19.04.2012

Miniffice

Cannon Fodder – Specyfikacja Techniczna

Bartosz Kalinowski X1

MiNI, 218584

Dokumentacja techniczna gry Miniffice - Cannon Fodder, na podstawie specyfikacji użytkownika autorstwa Mateusza Skalniaka. Opisane zostaną: struktura i działanie klas, algorytmy w grze.

Zawartość

1. Ogólny zarys programu

Gra zostanie zaimplantowana za pomocą biblioteki XNA 4.0 do tworzenia gier. Schemat UML znajduje się w dołączonym pliku ze względu na jego rozmiary.

Aplikacja do wyświetlania wszystkiego będzie korzystała z klasy ScreenManager który obsługuje wyświetlanie warstw ekranu (GameScreen). Odpowiednia warstwa (GamePlayScreen) będzie przechowywać menadżera gry (GameManager), w którym będzie obsłużona cała logika gry. Na początku zgodnie z dokumentacją użytkownika gra nie posiada żadnych dodatkowych ekranów (w trakcie gry), ale gdyby zaszła taka potrzeba na dodanie nowych (na przykład ekranu z wyposażeniem postaci, czy jakimiś bardziej zaawansowanymi statystykami itp.) to można by to bardzo prosto rozwiązać za pomocą ScreenManager'a.

Ładowanie ustawień gry, oraz kolejnych poziomów odbywać się będzie poprzez serializację XML. Wszystkie klasy odpowiedzialne za samą grę będą serializowalne, więc będzie się je dało łatwo przenieść do plików, oraz z powrotem. Dzięki temu, można w dość łatwy sposób zrealizować zapisywanie i wczytywanie gry (jednak nie było tego w specyfikacji użytkownika, więc zostaje to opcją którą można później zaimplementować).

2. Framework gry

2.1 Minifice

Klasa dziedzicząca po Microsoft.XNA.Framework.Game czyli podstawowej klasie biblioteki XNA. Nie zawiera niczego istotnego oprócz przechowywanego jako prywatne pole `ScreenManager`'a , oraz `GraphicsDeviceManager`'a. Ten drugi to klasa z biblioteki XNA zarządzająca urządzeniami graficznymi komputera.

Konstruktor

```
public Minifice()
{
	Inicjalizacje;
	Components.Add(screenManager);	// Dodanie screenManagera jako komponentu gry.
					// Więcej wyjaśnione w odpowiednim rozdziale.
	screenManager.AddScreen(odpowiednie warstwy);
}
```

Draw

Tutaj nic nie zachodzi oprócz czyszczenia ekranu. Prawdziwe rysowanie zachodzi w klasie ScreenManager.

2.2 ScreenManager

Klasa dziedzicząca po Microsoft.XNA.Framework.DrawableGameComponent czyli komponencie gry z biblioteki XNA. Dzięki temu, że została dodana do komponentów w klasie Game, będzie odświeżana i rysowana 60 razy na sekundę przy pomocy metod Update() oraz Draw().

Pola

```
List<GameScreen> screens – Lista wyświetlanych warstw
List<GameScreen> screensToUpdate – Pomocnicza lista wyświetlanych warstw
SpriteBatch spriteBatch – klasa XNA zajmująca się wyświetlaniem tekstur
```

Settings Property

Właściwość specjalna, zajmująca się ustawieniami gry. Pobranie ustawień sprawdza czy istnieje plik i czy jest poprawny, jeśli nie to tworzy takowy. Zapisanie ustawień robi to czego się można spodziewać.

Update

```
public override void Update(GameTime gameTime)
{
    Zaktualizuj stan klawiatury i myszy;
    Wyczyść WarstwyDoAktualizacji;
    Skopiuj Warstwy do WarstwyDoAktualizacji;
    InnaWarstwaMaFocus = !Game.IsActive;

    while (WarstwyDoAktualizacji.Count > 0)
    {
        warstwa = WarstwyDoAktualizacji.Ostatnia;
        WarstwyDoAktualizacji.UsuńOstatnią();
        warstwa.Update();

        if (warstwa jest aktywna || warstwa jest w fazie pojawiania się)
        {
            if (!InnaWarstwaMaFocus)
            {
                Zaktualizuj stan klawiatury i myszy w warstwie;
                InnaWarstwaMaFocus = true;
            }
        }
    }
}
```

Draw

```
public override void Draw(GameTime gameTime)
{
    Dla każdej warstwy w Warstwach)
    {
        if (warstwa nie jest ukryta)
            warstwa.Rysuj();
    }
}
```

