13.06.2015

PROGRAMOWANIE KOMPUTERÓW

Temat projektu:
GRA BOULDER DASH

prowadzący: dr inż. Roman Starosolski

autor: Mateusz Nurek gr.2

1. Temat
Tematem projektu było napisanie w C++ własnej wersji gry 2D Boulder Dash.

2. Analiza tematu
Głównym celem gry jest zbieranie diamentów i unikanie kamieni. Realizując swój projekt chciałem rozszerzyć grę o różne rodzaje ww. Przedmiotów.
Wstępny wybór klas wyglądał następująco:
- klasa Player
- klasa Level – przechowująca informacje na temat poziomu (rozmieszczenie wszystkich obiektów na planszy)
- klasa Stone – klasa bazowa po której dziedziczą wszystkie dodatkowe rodzaje kamieni.
- klasa Diamond – klasa bazowa po której dziedziczą wszystkie dodatkowe rodzaje diamentów.

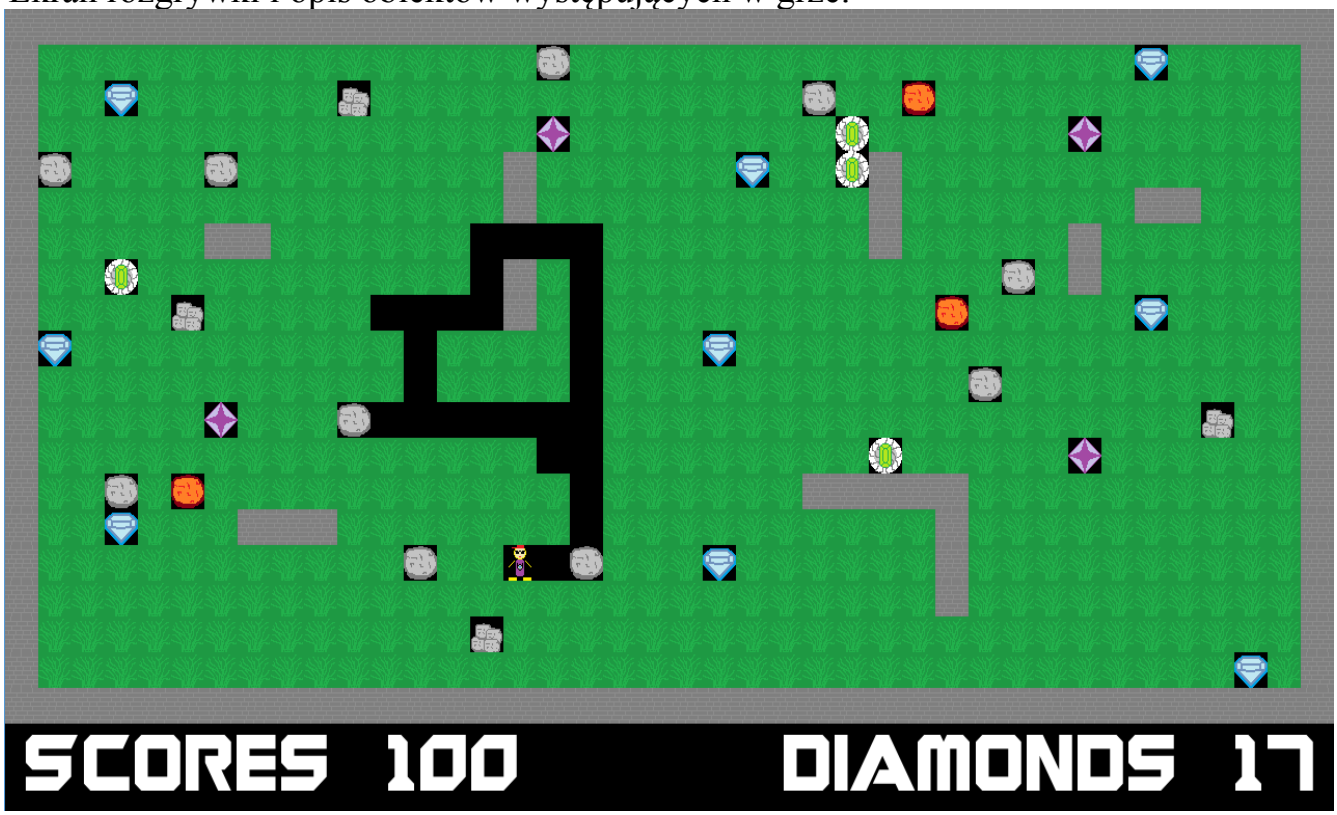
Do stworzenia gry użyłem biblioteki SFML w wersji 2.1. Główną zaletą tej biblioteki jest jej obiektowość co znacznie upraszcza pracę z nią. Kolejną zaletą SFML jest jej wieloplatformowość. W swoim projekci używałem głównie następujących modułów tej biblioteki:
- Window- obsługa okna i interakcji z użytkownikiem
- Graphics – renderowanie grafiki
- System – obsługa czasu

3. Specyfikacja zewnętrzna
Boulder Dash jest grą logiczno – zręcznościową. Celem gry jest zebranie wszystkich diamentów na planszy oraz unikanie kamieni.

