Dokumentacja techniczna

Digger

Napora Łukasz Y2

1 kwietnia 2014

Spis treści

1. Wprowadzenie

Niniejszy dokument jest dokumentacją techniczną gry zręcznościowej Digger, opartej na specyfikacji użytkownika autorstwa Pawła Mitrusia . W dokumentacji wyszczególniono: strukturę i działanie klas, algorytmy sztucznej inteligencji oraz instrukcję użytkownika.

2. Ogólny diagram składowych gry

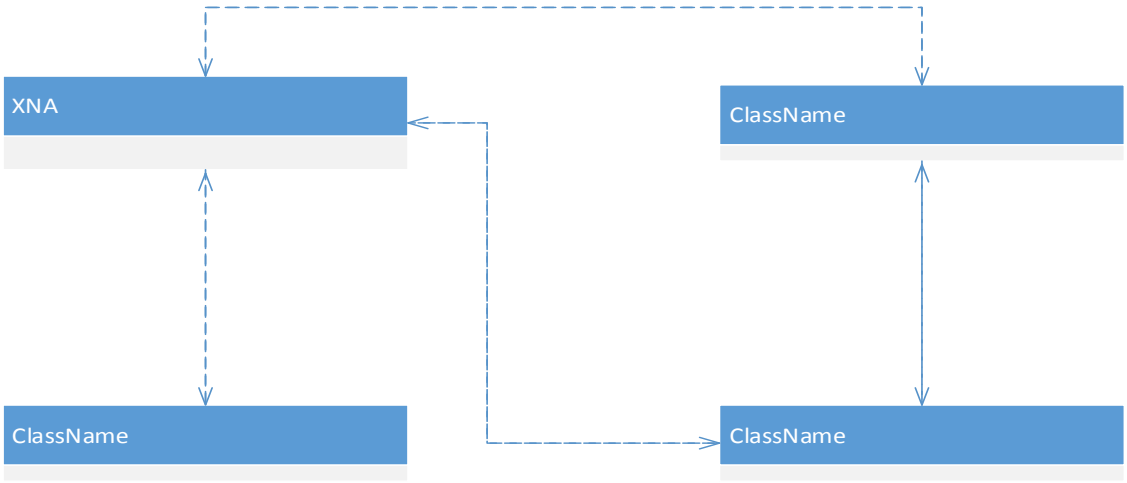


Diagram prezentuje cztery główne składowe gry – klasy biblioteki XNA, klasy interfejsu, klasy logiki gry oraz klasy związane z obiektami występującymi w grze.

3. XNA

Implementacja gry odbędzie się przy użyciu biblioteki XNA. Biblioteka XNA udostępnia klasę Game, w której znajdują się następujące metody:

- Initialize() – pozwala grze wykonać inicjalizację wszystkich zasobów potrzebnych do uruchomienia. Tutaj można kierować zapytania do potrzebnych usług oraz załadować nie-graficzną zawartość gry.

