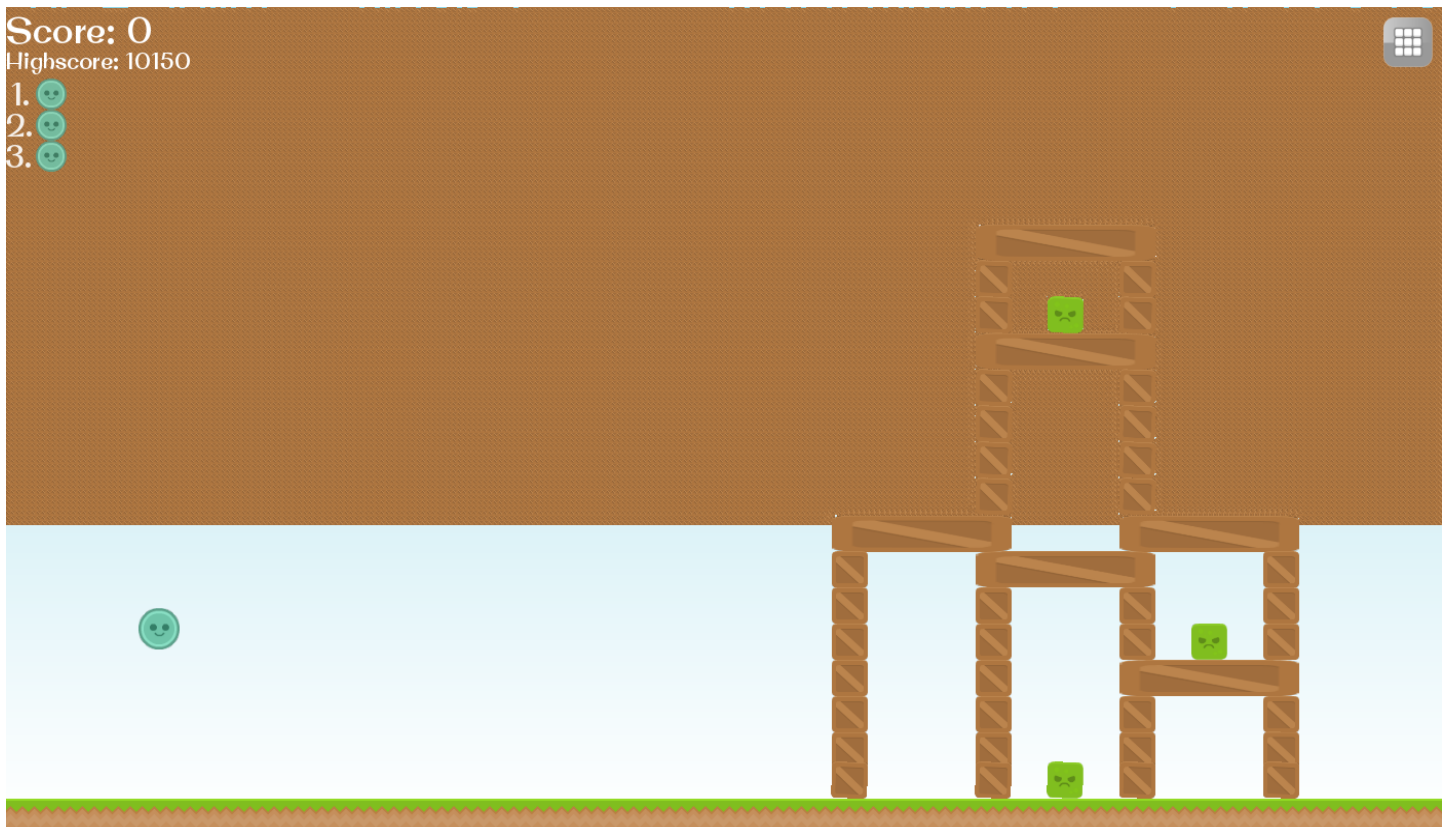
Program został przetestowany pod kątem ilości obiektów, które może stworzyć i renderować przy rozsądnych klatkach na sekundę - program działa przy grywalnych klatkach na sekundę do około 500 obiektów na poziomie. Przy większych ilościach stworzonych obiektów program działa powoli aż do momentu zniszczenia się obiektów przez kolizję do ilości około 500 obiektów. Uruchomienie poziomu z 10000 obiektami trwa 16 sekund. Przy 40000 obiektach uzyskuje się błąd:

```
D:\Google Drive\GitHub\2466a34f-gr21-repo\Projekt\AngryBlocks\Debug\AngryBlocks.exe (proces 14996)
zakończono z kodem -1073741819.
```

Przy 50000 obiektach występuje inny błąd:

```
Failed to load image "Textures/box.png". Reason: Corrupt JPEG
Failed to load image "Textures/box_damaged.png". Reason: Corrupt JPEG
```

Poziom z największą ilością obiektów, jaką udało się uruchomić to 30000 (są to drewniane pudełka wielkości 5×5 , gdzie standardowe są wielkości 35×35).




Stworzenie poziomu z 30000 obiektów trwało 56 sekund.

Odpalenie programu po usunięciu folderu „Levels” gdzie znajdują się punkty i design poziomów skutkuje wyzerowaniem uzyskanych punktów dla każdego z poziomów, a każdy poziom tworzy się pusty. Jeżeli przy tworzeniu poziomów w pliku tekstowym popełni się błąd otrzymuje się komunikat "Line is not 41 chars!" interpretacja pliku się kończy i od tego momentu design jest określany przez plik binarny.

Program został sprawdzony pod kątem wycieków pamięci z użyciem biblioteki CRT.

Program uruchamia się plikiem wykonywalnym „AngryBlocks.exe”.