Główne menu:

Użytkownik ma do wyboru 3 opcje (wybór przy pomocy myszki)
START – rozpoczęcie gry
BEST SCORES – wyświetlenie listy pięciu najlepszych graczy
EXIT – wyjście z programu

Ekran rozgrywki i opis obiektów występujących w grze:



W lewym dolnym rogu widoczna jest informacja o zebranych punktach, a w prawym dolnym rogu widnieje informacja o liczbie diamentó pozostałych do zebrania.

 Postać gracza

 Zwykły diament

 Kruchy diament. Jeśli spadnie z wysokości lub zleci na niego kamień rozbije tym samym uniemożliwiając zebranie wszystkich diamentów co kończy grę.

 Diament w muszli, aby go zebrać należy na niego zrzucić kamień aby go rozbić

 Diament po rozbiciu muszli, możliwy do zebrania.

 Zwykły kamień, jeśli spadnie na gracza to gra się kończy. Można przesuwać go na boki.

 Wybuchający kamień. Kiedy spadnie z wysokości wybucha niszcząc sąsiadujące pola jeśli są trawą lub zabija gracza jeśli znajduje się w strefie wybuchu. Można przesuwać na boki.

 Sterta kamieni – po spadnięciu z wysokości tworzy zwykłe kamienie na sąsiadujących polach które są trawą. Nie można przesuwać na boki.

 Trawa jest barierą dla kamieni I diamentów. Jeśli gracz wejdzie na takie pole to trawa zniknie.

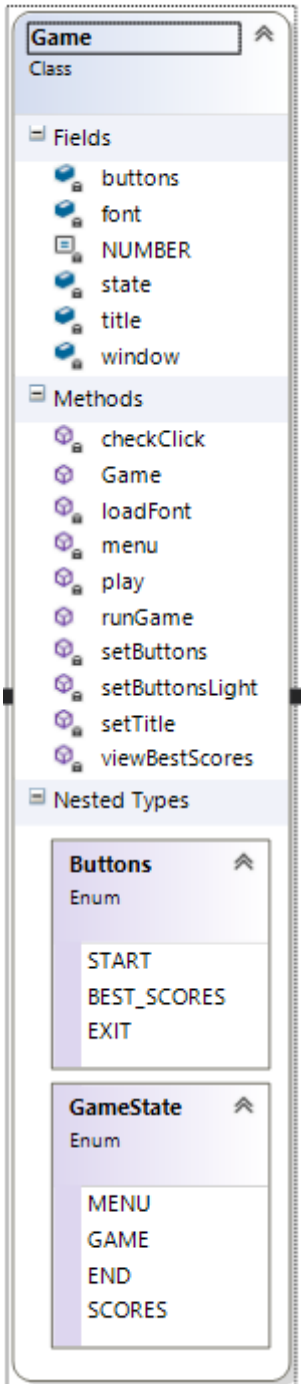
 Kamienna ściana stanowi barierę dla gracza I wszystkich obiektów w grze.

Sterowanie:
W – góra
S - dół
A – lewp
D – prawo
SPACJA + A – przesuwa kamień w lewo.
SPACJA + D – przesuwa kamień w prawo.
ESC – powrót do głównego menu.

Jeśli gracz zbierze wystarczająco dużą ilość punktów to na koniec gry będzie mógł wpisać się na listę najlepszych graczy.

4. Specyfikacja wewnętrzna
Istotne klasy:

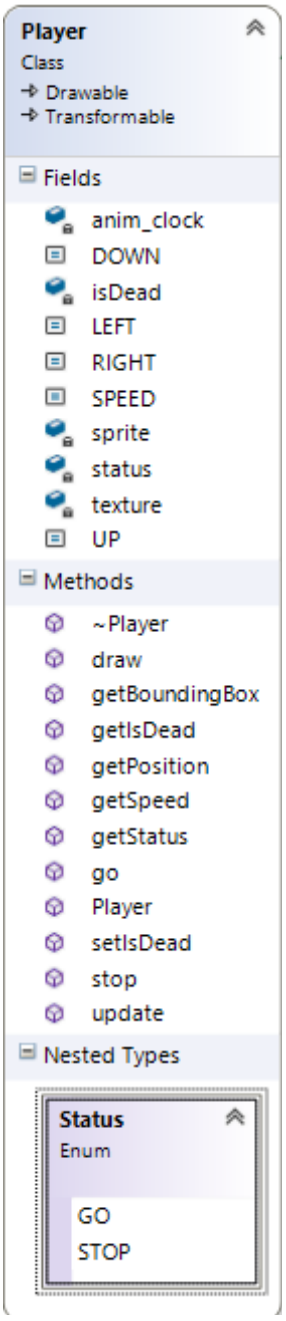
Klasa Game – główna klasa zarządzająca stanem programu. Jej zadaniem jest odpowiednia reakcja gry na polecenia użytkownika: wyświetlenie menu, uruchomienie rozgrywki, pokazanie listy najlepszych wyników lub wyjście z aplikacji.