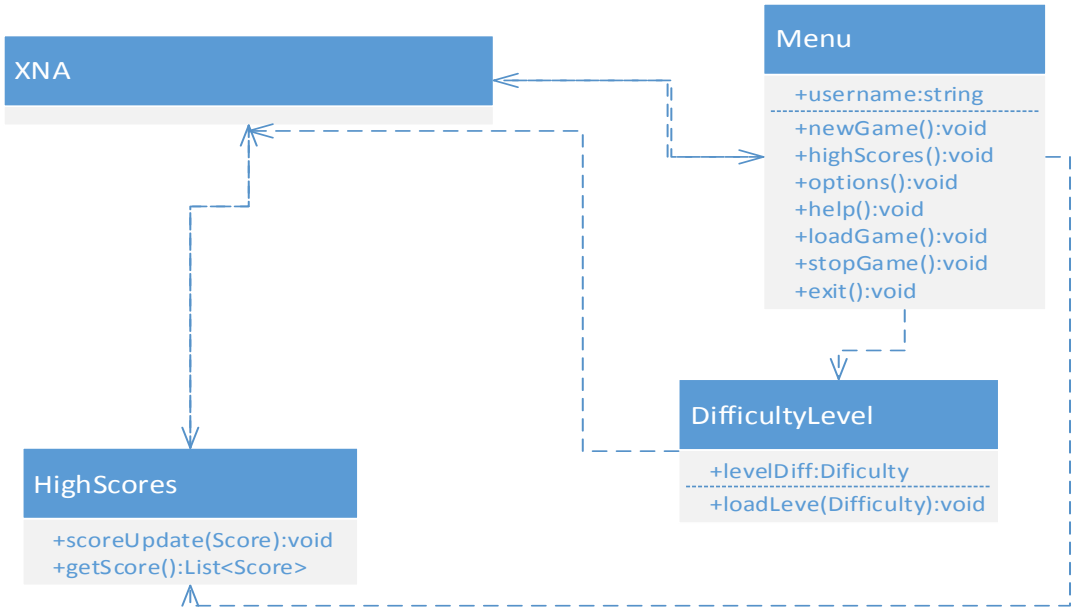
- LoadContent() – wywoływany jednokrotnie, jest miejscem na załadowanie całej zawartości gry. Ładuje zasoby graficzne przy pomocy klas GraphicsDevice, Sprite i SpriteBatch.

- UnloadContent – posłuży do zwalniania zasobów.

- Update(GameTime) – w XNA główna pętla gry jest ukryta przed programistą . Metoda Update aktualizuje świat, umożliwia sprawdzanie kolizji, odtwarzanie dźwięków. Będzie interpretować input użytkownika. W tej metodzie wywoływane będą metody Update poszczególnych obiektów. Parametrem wywołania funkcji jest licznik czasu gry, zawierający 2 timery: czas upłynąwszy od rozpoczęcia rozgrywki oraz od ostatniego wywołania Update.

- Draw(GameTime) – renderuje obiekty graficzne na ekran.

4. Interface



4.1 Menu

4.1.1. newGame()

Przechodzi do ekranu logowania, gracz jest proszony o podanie imienia następnie przechodzimy do wyboru poziomu trudności.

```
void newGame()
{
    wyświetl panel logowania
     wyświetl wybór poziomu trudności
     levelDiff = wybrany poziom;
     heroChosen = heroChoice();
     loadLevel(1,levelDiff,heroChosen);
}
```

4.1.2. highScores()

Wyświetla ekran z listą najlepszych wyników (high-scores) zawierającą nazwę użytkownika, datę uzyskania wyniku i ilość zdobytych punktów. Wyniki są wczytywane z pliku .xml i sortowane malejąco.

```
void highScores()
{
        List<Score> hs = HighScores.getScores();
        sformatuj wyniki hs;
        wyświetl wyniki hs;
}
```

4.1.3. options()

Wyświetla ekran opcji z możliwościami: wł/wył. muzykę, wł/wył. efekty dźwiękowe oraz ustawienie własnego sterowania. Wybrane opcje są zapamiętywane globalnie i używane przez klasy gry.

4.1.4. help()

Wyświetla ekran pomocy z opisaną klawiszologią i celami gry.

4.1.5. loadGame()

Wczytuje zapisaną w pliku xml grę.

4.1.6. stopGame()

Wywoływane, gdy podczas gry użytkownik wciśnie przycisk Escape. Wyświetla okienko z pytaniem ‘Czy chcesz wyjść do menu głównego?’ i opcjami ‘Tak’/’Nie’. Ponadto od momentu wywołania funkcji do odpowiedzi użytkownika wszystkie funkcje gry (w tym timery) są zatrzymywane, gra zostaje zapisana za pomocą struktury Save do pliku xml.

4.1.7. exit ()

Zamyka aplikację.