FadeBackBufferToBlack

Metoda pomocnicza, rysuje czarny prostokąt na całym ekranie. Ważna, ponieważ jest aktywowana podczas efektu przechodzenia między warstwami, oraz gdy pojawia się okienko typu PopUp.

```
public void FadeBackBufferToBlack(float alpha)
{
    spriteBatch.Begin();
    spriteBatch.Draw(pustaTekstura, new Rectangle(0, 0, Szerokość ekranu, Wysokość
ekranu), Color.Black * alpha);
    spriteBatch.End();
}
```

2.3 GameScreen

Klasa wyświetlanych warstw. Każda warstwa może być typu PopUp (wtedy gdy jest wyświetlana to warstwa pod spodem nadal jest wyświetlana, tylko zaciemniona).

Update

Pozwala warstwie uruchamiać operacje takie jak aktualizacja efektu Transition. Metoda Update wykonywana jest zawsze (nawet jak okno jest nieaktywne).

```
public virtual void Update(GameTime gameTime, bool innaWarstwaMaFocus, bool przykryta)
{
    if (warstwa się wyłącza)
    {
        // Jeśli warstwa ma zostać usunięta to powinna też zacząć znikać
        Stan warstwy = znikająca;
        if (!UpdateTransition(gameTime, czasTransitionOff, 1))
            // Kiedy efekt znikania się skończy to usuwamy warstwę
            Usuń tą warstwę;
    }
    else if (przykryta)
    {
        // Jeśli warstwa jest przykryta przez inną to powinna zacząć znikać
        if (UpdateTransition(gameTime, czasTransitionOff, 1))
            Stan warstwy = znikająca;
        else
            Stan warstwy = ukryta;
    }
    else
    {
        // Wyświetlamy warstwę żeby uczynić ją aktywną
        if (UpdateTransition(gameTime, czasTransitionOn, -1))
            Stan warstwy = pojawiająca się;
        else
            Stan warstwy = aktywna;
    }
}
```

UpdateTransition

```
bool UpdateTransition(GameTime gameTime, TimeSpan czas, int kierunek)
{
    float Delta;
    if (czas == 0)
        Delta = 1;
    else
        Delta = (Pozostały czas / czas);

    Pozycja przejścia += Delta * kierunek;

    if (((kierunek < 0) i (Pozycja przejścia <= 0)) lub
        ((kierunek > 0) i (Pozycja przejścia >= 1)))
    {
        // Clamp zwraca pozycje przejścia jeśli jest pomiędzy 0 a 1
        // Jak jest mniej to zwraca 0, jak więcej to 1
        Pozycja przejścia = Clamp(Pozycja przejścia, 0, 1);
        return false;
    }
    return true;
}
```

2.4 InputState

Update

```
public void Update()
{
    poprzedniStanKlawiatury = aktualnyStanKlawiatury;
    aktualnyStanKlawiatury = Keyboard.GetState();

    poprzedniStanMyszy  = aktualnyStanMyszy;
    aktualnyStanMyszy = Mouse.GetState();
}
```

IsNewKeyPress

```
public bool IsNewKeyPress(Keys key)
{
    return (czy key jest wciśnięty i czy poprzednio był odciśnięty);
}
```

IsNewMouseLeftClick

Na tej samej zasadzie zaimplementowane mają być pozostałe dwie metody dotyczące myszki (RightClick i MouseMove).

```
public bool IsNewMouseLeftClick(out Vector2 Punkt)
{
    Punkt = new Vector2(aktualnyStanMyszy.X, aktualnyStanMyszy.Y);
    return (czy aktualnie lewy przycisk myszy wciśnięty i poprzednio odciśnięty);
}
```

2.5 FileManager

Klasa ta służy tylko do uniwersalnego serializowania klas do XML i na odwrót. Nie ma tu specjalnie skomplikowanych algorytmów.

2.6 Settings

Ustawienia gry, takie jak rozdzielczość ekranu, czy ustawienia dźwięku. Klasa to zwykła struktura przechowująca dane.

3. Menu Gry

3.1 MenuScreen

Abstrakcyjna klasa służąca do wyświetlania warstw menu. Każda instancja przechowuje listę wpisów w menu (MenuEntry) które wyświetlane są w odpowiednim miejscu na warstwie.

HandleInput

Metoda bada sytuację na klawiaturze i myszce i wywołuje odpowiednie eventy jeśli zostanie wybrana jakaś pozycja menu.

