Grafika Komputerowa						
Rok akademicki	Termin	Rodzaj studiów	Kierunek	Prowadzący	Grupa	Sekcja
2014/2015	Wtorek 12:45 - 15:00	SSI	INF	dr Ewa Lach	GKiO3	1

Karta projektu

Danmaku Shooter

Skład sekcji:

Buchała Bartłomiej

Forczmański Mateusz

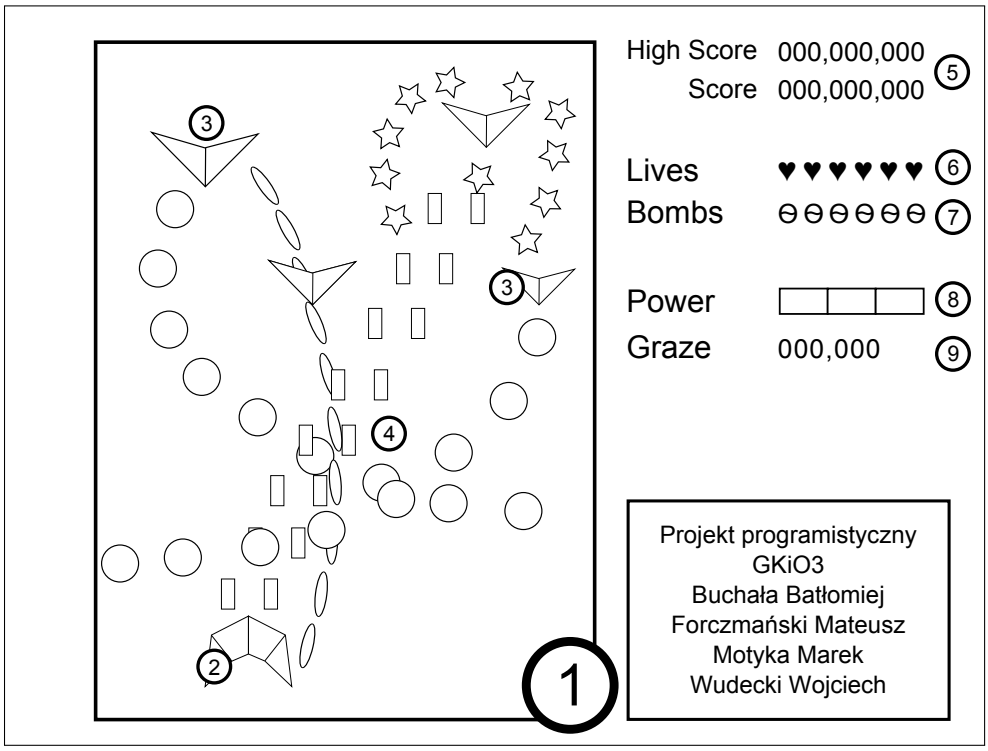
Motyka Marek

Wudecki Wojciech

Krótki opis aplikacji

Shoot' em up (w skrócie zwany shmup) jest gatunkiem gier akcji wywodzącym się w prostej linii od gier typu *Space Invaders* lub *River Raid*. Kontrolowana przez gracza postać (np. statek) w pojedynkę stawia czoło przeciwnikom, niszcząc ich za pomocą wystrzeliwanych pocisków, jednocześnie unikając ich ataków. Podgatunek shmupów, zwany *danmaku* (z jap. *ściana pocisków* lub *piekło pocisków*) kładzie większy nacisk na omijanie wrogich ataków, niż na ofensywie. Przykładowymi danmaku są np. *Ikaruga* czy większość gier z uniwersum *Touhou Project*.

Gra powstawać będzie jako projekt łączony z przedmiotów Projekt Programistyczny i Grafika Komputerowa.