Najważniejsze pola:
GameState **state –** przechowuje aktualny stan programu: menu, rozgrywka, najlepsze wyniki, wyjście
sf::RenderWindow **window** – okno programu

Najważniejsze metody:
void **runGame( )** - metoda zawiera główną pętlę programu w której zależnie od zmiennej state wywoływane są metody odpowiedzialne za poszczególne moduły programu
void **menu( )** - metoda odpowiedzialna za wyświetlanie głównego menu programu.
void **play( )** - w metodzie jest tworzony obiekt klasy Engine odpowiedzialny za włączenie gry.
void **viewBestScores( )** - metoda odpowiedzialna za wyświetlenie listy najlepszych graczy.

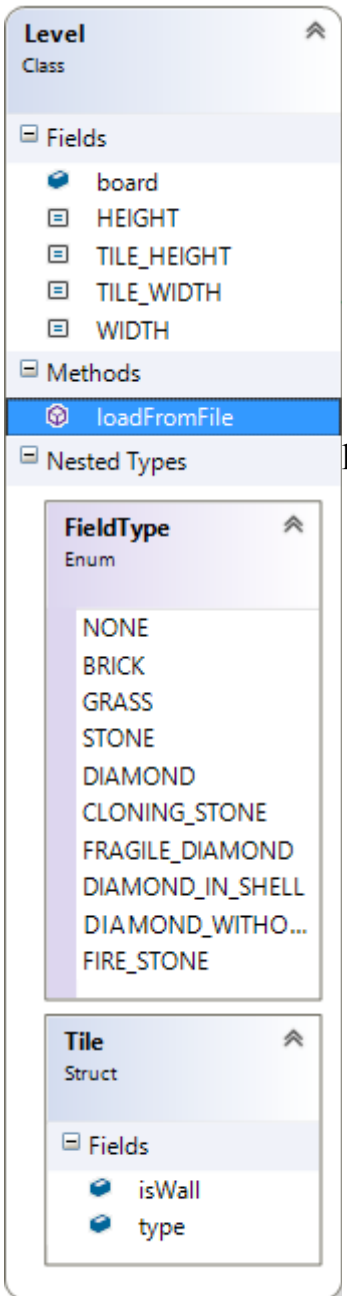
Klasa Player – odpowiedzialna za postać gracza w grze.



Najważniejsze pola:
bool **isDead** – zmienna przechowuje informacje na temat stanu gracza (żywy / martwy).
const double **SPEED –** określa prędkość gracza
Status **status –** określa czy gracz aktualnie się porusza (wartość GO) czy stoi (wartość STOP)

Najważniejsze metody:
sf::FloatRect **getBoundingBox( )** - zwraca współrzędne lewego górnego wierzchołka prostokąta opakowującego postać gracza. Metoda wykorzystywana do obliczenia współrzędnych gracza na planszy.
void **update(**int direction**)** – aktualizuje położenie gracza przesuwając go w kierunku przekazanym jako argument metody.

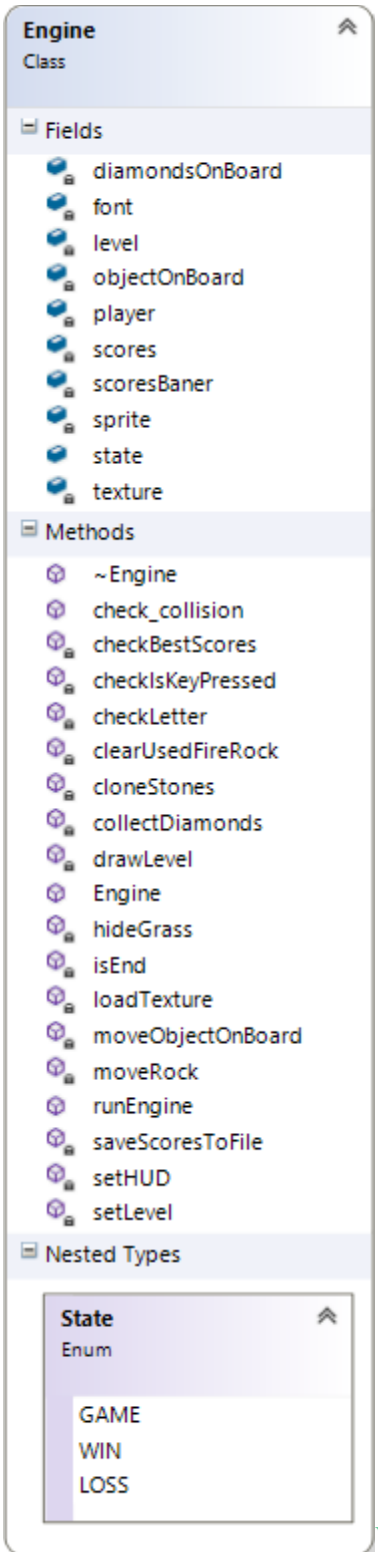
Klasa Level – przechowuje informacje na temat wyglądu poziomu.