4.2. DifficultyLevel

Okno wyboru poziomu trudności.

4.2.1. loadLevel (Difficulty)

```
void loadLevel(int level, Difficulty levelDiff, HeroClass heroChosen)
{
        ustaw parametry gry zależne od poziomu trudności
```

```
        przekaż do klas XNA dane dot. parametrów
        game.Run()
}
```

4.3. HighScores

Klasa obsługująca zapis i wczytywanie najlepszych wyników zapisywanych w plikach .xml, za pomocą narzędzi deserializacji i serializacji.

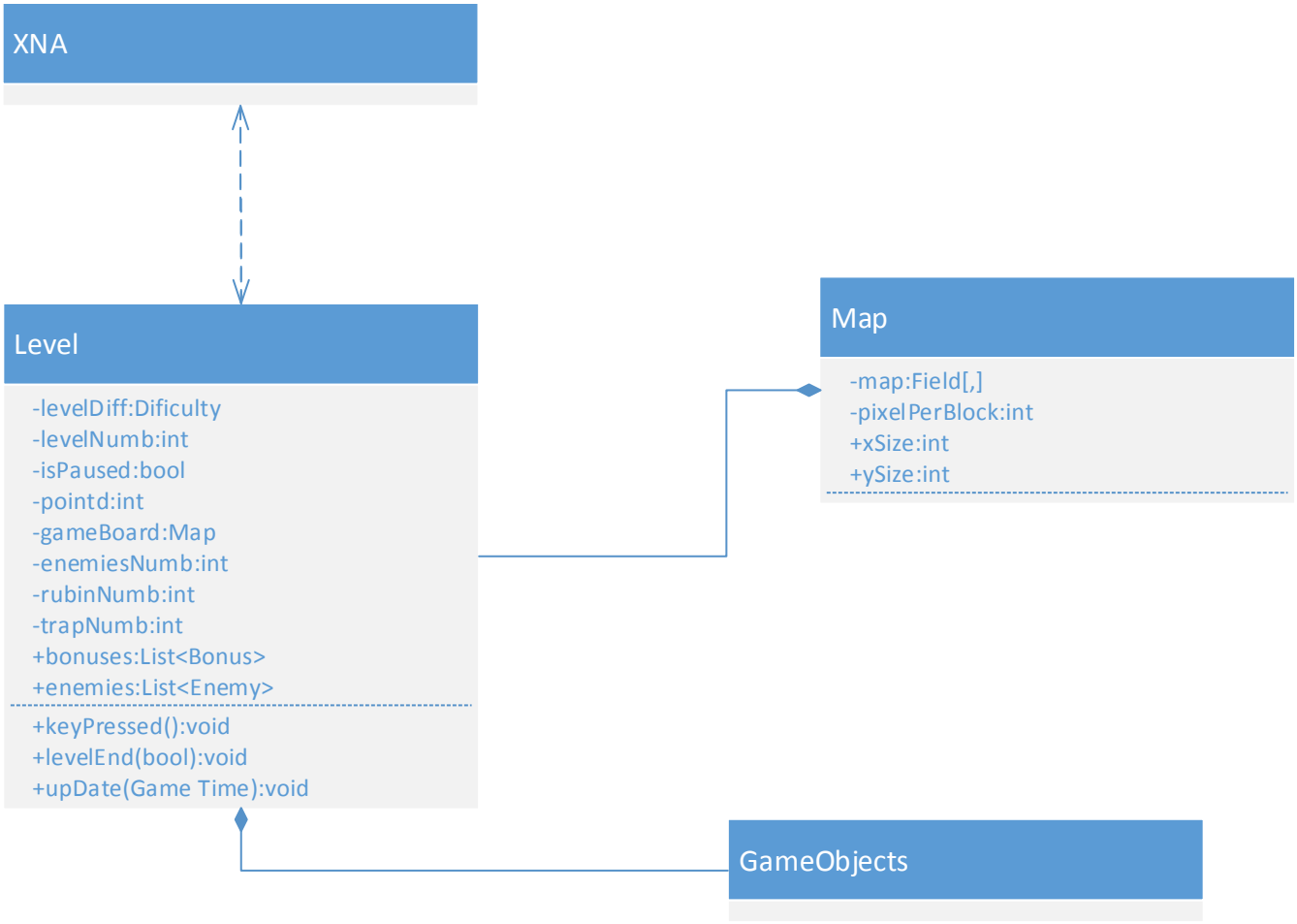
4.3.1. scoreUpdate(Score)

```
void scoreUpdate(Score)
{
        List<Score> hs = getScores();
        if(wynik lepszy od 10-go na liście)
        {
                dodaj Score do listy
                if(więcej niż 10 elementów na liście)
                usuń ostatni element
        }
}
```

