Grafika Komputerowa						
Rok akademicki	Termin	Rodzaj studiów	Kierunek	Prowadzący	Grupa	Sekcja
2014/2015	Piątek 09:30 - 11:00	SSI	INF	dr Ewa Lach	GKiO3	1

Raport wersji Beta

Danmaku Shooter

Skład sekcji:

Buchała Bartłomiej

Forczmański Mateusz

Motyka Marek

Wudecki Wojciech

Realizacja zadań

1. Interfejs gracza

1.1. Ekran powitalny

Znajduje się na nim logo naszej gry oraz menu z możliwością wyboru jednego z nich. Ekran jest w pełni narysowany, jednak w tej chwili umożliwia wyłącznie uruchomienie gry.

1.2. Ekran gry

W pełni zrealizowany. Na ekranie widoczne są wszystkie informacje potrzebne dla gracza jak liczba żyć, bomb i zdobyte punkty. Ekran został wyraźnie podzielony na część interaktywną, gdzie dzieje się właściwa gra, oraz część interfejsu. Informacja o rozmiarze i umiejscowieniu tych pól znajduje się w stałych gry.

2. Obiekty gry

2.1. Gracz

Może poruszać się tylko w obrębie części interaktywnej, posiada możliwość strzelania i niszczenia wrogich elementów.

2.2. Wrogowie

Mogą strzelać, poruszać się i otrzymywać obrażenia. Wróg umiera przy zderzeniu z graczem, straceniu całego życia lub przebyciu swojej drogi.

2.3. Pociski

Wszystkie posiadają swój sprajt, ich kształt i droga są zależny od Wzoru do której należą. Pocisk znika przy zderzeniu z graczem lub wyjściu poza planszę. Zostały zaimplementowane zarówno wrogie pociski, jak i gracza, a kolizje sprawdzane są dla obu grup z osobna.

2.4. Wzory

Wzór jest zbiorem pocisków. Rezerwuje dla nich pamięć, inicjalizuje je oraz aktualizuje. Oparty o silną agregację - gdy wszystkie pociski zostaną skasowane wzór też zostaje usunięty.

2.5. Tory

Wrogowie, pociski i bonusy mogą poruszać się po 3 rodzajach tras: liniowej, eliptycznej oraz spiralnej. Istnieje również tor typu None, wymuszający obiekt do stania w miejscu.
Istnieje możliwość zagnieżdżania torów. Przykładowo, każdy pocisk posiada swój tor, ale też musi znajdować się jako element wzoru. Z kolei sam wzór również posiada swój tor. Jeżeli Wzór się przemieszcza, tory wszystkie należących do niego pocisków ulegają translacji.

2.6. Bonusy

Istnieją 4 bonusy, które gracz może zbierać: *score* i *power*, które dodają przechowywaną wartość do wyniku punktowego lub siły, oraz *life* i *bomb*, które zwiększają liczbę żyć lub bomb o jeden.

2.7. Bomba

Specjalny atak, który można wykorzystać tylko kilka razy w grze. Posiada ogromny zasięg, zadaje duże obrażenia i likwiduje wszystkie pociski na planszy.

3. Wczytywanie gry z pliku

Istnieje klasa Stage, która jest małym parserem pliku XML w którym zawarte są dane na temat planszy. Format pliku obejmuje małą obsługę zdarzeń: w której sekundzie pojawiają się wrogowie, w jakiej ilości i jak atakują. Klasa Game odpytuje Stage i otrzymuje wskaźnik na kolekcję przeciwników.
W ramach mechanizmu serializacji na chwilę obecną zostały zaimplementowane fabryki obiektów: bonusów oraz trajektorii. W czasie uruchamiania programu klasy te rejestrują wszystkie klasy, których obiekty mogą produkować i przy wywołaniu funkcji z podaniem id klasy zawracają wskaźnik do nowego obiektu.

4. Cechy obiektów

4.1. Ruch

Wszystkie obiekty mogą się poruszać po swoim torze z pewną szybkością i przyspieszeniem.

4.2. Sprajt

Każdy obiekt dziedziczący po GameObject posiada wskaźnik na sprajt, który należy przydzielić w trakcie ładowania gry. Wszystkie sprajty są tworzone podczas uruchamiania gry i przechowywane w jednym miejscu pamięci - obiekty gry tylko się do nich odnoszą i każą rysować to czego potrzebują.