Draw

```
public override void Draw(GameTime gameTime)
{
    Ustaw lokalizacje wszystkich pozycji z menu
    spriteBatch.Begin();

    Dla każdej pozycji w Menu
    {
```

```
            Rysuj pozycję (jeśli jest wybrana to pokoloruj)
        }

        // Wyliczanie efektu "wjeżdżania" który wykonują pozycje w menu
        przesunięcie = pozycja przesunięcia podniesiona do kwadratu;

        spriteBatch.DrawString(tytuł menu, przesunięcie);

        spriteBatch.End();
    }
```

3.2 MenuEntry

Obiekty tej klasy rysowane są w klasach dziedziczących po MenuScreen.
Będą także przy wyświetlaniu „wjeżdżać" i wygasać lub zapalać się przy zmianie warstw. Po to jest pole selectionFade.

```
public event EventHandler<EventArgs> Selected;
// Event uruchamia się gdy pozycja zostanie wybrana
```

Update

```
public virtual void Update(GameScreen screen, bool isSelected, GameTime gameTime)
{
    SzybkośćFade = czas który minął * 4;

    if (zaznaczone menu)
        Fade = Min(Fade + SzybkośćFade, 1);
    else
        Fade = Max(Fade - SzybkośćFade, 0);
}
```

Draw

```
public virtual void Draw(GameScreen screen, bool isSelected, GameTime gameTime)
{
    Ustaw kolor, jeśli zaznaczone menu to inny niż jak

    pulsacja = Sinus(czas gry * 6) + 1;

    skala = 1 + pulsacja * 0.05f * Fade;

    kolor *= screen.TransitionAlpha;

    DrawString(tekst menu, pozycja, kolor, skala);
}
```

IsContained

```
public bool IsContained(Vector2 point, SpriteFont font)
{
    return (czy punkt zawiera się w obszarze zajmowanym przez napis);
}
```

3.3 MessageBoxScreen

Tak jak MenuScreen posiada listę MenuEntry i wyświetla je na podobnej zasadzie. Jest typu PopUp więc wyskakuje przed innymi okienkami nie zasłaniając ich całkowicie tylko wygasza je.

```
public event EventHandler<EventArgs> Accepted;
// Event uruchamia się gdy okienko zostanie zaakceptowane
public event EventHandler<EventArgs> Cancelled;
// Event uruchamia się gdy okienko zostanie zanulowane
```

HandleInput

```
public override void HandleInput(InputState input)
{
    Pobieraj stan klawiatury i myszki;
    Obsłuż wciśnięcia odpowiednio dla wciśniętych klawiszy myszki i klawiatury;
}
```

Update

```
public override void Update(GameTime gameTime, bool otherScreenHasFocus, bool
coveredByOtherScreen)
{
    base.Update();
    Dla każdej pozycji w menu
    {
        pozycja.Update(czy zaznaczona);
    }
}
```

Draw

```
public override void Draw(GameTime gameTime)
{
    Wygaś tło;
    Prostokat = Ustaw pozycje napisów na środku;
    spriteBatch.Begin();
    spriteBatch.Draw(gradient tła);
    spriteBatch.DrawString(tekst, prostokąt);
    dla każdej pozycji w menu
    {
        pozycja.Draw(na środku);
    }
    spriteBatch.End();
}
```

3.4 LoadingScreen

Klasa z prywatnym konstruktorem, powinna być tworzona tylko poprzez statyczną metodę Load. Służy do ładowania nowej warstwy (najczęściej po prostu gry). Ponieważ podczas ładowania gry może być wczytywane sporo zasobów, ta klasa będzie czekała i wyświetlała czarne okno „Ładowanie” dopóki wszystkie zasoby się nie załadują, jeśli odpowiednia flaga została ustawiona (loadingIsSlow).

Load

```
public static void Load(ScreenManager screenManager, bool loadingIsSlow, params
GameScreen[] screensToLoad)
{
    Dla każdej warstwy w ScreenManagerze
        warstwa.ExitScreen();
    LoadingScreen loadingScreen = new LoadingScreen(screenManager, loadingIsSlow,
screensToLoad);
    screenManager.AddScreen(loadingScreen);
}
```

Update

```
public override void Update(GameTime gameTime, bool otherScreenHasFocus, bool
coveredByOtherScreen)
{
    base.Update();

    if (nie ma innych warstw)
    {
        ScreenManager.RemoveScreen(this);

        Dla każdej warstwy w warstwyDoZaładowania
        {
            if (warstwa != null)
            {
                ScreenManager.AddScreen(warstwa);
            }
        }
        Zresetuj czas gry;
    }
}
```