1. Ekran gry właściwej. W jej obrębie znajduje się gracz, pociski oraz wszyscy wrogowie.
2. Grywalna postać, porusza się po ekranie gry, unikając pocisków oraz strzelając do wrogów.
3. Wrogowie, których należy pokonać.
4. Chmara pocisków. Jak widać na rysunku, nie wchodzą ze sobą w żadną interakcję, każdy leci swoim wyznaczonym torem. Sprajty wrogów są niewrażliwe na swoje pociski, nie występuje *friendly fire*.
5. Liczba zdobytych punktów oraz porównywanie ich z największym wynikiem.
6. Liczba pozostałych żyć. W trakcie gry można zdobywać kolejne. Utrata wszystkich kończy grę.
7. Liczba pozostałych bomb. Każda wykorzystana bomba zapewnia kilkusekundową odporność na wrogie pociski oraz umożliwia pojedynczy silniejszy atak. Można je zdobyć w trakcie gry.
8. Pasek mocy, napełnia się w trakcie gry wraz ze zdobytymi punktami. Utrata życia skutkuje zmniejszeniem paska o 1 segment.
9. Liczba "otarć", czyli uniknięć bardzo blisko pocisku. Aby umożliwić większe wyzwanie, ostateczny wynik pomnożony jest przez licznik Graze.

Analiza zadania

1. Podstawy teoretyczne problemu

1.1. Przestrzeń fizyczna

W naszej grze w interakcję ze sobą będzie wchodzić bardzo dużo elementów, począwszy od gracza, a na ostatnim z setki pocisków skończywszy. Bardzo ważne jest, aby ruchy i interakcje wszystkich elementów były ze sobą zsynchronizowane, a droga o długości 1 była tym samym dla każdego z nich.
W tym celu zdecydowaliśmy się na wprowadzenie do gry przestrzeni fizycznej, w której będą zachodzić wybrane przez nas zjawiska fizyczne, potrzebne dla realizacji gry.

1.2. Ruch

Każdy obiekt, który może się poruszać i zmienić swoje położenia, posiada swoją prędkość. W naszej grze wyróżniliśmy 3 rodzaje ruchu:

1.2.1. Ruch jednostajny

W jednostce czasu ciało pokonuje jednakową drogę, a przebyta droga jest proporcjonalna do czasu.

$$v = \frac{s}{t} = const$$

Gdzie v to prędkość, s - droga, a t to czas. W naszym obiekty poruszają się na ekranie, więc jednostką długości jest piksel.

1.2.2. Ruch jednostajnie przyspieszony

W jednostce czasu prędkość ciała ulega zwiększeniu o stałą wartość.

$$v(t) = v_0 + a \cdot t$$

1.2.3. Ruch jednostajnie opóźniony

W jednostce czasu prędkość ciała ulega pomniejszeniu o stałą wartość.

$$v(t) = v_0 - a \cdot t$$

1.3. Tor pocisków

Jednym z podstawowych problemów w naszym zadaniu jest tor po jakim poruszają się pociski. Obiekty wrogów będą poruszać się w prosty sposób, jednak zbiory pocisków będą układać się w skomplikowane wzory, tworzyć ze sobą specyficzny układ. W naszej grze chcemy zaimplementować pociski, które będą poruszać się m.in. po takich torach jak okrąg i elipsa.

1.3.1. Okrąg

$$\begin{cases} x = x_0 + r \cdot \cos\theta \\ y = y_0 + r \cdot \sin\theta \end{cases}$$

Gdzie punkt $O(x_0, y_0)$ jest środkiem okręgu, r promieniem, a parametr $\theta \in [0, 2\pi)$.

1.3.2. Elipsa

Podobnie jak wyżej, elipsę zdefiniowaliśmy równaniem parametrycznym:

$$\begin{cases} x = x_0 + a \cdot \cos\theta \\ y = y_0 + b \cdot \sin\theta \end{cases}$$

Gdzie punkt $O(x_0, y_0)$ jest środkiem elipsy, a, b długościami półosi, a parametr $\theta \in [0, 2\pi)$.

1.3.3. Spirala

Będziemy wykorzystywać spirale z gatunku spiral Archimedesa. Tor takiej spirali jest zdefiniowany równaniem parametrycznym:

$$\begin{cases} x = x_0 + a \cdot \theta \cdot \cos\theta \\ y = y_0 + a \cdot \theta \cdot \sin\theta \end{cases}$$

Gdzie punkt $O(x_0, y_0)$ jest punktem centralnym spirali, $a \in R$, a parametr $\theta \in [0, \text{inf})$.