5. Przekształcenia afiniczne

- Wszystkie obiekty implementujące interfejs ITransformable posiadają możliwość translacji, skalowania i obrotu.
- Dotyczy to zarówno sprajtów, których efekty są widoczne, oraz dróg po których obiekty gry się poruszają. Na przykład droga w kształcie elipsy może być skalowana, co powoduje jej zmianę jej rozmiarów i tym samym sposobu wszystkich obiektów, które się po niej poruszają.
- Podczas wczytywania obiektów z pliku XML jest możliwe ustalenie zarówno przekształceń początkowych, jak i zmieniających się regularnie w czasie.

6. Wykrywanie kolizji

Wykrywanie zderzeń obiektów odbywa się poprzez sprawdzenie ich "hitboxy", obiekty w kształcie elipsy znajdujące się w centrum sprajta, nie nachodzą na siebie. Wykrywanie kolizji odbywa się na kilka sposobów:

- Czy pocisk gracz uderzył wroga?
- Czy pocisk wroga uderzył gracza?
- Czy pocisk wroga otarł się o gracza?
- Czy sam wróg uderzył w gracza?
- Czy bonus trafił w gracza?

Każda z tych 5 procedur jest wywoływana przez osobną funkcję. W czasie gdy zdarzają się momenty, że gracz przez pewien czas jest niezniszczalny: podczas wykorzystywania bomby oraz przez kilka sekund po stracie życia. Wówczas sprawdzanie kolizji wrogich obiektów z graczem jest pomijane.

Napotkane problemy

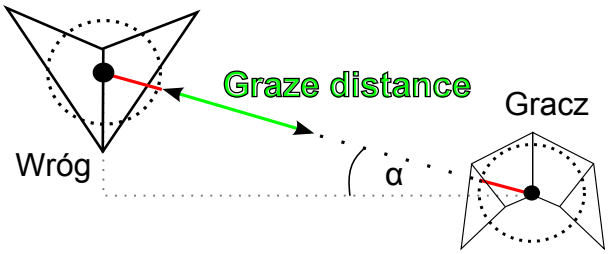
1. Źródło pocisków

Wyłącznie obiekt klasy Enemy może być źródłem wrogich pocisków, które mogą zaszkodzić Graczowi. Jednak w momencie gdy wróg zostaje zabity, a jego obiekt usunięty, dostęp do pocisków zostaje utracony. Dlatego zdecydowaliśmy wystrzelone pociski przez Wroga pociski przekazywać do Gry w momencie jego śmierci.

Jednakże to rozwiązanie wymaga konieczności iterowania po dwóch rożnych kolekcjach: pociskach przez wroga i pociskach na planszy, a których obsługa jest dokładnie taka sama. Aktualnie myślimy nad bardziej efektywnym rozwiązaniem - rozdzieleniem generowania i aktualizowania pocisków i ich stanu.

2. Algorytm rozwiązywania kolizji

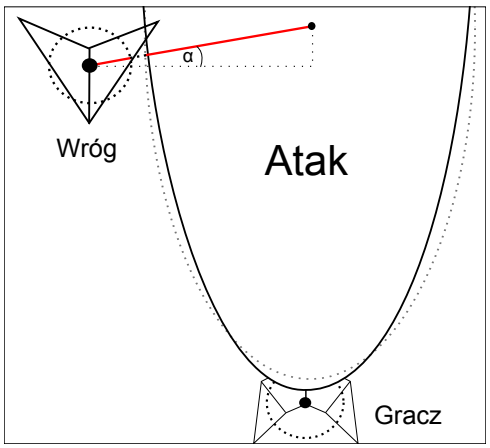
2.1. Dystans i otarcia



Każdy obiekt gry posiada swojego tzw. hitboxa, elipsę lub koło, które wyznacza obszar zdarzeń obiektu. Podczas testowania kolizji algorytm oblicza:

1. Dystans i kąt pomiędzy wrogimi obiektami
2. Dla każdego hitobxa dystans pomiędzy punktem środkowym, a punktem znajdującym się na linii zderzenia
3. Jeżeli oba hitboxy są ze sobą styczne lub na siebie nachodzą, następuje kolizja
4. Jeżeli nie doszło do kolizji, ale obiekt Gracza znajduje się w pewnej określonej odległości między wrogim obiektem, tak, że można powiedzieć, że się o niego "otarł", zdobywa dodatkowe punkty.

2.2. Kształt hitboxa



Kształt naszego "pola zderzeń" nie jest stricte kołem, ale elipsą. Decyzja wynikła z tego, że w naszej grze będą duże, wypukłe obiekty dla których pole zderzeń w postaci koła może wyglądać nieatrakcyjnie. Dlatego zdecydowaliśmy się na elipsę, która dostosowuje wymiary swoich półosi do boków sprajta.