4.3.2. getScores ()

Pobiera wyniki z pliku i zapisuje w postaci listy obiektów struktury Score.

5. Game Logic



5.1. Level

Klasa obsługująca logikę poziomu. Zawiera niezbędne informacje o bieżącej rozgrywce – punkty, liczbę przeciwników i obiektów, ich położenie itd.

5.1.1. keyPressed ()

Reaguje na event zmiany stanu klawiatury.

```
void KeyPressed()
{

        case Strzalka lub W/S/A/D:
         direction = odpowiedni kierunek;
         Hero.moveHero(direction);
        Case spacja:
        Hero.Shot();
        Case left Alt:
        Hero.ChangeGun();
}
```

5.1.2. levelEnd (bool)

Funkcja wykonywana po zakończeniu poziomu.

```
void levelEnd(bool win)
{
        if(win)
        {
                utwórz zmienną typu Score z odpowiednimi danymi;
                wykonaj funkcję updateScore;
        }
        else
        {
                jeśli lifes>0
                        life = life -1
                        kontynuuj grę
                w przeciwnym przypadku
                        spytaj gracza czy chce powtórzyć poziom
                        jeśli tak
                                points = pointsAtLevelStart
                                lifes = lifesAtLevelStart
                                rozpocznij poziom od nowa
                        jeśli nie
                                wywołaj UpdateScore
        }
}
```

5.1.4. Update (GameTime)

Wywołuje odpowiednie funkcje Update dla wszystkich obiektów znajdujących się na planszy (na podstawie list poszczególnych typów obiektów). Sama jest wywoływana przez funkcję Update klasy Game z głównej pętli gry.