Najważniejsze pola:
Tile **board[**height**][**width**]** – tablica przechowuje informacje o rozmieszczeniu wszystkich obiektów na planszy.
static const int **HEIGHT** – długość poziomu (ilość kafelek)
static const int **WIDTH** – szerokość poziomu (ilość kafelek)
static const int **TILE_HEIGHT** – wysokość tekstury kafelka w pikselach.
static const int **TILE_WIDTH** – długość tekstury kafelka w pikselach.
void **loadFromFile(**std::string filename**)** – wczytuje z pliku poziom

Klasa zawiera także zagnieżdżoną strukturę **Tile** która zawiera następujące pola:
bool **isWall** - określenie czy dany kafelek jest barierą dla gracza
FieldType **type** - informacja o typie kafelka.

Klasa Engine – klasa będąca silnikiem gry zawierająca logikę odpowiedzialną za rozgrywkę.

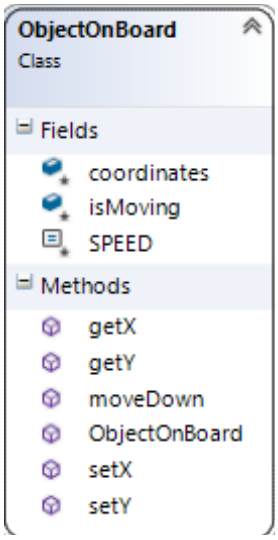


Najważniejsze pola:
sf::Text **diamondsOnBoard -** napis pokazujący liczbę pozostałych na planszy diamentów.
Level **level** – obiekt klasy Level przechowujący informacje o poziomie.
list<ObjectOnBoard*> **objectOnBoard –** lista przechowująca wszystkie diamenty i kamienie znajdujące się na planszy.
Player **player –** obiekt klasy Player reprezentujący postać gracza.
int **scores –** przechowuje liczbę zebranych punktów
sf::Text **scoresBaner -** napis pokazujący liczbę zebranych punktów
sf::Sprite **sprite[**Level::HEIGHT**][**Level::WIDTH**] –** tablica przechowuje rozmieszczenie tekstur na planszy.
State **state** – przechowuje informację o stanie gry: gra w trakcie (wartość GAME), wygrana (wartość WIN), przegrana (wartość LOSS).
sf::Texture **texture –** przechowuje tekstury gry.

Najważniejsze metody:
void **check_collision(**int direction**) –** sprawdza kolizję gracza ze ścianami. Metoda otrzymuje jako argument kierunek w którym gracz chce się przesunąć i jeścli ruch jest niedozwolony zatrzymuje gracza zmieniając jego status na STOP.
void **checkBestScores(**sf::RenderWindow& window**) -** sprawdza czy po skończonej grze uzyskany wynik kwalifikuje się do listy najlepszych graczy i jeśli tak to przyjmuje dane od gracza.
void **checkIsKeyPressed(**int& direction, Player& player, bool& menu**) –** metoda odpowiedzialna za wprowadzanie danych z klawiatury przez gracza. Wczytuje kierunek ruchu lub chęć wyjścia z gry.
void **clearUsedFireStone( )** - metoda przeszukuje listę obiektów na planszy i usuwa z niej kamienie które już wybuchły.
void **cloneStones( ) -** metoda przeszukuje listę obiektów na planszy wyszukując obiektów ClonningStone, które mają dodać do listy obiektów na planszy dodatkowe kamienie, a następnie usuwa znaleziony obiekt.

void **collectDiamonds( )** - metoda odpowiedzialna za zbieranie diamentów. Przeszukuje listę obiektów na planszy. Jeśli jakiś obiekt ma takie same współrzędne jak gracz to metoda sprawdza czy dany obiekt jest diamentem i jeśli tak to przyznaje punkty, usuwa zebrany diament z listy obiektów i aktualizuje wyświetlane informacje o stanie diamentów i punktów.
void **drawLevel(**sf::RenderWindow& **window) –** metoda odpowiedzialna za rysowanie poziomu w oknie programu.
void **hideGrass( )** - metoda oblicza współrzędne gracza i jeśli pole na którym stoi gracz jest typu GRASS to metoda zmienia ten typ na puste pole NONE.
bool **isEnd(**State state, sf::RenderWindow & window**) –** sprawdza aktualny stan gry. Jeśli pole state ma wartość WIN lub LOSS lub jeśli gracz jest martwy to metoda wyświetla odpowiedni komunikat o wygranej lub przegranej i wywołuje funkcję checkBestScores w celu sprawdzenia czy uzyskany wynik może zostać do listy najlepszych.
void **moveObjectOnBoard( ) -** metoda przechodzi po całej liście obiektów na planszy i jeśli trzeba to przesuwa je.
bool **moveStone(**int direction**) –** metoda odpowiadająca za przesuwanie przez gracza kamieni na boki. Zwracany rezultat informuje czy przesunięcie w podanym jako argument kierunku jest możliwe.
void **runEngine(**sf::RenderWindow & window**) –** metoda zawierająca główną pętle rozgrywki. Metoda mieści całą logike gry.