1.3.4. Spirala Fermata

Typ spirali zdefiniowany równaniem parametrycznym:

$$\begin{cases} x = x_0 + a \cdot \sqrt{\theta} \cdot \cos\theta \\ y = y_0 + a \cdot \sqrt{\theta} \cdot \sin\theta \end{cases}$$

Gdzie punkt $O(x_0, y_0)$ jest punktem centralnym spirali, $a \in R$, a parametr $\theta \in [-inf, \text{inf})$.

1.3.5. Spirala hiperboliczna

Typ spirali zdefiniowany równaniem parametrycznym:

$$\begin{cases} x = x_0 + ae^{b\theta} \cdot \cos\theta \\ y = y_0 + ae^{b\theta} \cdot \sin\theta \end{cases}$$

Gdzie punkt $O(x_0, y_0)$ jest punktem centralnym spirali, $a, b \in R$, a parametr $\theta \in [0, \text{inf})$.

2. Wykorzystywane zagadnienia grafiki komputerowej

2.1. Przekształcenia afiniczne

Postanowiliśmy wykorzystać poznane na laboratorium przekształcenia afiniczne. Wszystkie ruchome obiekty w naszej grze, które zmieniają dynamicznie swoją pozycję w przestrzeni dwuwymiarowej, będą korzystać z trzech wybranych przekształceń:

2.1.1. Translacja

Przesunięcie punktu $P = (x, y)$ o wektor $d = [d_x, d_y]$, w wyniku którego powstaje obraz $P = (x', y')$, gdzie:

$$\begin{cases} x' = x + d_x \\ y' = y + d_y \end{cases}$$

2.1.2. Skalowanie

W naszym programie wszystkie przekształcenia skalowania będą zachowywać proporcje, więc zdefiniować je jako: przesunięcie punktu $P = (x, y)$ o współczynnik skalowania s, w wyniku którego powstaje obraz $P = (x', y')$, gdzie:

$$\begin{cases} x' = s \cdot x \\ y' = s \cdot y \end{cases}$$

2.1.3. Obrót

Obrót punktu $P = (x, y)$ wokół początku układu $O = (0, 0)$ da nam obraz $P = (x', y')$, gdzie:

$$\begin{cases} x' = x\cos(x) - y\sin(x) \\ y' = x\sin(x) + y\cos(x) \end{cases}$$

2.2. Krzywe i powierzchnie parametryczne

W naszym projekcie pojawią się krzywe parametryczne. Będą spełniać rolę toru pocisków. W celu opisania krzywej na przestrzeni n-wymiarowej są potrzebne n funkcji, które opisują współrzędne punktów tej krzywej. My wykorzystamy z wielomianów 3 stopnia, ponieważ są wystarczająca elastyczne jeśli chodzi o zmianę kształtu krzywej i nie powodują wykonywania dużej ilości dodatkowych obliczeń. Wielomian opisujący segment krzywej:

$$Q(t) = [\, x(t)\; y(t)\; z(t)\,]^T \qquad 0 \leqslant t \leqslant 1$$

Gdzie:

- $x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$
- $y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$
- $z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$

2.2.1. Krzywe Beziera

Fragment krzywej Beziera opisywany jest przez dwa punkty końcowe P_0 i P_3 (interpolacja) i dwa punkty pośrednie P_1 i P_2 (aproksymacja), które decydują o postaci wektorów w punktach końcowych.
Krzywą Beziera trzeciego stopnia definiuje równanie wielomianowe:

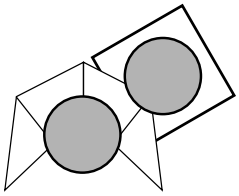
$$Q(t) = (1-t)^3 \cdot P_0 + 3t \cdot (1-t)^2 \cdot P_1 + 3t^2(1-t) \cdot P_2 + t^3 \cdot P_3$$

2.3. Wykrywanie kolizji

Jako kolejne zagadnienie wykorzystamy wykrywanie kolizji. Jest ono potrzebne ze względu na zderzenia pocisków z obiektami i przeciwnikami. Naszym zadaniem będzie wykrycie, że kolizja miała miejsce i odpowiednia reakcja na tę sytuację (utrata życia, animacja zderzenia itd.).