 AngryBlocks.exe	14.05.2021 12:26	Application	135 KB
---	------------------	-------------	--------

6 Źródła

- <https://www.sfml-dev.org/tutorials/2.5/>
- <https://box2d.org/documentation/>
- <https://www.iforce2d.net/b2dtut/collision-callbacks>
- <http://www.iforce2d.net/b2dtut/projected-trajectory>

Tekstury:

- <https://opengameart.org/content/platformer-art-complete-pack-often-updated>
- <https://opengameart.org/content/bevouliin-free-mountain-game-background>
- <https://opengameart.org/content/gui-buttons-vol1>
- <https://opengameart.org/content/galaxy-skybox>

Dodatek
Szczegółowy opis klas i metod

AngryBlocks

Wygenerowano przez Doxygen 1.9.1

1 Indeks hierarchiczny	1
1.1 Hierarchia klas	1
2 Indeks klas	3
2.1 Lista klas	3
3 Dokumentacja klas	5
3.1 Dokumentacja klasy Bombarding	5
3.1.1 Dokumentacja konstruktora i destruktor	5
3.1.1.1 Bombarding()	5
3.2 Dokumentacja klasy Button	6
3.2.1 Dokumentacja konstruktora i destruktor	7
3.2.1.1 Button() [1/3]	7
3.2.1.2 Button() [2/3]	8
3.2.1.3 Button() [3/3]	8
3.2.2 Dokumentacja funkcji składowych	9
3.2.2.1 drawTo()	9
3.2.2.2 isClicked()	9
3.2.2.3 isHovered()	9
3.2.2.4 setActiveTexture()	10
3.2.2.5 setButtonPosition()	10
3.2.2.6 setPassiveTexture()	10
3.3 Dokumentacja klasy Cluster	11
3.3.1 Dokumentacja konstruktora i destruktor	11
3.3.1.1 Cluster()	11
3.4 Dokumentacja klasy ContactListener	12
3.5 Dokumentacja klasy DefaultThrowable	12
3.5.1 Dokumentacja konstruktora i destruktor	12
3.5.1.1 DefaultThrowable()	12
3.6 Dokumentacja klasy Enemy	13
3.6.1 Dokumentacja konstruktora i destruktor	13
3.6.1.1 Enemy()	13
3.7 Dokumentacja klasy Ground	14
3.7.1 Dokumentacja konstruktora i destruktor	14
3.7.1.1 Ground() [1/2]	14
3.7.1.2 Ground() [2/2]	15
3.8 Dokumentacja klasy Level	15
3.8.1 Dokumentacja konstruktora i destruktor	16
3.8.1.1 Level()	16
3.8.2 Dokumentacja funkcji składowych	17
3.8.2.1 createEnemy()	17
3.8.2.2 createGround()	17
3.8.2.3 createObstacle()	18

3.8.2.4	getActiveThrowable()	18
3.8.2.5	getBackgroud()	18
3.8.2.6	getLevelNumber()	19
3.8.2.7	getLevelType()	19
3.8.2.8	getObjects()	19
3.8.2.9	getScore()	19
3.8.2.10	getThrowableByNumber()	19
3.8.2.11	getWorldPointer()	20
3.8.2.12	isLevelLost()	20
3.8.2.13	isLevelWon()	20
3.8.2.14	setActiveThrowable()	20
3.9	Dokumentacja klasy Obstacle	21
3.9.1	Dokumentacja konstruktora i destruktor	21
3.9.1.1	Obstacle()	21
3.10	Dokumentacja klasy Screen	22
3.10.1	Dokumentacja konstruktora i destruktor	22
3.10.1.1	Screen()	22
3.10.2	Dokumentacja funkcji składowych	22
3.10.2.1	loadLevelsLayout()	22
3.10.2.2	loadScore()	23
3.10.2.3	saveScore()	23
3.11	Dokumentacja klasy SimulatedObject	23
3.11.1	Dokumentacja funkcji składowych	24
3.11.1.1	initializePosition()	24
3.12	Dokumentacja klasy Throwable	25
3.12.1	Dokumentacja funkcji składowych	26
3.12.1.1	getThrowableState()	26
3.12.1.2	getTrajectoryX()	26
3.12.1.3	getTrajectoryY()	26
3.12.1.4	launch()	27
3.12.1.5	setThrowableState()	27

Rozdział 1

Indeks hierarchiczny

1.1 Hierarchia klas

Ta lista dziedziczenia posortowana jest z grubsza, choć nie całkowicie, alfabetycznie:

b2ContactListener	
ContactListener	12
Level	15
Screen	22
SimulatedObject	23
Enemy	13
Ground	14
Obstacle	21
Throwable	25
Bombarding	5
Cluster	11
DefaultThrowable	12
sf::Sprite	
Button	6

Rozdział 2

Indeks klas

2.1 Lista klas

Tutaj znajdują się klasy, struktury, unie i interfejsy wraz z ich krótkimi opisami:

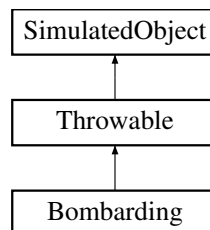
Bombarding	5
Button	6
Cluster	11
ContactListener	12
DefaultThrowable	12
Enemy	13
Ground	14
Level	15
Obstacle	21
Screen	22
SimulatedObject	23
Throwable	25

Rozdział 3

Dokumentacja klas

3.1 Dokumentacja klasy Bombarding

Diagram dziedziczenia dla Bombarding



Metody publiczne

- **Bombarding** (b2World *world, float radius, bool is_add=false, float x=150, float y=600)

*Konstruktor obiektu do rzucania, bombardującego typu (**Bombarding**). Ustalony zostaje typ obiektu, jego lokacja, tekstura i rozmiar.*

Atrybuty publiczne

- int adds_used = 0

Ilość stworzonych dodatkowych obiektów bombardujących.

3.1.1 Dokumentacja konstruktora i destruktora

3.1.1.1 Bombarding()

```
Bombarding::Bombarding (
    b2World * world,
    float radius,
    bool is_add = false,
    float x = 150,
    float y = 600 )
```

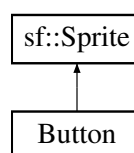
Konstruktor obiektu do rzucania, bombardującego typu (**Bombarding**). Ustalony zostaje typ obiektu, jego lokacja, tekstura i rozmiar.