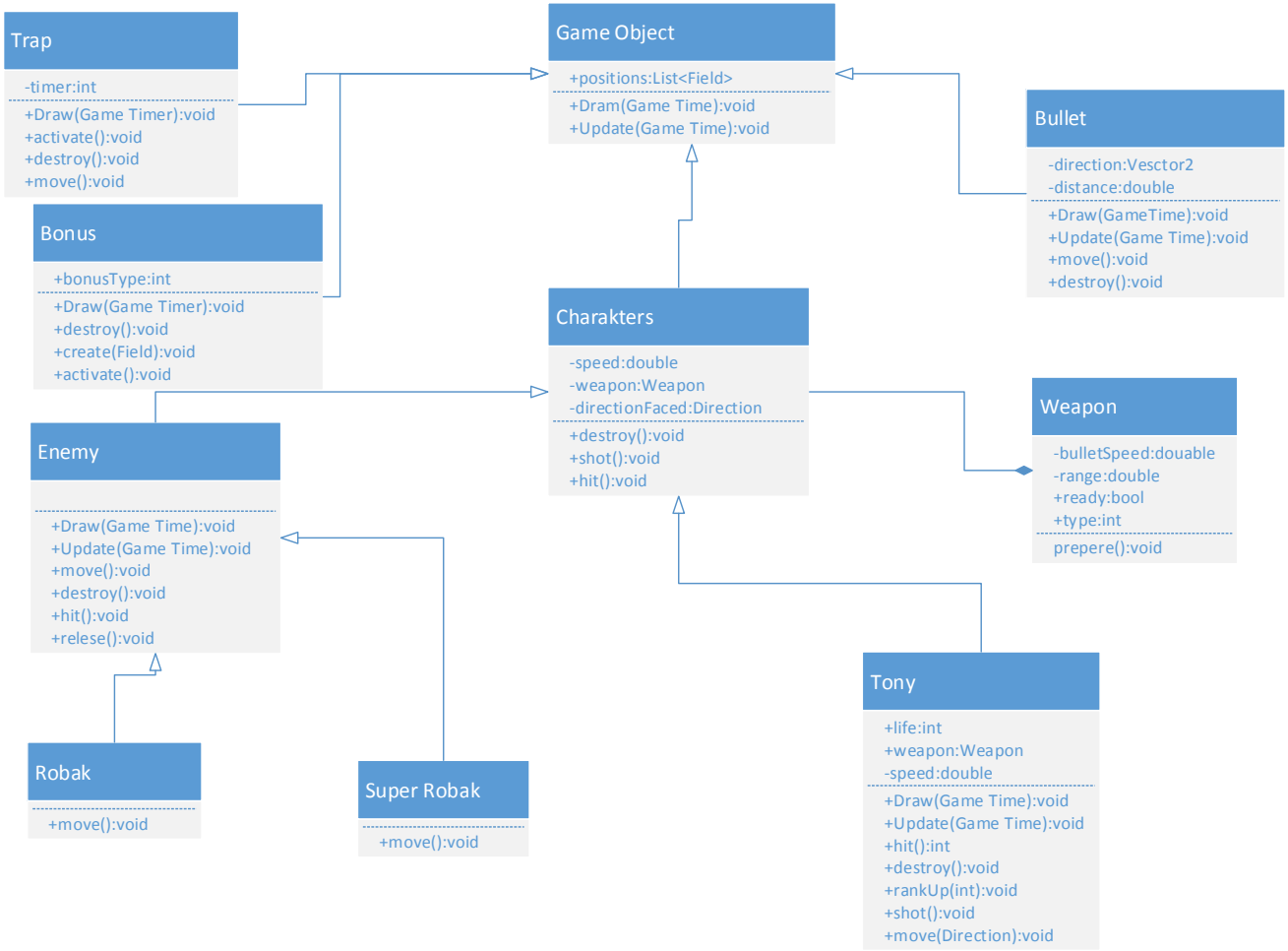
5.2. Map

Klasa obsługująca mapę. Mapa jest dwuwymiarową tablicę obiektów typu Field. Struktura Field składa się ze współrzędnych (w formacie Vector2) oraz boolowskiej informacji IsEmpty informującej czy na danym polu jest jakaś przeszkoda . Parametr pixelsPerBlock określa ile pikseli składa się na każde pole Field, co pomoże przy rysowaniu mapy na ekran.

6. Game Objects

Klasa związana ze wszystkimi obiektami pojawiającymi się na planszy. Wyróżniamy cztery typy obiektów: postacie (Characters), bonusy (Bonuses), obiekty statyczne (Barriers) i pocisk (Bullet). Klasę

postaci dzielimy na bohatera (Hero) i przeciwników (Enemy). Wprowadzam także dwie klasy pomocnicze, odpowiedzialne za bronie (Weapon).