Draw

```
public override void Draw(GameTime gameTime)
{
    if (warstwa jest aktywna i jest tylko jedna)
    {
        Nie ma innych okien  = true;
    }

    if (powolne ladowanie)
    {
        tekst = "Ładowanie...";

        spriteBatch.Begin();
        spriteBatch.DrawString(tekst, pozycja w Polowie ekranu);
        spriteBatch.End();
    }
}
```

3.3 ResultsScreen

Warstwa wyświetla wynik jaki otrzymał gracz i pyta gracza o jego imię, żeby zapisać do wyników głównych. Na końcu po wyjściu przełącza LoadingScreenem gracza do menu.

4. Warstwa Gry

4.1 GameplayScreen

Warstwa przechowująca cały zestaw tekstur potrzebnych do gry, oraz klasę GameManager. Poza tym cała logika gry oraz wyświetlania jest wykonywana w GameManager i podrzędnych klasach. Możliwości połączenia tej warstwy z GameManagerem rozwiązane są w taki sposób, że w metodach Draw, Update oraz HandleInput wykonywane są odpowiednie metody z klasy GameManager.

Draw

Metoda otwiera spriteBatch'a a potem uruchamia GameManager.Draw(spriteBatch) przed zamknięciem spriteBatcha'a. W ten sposób reszta rysowania mimo że wykonywana jest w innych klasach, nadal jest robiona na tej samej instancji spriteBatch. Dzięki temu można zachować

warstwowość wyświetlanych grafik (to co jest bliżej, zasłania to co dalej), można dzięki temu robić pseudo trójwymiarowość (2,5d).

LoadContent

```
public override void LoadContent()
{
    gameManager = GameManager.Load(content);
}
```

4.2 GameManager

Klasa zarządzająca całą logiką gry, oraz wyświetlaniem elementów gry w odpowiedni sposób. Przechowuje informacje na temat misji w której aktualnie znajduje się gracz, poziomu trudności, mapy i wszystkich jej elementów, wojowników którymi porusza gracz, przeciwników (na razie tylko jeden typ przeciwników, ze względu na to, że w specyfikacji użytkownika tak było, jednak nie stanowi problemu dodanie nowego typu przeciwników), bonusów które zebrał gracz, punktów które zdobył, czasu oraz broni jakie jeszcze pozostały w arsenale i w jakich ilościach. Klasa ta także musi być serializowalna, żeby można było w prostu sposób wczytywać dane z pliku oraz do pliku gdyby w przyszłości dorobić zapisywanie i wczytywanie gry.

Dodatkowo ma prywatny konstruktor, gdyż powinna być uruchamiana tylko poprzez metodę Load i w zasadzie nawet nie potrzeba z niego korzystać. Upublicznić konstruktor można by tylko jeśli zrobiłoby się aplikację do tworzenia map.

Draw

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Begin();

    Rysuj Mapę;
    Rysuj wszystkie postaci sterowane przez gracza;
    Rysuj wszystkich wrogów;
    Rysuj bronie;
    Rysuj Interfejs gry po lewej stronie od mapy;

    spriteBatch.End();
}
```

Update

```
public void Update(GameTime gameTime)
{
    Dla każdej postaci
    {
        if (postać jest żywa i nie jest kontrolowana przez gracza)
        {
            Ruszaj odpowiednio postacią;
            // Za pomocą strategii ruchu – te postaci mają ustawioną strategię Follow
            Strzelaj();
        }
    }
    if (wszystkie postaci gracza martwe)
    {
        Załaduj okno z wynikami i prośbą o podanie imienia do zapisania w Top5;
    }
    Dla każdego wroga
    {
        Sprawdź graczy w zasięgu wzroku;
```

```
            Ruszaj odpowiednio wrogiem;
            // Za pomocą strategii ruchu
            if (wróg jest aktywny)
                Strzelaj();
        }
        Dla każdego bonusu
        {
            Modyfikuj listę postaci sterowanych przez gracza;
        }
        Dla każdej broni
        {
            if  (broń.Update())
            {
                Usuń broń z listy broni;
            }
        }