Parametry

<i>world</i>	Świat w którym ma zostać stworzony obiekt
<i>radius</i>	Promień obiektu do rzucania
<i>is_add</i>	Określa czy zostaje stworzony obiekt Bombarding czy jego odłamek
<i>x</i>	Lokacja odłamka na płaszczyźnie x
<i>y</i>	Lokacja odłamka na płaszczyźnie y

3.2 Dokumentacja klasy Button

Diagram dziedziczenia dla Button



Metody publiczne

- [Button](#) (std::string file_location_passive, std::string file_location_active, float scale_x=1, float scale_y=1)
Konstruktor przycisku opartego o teksturę.
- [Button](#) (float position_x, float position_y, float size_x, float size_y, sf::Color color_passive_fill=sf::Color(245, 227, 169), sf::Color color_passive_outline=sf::Color(241, 195, 131), sf::Color color_active_fill=sf::Color(245, 227, 169), sf::Color color_active_outline=sf::Color(200, 145, 100))
Konstruktor przycisku opartego o prostokątną grafikę.
- [Button](#) (float position_x, float position_y, float radius, sf::Color color_passive_fill=sf::Color(245, 227, 169), sf::Color color_passive_outline=sf::Color(241, 195, 131), sf::Color color_active_fill=sf::Color(245, 227, 169), sf::Color color_active_outline=sf::Color(200, 145, 100))
Konstruktor przycisku opartego o okrągłą grafikę.
- bool [isHovered](#) (sf::Vector2f mouse_position)
Sprawdza czy kursor myszy znajduje się na przycisku. Na tej podstawie zmienia wygląd przycisku z pasywnego na aktywny i umożliwia kliknięcie.
- bool [isClicked](#) (sf::RenderWindow &>window, sf::Event &event)
Sprawdza czy przycisk został kliknięty.
- void [setPassiveTexture](#) (std::string file_location)
Ustawia pasywną teksturę przycisku opartego o teksturę.
- void [setActiveTexture](#) (std::string file_location)
Ustawia aktywną teksturę przycisku opartego o teksturę.
- void [drawTo](#) (sf::RenderWindow &>window)
Wyświetla przycisk na ekranie.
- void [setButtonPosition](#) (float x, float y)
Ustawia pozycję przycisku.
- void [setUnclickableColor](#) ()
Ustawia wygląd przycisku na przezroczysty jeżeli jest on ustawiony jako nieaktywny na kliknięcia.

Atrybuty publiczne

- sf::CircleShape [circle_button](#)
Grafika przycisku okrągłego.
- sf::RectangleShape [rectangle_button](#)
Grafika przycisku prostokątnego.
- sf::Text [text](#)
Tekst w przycisku.
- sf::Color [color_passive_fill](#)
Kolor wnętrza przycisku na którym nie znajduje się kursor myszy.
- sf::Color [color_passive_outline](#)
Kolor zarysu przycisku na którym nie znajduje się kursor myszy.
- sf::Color [color_active_fill](#)
Kolor wnętrza przycisku na którym znajduje się kursor myszy.
- sf::Color [color_active_outline](#)
Kolor zarysu przycisku na którym znajduje się kursor myszy.
- float [size_x](#)
Szerokość przycisku.
- float [size_y](#)
Wysokość przycisku.
- float [radius](#)
Promień przycisku.
- bool [allow_to_click](#) = true
Zmienna określająca czy przycisk jest do kliknięcia.

3.2.1 Dokumentacja konstruktora i destruktor

3.2.1.1 Button() [1/3]

```
Button::Button (
    std::string file_location_passive,
    std::string file_location_active,
    float scale_x = 1,
    float scale_y = 1 )
```

Konstruktor przycisku opartego o teksturę.

Parametry

file_location_passive	Lokacja pliku z teksturą przycisku na którym nie znajduje się kursor myszy
file_location_active	Lokacja pliku z teksturą przycisku na którym znajduje się kursor myszy
scale_x	Skala szerokości tekstury przycisku
scale_y	Skala wysokości tekstury przycisku

3.2.1.2 Button() [2/3]

```
Button::Button (
    float position_x,
    float position_y,
    float size_x,
    float size_y,
    sf::Color color_passive_fill = sf::Color(245, 227, 169),
    sf::Color color_passive_outline = sf::Color(241, 195, 131),
    sf::Color color_active_fill = sf::Color(245, 227, 169),
    sf::Color color_active_outline = sf::Color(200, 145, 100) )
```

Konstruktor przycisku opartego o prostokątną grafikę.

Parametry

<i>position_x</i>	Pozycja przycisku na płaszczyźnie x
<i>position_y</i>	Pozycja przycisku na płaszczyźnie y
<i>size_x</i>	Szerokość przycisku
<i>size_y</i>	Wysokość przycisku
<i>color_passive_fill</i>	Kolor wnętrza przycisku na którym nie znajduje się kursor myszy
<i>color_passive_outline</i>	Kolor zarysu przycisku na którym nie znajduje się kursor myszy
<i>color_active_fill</i>	Kolor wnętrza przycisku na którym znajduje się kursor myszy
<i>color_active_outline</i>	Kolor zarysu przycisku na którym znajduje się kursor myszy

3.2.1.3 Button() [3/3]

```
Button::Button (
    float position_x,
    float position_y,
    float radius,
    sf::Color color_passive_fill = sf::Color(245, 227, 169),
    sf::Color color_passive_outline = sf::Color(241, 195, 131),
    sf::Color color_active_fill = sf::Color(245, 227, 169),
    sf::Color color_active_outline = sf::Color(200, 145, 100) )
```

Konstruktor przycisku opartego o okrągłą grafikę.

Parametry

<i>position_x</i>	Pozycja przycisku na płaszczyźnie x
<i>position_y</i>	Pozycja przycisku na płaszczyźnie y
<i>radius</i>	Promień przycisku
<i>color_passive_fill</i>	Kolor wnętrza przycisku na którym nie znajduje się kursor myszy
<i>color_passive_outline</i>	Kolor zarysu przycisku na którym nie znajduje się kursor myszy
<i>color_active_fill</i>	Kolor wnętrza przycisku na którym znajduje się kursor myszy
<i>color_active_outline</i>	Kolor zarysu przycisku na którym znajduje się kursor myszy

3.2.2 Dokumentacja funkcji składowych

3.2.2.1 drawTo()

```
void Button::drawTo (
    sf::RenderWindow & window )
```

Wyświetla przycisk na ekranie.

Parametry

<i>window</i>	Ekran na którym wyświetlony ma zostać przycisk
---------------	--

3.2.2.2 isClicked()

```
bool Button::isClicked (
    sf::RenderWindow & window,
    sf::Event & event )
```

Sprawdza czy przycisk został kliknięty.

Parametry

<i>window</i>	Ekran na którym znajduje się przycisk
<i>event</i>	Wydarzenie kliknięcia myszy na ekranie

Zwraca

true Przycisk został kliknięty

false Przycisk nie został kliknięty

3.2.2.3 isHovered()

```
bool Button::isHovered (
    sf::Vector2f mouse_position )
```

Sprawdza czy kursor myszy znajduje się na przycisku. Na tej podstawie zmienia wygląd przycisku z pasywnego na aktywny i umożliwia kliknięcie.