6.1. Character

Klasa obsługująca wszystkie postaci pojawiające się na planszy. Każda z postaci jest opisana przez parametry:
speed – prędkość ruchu
directionFaced – kierunek w którym ‘patrzy’ postać

6.2. Tony

Klasa obsługująca postać bohatera, dziedzicząca z klasy Character. Bohatera opisują dodatkowe parametry:
life – pozostała ilość żyć
weapon –aktualna broń bohatera

6.2.1. rankUp (int)

Funkcja zmiany broni bohatera. Jako parametr wejściowy przyjmuje hero.weapon.type, czyli rodzaj aktualnej broni.

```
void rankUp(int oldType)
{
        if(oldType == 1)
        Tony.weaponType=2;
        else if(oldType == 2)
        Tony.weaponType=1;
}
```

6.2.2. move (Direction)

Wywoływana przez event keyPressed, jeśli wciśnięto przycisk strzałek lub W/S/A/D.

```
void move(Direction dir)
{
 Tony.direction = dir;
 if(pole w danym kierunku.IsEmpty = true)
 {
        Tony.position = to pole;
        if(pole zawiera bonus)
                Bonus.destroy();
        if(pole zawiera przeciwnika)
                Enemy.destroy();
                Tony.destroy();
        If(Tony kopie)
                Weapon.prepere();

 }
}
```

6.2.3. shot ()

Wywoływana przez event keyPressed, jeśli wciśnięto spacje.

```
void shot()
{
        stwórz obiekt typu Bullet i dodaj go na listę;
        Bullet.direction = Tony.direction;
        uruchom BulletTimer;
        weapon.ready=fales
}
```

6.2.4. destroy ()

Wywoływana w momencie śmierci bohatera.

```
void destroy()
{
  Level.levelend(false);
}
```

6.2.5. hit (int)

Wywoływana w momencie trafienia bohatera na przeciwnika.

```
void hit()
{
        Tony.destroy();
}
```

6.3 Weapon

Kalasa opisuje bron której używa bohater.

6.3.1. prepere()

Wywoływana w momencie zaczęcia kopanie przez bohatera, ładuje obecna broń

```
prepere()
{
        Zacznij odliczać czas dla typubroni
        If(czas upłyną)
                ready=true;
}
```

6.4. Enemy

Klasa opisująca wrogie jednostki, dziedzicząca po klasie Character. Klasa zawiera wirtualną metodę move(), przeciążoną dla poszczególnych typów jednostek (między typem ruchów jednostek będą pewne różnice). Funkcja move jest wywoływana bez przerwy z jej wnętrza.

6.4.1. destroy ()

Wywoływana przy śmierci przeciwnika.

```
void destroy()
{
        wygeneruj liczbę od 1 do 100
        jeśli liczba jest większa od n (jeszcze nieokreślona wartość)
        stwórz w miejscu przeciwnika losowy bonus
        usuń przeciwnika z listy Enemies
```

```
        zmniejsz enemiesNumb
}
```

6.4.2. hit (int)

Wywoływana w momencie trafienia przeciwnika przez pocisk bohatera.

```
void hit()
{
destroy();
}
```

6.4.3. relese()

Funkcja generuje nowych przeciwników w odpowiednich odstępach czasowych (8 sekund), sprawdza czy może wygenerować przeciwnika(czy jest osiągnięta maksymalna liczba przeciwników na dany level.

6.5. Klasy typów przeciwnika

Klasy zawierają tylko funkcję move(), określającą sposób ruchu przeciwników. Przykladowa funkcja dla robaka:

```
void move()
{
        Wyznacz najkrótsza ścieżkę z obecnego pola do pola gracza
        Wyznacz wektor kierunkowy do najbliższego punktu tej ścieżki
        Enemy.position+=enemy.speed*wektor_kierunku
}
```

6.6 Trap

Klasa obsługująca pułapki znajdujące się na planszy. Zawiera pole timer, które określa czas po który pułapka zostanie aktywowana.