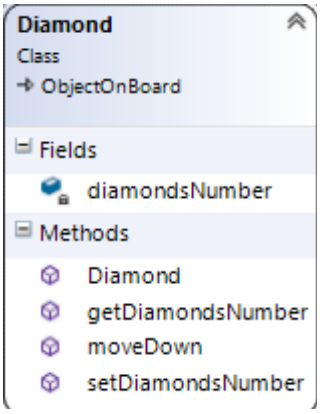
Klasa ObjectOnBoard – klasa abstrakcyjna. Jest klasą bazową dla klas Diamond i Stone.



Najważniejsze pola:
array<int, 2> **coordinates** – tablica przechowująca współrzędne x i y obiektu.
bool **isMoving -** przechowuje informacje czy obiekt jest w ruchu czy w spoczynku.
const double **SPEED -** przechowuje informacje o prędkości obiektu

Najważniejsze metody:
virtual bool **moveDown(**Level* level, sf::Sprite [][40], sf::Texture* texture, Player& player**) –** metoda czysto wirtualna odpowiadająca za przemieszczanie w dół obiektów.

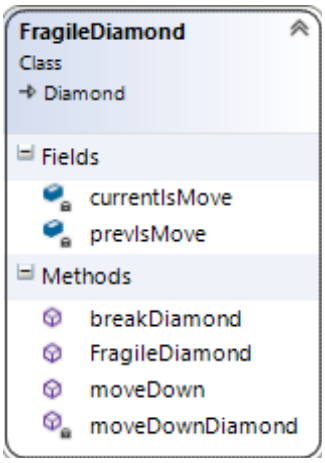
Klasa Diamond – klasa dziedzicząca po klasie ObjectOnBoard



Najważniejsze pola:
static int **diamondsNumber** – liczba diamentów jaka znajduje się na planszy.

Najważniejsze metody:
virtual bool **moveDown(**Level* level, sf::Sprite [][40], sf::Texture* texture, Player& player**) –** funkcja odpowiedzialna za spadanie diamentów.

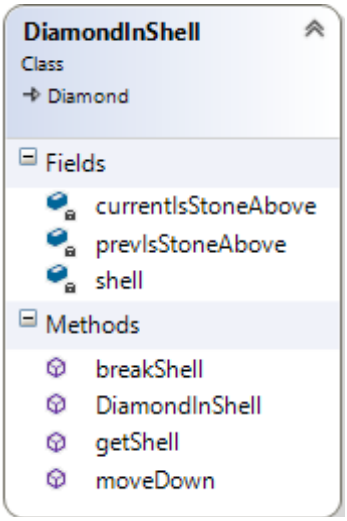
Klasa FragileDiamond - klasa dziedzicząca po klasie Diamond



Najważniejsze pola:
bool **currentIsMove** – informacja o tym czy obiekt aktualnie się porusza.
bool **prevIsMove** – informacja o tym czy poprzednio obiekt się poruszał.

Najważniejsze metody:
virtual bool **moveDown(**Level* level, sf::Sprite [][40], sf::Texture* texture, Player& player**)** – funkcja odpowiedzialna za spadanie diamentów. Metoda wywołuje pomocniczą metodę moveDownDiamond i na podstawie zwróconego rezultatu ustawia pola currentIsMove i prevIsMove na odpowiednie wartości.
bool **moveDownDiamond(**Level* level, sf::Sprite[][40], sf::Texture* texture, Player& player**)** – metoda odpowiedzialna za przemieszczanie w dół diamentów.
bool **breakDiamond(**std::list<ObjectOnBoard*> objectOnBoard, Player& player, sf::Sprite sprite[][40]**)** – metoda wykrywająca rozbicie diamentu na podstawie pól currentIsMove i prevIsMove.

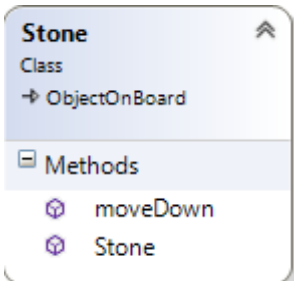
Klasa DiamondInShell - klasa dziedzicząca po klasie Diamond



Najważniejsze pola:
bool **currentIsStoneAbove** – informacja o tym czy aktualnie nad obiektem znajduje się kamień.
bool **prevIsStoneAbove** – informacja o tym czy poprzednio nad obiektem znajdował się kamień.
bool **shell** – informacja czy diament znajduje się jeszcze w muszli czy nie.

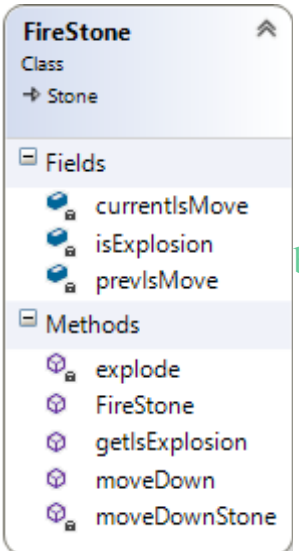
Najważniejsze metody:
virtual bool **moveDown(**Level* level, sf::Sprite [][40], sf::Texture* texture, Player& player**)** – funkcja odpowiedzialna za spadanie diamentów.
bool **breakShell(**std::list<ObjectOnBoard*> objectOnBoard, Level* level, sf::Sprite[][40], sf::Texture* texture**)** – metoda sprawdza na podstawie pól currentIsStoneAbove i prevIsStoneAbove czy muszla została rozbita i jeśli tak to podmienia teksturę obiektu.