Parametry

<i>mouse_position</i>	Pozycja myszy w pikselach
-----------------------	---------------------------

Zwraca

true Kursor myszy znajduje się na przycisku

false Kursor myszy nie znajduje się na przycisku

3.2.2.4 setActiveTexture()

```
void Button::setActiveTexture (
    std::string file_location )
```

Ustawia aktywną teksturę przycisku opartego o teksturę.

Parametry

<i>file_location</i>	Lokacja pliku z teksturą aktywną
----------------------	----------------------------------

3.2.2.5 setButtonPosition()

```
void Button::setButtonPosition (
    float x,
    float y )
```

Ustawia pozycję przycisku.

Parametry

<i>x</i>	Pozycja na płaszczyźnie x
<i>y</i>	Pozycja na płaszczyźnie y

3.2.2.6 setPassiveTexture()

```
void Button::setPassiveTexture (
    std::string file_location )
```

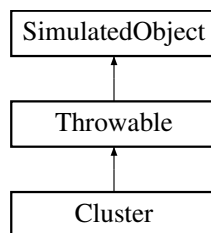
Ustawia pasywną teksturę przycisku opartego o teksturę.

Parametry

<i>file_location</i>	Lokacja pliku z teksturą pasywną
----------------------	----------------------------------

3.3 Dokumentacja klasy Cluster

Diagram dziedziczenia dla Cluster



Metody publiczne

- **Cluster** (b2World *world, float radius, bool is_add=false, float x=150, float y=600)

Konstruktor obiektu do rzucania, odłamkowego typu (**Cluster**). Ustalony zostaje typ obiektu, jego lokacja, tekstura i rozmiar.

Atrybuty publiczne

- bool collided = false

Określa czy obiekt brał udział w kolizji.

3.3.1 Dokumentacja konstruktora i destruktora

3.3.1.1 Cluster()

```

Cluster::Cluster (
    b2World * world,
    float radius,
    bool is_add = false,
    float x = 150,
    float y = 600 )
  
```

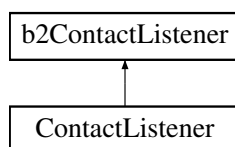
Konstruktor obiektu do rzucania, odłamkowego typu (**Cluster**). Ustalony zostaje typ obiektu, jego lokacja, tekstura i rozmiar.

Parametry

<i>world</i>	Świat w którym ma zostać stworzony obiekt
<i>radius</i>	Promień obiektu do rzucania
<i>is_add</i>	Określa czy zostaje stworzony obiekt Cluster czy jego odłamek
<i>x</i>	Lokacja odłamka na płaszczyźnie x
<i>y</i>	Lokacja odłamka na płaszczyźnie y

3.4 Dokumentacja klasy ContactListener

Diagram dziedziczenia dla ContactListener

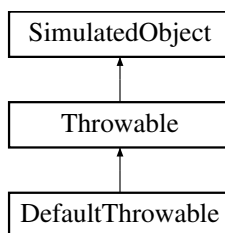


Atrybuty publiczne

- bool `cluster_contact` = false
Czy obiekt do rzucania uległ już kolizji.

3.5 Dokumentacja klasy DefaultThrowable

Diagram dziedziczenia dla DefaultThrowable



Metody publiczne

- `DefaultThrowable` (b2World *`world`, float `radius`)
Konstruktor obiektu do rzucania, standardowego typu (Default). Ustalony zostaje typ obiektu, jego lokacja, tekstura i rozmiar.

Dodatkowe Dziedziczone Składowe

3.5.1 Dokumentacja konstruktora i destruktor

3.5.1.1 DefaultThrowable()

```
DefaultThrowable::DefaultThrowable (
    b2World * world,
    float radius )
```

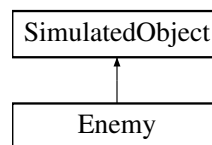
Konstruktor obiektu do rzucania, standardowego typu (Default). Ustalony zostaje typ obiektu, jego lokacja, tekstura i rozmiar.

Parametry

<i>world</i>	Świat w którym ma zostać stworzony obiekt
<i>radius</i>	Promień obiektu do rzucania

3.6 Dokumentacja klasy Enemy

Diagram dziedziczenia dla Enemy



Metody publiczne

- **Enemy** (b2World *world, float block_size_x, float block_size_y, float x, float y)

Konstruktor przeciwnika. Ustalona zostaje lokacja przeciwnika, tekstura, rozmiar i maksymalne HP.

Dodatkowe Dziedziczone Składowe

3.6.1 Dokumentacja konstruktora i destruktora

3.6.1.1 Enemy()

```

Enemy::Enemy (
    b2World * world,
    float block_size_x,
    float block_size_y,
    float x,
    float y )
  
```

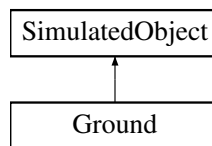
Konstruktor przeciwnika. Ustalona zostaje lokacja przeciwnika, tekstura, rozmiar i maksymalne HP.

Parametry

<i>world</i>	Świat w którym ma zostać stworzony przeciwnik
<i>block_size_x</i>	Szerokość przeciwnika
<i>block_size_y</i>	Wysokość przeciwnika
<i>x</i>	Położenie przeciwnika na płaszczyźnie x
<i>y</i>	Położenie przeciwnika na płaszczyźnie y

3.7 Dokumentacja klasy Ground

Diagram dziedziczenia dla Ground



Metody publiczne

- **Ground** (b2World *world, float block_size_x, float block_size_y, GroundType ground_type, float x, float y)
Konstruktor podłoża w oparciu o teksturę. Ustalony zostaje typ podłoża, jej lokacja, tekstura i rozmiar.
- **Ground** (b2World *world, float block_size_x, float block_size_y, sf::Color color, float x, float y)
Konstruktor podłoża w oparciu o kolor. Ustalona zostaje lokacja podłoża, kolor i rozmiar.

Dodatkowe Dziedziczone Składowe

3.7.1 Dokumentacja konstruktora i destruktora

3.7.1.1 Ground() [1/2]

```

Ground::Ground (
    b2World * world,
    float block_size_x,
    float block_size_y,
    GroundType ground_type,
    float x,
    float y )
  
```

Konstruktor podłoża w oparciu o teksturę. Ustalony zostaje typ podłoża, jej lokacja, tekstura i rozmiar.