Z dwóch rodzajów kolizji, statycznych i dynamicznych, zdecydowaliśmy się na realizację tych pierwszych. Są prostsze do realizacji, wymagają mniej obliczeń i szerzej omówione w ramach laboratorium. Aby uniknąć potencjalnych problemów związanych ze sprawdzaniem stanu klatka po klatce będziemy ograniczać prędkość pocisków tak, by nie występowało omijanie kolizji.

W celu optymalizacji tego procesu będziemy stosować zmodyfikowaną metodę sfery otaczającej - zamiast otaczać cały model kulą, będzie się ona znajdować w jego wnętrzu na scenie 2D jako tzw. *Hitbox*. Podobnie jak przy sferze otaczającej, jego środek równy środku sprajta, lecz jego promień będzie odpowiednio mniejszy. Każdy element gry, z którym gracz może wejść w kontakt, będzie posiadał swój Hitbox. Elementy takie jak tło czy pasek życia nie będą go miało i pozostaną tylko rysunkami.
W celu testowania kolizji będzie stosować prostą metodę *sphere - sphere*.



Ilustracja: jak widać, chociaż oba sprajty na siebie na nachodzą, do kolizji nie dochodzi gdyż oba Hitboxy są od siebie oddalone.

3. Wykorzystywane biblioteki i narzędzia programistyczne

3.1. DirectX 9

Jako narzędzie do budowania obiektów graficznych zdecydowaliśmy się na DirectX w wersji 9. Jako alternatywne rozwiązanie rozważaliśmy DirectX 11, jednak nasza gra będzie budowana na scenie 2D, a DirectX 9 oferuje wygodniejsze narzędzia - wciąż operuje na takich klasach jak Sprite lub Texture, które są lepsze dla naszej gry. DirectX 11 wszystkie te klasy zastępuje interfejsem IResource, który wymaga odpowiedniej konwersji (większe nastawienie na grafikę trójwymiarową). By móc wykorzystać DirectX 11 do pracy na zwykłych sprajtach, wymagane byłyby dodatkowe zestawy narzędzi, jak np. DirectX Tool Kit.
Jako konkurencyjne rozwiązanie dla samego DirectXa rozważaliśmy OpenGL, jednak nasz program będzie zorientowany obiektowo, a DirectX daje lepsze możliwości enkapsulacji.
Ponadto, dzięki DirectXowi wygodniejsze jest pracowanie m.in. z przekształceniami afinicznymi modeli 3D.

3.2. Microsoft Visual Studio 2012

Wybraliśmy to środowisko ze względu na przyjazny interfejs, który nawet w najtrudniejszych sytuacjach potrafi okazać się bardzo pomocny. Pracujemy na nim od dłuższego czasu co ułatwi nam szybki dostęp do potrzebnych narzędzi. Etap debugowania posiada wiele możliwości kontroli naszego programu, co na pewno pozwoli uniknąć kilku czasochłonnych błędów. Ponieważ czasami pojawiają się problemy w kompilacji związane z różnymi wersjami VS, wspólnie wybraliśmy jedną wersję środowiska w celu usunięcia tych przeszkód.

3.3. Enterprise Architect

Jako narzędzie do zaprojektowania gry w modelu zorientowanym obiektowo wybraliśmy Enterprise Architect. Głównym powodem była nasza znajomość języka UML, którego uczymy się na studiach od ponad roku, a także doświadczenie z tym środowiskiem. Zdecydowaną zaletą wcześniejszego zaprojektowania aplikacji w tym środowisku jest wygoda obsługi, łatwość tworzenia diagramów oraz możliwość generacji kodu. Samo zdecydowanie się na wcześniejsze utworzenie diagramów UML umożliwi nam lepszą kontrolę nad pracą oraz zapewnienie wszystkich potrzebnych możliwości naszej aplikacji.