}
```

HandleInput

```
public void HandleInput(InputState input)
{
    if (wciśnięty lewy przycisk myszy)
    {
        if (wciśnięty w obszarze mapy)
        {
            Ruszaj odpowiednio postacią kontrolowaną przez gracza;
        }
        if (wciśnięty guzik w bocznym interfejsie)
        {
            switch (guzik)
            {
            case ikona postaci:
                Aktywuj lub dezaktywuj postać jeśli żyje i nikt nie walczy;
                break;
            case ikona broni:
                Zmień broń głównej postaci jeśli jeszcze jest amunicja w tej broni;
                break;
            }
        }
    }
    if (wciśnięty prawy przycisk myszy)
    {
        if (wciśnięty w obszarze mapy)
        {
            Stwórz odpowiednią broń;
            Strzelaj wszystkimi postaciami gracza;
        }
    }
    if (wciśnięty klawisz Escape)
    {
       Pauza;
       Wyświetl PopUp „Menu Gry” w którym jest możliwość wyjścia z gry;
       // W przyszłości tutaj też powinna być możliwość zapisania gry
    }

}
```

Load

```
public static GameManager Load(ContentManager content)
{
    Deserializuj wszystkie obiekty GameManager'a z misji 1;
    Mapa.Load(content);
    Dla każdej postaci oraz wroga
    {
        Unit.Load(content);
    }
}
```

LoadMission

```
public void LoadMission(int missionId)
{
    Wyczyść klasę instancję klasy GameManager, zachowaj tylko punktacje;
    if (to była ostatnia misja)
        Załaduj okno z wynikami i prośbą o podanie imienia do zapisania w Top5;
    Deserializuj misje z odpowiedniego pliku;
    Mapa.Load(content);
    Dla każdej postaci oraz wroga
    {
        Unit.Load(content);
    }
}
```

4.3 GameMap

Przechowuje wszystkie dane na temat planszy po której porusza się gracz. W klasie zawarte są informacje na temat rozmiarów mapy (w ilości komórek, nie pikseli), skali obliczonej na podstawie rozdzielczości, punktu początkowego (lewego górnego rogu) mapy od którego zaczynać rysowanie (wyświetlany jest tylko fragment mapy, na którym akurat znajduje się aktywna postać gracza).

Draw

```
public void Draw(SpriteBatch spriteBatch)
{
    for (int i = 1; i < height; i++)
    {
        for (int j = 1; j < width; j++)
        {
            Mapa[i,j].Draw(spriteBatch, współrzędne(i,j));
        }
    }
}
```

Load

```
public void Load(ContentManager content)
{
    for (int i = 1; i < height; i++)
    {
        for (int j = 1; j < width; j++)
        {
            Mapa[i,j].Load(content);
        }
    }
}
```

CalculateCoord

```
public Point CalculateCoord(Vector2 coord)
{
    coord.X -= 200; // Rozmiar lewego interfejsu
    coord += przesuniecie mapy * skala ;
    punkt = coord / (przesuniecie mapy * skala);
    return punkt;
}
```

4.3.1 Sprite

Abstrakcyjna klasa przechowująca informacje o teksturze i jej wycinku który przeznaczony jest do rysowania.

Draw

```
public virtual void Draw(SpriteBatch spriteBatch, Point location, float layerDepth)
{
    spriteBatch.Draw(wycinek tekstury, location * rozmiar kafelka, layerDepth);
}
```

Load

```
public void Load(ContentManager content)
{
    Tekstura = content.Load<Texture2D>(textureLocation);
}
```

4.3.2 MapTile

Klasa służąca do przechowywania podkładu mapy, oraz wszystkich elementów leżących na jednym kafelku mapy.

Draw

```
public void Draw(SpriteBatch spriteBatch, Point location)
{
    Tło.Draw(spriteBatch, location);
    Dla każdego obiektu mapy
    {
        obiekt.Draw(spriteBatch, location);
    }
}
```

Load

```
public void Load(ContentManager content)
{
    Tło.Load(content);
    Dla każdego obiektu mapy
    {
        obiekt.Load(content);
    }
}
```

4.3.3 MapObject

Klasa przechowująca wszystkie obiekty które mogą wystąpić na mapie, oprócz przeciwników i postaci. Także może to być zwykła przeszkoda, jak słup, ściana czy krzesło, a może także to być bonus do zebrania.

Draw

Rysowanie Obiektu odbywa się w miejscu określonym przez rozmiar kafelka razy współrzędne kafelka. Dodatkowo granice obiektu wyznaczają gdzie na kafelku będzie obiekt. layerDepth jest zmienną która indykuje jak głęboko powinna być narysowana tekstura. Im niżej tekstura tym mniejszy powinna być ta zmienna, żeby rysowała się na wierzchu. W ten sposób można otrzymać pseudo 3D.