Parametry

<i>world</i>	Świat w którym ma zostać stworzony blok podłoża
<i>block_size_x</i>	Szerokość podłoża
<i>block_size_y</i>	Wysokość podłoża
<i>ground_type</i>	Typ podłoża
<i>x</i>	Położenie podłoża na płaszczyźnie x
<i>y</i>	Położenie podłoża na płaszczyźnie y

3.7.1.2 Ground() [2/2]

```
Ground::Ground (
    b2World * world,
    float block_size_x,
    float block_size_y,
    sf::Color color,
    float x,
    float y )
```

Konstruktor podłoża w oparciu o kolor. Ustalona zostaje lokacja podłoża, kolor i rozmiar.

Parametry

<i>world</i>	Świat w którym ma zostać stworzony blok podłoża
<i>block_size_x</i> ↔ <i>_x</i>	Szerokość podłoża
<i>block_size_y</i> ↔ <i>_y</i>	Wysokość podłoża
<i>color</i>	Kolor podłoża
<i>x</i>	Położenie podłoża na płaszczyźnie x
<i>y</i>	Położenie podłoża na płaszczyźnie y

3.8 Dokumentacja klasy Level

Metody publiczne

- [Level](#) (int level_number, int throwable_number, GroundType level_type, int screen_width=1400, int screen_height=800)
Konstruktor obiektu typu [Level](#), określa grawitację i tło.
- [~Level](#) ()
Destruktor obiektu typu [Level](#) który jednocześnie niszczy i zwalnia pamięć wszystkich stworzonych obiektów.
- void [createGround](#) (GroundType ground_type, float x, float y)
Tworzy pojedynczy blok podłoża i umieszcza pointer stworzonego obiektu w wektorze objects. Rozmiar bloku podłoża jest stały.
- void [createObstacle](#) (ObstacleType obstacle_type, float x, float y, float box_size_x=35.f, float box_size_y=35.f)
Tworzy pojedynczy blok przeszkody i umieszcza pointer stworzonego obiektu w wektorze objects.
- void [createEnemy](#) (float x, float y, float box_size_x=35.f, float box_size_y=35.f)
Tworzy pojedynczy blok przeciwnika i umieszcza pointer stworzonego obiektu w wektorze objects.
- void [makeLevel](#) ()
Tworzy całe podłoże, granice i obiekty do rzutu poziomym. Pozwala stworzyć obiekty - w dowolnych rozmiarach i dowolnych pozycjach po edycji definicji funkcji.
- void [throwableUsed](#) ()
Wprowadza aktywny obiekt do rzutu do kontenera wszystkich obiektów. Czyni go to obiektem do fizycznej symulacji który może wchodzić w interakcje z innymi obiektami. Oznacza rzucony obiekt jako użyty.
- void [createBombardingAdds](#) ()
Tworzy maksymalnie 3 odłamki od obiektu typu [Bombarding](#). Odłamki są wystrzeliwane z prędkością liniową skierowaną w dół.
- void [createClusterAdds](#) ()

Tworzy 8 odłamków po kolizji obiektu typu [Cluster](#). Stworzone obiekty są wstrzeliwane z prędkością liniową w 4 różnych kierunkach.

- void [checkThrowableMovement](#) ()
Sprawdza czy obiekty do rzucania skończyły się poruszać. Obiekt który przestał się poruszać zostaje oflagowany jako nieporuszający się.
- void [destroyFlaggedObjects](#) ()
Niszczy obiekt którego HP jest mniejsze lub równe 0 przyznając punkty równe jego maksymalnego HP * 10. Jeżeli HP spadnie poniżej połowy maksymalnego HP to obiekt zostaje oznaczony jako poniszczony i zmienia swoją teksturę.
- bool [isLevelWon](#) ()
Iteruje poprzez wszystkie obiekty ze złożonością obliczeniową $O(n)$ i sprawdza czy istnieje obiekt typu [Enemy](#).
- bool [isLevelLost](#) ()
Sprawdza czy obiekty do rzucania skończyły się poruszać.
- void [addPointsForUnusedThrows](#) ()
Przyznaje punkty za każdy z nieużytych obiektów do rzucania. 3000 punktów za Default, 5000 punktów za [Cluster](#) i [Bombarding](#).
- b2World * [getWorldPointer](#) ()
Zwraca wskaźnik do świata.
- std::vector< [SimulatedObject](#) * > * [getObjects](#) ()
Zwraca wskaźnik do obiektów.
- [Throwable](#) * [getActiveThrowable](#) ()
Zwraca aktywny obiekt do rzucania.
- sf::RectangleShape [getBackgroud](#) ()
Zwraca grafike tła.
- int [getScore](#) ()
Zwraca ilość punktów.
- int [getLevelNumber](#) ()
Zwraca numer poziomu.
- GroundType [getLevelType](#) ()
Zwraca typ poziomu.
- [Throwable](#) * [getThrowableByNumber](#) (int throwable_number)
Zwraca wskaźnik do obiektu do rzucania na podstawie jego numeru w tablicy. Dla stopnia ulepszenia rzutów 1 - wszystkie są Default. Dla stopnia ulepszenia rzutów 2 - rzut numer 2 to [Cluster](#), reszta Default. Dla stopnia ulepszenia rzutów 3 - rzut numer 2 to [Bombarding](#), rzut numer 1 to [Cluster](#), rzut numer 0 to Default.
- void [setActiveThrowable](#) (int throwable_number)
Ustawia obiektu do rzucania jako aktywny na podstawie jego numeru w tablicy. Dla stopnia ulepszenia rzutów 1 - wszystkie są Default. Dla stopnia ulepszenia rzutów 2 - rzut numer 2 to [Cluster](#), reszta Default. Dla stopnia ulepszenia rzutów 3 - rzut numer 2 to [Bombarding](#), rzut numer 1 to [Cluster](#), rzut numer 0 to Default.

3.8.1 Dokumentacja konstruktora i destruktora

3.8.1.1 Level()

```
Level::Level (
    int level_number,
    int throwable_number,
    GroundType level_type,
    int screen_width = 1400,
    int screen_height = 800 )
```

Konstruktor obiektu typu [Level](#), określa grawitację i tło.

Parametry

<i>level_number</i>	Liczba całkowita jednoznacznie identyfikująca Level
<i>throwable_amount</i>	Określa stopień ulepszonych rzutów, 1 - wszystkie Default, 2 - jeden Cluster , 3 - jeden Cluster i Bombarding
<i>level_type</i>	Typ Levelu określający: podłoże, tło i grawitację
<i>screen_width</i>	Szerokość ekranu
<i>screen_height</i>	Wysokość ekranu

3.8.2 Dokumentacja funkcji składowych

3.8.2.1 createEnemy()

```
void Level::createEnemy (
    float x,
    float y,
    float box_size_x = 35.f,
    float box_size_y = 35.f )
```

Tworzy pojedynczy blok przeciwnika i umieszcza pointer stworzonego obiektu w wektorze objects.