Klasa Stone - klasa dziedzicząca po klasie ObjectOnBoard



Najważniejsze pola:
virtual bool **moveDown(**Level* level, sf::Sprite [][40], sf::Texture* texture, Player& player**)** – funkcja odpowiedzialna za spadanie kamieni.

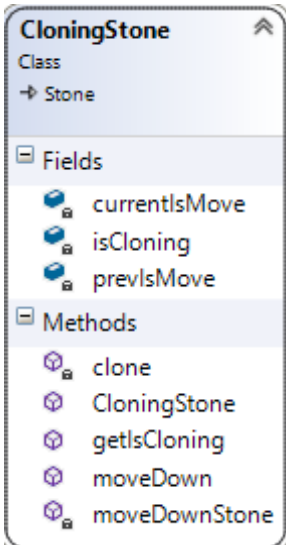
Klasa FireStone – klasa dziedzicząca po klasie Stone



Najważniejsze pola:
bool **currentIsMove** – informacja o tym czy obiekt aktualnie się porusza.
bool **prevIsMove** – informacja o tym czy poprzednio obiekt się poruszał.
bool **isExplode** – pole informujące czy dany kamień wybuchł.

Najważniejsze metody:
virtual bool **moveDown(**Level* level, sf::Sprite [][40], sf::Texture* texture, Player& player**)** – funkcja odpowiedzialna za spadanie kamieni. Metoda wywołuje pomocniczą metodę moveDownStone i na podstawie zwróconego rezultatu ustawia pola currentIsMove i prevIsMove na odpowiednie wartości. Metoda jest także odpowiedzialna za wywołanie metody explode.
bool **moveDownStone(**Level* level, sf::Sprite[][40], sf::Texture* texture, Player& player**)** – metoda odpowiedzialna za przemieszczanie w dół kamieni.
bool **explode(**std::list<ObjectOnBoard*> objectOnBoard, Player& player, sf::Sprite sprite[][40]**)** – metoda niszcząca sąsiednie pola jeśli są typu GRASS to ich typ zmieniany jest na NONE puste pole oraz jeśli gracz znajduje się w zasięgu wybuchu to metoda zmieni jego status na martwy.

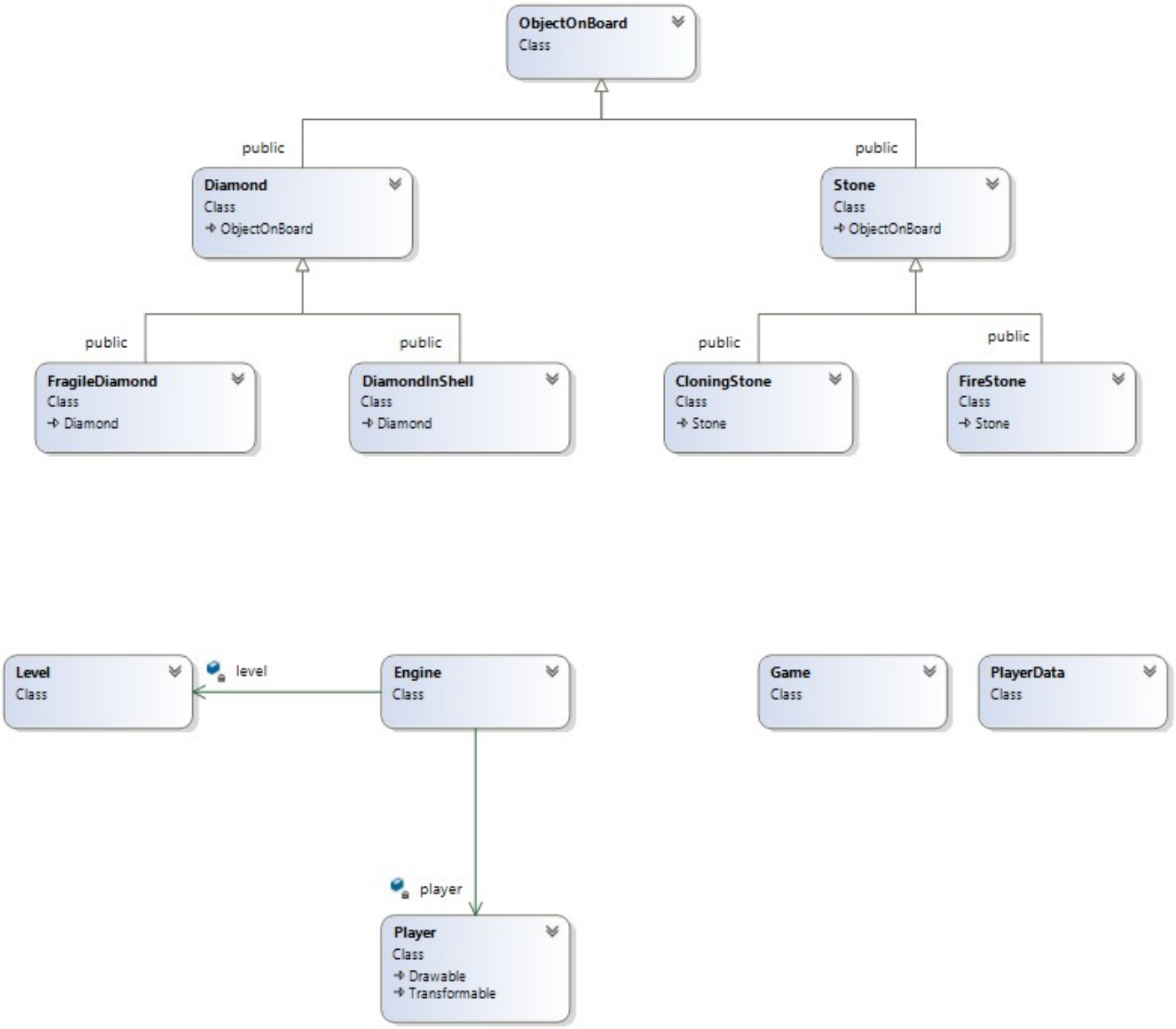
Klasa CloningStone – klasa dziedzicząca po klasie Stone