6.6.1. activate()

Funkcja wywołana gdy bohater przejdzie pod pułapką. Odlicza czas i uruchamia pułapkę.

```
activate()
{
        Odlicz 2 sekundy
        Pułapka spada do końca pionowego tunelu
        This.destroy()
}
```

6.6.2. move()

Funkcja wywoływana gdy gracz przesuwa pułapkę.

```
move()
{
        If(pole obok pułapki w kierunku Tony.directionFaced && Tony.directionFaced jest w poziomi)
```

```
                Przesuń pułapkę
                If( na polu jest przeciwnik)
                        Enemy.destroy()
                        This.destroy()
        If(jest tunel w dół)
                This.activate
}
```

6.6.3. destroy()

Niszczy pułapkę, usuwa z planszy.

6.7. Bonus

Klasa obsługująca wszystkie nieruchome obiekty na mapie, na które da się wejść – tj. wszystkie bonusy, rubiny oraz diamenty. Zmienna bonus Type, podobnie jak w przypadku barier, oznacza typ bonusu (typu int lub enum, do ustalenia w trakcie implementacji).

6.7.1. destroy ()

Funkcja wywoływana przy wejściu bohatera na pole z bonusem.

```
void destroy()
{
        wykryj typ bonusu na podstawie danych z listy bonusów
        uwzględnij działanie bonusa
        usuń bonus
}
```

6.7.2. create (Field)

Funkcja generuje losowy bonus w podanym przez parametr miejscu.

```
void create(Field position)
{
        bonus.Position = position
        bonus.bonusType = losowy bonus
}
```

6.7.3. activate()

Funkcja wywoływana podczas gdy bohater wejdzie na bonus.

```
Activate()
{
        If(bonus to rubin)
                Level.points+=30
                Level.rubinNumb--;
                If(Level.rubinNumb == 0)
                        Stwórz na mapie diament
        else if(bonus to diament)
```

```
            Level.points+=200
            Levleend(true);
        else
            sprawdz typ bonusa
            wykonaj odpowiednie dzialanie

}
```

6.8. Bullet

Klasa obsługująca wystrzelone pociski. Jest opisany przez zmienne: wektor direction określający w jakim kierunku porusza się pocisk, oraz wartość distance określającą dystans, który pokonał pocisk.

6.8.1. move ()

Metoda ruchu pocisku, wywoływana przez jego własny timer. Funkcja zależy do urzywanej obecnie broni. Przykładowa funkcja dla pistoletu:

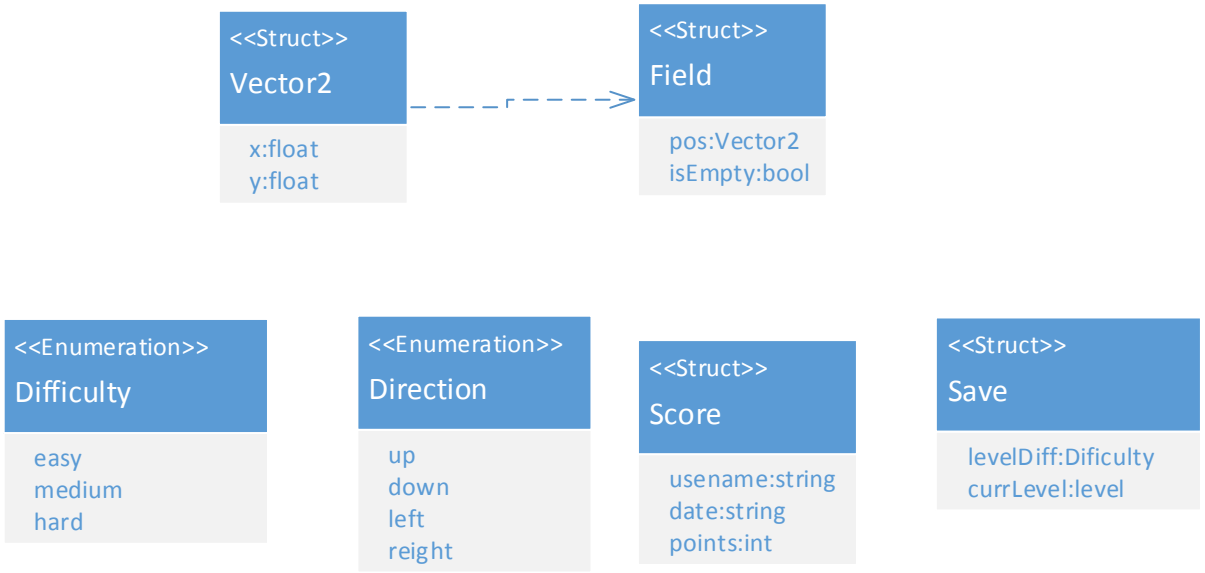
```
void move()
{
 this.position += this.weapon.bulletSpeed * bullet.direction;
 Field P = bullet.position;

jeśli na polu P znajduje się przeciwnik
        enemy.destroy();
        this.destroy();
 else jeśli na polu P znajduje się przeszkoda
        this.destroy();
}
```