Parametry

<i>x</i>	Lokacja stworzenia obiektu na koordynacie x
<i>y</i>	Lokacja stworzenia obiektu na koordynacie y
<i>box_size</i> ↔ <i>_x</i>	Szerokość przeciwnika w pikselach
<i>box_size</i> ↔ <i>_y</i>	Wysokość przeciwnika w pikselach

3.8.2.2 createGround()

```
void Level::createGround (
    GroundType ground_type,
    float x,
    float y )
```

Tworzy pojedynczy blok podłoża i umieszcza pointer stworzonego obiektu w wektorze objects. Rozmiar bloku podłoża jest stały.

Parametry

<i>ground_type</i>	Typ podłoża: Normal, Moon
<i>x</i>	Lokacja stworzenia obiektu na koordynacie x
<i>y</i>	Lokacja stworzenia obiektu na koordynacie y

3.8.2.3 createObstacle()

```
void Level::createObstacle (
    ObstacleType obstacle_type,
    float x,
    float y,
    float box_size_x = 35.f,
    float box_size_y = 35.f )
```

Tworzy pojedynczy blok przeszkody i umieszcza pointer stworzonego obiektu w wektorze objects.

Parametry

<i>obstacle_type</i>	Typ przeszkody: Wood, Stone, Metal
<i>x</i>	Lokacja stworzenia obiektu na koordynacie x
<i>y</i>	Lokacja stworzenia obiektu na koordynacie y
<i>box_size_x</i>	Szerokość przeszkody w pikselach
<i>box_size_y</i>	Wysokość przeszkody w pikselach

3.8.2.4 getActiveThrowable()

```
Throwable * Level::getActiveThrowable ( )
```

Zwraca aktywny obiekt do rzucania.

Zwraca

Throwable* Wskaźnik do aktywnego obiektu do rzucania

3.8.2.5 getBackgroud()

```
sf::RectangleShape Level::getBackgroud ( )
```

Zwraca grafike tła.

Zwraca

sf::RectangleShape Grafika tła

3.8.2.6 getLevelNumber()

```
int Level::getLevelNumber ( )
```

Zwraca numer poziomu.

Zwraca

int Liczba całkowita jednoznacznie identyfikująca [Level](#)

3.8.2.7 getLevelType()

```
GroundType Level::getLevelType ( )
```

Zwraca typ poziomu.

Zwraca

GroundType Typ Levelu określający: podłoże, tło i grawitację

3.8.2.8 getObjects()

```
std::vector< SimulatedObject * > * Level::getObjects ( )
```

Zwraca wskaźnik do obiektów.

Zwraca

std::vector<SimulatedObject*>* Wskaźnik na kontener wskaźników do wszystkich stworzonych obiektów: przeszkód, przeciwników, podłoża, rzutów

3.8.2.9 getScore()

```
int Level::getScore ( )
```

Zwraca ilość punktów.

Zwraca

int Punkty zdobyte na poziomie

3.8.2.10 getThrowableByNumber()

```
Throwable * Level::getThrowableByNumber (
    int throwable_number )
```

Zwraca wskaźnik do obiektu do rzucania na podstawie jego numeru w tablicy. Dla stopnia ulepszenia rzutów 1 - wszystkie są Default. Dla stopnia ulepszenia rzutów 2 - rzut numer 2 to [Cluster](#), reszta Default. Dla stopnia ulepszenia rzutów 3 - rzut numer 2 to [Bombarding](#), rzut numer 1 to [Cluster](#), rzut numer 0 to Default.

Parametry

<code>throwable_number</code>	Numer obiektu do rzucania od 0 do 2
-------------------------------	-------------------------------------

Zwraca

Throwable* Wskaźnik obiektu do rzucania

3.8.2.11 getWorldPointer()

```
b2World * Level::getWorldPointer ( )
```

Zwraca wskaźnik do świata.

Zwraca

b2World* Wskaźnik do świata w którym znajdują się obiekty na których przeprowadzane są obliczenia ich: położenia, przyspieszenia, prędkości, rotacji i kolizji

3.8.2.12 isLevelLost()

```
bool Level::isLevelLost ( )
```

Sprawdza czy obiekty do rzucania skończyły się poruszać.

Zwraca

true Każdy z obiektów do rzucania skończył się poruszać
false Istnieje przynajmniej 1 obiekt do rzucania który jeszcze się porusza

3.8.2.13 isLevelWon()

```
bool Level::isLevelWon ( )
```

Iteruje poprzez wszystkie obiekty ze złożonością obliczeniową $O(n)$ i sprawdza czy istnieje obiekt typu [Enemy](#).

Zwraca

true Nie znaleziono obiektu typu [Enemy](#)
false Znaleziono obiekt typu [Enemy](#)

3.8.2.14 setActiveThrowable()

```
void Level::setActiveThrowable (
    int throwable_number )
```

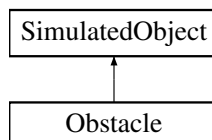
Ustawia obiektu do rzucania jako aktywny na podstawie jego numeru w tablicy. Dla stopnia ulepszenia rzutów 1 - wszystkie są Default. Dla stopnia ulepszenia rzutów 2 - rzut numer 2 to [Cluster](#), reszta Default. Dla stopnia ulepszenia rzutów 3 - rzut numer 2 to [Bombarding](#), rzut numer 1 to [Cluster](#), rzut numer 0 to Default.

Parametry

<code>throwable_number</code>	Numer obiektu do rzucania od 0 do 2
-------------------------------	-------------------------------------

3.9 Dokumentacja klasy Obstacle

Diagram dziedziczenia dla Obstacle



Metody publiczne

- **Obstacle** (`b2World *world`, `float block_size_x`, `float block_size_y`, `ObstacleType obstacle_type`, `float x`, `float y`)
Konstruktor przeszkody. Ustalony zostaje typ przeszkody, jej lokacja, tekstura i rozmiar.

Dodatkowe Dziedziczone Składowe

3.9.1 Dokumentacja konstruktora i destruktora

3.9.1.1 Obstacle()

```

Obstacle::Obstacle (
    b2World * world,
    float block_size_x,
    float block_size_y,
    ObstacleType obstacle_type,
    float x,
    float y )
  
```

Konstruktor przeszkody. Ustalony zostaje typ przeszkody, jej lokacja, tekstura i rozmiar.