Najważniejsze pola:
bool **currentIsMove** – informacja o tym czy obiekt aktualnie się porusza.
bool **prevIsMove** – informacja o tym czy poprzednio obiekt się poruszał.
bool **isCloning** – pole informujące czy dany obiekt ma stworzyć nowe kaienie.

Najważniejsze metody:
virtual bool **moveDown(**Level* level, sf::Sprite [][40], sf::Texture* texture, Player& player**) –** funkcja odpowiedzialna za spadanie kamieni. Metoda wywołuje pomocniczą metodę moveDownStone i na podstawie zwróconego rezultatu ustawia pola currentIsMove i prevIsMove na odpowiednie wartości. Metoda jest także odpowiedzialna za wywołanie metody clone.
bool **moveDownStone(**Level* level, sf::Sprite[][40], sf::Texture* texture, Player& player**) –** metoda odpowiedzialna za przemieszczanie w dół kamieni.
bool **clone(**std::list<ObjectOnBoard*> objectOnBoard, Player& player, sf::Sprite sprite[][40]**) –** metoda ustawiające sąsiednie pola jeśli są typu GRASS na typ STONE. Później metoda z klasy Engine cloneStone stworzy na tych polach nowe kamienie.

Diagram hierarchii klas:



Istotne struktury danych i algorytmy.

Do reprezentacji poziomu użyłem tablicy dwuwymiarowej. Zdecydowałem się na taką strukturę danych ponieważ założyłem, że rozmiar poziomu jest zawsze stały i nie zmienia się podczas rozgrywki.

Do przechowywania obiektów na planszy wykorzystałem listę z biblioteki STL ponieważ podczas rozgrywki wielokrotnie trzeba usuwać i dodawać kamienie oraz diamenty.

W swoim projekcie nie miałem potrzeby wykorzystania złożonych algorytmów jednakże istotną rzeczą była implementacja algorytmu wykrywania kolizji gracza ze ścianami. Algorytm polega na pobraniu pozycji gracza na planszy, obliczeniu nowej pozycji po przesunięciu w zadanym kierunku, następnie wyznaczeniu środka tekstury i obliczeniu współrzędnych x,y dzieląc współrzędne środka tekstury przez rozmiar kafelka. Mając współrzędne x,y możemy odzukać w tablicy przechowującej poziom daną pozycję i sprawdzić czy ruch jest dozwolony. Jeśli ruch jest niepoprawny status gracza zostanie zmieniony na STOP.

Wykorzystane techniki obiektowe:
- Polimorfizm
Klasa abstrakcyjna ObjectOnBoard posiada metodę czysto wirtualną moveDown. Wszystkie klasy dziedziczące po ObjectOnBoard implementują metodę moveDown odpowiednio do tego jak powinny się poruszać.
Klasa Engine zawiera listę wskaźników do obiektów ObjectOnBoard i wypełniona jest różnymi obiektami klas pochodnych. W klasie Engine metoda moveObjectOnBoard przechodzi po całej liście wywołując dla każdego obiektu metodę wirtualną moveDown i każdy obiekt porusza się zgodnie ze swoją implementacją tej metody.

- Kontenery STL
```
list<ObjectOnBoard*> objectOnBoard; // lista przechowujaca wszystkie diamenty I kamienie na planszy
vector<PlayerData*> players;//vector sluzacy do przechowywania danych graczy wczytanych z pliku
array<int, 2> coordinates;  // tablica do przechowywania współrzędnych x,y
```

- RTTI
Jako że wszystkie obiekty trzymam w liście wskaźników do klasy bazowej często używam RTTI do rozpoznawania jakiej dokładnie klasy jest dany obiekt np. W klasie Engine w metodzie collectDiamonds przechodzę przez listę obiektów na planszy poszukując obiektu o współrzędnych indentycznych jak gracza, a następnie sprawdzam za pomocą typeid czy dany obiekt jest jednym z rodzajów diamentów.