6.8.2. destroy ()

Wywoływana, gdy pocisk w coś trafi.

```
void destroy()
{
 usuń pocisk;
}
```

7. Elementy pomocnicze



8. Algorytm A*

Algorytmy sztucznej inteligencji oprą się na algorytmie A*, dostosowane do gry w następujący sposób:

- wierzchołkami grafu są pola (Field) na mapie, dla których wartość isEmpty = true, czyli takie na które da się ‘wejść’
- krawędź istnieje, jeśli pola mają wspólną krawędź
- wierzchołek początkowy to pole na którym znajduje się jednostka
- wierzchołek końcowy to pole, na którym aktualnie znajduje się gracz
- w przypadku nieodnalezienia drogi (na zaprojektowanych przeze mnie mapach nie powinno się zdarzyć) algorytm zwróci false.

9. Instrukcja użytkownika

9.1 Krótki opis gry

Digger jest dwuwymiarową grą zręcznościową. Gracz wciela się w rolę Tonego i jego zadaniem jest splądrowanie całego podziemia „Krzemowej Doliny”, mimo tego, ze czai się w nim wiele niebezpieczeństw, czyhających na jego życie.

Wszelkie odległości w grze są mierzone metryką taksówkową, chyba że zaznaczono inaczej.

9.2 Poziomy trudność

W Digger mamy do wyboru 3 poziomy trudności. W zależności od poziomu trudności zmienia się liczba przeciwników, rubinów i pułapek.

Łatwy

Etap	Robak	Super Robak	Rubin	Pułapka
1	2	0	10	3
2	3	0	15	4
3	3	1	15	5
4	4	1	15	5
5	5	2	20	5

Średni

Etap	Robak	Super Robak	Rubin	Pułapka
1	3	0	13	2
2	4	0	17	3
3	4	1	17	4
4	5	2	17	4
5	6	2	23	4

Trudny

Etap	Robak	Super Robak	Rubin	Pułapka
1	4	0	15	2
2	5	0	20	2
3	5	1	20	3
4	6	2	20	3
5	7	3	25	4

9.3 Sterowanie

Klawisz	Działanie
Strzałka w górę	Poruszanie się postacią w górę
Strzałka w dół	Poruszanie się postacią w dół
Strzałka w lewo	Poruszanie się postacią w lewo
Strzałka w prawo	Poruszanie się postacią w prawo
Spacja	Strzał
Lewy alt	Zmiana broni
P	Pauza

Powyższe sterowanie jest domyślne, gracz może je dostosować do swoich potrzeb.

9.4 Menu główne

New Game – rozpoczyna nową grę

Load Game – pozwala na wczytanie ostatniej gry

Help – wyświetla informacje dotyczące klawiszologi oraz wskazówki do gry, umożliwia zmianę ustawień (sterowanie, dźwięk)

Hight Scores – wyświetla listę z najlepszymi wynikami

Exit – powoduje zamknięcie programu