3.4. Inkscape & Photoshop

Naszymi narzędziami graficznymi zostały:

- **Inkscape**: bardzo wydajny i zaawansowany program do grafiki wektorowej. Posiada ogromne możliwości w tworzeniu prostych i złożonych figur geometrycznych, ma bardzo dużą gamę tekstur oraz dzięki dobrym filtrom umożliwia tworzenie ciekawych efektów niewielkim kosztem. Dzięki operowaniu na grafice wektorowej umożliwia pracę na figurach i ich konwersję do popularnych formatów obrazków (JPEG, PNG) w wybranej rozdzielczości bez utraty jakości. Ponadto Inkscape posiada wsparcie dla grafiki 3D.
- **Adobe Photoshop**: drugi, również bardzo dobry program do tworzenia zaawansowanej grafiki 2D. Operuje na pikselach, lecz posiada klasyczny zestaw narzędzi (np. pędzel, gumka), które umożliwiają bardziej intuicyjne tworzenie grafiki. Dzięki wygodnej obsłudze warstw, licznym filtrom i narzędziom w szczególności umożliwia dobrą edycję rysunków.

4. Algorytmy, struktury danych, ograniczenia specyfikacji

4.1. Algorytm De Casteljau

W naszym programie konieczne okazało się zastosowanie tylko jednego algorytmu, do tworzenia krzywych Beziera. Wybraliśmy algorytm De Casteljau, który jest prosty w zrozumieniu i implementacji, ponadto był omawiany na wykładach profesora Wojciechowskiego.

Zasada działania: krzywa sześcienna przedstawiona jest jako 4 punkty kontrolne. Algorytm dzieli odcinki P_0P_1, P_1P_2, P_2P_3 w proporcji $t : 1-t$, gdzie $t \in [0,1]$. Operację tę należy powtarzać aż do momentu otrzymania pojedynczego punktu $P(t)$. Uzyskany punkt leży na krzywej Beziera i dzieli ją na dwie części o punktach kontrolnych $P_0P_0^1P_0^2P(t)$ i $P(t)P_1^2P_2^1P_3$.

Pseudokod:

```
-- P[n] - tablica n punktow kontrolnych
-- t - wspolczynnik z przedzialu od 0 do 1
function deCasteljau(P[n], t)
begin
    for  i:=0 to n do
        Q[i] := P[i];
        for k:=1 to n-k do
            Q[i] := (1-t) * Q[i] + t * Q[i+1];
        end for;
    end for;
    return Q[0]; -- punkt na krzywej Q(t)
end.
```

Plan pracy

1. Zaprojektowanie gry w języku UML

a. Utworzenie modelu przypadków użycia aktora Gracza

b. Utworzenie modelu klas, w postaci diagramu, z wszystkimi potrzebnymi relacjami.

c. Zarysowanie scenariuszy pierwszoplanowych.

2. Przygotowanie szkieletu aplikacji

a. Napisanie okna w WIN API jako punktu wejściowego aplikacji.

b. Inicjalizacja silnika graficznego Direct 3D w celu umożliwienie pisania i testowania obiektów graficznych od samego początku.

c. Zdefiniowanie zegara gry, do którego obiektu muszą dostosowywać swoje zachowanie.

d. Napisanie klasy nadrzędnej GameObject dla ruchomych obiektów gry - podstawa dla przekształceń afinicznych.

e. Klasa Sprite służąca do rysowania i kontroli elementów gry. W późniejszej części pracy, dopóki modele sprajtów końcowych nie będą gotowe, będziemy pracować na prymitywach.

3. Części niezależne od siebie

a. Implementacja klasy Gracza, synchronizacja z klawiaturą, umożliwienie strzelania i wykorzystywania bomb.

b. Napisanie klasy Pocisk oraz Wzorzec, które kontrolują pociski i układają je we układy oparte na funkcjach matematycznych

c. Napisanie klas Wrogów oraz integracja ich z typem wrogich pocisków

d. Utworzenie ekranu powitalnego (tzw. *title screen*) umożliwiającego zmianę konfiguracji i rozpoczęcie gry.

e. Stworzenie uniwersalnej konfiguracji klawiszy - zdefiniowanie kontrolek dla Gracza oraz możliwości ich zmiany przez użytkownika.

4. Części zależne od siebie

a. Umożliwienie interakcji Hitboxów - implementacja otarć, kolizji, straty życia.

b. Szczegółowa integracja obiektów gry na scenie: określenie w których momentach pojawiają się wrogowie, kiedy strzelają, kiedy zostają wyeliminowani.

c. Naliczanie zdobytych punktów.