- Algorytmy i iteratory
Iteratory były mi niezbędne przy praktycznie każdym przechodzeniu przez listę obiektów znajdujących się na planszy.

```
for (auto iter = objectOnBoard.begin(); iter != objectOnBoard.end(); ++iter)
{
	// ...
}
```

W klasie Engine w metodzie checkBestScores użyłem algorytmu min_element w celu znalezienia minimalnego najlepszego wyniku i porównania go z rezultatem gracza.

```
min_element(players.begin(), players.end(), [](PlayerData* p1, PlayerData* p2) { return p1-
>getScore() < p2->getScore(); });
```

W tej funkcji użyłem także algorytmu sort w celu posortowania wyników po dodaniu nowego rekordu na koniec vectora.

```
sort(players.begin(), players.end(), [](PlayerData* p1, PlayerData* p2) { return p1->getScore()
> p2->getScore(); }); // posortowanie wynikow w kolejnosci malejacej
```

W klasie Engine w metodzie saveScoresToFile wykorzystuje algorytm for_each w celu przejścia przez cały vector zawierający listę graczy.

Ogólny schemat działania programu.
1. Po uruchomieniu program wyświetla główne menu i czeka na wygenerowanie zdarzeń przez użytkownika.
2. W zależności od tego co wybierze użytkownik program uruchomi odpowiedni moduł START, BEST_SCORES lub EXIT
3. Jeśli użytkownik wybrał opcję START zostanie stworzony obiekt klasy Engine i gra się rozpocznie.
3.1. Program załaduje poziom.
3.2. Program będzie pobierał od użytkownika kierunek ruchu postaci.
3.3. Sprawdzenie kolizji i przemieszczenie gracza
3.4. Ruch obiektów na planszy.
3.5. Wywołanie dodatkowych metod związanych z poszczególnymi obiektami (np. sprawdzenie czy diament został zebrany lub czy jakiś kamień wybuchł)
3.6. Sprawdzenie czy wystąpił koniec gry.
3.7. Jeśli wystąpił koniec gry to jest sprawdzany wynik gracza czy może dopisać się na listę najlepszych wyników i jeśli tak to pobierany jest nick gracza i następuje zapis do pliku.
3.8. Powrót do menu.

4. Jeśli użytkownik wybrał opcję BEST_SCORES.
4.1. Z pliku zostanie pobrana lista najlepszych graczy.
4.2. Program wyświetli pobrane dane.
4.3. Oczekiwanie aż użytkownik wciśnie klawisz ESC.
4.4 Powrót do menu.
5. Jeśli użytkownik wybrał opcję EXIT to program zakończy działanie.

Testowanie i uruchamianie
Wycieki pamięci były sprawdzane za pomocą programu Visual Leak Detector
Podczas testów wykryto różne wycieki pamięci:

```
Visual Leak Detector detected 1 memory leak (100 bytes).
Largest number used: 10824 bytes.
Total allocations: 44842168 bytes.
Visual Leak Detector is now exiting.
The program '[6636] Boulder Dash.exe' has exited with code 0 (0x0).
```

Źródłem wycieków pamięci okazało się być zapomnienie o zwolnieniu pamięci przed usunięciem wskaźnika na obiekt z listy w kilku miejscach programu.

```
Visual Leak Detector detected 10 memory leaks (840 bytes).
Largest number used: 9024 bytes.
Total allocations: 4572760 bytes.
Visual Leak Detector is now exiting.
The program '[15016] Boulder Dash.exe' has exited with code 0 (0x0).
```

Źródłem wycieków pamięci okazało się być zapomnienie o zwolnieniu pamięci przed usunięciem wskaźnika na obiekt z listy w kilku miejscach programu.
Drugą przyczyną wycieków było zapomnienie o zwolnieniu pamięci w przypadku przegrania gracza w metodzie checkBestScores w klasie Engine.

```
Visual Leak Detector detected 2 memory leaks (144 bytes).
Largest number used: 328 bytes.
Total allocations: 1008 bytes.
Visual Leak Detector is now exiting.
The program '[13952] Boulder Dash.exe' has exited with code 1 (0x1).
```

Wyciek pamięci pojawiał się podczas gdy program nie mógł załadować tekstur.
Pomogło jawne wywołanie destruktora przed zakończeniem pracy programu.

```
this->~Engine();
exit(1);
```

Wycieki zostały zlikwidowane i podczas ponownych testów żadne nowe nie wystąpiły.

```
No memory leaks detected.
Visual Leak Detector is now exiting.
The program '[6960] Boulder Dash.exe' has exited with code 0 (0x0).
```

Jeśli program nie będzie mógł załadować jakiejś tekstury wyświetli komunikat o błędzie
I zakończy działanie.