Parametry

<code>world</code>	Świat w którym ma zostać stworzona przeszkoda
<code>block_size_x</code>	Szerokość przeszkody
<code>block_size_y</code>	Wysokość przeszkody
<code>obstacle_type</code>	Typ przeszkody: drewno, kamień, metal
<code>x</code>	Położenie przeszkody na płaszczyźnie x
<code>y</code>	Położenie przeszkody na płaszczyźnie y

3.10 Dokumentacja klasy Screen

Metody publiczne

- `Screen` (int screen_width=1400, int screen_height=800)
Konstruktor okna. Tworzy wszystkie potrzebne okna, przyciski i napisy potrzebne do wyświetlenia gry.
- `~Screen` ()
Destruktor okna. Niszczy i zwalnia pamięć aktywnego poziomu i wszystkich przycisków.
- void `loadScore` (SaveType save_type)
Wczytuje punkty do tablicy scores z pliku binarnego lub tekstowego.
- void `loadLevelsLayout` (SaveType save_type)
Wczytuje design poziomów do tablicy level_design z pliku binarnego lub tekstowego.
- void `saveScore` (SaveType save_type)
Zapisuje tablice scores do pliku binarnego lub tekstowego.
- void `createWindow` ()
Tworzy okno w którym toczy się cała gra. Wywołuje funkcje i operacje w zależności od stanu gry. Odpowiada za wyświetlanie grafik i odbieranie inputu użytkownika w postaci zmiennej event.

3.10.1 Dokumentacja konstruktora i destruktora

3.10.1.1 Screen()

```
Screen::Screen (
    int screen_width = 1400,
    int screen_height = 800 )
```

Konstruktor okna. Tworzy wszystkie potrzebne okna, przyciski i napisy potrzebne do wyświetlenia gry.

Parametry

<code>screen_width</code>	Szerokość ekranu
<code>screen_height</code>	Wysokość ekranu

3.10.2 Dokumentacja funkcji składowych

3.10.2.1 loadLevelsLayout()

```
void Screen::loadLevelsLayout (
    SaveType save_type )
```

Wczytuje design poziomów do tablicy level_design z pliku binarnego lub tekstowego.

Parametry

<code>save_type</code>	Wybór pliku binarnego lub tekstowego do wczytania
------------------------	---

3.10.2.2 loadScore()

```
void Screen::loadScore (
    SaveType save_type )
```

Wczytuje punkty do tablicy scores z pliku binarnego lub tekstowego.

Parametry

<code>save_type</code>	Wybór pliku binarnego lub tekstowego do wczytania
------------------------	---

3.10.2.3 saveScore()

```
void Screen::saveScore (
    SaveType save_type )
```

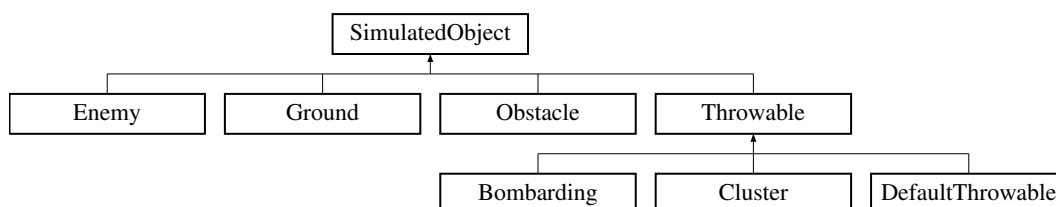
Zapisuje tablice scores do pliku binarnego lub tekstowego.

Parametry

<code>save_type</code>	Wybór pliku binarnego lub tekstowego do zapisu
------------------------	--

3.11 Dokumentacja klasy SimulatedObject

Diagram dziedziczenia dla SimulatedObject

**Metody publiczne**

- virtual void [createObject](#) ()=0

Metoda czysto wirtualna.

- void `initializePosition` (b2World *`world`, float `x`, float `y`)
Ustawia świat w którym znajduje się obiekt i jego położenie.
- void `updatePosition` ()
Ustawia odpowiednie położenie i rotację grafiki obiektu, w oparciu o jego interakcje z innymi obiektami w świecie.
- void `deleteObject` ()
Usuwa obiekt ze świata.
- void `setAsDamaged` ()
Ustawia teksturę obiektu na wersję poniszczoną.

Atrybuty publiczne

- float `x`
Położenie obiektu na płaszczyźnie x.
- float `y`
Położenie obiektu na płaszczyźnie y.
- float `hp` = 1
HP obiektu.
- float `max_hp` = 1
Maksymalne HP obiektu.
- float `block_size_x` = 0
Szerokość bloku.
- float `block_size_y` = 0
Wysokość bloku.
- float `radius` = 0
Promień obiektu.
- sf::RectangleShape `graphics`
Grafika obiektu.
- sf::Texture `texture`
Tekstura obiektu.
- sf::Texture `texture_damaged`
Tekstura poniszczonego obiektu.
- b2World * `world` = nullptr
Świat do którego należy obiekt.
- b2Body * `physics`
Fizyczne właściwości obiektu.
- ObjectType `object_type`
Typ obiektu: przeszkoda, podłoże, przeciwnik, obiekt do rzucania.

3.11.1 Dokumentacja funkcji składowych

3.11.1.1 initializePosition()

```
void SimulatedObject::initializePosition (
    b2World * world,
    float x,
    float y )
```

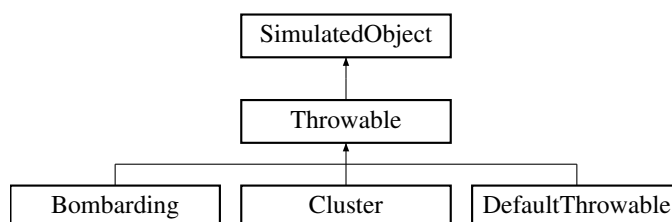
Ustawia świat w którym znajduje się obiekt i jego położenie.