5. Wykonanie modeli graficznych

a. Narysowanie obiektu statku w technice 3D.

b. Narysowanie tła ekranu powitalnego oraz gry, a także panelu ze statystyką.

c. Narysowanie modeli wrogów.

d. Narysowanie kształtów pocisków oraz bonusów, opartych o prymitywy.

Podział pracy

1. Buchała Bartłomiej

a. **Klasa Player**: uniemożliwienie wyjścia graczowi poza planszę, utworzenie trybu *focus*, umożliwienie mu strzelania pociskami i bombami, narysowanie bomby i pocisków, synchronizacja z klasą Input.

b. **Klasa Game**: narysowanie interfejsu gry, wyświetlenie wszystkich danych: liczby żyć i bomb, przechowywanych w klasie Player oraz wyniku i liczby grejzu, przechowywanych w klasie Game, usunięcie pocisków gracza i wrogów z pamięci gdy wyjdą poza planszę, umożliwienie zatrzymania i zakończenia gry, wejścia do menu oraz zapisania wyniku gry do pamięci.

c. **Klasy typu Bullet**: zrealizowanie ruchu pocisków zgodnie ze wskazanymi torami - domyślnie po prostym wektorze, z możliwością zmiany na wskazany tor, umożliwienie przyspieszenia, silna parametryzacja, narysowanie kilku kształtów pocisków w różnych kolorach.

2. Forczmański Mateusz

a. **Projekt UML**: utworzenie modelu przypadków użycia, zarysowanie scenariuszy pierwszoplanowych, narysowanie diagramu klas z właściwymi relacjami.

b. **Okno gry**: utworzenie okna w języku WinAPI, inicjalizacja i konfiguracja Direct3D 9, implementacja zegara gry.

c. **Klasy Pattern i Spellcard**: generowanie i układanie wzorów z pocisków, reagowanie na usunięcie pocisków gdy wyjdą poza planszę lub zostaną usunięte przez bombę, realizacja bonusów za pokonanie karty czarów bez straty życia.

d. **Klasa Sprite**: rysowanie sprajtów na ekranie gry, implementacja translacji, skalowania i rotacji, umożliwienie przechowywania i wyboru większej liczby tekstur.

3. Motyka Marek

a. **Klasa Input**: enkapsulacja wszystkich funkcji do obsługi klawiatury, synchronizacja z klasami integralnymi (gra oraz ekran powitalny). Stworzenie uniwersalnych klawiszy – *Shoot*, *Bomb*, *Focus* itp., które będą połączone z odpowiadającymi im wartościami z klasy Input.

b. **Klasa Hitbox**: implementacja, narysowanie sprajta hitboxa, zrealizowanie obsługi zdarzeń i styków dwóch hitboxów, wykrywanie *graze* dla hitboxów klas Player i EnemyBullet (poprzez ustawienie pewnej odległości) oraz obsługa utraty życia przy nałożeniu się hitboxów tych klas.

c. **Klasy typu Bonus**: implementacja, narysowanie sprajtów, synchronizacja z klasami Player (zebranie, zwiększenie powera) oraz Game (zwiększenie wyniku), zwolnienie pamięci po zebraniu przez gracza lub wyleceniu z planszy.

4. Wudecki Wojciech

a. **Klasa TitleScreen**: narysowanie ekranu powitalnego, utworzenie menu z wszystkimi możliwościami, utworzenie klasy Menu z szczegółowymi możliwościami zmian, zapisywanie i odczytywanie ustawień z pamięci, utworzenie sprawnego przejścia z ekranu powitalnego do gry i na odwrót.

b. **Klasa Enemy**: implementacja, umożliwienie strzelania i wykorzystywania patternów, realizacja wrażliwości na pociski gracza, narysowanie sprajtów wrogów.

c. **Klasa Stage**: przechowywanie wszystkich obiektów wrogów, decydowanie który jakiego wzoru pocisków używa, silna integracja z timerem – to Stage ma wiedzieć, w której sekundzie pojawiają się wrogowie, kiedy mają strzelić i zejść ze sceny, narysowanie tła.