```
public void Draw(SpriteBatch spriteBatch, Point location, layerDepth)
{
    layerDepth = 0.001 – graniceObiektu.Min.Y * 0.000001;
    spriteBatch.Draw(tekstura, rozmiarKafelka * location + graniceObiektu,
layerDepth);
}
```

4.3.4 BackgroundSprite

Klasa pro-forma, służąca tylko do tego, żeby indykować że służy do wyświetlania tła planszy.

4.4 Unit

Klasa abstrakcyjna przechowująca informacje na temat postaci które mogą się przemieszczać po planszy, zarówno wrogów jak i wojowników gracza. Co do struktur które przechowuje to przechowywane są granice postaci, które indykują jak dużo miejsca na mapie zajmuje jednostka, przechowywane są animacje ruchu oraz śmierci jednostki, przechowywana jest szybkość ruchu, oraz rodzaj przemieszczania się dla jednostek nie kontrolowanych przez gracza.

Move

Abstrakcyjna metoda, do przeciążenia w dzieciach.

Shoot

Abstrakcyjna metoda, do przeciążenia w dzieciach.

Load

```
public void Load(ContentManager content)
{
    animation.Load(content);
    animationDeath.Load(content);
}
```

4.4.1 Fighter

Move

```
public void Move(GameMap gameMap, List<Fighter> fighters, List<Enemy> enemies,
InputState input, GameTime gameTime)
{
    if (kontrolowany przez gracza)
    {
        Przelicz pozycję myszy na współrzędne z mapy; (CalculateCoord)
        Oblicz odległość od punktu docelowego;
        // Oblicz drogę jaką przeszła postać w czasie od ostatniego przesunięcia:
        s = prędkość postaci * (czas teraz – czas ostatniego przesunięcia);
        Normalizuj wektor pozycji gracza względem pozycji myszy;

        Przesuń skopiowana pozycje postaci o s * znormalizowany wektor;
        Dla każdego przeciwnika, wojownika i obiektu mapy
        {
```

```
            Sprawdź czy nie zachodzi kolizja;
            if (koliduje z bonusem)
                Dodaj Bonus;
        }
        if (nie zachodzi kolizja)
            Przesuń postać(wektor);

        Animuj ruch(kierunek);
    }
}
```

Shoot

```
public void Shoot(InputState input, Weapon weapon, GameTime gameTime, ref List<Weapon>
weapons)
{
    if (czas od ostatniego strzału < 2 sekund)
        return;

    if (kontrolowany przez gracza)
    {
        Przelicz pozycję myszy na współrzędne z mapy;

        Ustaw punkt docelowy z losowym odchyleniem, zależnie od poziomu trudności (im
        wyższy, tym wyższa szansa na strzelanie po bokach). Punkt docelowy ustalony na
        podstawie zasięgu broni oraz wylosowanego odchylenia.

        Normalizuj wektor utworzony pomiedzy punktem docelowym, a pozycja postaci.

        Punkt docelowy = znormalizowany wektor * zasieg broni;
        Punkt początkowy = pozycja gracza;
    }
    else if (wrogowie w zasiegu)
        Strzelaj do pierwszego w zasiegu z bardzo słabą celnością;
    Dodaj broń do listy broni aktualizowanych w GameManagerze.
}
```

IsAlive

```
return (health > 0);
```

4.4.2 Enemy

Shoot

```
public void Shoot(InputState input, Weapon weapon, GameTime gameTime, ref List<Weapon>
weapons)
{
    if (czas od ostatniego strzału < 2 sekund)
        return;

    Jeśli jakaś postać gracza jest widoczna (nie zasłoniona przez żadne przeszkody) to
    strzelaj z losowym odchyleniem w jej kierunku. Im wyższy poziom trudności tym
mniejsze
    odchylenie.
    Normalizuj wektor utworzony pomiedzy punktem docelowym, a pozycja postaci.