Parametry

<i>world</i>	Świat w którym znajduje się obiekt
<i>x</i>	Położenie obiektu na płaszczyźnie x
<i>y</i>	Położenie obiektu na płaszczyźnie y

3.12 Dokumentacja klasy Throwable

Diagram dziedziczenia dla Throwable



Metody publiczne

- void `createObject` ()
Tworzy obiekt który jest gotowy do symulacji fizyki. Określa jego parametry po stworzeniu: pozycje, rozmiar, gęstość, tarcie, sprężystość, tłumienie kątowe.
- float `getTrajectoryX` (float thr_shifted_x, float n)
Zwraca przyszłą lokację obiektu na płaszczyźnie x. Bierze pod uwagę startową prędkość po wystrzeleniu i grawitację.
- float `getTrajectoryY` (float thr_shifted_y, float n)
Zwraca przyszłą lokację obiektu na płaszczyźnie y. Bierze pod uwagę startową prędkość po wystrzeleniu i grawitację.
- void `launch` (float thr_shifted_x, float thr_shifted_y)
Tworzy obiekt który został rzucony, z prędkością liniową określoną pozycją myszy.
- void `checkIfMoving` ()
Ustawia obiekt jako nieruchomy jeżeli suma jego prędkości na płaszczyźnie x i y nie przekracza 0.2f.
- ThrowableState `getThrowableState` ()
Zwraca stan obiektu do rzucania.
- void `setThrowableState` (ThrowableState state)
Ustawia stan obiektu do rzucania.

Atrybuty publiczne

- float `thr_start_x` = 150
Pozycja startowa (na płaszczyźnie x) obiektu do rzucania.
- float `thr_start_y` = 600
Pozycja startowa (na płaszczyźnie y) obiektu do rzucania.
- bool `is_moving` = true
Określa czy obiekt jest wciąż w ruchu.
- ThrowableType `throwable_type`
Typ obiektu do rzucania.

3.12.1 Dokumentacja funkcji składowych

3.12.1.1 getThrowableState()

```
ThrowableState Throwable::getThrowableState ( )
```

Zwraca stan obiektu do rzucania.

Zwraca

ThrowableState Stan obiektu (oczekujący, celujący, użyty)

3.12.1.2 getTrajectoryX()

```
float Throwable::getTrajectoryX (
    float thr_shifted_x,
    float n )
```

Zwraca przyszłą lokację obiektu na płaszczyźnie x. Bierze pod uwagę startową prędkość po wystrzeleniu i grawitację.

Parametry

<i>thr_shifted_x</i>	Pozycja kursora myszy na płaszczyźnie x po wystrzeleniu
<i>n</i>	Ilość skoków w czasie

Zwraca

float Przyszłą lokację obiektu na płaszczyźnie x

3.12.1.3 getTrajectoryY()

```
float Throwable::getTrajectoryY (
    float thr_shifted_y,
    float n )
```

Zwraca przyszłą lokację obiektu na płaszczyźnie y. Bierze pod uwagę startową prędkość po wystrzeleniu i grawitację.

Parametry

<i>thr_shifted</i> _↔ <i>_y</i>	Pozycja kursora myszy na płaszczyźnie y po wystrzeleniu
<i>n</i>	Ilość skoków w czasie

Zwraca

float Przyszłą lokację obiektu na płaszczyźnie y

3.12.1.4 launch()

```
void Throwable::launch (
    float thr_shifted_x,
    float thr_shifted_y )
```

Tworzy obiekt który został rzucony, z prędkością liniową określoną pozycją myszy.

Parametry

<i>thr_shifted</i> _↔ <i>_x</i>	Pozycja myszy na płaszczyźnie x
<i>thr_shifted</i> _↔ <i>_y</i>	Pozycja myszy na płaszczyźnie y

3.12.1.5 setThrowableState()

```
void Throwable::setThrowableState (
    ThrowableState state )
```

Ustawia stan obiektu do rzucania.

Parametry

<i>state</i>	Stan obiektu (oczekujący, celujący, użyty)
--------------	--

Indeks

- Bombarding, [5](#)
 - Bombarding, [5](#)
- Button, [6](#)
 - Button, [7](#), [8](#)
 - drawTo, [9](#)
 - isClicked, [9](#)
 - isHovered, [9](#)
 - setActiveTexture, [10](#)
 - setButtonPosition, [10](#)
 - setPassiveTexture, [10](#)
- Cluster, [11](#)
 - Cluster, [11](#)
- ContactListener, [12](#)
- createEnemy
 - Level, [17](#)
- createGround
 - Level, [17](#)
- createObstacle
 - Level, [18](#)
- DefaultThrowable, [12](#)
 - DefaultThrowable, [12](#)
- drawTo
 - Button, [9](#)
- Enemy, [13](#)
 - Enemy, [13](#)
- getActiveThrowable
 - Level, [18](#)
- getBackgroud
 - Level, [18](#)
- getLevelNumber
 - Level, [18](#)
- getLevelType
 - Level, [19](#)
- getObjects
 - Level, [19](#)
- getScore
 - Level, [19](#)
- getThrowableByNumber
 - Level, [19](#)
- getThrowableState
 - Throwable, [26](#)
- getTrajectoryX
 - Throwable, [26](#)
- getTrajectoryY
 - Throwable, [26](#)
- getWorldPointer
 - Level, [20](#)
- Ground, [14](#)
 - Ground, [14](#)
- initializePosition
 - SimulatedObject, [24](#)
- isClicked
 - Button, [9](#)
- isHovered
 - Button, [9](#)
- isLevelLost
 - Level, [20](#)
- isLevelWon
 - Level, [20](#)
- launch
 - Throwable, [27](#)
- Level, [15](#)
 - createEnemy, [17](#)
 - createGround, [17](#)
 - createObstacle, [18](#)
 - getActiveThrowable, [18](#)
 - getBackgroud, [18](#)
 - getLevelNumber, [18](#)
 - getLevelType, [19](#)
 - getObjects, [19](#)
 - getScore, [19](#)
 - getThrowableByNumber, [19](#)
 - getWorldPointer, [20](#)
 - isLevelLost, [20](#)
 - isLevelWon, [20](#)
 - Level, [16](#)
 - setActiveThrowable, [20](#)
- loadLevelsLayout
 - Screen, [22](#)
- loadScore
 - Screen, [23](#)
- Obstacle, [21](#)
 - Obstacle, [21](#)
- saveScore
 - Screen, [23](#)
- Screen, [22](#)
 - loadLevelsLayout, [22](#)
 - loadScore, [23](#)
 - saveScore, [23](#)
 - Screen, [22](#)
- setActiveTexture
 - Button, [10](#)

setActiveThrowable

Level, [20](#)

setButtonPosition

Button, [10](#)

setPassiveTexture

Button, [10](#)

setThrowableState

Throwable, [27](#)

SimulatedObject, [23](#)

initializePosition, [24](#)

Throwable, [25](#)

getThrowableState, [26](#)

getTrajectoryX, [26](#)

getTrajectoryY, [26](#)

launch, [27](#)

setThrowableState, [27](#)