2.3. Efektywność testów

Ponieważ kształtem hitboxa nie jest okrąg, ale elipsa, w czasie sprawdzania wymagana jest większa liczba informacji: wymagane jest obliczenie kąta pomiędzy obiektami oraz uwzględnienie obrotu obiektu. W czasie pojedynczego wykonania aktualizacji stanu gry wykonywane są tysiące testów kolizji, co dla obiektów, dla których wystarczyłby okrąg prowadzi do wielu niepotrzebnych obliczeń.
Na chwilę obecną rozważamy utworzenie osobnych klas dla okręgu oraz elipsy - obliczanie promienia byłoby zakapsułkowane, a obliczenia dla elipsy wykonywane gdy byłoby to naprawdę konieczne.

3. Zarządzanie zasobami

3.1. Problem

W naszych pierwszych krokach każdy obiekt gry posiadał osobny dla siebie sprajt. Nawet jeżeli wszystkie Pociski we Wzorze posiadały ten sam sprajt, każdy miał osobno przydzieloną pamięć. Powodowało to znaczne opóźnienia zarówno we wczytywaniu jak i podczas samej gry.

3.2. Rozwiązanie

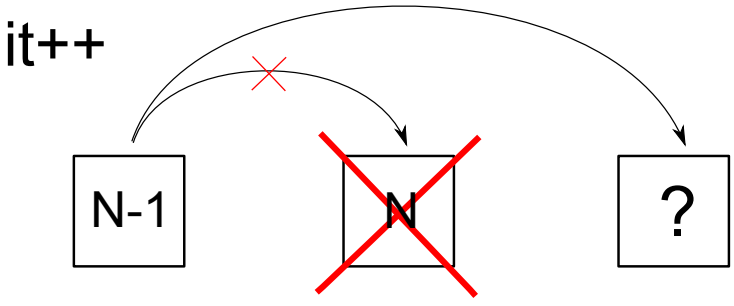
Później zdecydowaliśmy się utworzyć nowy typ klas *Resource*, które mapują i przechowują wskaźniki do utworzonych sprajtów. Z kolei każdy obiekt otrzymuje potrzebny wskaźnik, któremu jedynie przekazuje informację, w którym miejscu sprajt narysować. Każdy sprajt tworzony jest tylko raz oraz wyłącznie wtedy, gdy pojawia się na danej Planszy.

Wyjątkami są te rodzaje sprajtów, które są niezbędne przez cały czas trwania gry:

- Bonusy, które mogą pojawić się w każdej chwili, ich kształt jest ściśle związany z typem
- Pociski Gracza, ich rodzaj zmienia się zależnie od zebranej mocy w czasie gry, każdy z nich może być potrzebny w dowolnym momencie.

Referencje do obiektów przekazywane są w formie **wspólnego wskaźnika**. Jest to bardzo dobre rozwiązanie, które eliminuje wiele potencjalnych błędów. Nie musimy martwić się tym, czy któryś obiekt przypadkiem usunie sprajt na wskutek błędów w kodzie lub przez magiczne cechy destruktorów. Sprajt jest automatycznie kasowany gdy żaden obiekt na niego nie wskazuje - czyli dopóki nie zostanie usunięty obiekt klasy Resource.

4. Jednoczesne iterowanie i usuwanie z kolekcji



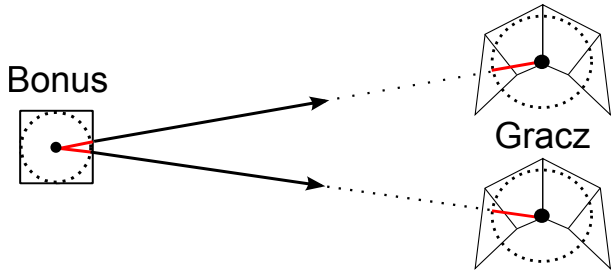
Sprawdzanie kolizji odbywa się wg wielokrotnej iteracji. Np. każdy obiekt pocisku sprawdza czy występuje kolizja z każdym możliwym obiektem.