    Punkt docelowy = znormalizowany wektor * zasieg broni;
    Punkt początkowy = pozycja gracza;

    Dodaj broń do listy broni aktualizowanych w GameManagerze.
}
```

Shout
W sytuacji gdy wróg się aktywuje ze względu na obecność gracza to wysyła sygnał do wszystkich innych sojuszników, którzy są od niego nie dalej niż na odległość zasięgu broni, żeby też się aktywowali.

CheckPlayers
Sprawdzanie czy nie ma jakiś graczy w zasięgu wzroku. Dzięki tej metodzie można w rozgrywce dodać ukrywanie się za przedmiotami przez graczy. Algorytm trywialny, przejrzenie całej tablicy postaci w poszukiwaniu bliskich wystąpień.

4.4.3 MoveStrategy
Klasa abstrakcyjna opisująca zachowanie postaci nie kontrolowanych przez gracza. Metoda Move służyć ma do wyliczania ścieżek które obrać ma postać żeby dostać się do gracza, oraz do faktycznego przesunięcia gracza na tej samej zasadzie co przesunięcie jest w metodzie Move przy aktywnym graczu.
Tablica z mapą przekształcana jest na graf za pomocą kafelków na których nic nie stoi. Tzn jeżeli kafelek jest zajęty przez przedmiot na który nie da się wejść (nawet jak jest mniejszy niż cały kafelek) to taki kafelek automatycznie jest wyłączany z grafu.

MoveStrategy::Follow::Move()

Strategia służąca do obsługi ruchu postaci niekontrolowanych przez gracza, tak żeby podążały za postacią gracza, ale starały się na nią nie wchodzić. Wyszukiwanie ścieżek odbywa się za pomocą algorytmu A*. Pozycja gracza aktywnego jest znana, tak samo jak pozycja aktualnej postaci, ograniczeniem do ruszania się jest krótka odległość za postacią gracza, tak żeby postaci na siebie nie nachodziły. Dozwolone jest żeby się przecinały, ale chodzi tylko o to, żeby algorytm nie powodował wejścia jednej jednostki na drugą.

MoveStrategy::Charge::Move()

Jest to podstawowa strategia ataku dla jednostek wroga. Aktywowana jest tylko, gdy wróg jest świadomy obecności gracza. Polega na wyszukaniu najlepszej ścieżki aby dojść do gracza, z założeniem, że nie podejdzie bliżej niż pół odległości zasięgu broni. Wyszukiwanie ścieżek odbywa się za pomocą algorytmu A*.

MoveStrategy::Patrol::Move()

Jest to strategia patrolowania. Postać nie stoi w miejscu, a chodzi pomiedzy jednym punktem a drugim (punkty zapisane jako pola klasy). W trakcie gdy patroluje jego prędkość poruszania się jest zmniejszona dwukrotnie.

MoveStrategy::Stand::Move()

Strategia ruchu która nic nie robi. Potrzebne dla postaci gracza które zostały dezaktywowane.

4.4.4 Weapon

Klasa abstrakcyjna z podwójnym przeznaczeniem. Po pierwsze służy do opisywania danego typu broni, jaki ma zasięg, szybkość itp. Drugie przeznaczenie służy do tworzenia instancji wystrzelonych pocisków na mapie. Każdy może mieć swoją oddzielną grafikę/animację. Kiedy pocisk wchodzi w kontakt z drużyną przeciwnika to zadaje tej jednostce obrażenia.

Draw

Rysuje animację w odpowiednim miejscu.

Load

```
public void Load(ContentManager content)
{
    Animacja.Load(content);
    AnimacjaWybuchu.Load(content);
}
```

Weapon::Pistol::Update()

```
public bool Update(GameMap gameMap, List<Fighter> fighters, List<Enemy> enemies,
GameTime gameTime)
{
    Droga = prędkość pocisku * (czas gry – czas startu lotu pocisku);
    Normalizuj wektor od startu pocisku do konca lotu pocisku;
    Pozycja pocisku = znormalizowany wektor * Droga;
    if (pocisk się z czymś styka)
    {
        If (to jest przeszkoda)
        {
            Zakończ lot pocisku;
            AnimacjaWybuchu.Animuj();
            return true;
        }
        If (to jest postać przeciwnej frakcji)
        {
            Zakończ lot pocisku;
            AnimacjaWybuchu.Animuj();
            If (postać.GetHit() i postać jest wrogiem)
            {
                Usuń postać z listy;
            }
            return true;
        }
    }
    Animacja.Animuj();
    return false;
}
```

Weapon::Grenade::Update()

```
public bool Update(GameMap gameMap, List<Fighter> fighters, List<Enemy> enemies,
GameTime gameTime)
{
    Oblicz krzywa do celu na bazie pozycji startowej i poczatkowej;
    Pozycja = krzywa(czas gry – czas startu pocisku);
    if (pocisk się z czymś styka)
    {
        Zakończ lot pocisku;
        AnimacjaWybuchu.Animuj();
        Dla każdej postaci w promienu r
        {
            If (postać.GetHit() i postać jest wrogiem)
            {
                Usuń postać z listy;
            }
        }
        return true;
    }
    Animacja.Animuj();
    return false;
}
```

Weapon::Mine::Update()

```
public bool Update(GameMap gameMap, List<Fighter> fighters, List<Enemy> enemies,
GameTime gameTime)
{
    if (czas od podlozenia miny > 3)
    {
        if (mina styka się z postacią)
        {
            Zakończ działanie miny;
            AnimacjaWybuchu.Animuj();
            if (postać.GetHit() i postać jest wrogiem)
            {
                Usuń postać z listy;
            }
            return true;
        }
    }
    Animacja.Animuj();
    return false;
}
```

4.4.5 Animation

Draw

```
public void Draw(SpriteBatch spriteBatch, Point location)
{
    Rysuj p-tą klatke z odpowiedniej listy w odpowiednim kierunku;
}
```

Animate

```
public void Draw(Direction direction, GameTime gameTime)
{
    Kierunek = direction;
    if (czas od ostatniej animacji < 1/fps)
        if (++progress > frames) progress = 0;
}
```

Load

```
public void Load(ContentManager content)
{
    Dla każdej klatki animacji
    {
        Klatka.Load(content);
    }
}
```

AnimationFrame

Klasa pro-forma, tylko po to, żeby indykować że dany Sprite jest klatką animacji.

4.4.6 Bonus

Bonusy to klasy które będą w jakiś sposób wpływały na grę. Czasem jako na grę jako całość, na przykład ładowanie nowego poziomu, a czasem na proste statystyki, jak szybsze chodzenie. Bonusy zbiera się stając na nich na mapie. Są obiektami po których można przechodzić, jednak znikają po przejściu przez nie.

Modify

Metoda abstrakcyjna która będzie dokonywać modyfikacji na grze jako całości.

Bonus::RebirthEgg::Modify()

```
public void Modify(Fighter fighter, GameManager gameManager)
{
    Sprawdź czy są jacyś martwi wojownicy;
    if (znalazł nieżywą postać gracza)
        Uzdrów postać;
}
```

Bonus::WelfareEye::Modify()

```
public void Modify(Fighter fighter, GameManager gameManager)
{
    Dodaj 2 granaty i 3 miny do zasobów gracza;
}
```

Bonus::DownFloorAbyss::Modify()

```
public void Modify(Fighter fighter, GameManager gameManager)
{
    Zabij dwie postaci gracza (kontrolowana przez gracza postać zabijana jako ostatnia
w kolejności);
}
```

Bonus::RushStar::Modify()

```
public void Modify(Fighter fighter, GameManager gameManager)
{
    if (czas działania < 15 sekund)
    {
        Dla każdej postaci gracza
            Przyspiesz postać dwukrotnie;
    }
}
```

Bonus::Rubble::Modify()

```
public void Modify(Fighter fighter, GameManager gameManager)
{
    if (czas działania < 20 sekund)
    {
        Dla każdej postaci gracza
            Spowolnij postać dwukrotnie;
    }
}
```

Bonus::NextLevel::Modify()

```
public void Modify(Fighter fighter, GameManager gameManager)
{
    gameManager.ŁadujNastępnyPoziom();
}
```

4.5 Boundaries

Klasa ma reprezentować zbiór punktów które ograniczają obiekt na mapie. Implementacja odbywa się za pomocą klasy z biblioteki XNA – BoundingBox. Jest to klasa napisana do ograniczania trójwymiarowych obiektów, więc aby przerobić ją na 2D wszędzie jako trzecią zmienną w wektorze podaję zero. Wklejam implementacje, bo nie ma tu zbyt wiele do tłumaczenia.

CreateFromPoints

```
public static Boundaries CreateFromPoints(IEnumerable<Vector2> points)
{
    List<Vector3> newPoints = new List<Vector3>();
    foreach (var p in points)
    {
        newPoints.Add(new Vector3(p.X, p.Y, 0));
    }
    return new Boundaries(BoundingBox.CreateFromPoints(newPoints));
}
```

GetCorners

```
public Vector2[] GetCorners()
{
    List<Vector2> corners = new List<Vector2>();
    foreach (var p in boundingBox.GetCorners())
    {
        corners.Add(new Vector2(p.X, p.Y));
    }
    return corners.ToArray();
}
```

Intersects

```
public bool Intersects(Boundaries boundaries)
{
    return boundingBox.Intersects(boundaries.boundingBox);
}
```