- Jeżeli kolizja nastąpiła:
  - pocisk zostaje usunięty, a wrogi obiekt otrzymuje odpowiednie obrażenia.
  - Jeżeli życia wroga tym samym spadło do zera, on również zostaje usunięty
- Jeżeli kolizja nie nastąpiła, pociski dalej posuwa się swojej drogi

Jednakże może dojść do sytuacji w której usunięty pocisk lub wróg jest ostatnim z kolejki. Usunięcie odbywa się w trakcie pojedynczej iteracji, a po jej zakończeniu iterator jest inkrementowany. Po usunięciu ostatniego elementu iterator wskazuje na koniec kolejki, a po wyjściu z zakończeniu iteracji mechanizm chce by wskazał jeszcze dalej, bo powoduje błąd.

Rozwiązaliśmy to przez dodanie dodatkowego warunku podczas iteracji, mianowicie, pod koniec iteracja jest sprawdzany warunek, czy iteracja jest możliwa (czyli czy nie został usunięty ostatni z kolejki). Wymaga to 2 razy więcej operacji typu IF, jednakże wygląda na to, że nie da się tego uniknąć.

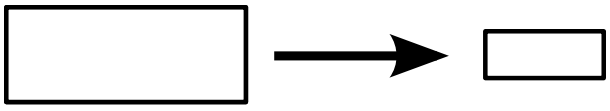
5. Naprowadzanie bonusów



Niektóre z obiektów są naprowadzane, np. gdy Gracz zbliży się na odległość $\frac{1}{5}$ wysokości Planszy, wszystkie istniejące bonusy zaczynają zbliżać się ku niemu. Zrealizowanie tego zadania wymagało napisania dodatkowej funkcji, przyjmującej jako argumenty pozycje Gracza oraz Bonusu, oraz funkcja ta musi być wywoływana do momentu zderzenia obojga.

Największą trudnością było napisanie jej tak, by zmiany w kodzie były jak najmniejsze. Jeśli Gracz stoi w miejscu, obiekt porusza się po linii prostej, więc na ten typ drogi się zdecydowaliśmy. Przy każdej aktualizacji, sprawdzane są obecne pozycje tych dwóch obiektów, wyliczany kąt pomiędzy nimi i ustawiana jest nowa droga. Żeby naprowadzania działało płynnie, za każdym razem punkt początkowy drogi był ustawiany jako aktualna pozycja bonusu, a dystans resetowany.

6. Niedoskonałość skalowania



W ekranie początkowym aktualnie wybrany przycisk pulsuje. Za pomocą przekształcenia afinicznego zwiększa i zmniejsza o zadaną wartość w czasie. Jednakże, po kilku minutach, przycisk stawał się znacznie mniejszy od pierwotnej wersji. Wynikało to z niedoskonałości obliczeń zmiennoprzecinkowych.
Rozwiązaliśmy to resetując skalowanie ($scale = 1.0$) w momencie gdy przycisk powinien być w swoim pierwotnym rozmiarze.

Podział pracy

1. Buchała Bartłomiej

- Interfejs graficzny i wyświetlanie danych
- Ruch gracza po planszy
- Strzelanie pociskami przez gracza
- Implementacja wzorców pocisków gracza
- Wystrzelenie bomby
- Narysowanie sprajtów pocisków gracza i bomby
- Umożliwienie pociskom ruchu po wybranych torach
- Usuwanie pocisków z pamięci gdy znajdą się poza planszą

2. Forczmański Mateusz

- Inicjalizacja urządzenia graficznego Direct3D 9
- Trajektorie obiektów i wyliczanie przesunięcia
- Przekształcenia afiniczne sprajtów i trajektorii
- Synchronizacja sprajtów z obiektami gry
- Wzory pocisków wrogów: kształt linii, elipsy i spirali
- Wczytywanie planszy gry z pliku XML
- Zarządzenie zasobami sprajtów
- Utworzenie fabryk obiektów

3. Motyka Marek

- Utworzenie hitboxa, jego typów oraz kształtów
- Obsługa kolizji między hitboxami
- Wykrywanie oraz zwiększanie otarć między obiektami
- Utworzenie i obsługa wszystkich bonusów
- Realizacja zmian po zderzeniu z bonusem
- Przyciąganie bonusów ku graczowi
- Usuwanie bonusów z pamięci

4. Wudecki Wojciech

- Narysowanie i napisanie ekranu powitalnego
- Utworzenie klasy nadrzędnej Playfield jako miejsca, gdzie mogą być wyświetlanie elementy gry
- Utworzenie wrogów, ich klasy i sprajtów
- Realizacja zależności pomiędzy wrogami, a ich wzorami pocisków
- Strzelanie pociskami przez wrogów
- Generowanie bonusów przez wrogów
- Narysowanie tła