Błażej Ejzak
Jakub Woźniak
Marcin Kowalczyk
Piotr Kowalski

Konfiguracja środowiska:

Platforma testowa: Visual Studio 22 oraz Visual Studio 19
Biblioteki zewnętrzne wymagające ściągnięcia: box2d; SFML

Preferowany sposób konfiguracji box2d:

1. Pobrać najnowszą wersje repozytorium z https://github.com/erincatto/box2d

2. W folderze box2d (z kopią repozytorium uruchomić w konsoli skrypt build.sh)

3. Dodać pliki nagłówkowe oraz pliki dll do konfiguracji projektu

Pozostałe możliwości opisane na GitHub'ie z repozytorium

Konfiguracja SFML:

1.Pobieramy wersję 64-bit z https://www.sfml-dev.org/download/sfml/2.5.1/

2. Z pobranego folderu kopiujemy katalogi lib oraz include do miejsca naszego pliku sln

3. Po kompilacji projektu (tryb release oraz debug) z folderu bin kopiujemy pliki dll. Do folderu x64\Release wklejamy pliki nie oznaczone symbolem -d, do x64\debug resztę. W obu folderach umieszczamy openall32.dll

4.We właściwościach projektu ustawiamy wcześniej dodane dodatkowe ścieżki plików nagłówkowych, dodatkowe katalogi plików lib, dodatkowe pliki lib

J**ak uruchomić gre:**

Gra uruchamia się automatycznie po uruchomieniu debugowania pliku main.cpp aby gra działała poprawnie trzeba mieć zainstalowane odpowiednie biblioteki opisane powyżej oraz trzeba w odpowiednim miejscu umieścić folder ,,animations,, zawierający animacje aby uruchomić grę należy także dodać do projektu w VS wszystkie pliki z folderu “BasicGraphicsUtilites” oraz pliki ,”object.h/cpp i contactListener.h” z folderu “box2d_code”.

W grę gra się za pomocą WASD oraz myszki, przyciski 1 2 3 służą do zmiany borni. WSAD służy do poruszania się, myszka do atakowania(można wykonać jeden atak na turę). Tury trwają 30s a po każdej turze następuje pauza aby gracze mogli zamienić się miejscami (jest to gra typu hotseat) jeśli dany gra jest już gotowy należy kliknąć przycisk Enter aby zaczęła się jego tura. Celem gry jest wyeliminowanie wszystkich robaków innych nacji kiedy cel ten zostanie osiągnięty gra zostanie zakończona. A aby zamknąć okno należy nacisnąć przycisk ESC(można to zrobić w dowolnym momencie gry ale progres nie zostanie zapisany) to wszystko ze strony instrukcji powodzenia i miłej rozgrywki.

Ogólny opis:

Reference

[Box2d]
Sekcja skupiająca się na implementacji związanej z biblioteką box2d, uwaga niektóre klasy będą miały opisane również w sekcji [SFML], z powodu synergii bibliotek w różnych klasach

Główne pliki zawierające implementacje klas używających biblioteki box2d:

Pliki korzystające biblioteki box2d i biblioteki standardowej c++:
1. objects.h / .cpp

Wstęp:
Największym problemem w tym przypadku było zrobienie odpowiedniego i przydatnego interfejsu klas który będzie wystarczająco uniwersalny by mógł służyć za podstawę do każdej klasy. Niestety definiowanie ciała w bibliotece box2d za pomocą specjalnej klasy b2BodyDef utrudniło to zadanie i ograniczyło nam niektóre możliwości i musieliśmy poprzestać na pewnej stałej charakterystyce każdej z klas. Używanie klasy string w późniejszym etapie również okazało się skomplikowane z powodu, że większość obiektów była innego typu i posiadała jedynie wskaźnik na klasę związaną z odpowiadającą za fizyczny model ciała. Udało się na szczęście to zrealizować zachowując bezpieczność i prywatność klasy dzięki zaprzyjaźnieniu funkcji ze sobą.

Opis działania:

Klasa Model
Klasą podstawowa dla pozostałych dwóch klas, jednym z jej atrybutów jest wskaźnik na ciało stworzone w świecie box2d, lecz wskaźnik pozostaje pusty ponieważ funkcja nie ma konstruktora, lecz ma destruktor który niszczy ciało powstałe w świecie boxa. Posiada też wszystkie potrzebne settery i gettery.

Klasa DynamicModel : public Model
Klasa definiująca konstruktor pozwalający stworzyć dynamiczny obiekt w box2d (tzn. działający zgodnie z prawami fizyki) oraz dodającym na niego fixture (czyli kształ, masę, tarcie i inne fizyczne wartości na podstawie przekazanych wierzchołków)

Klasa StaticModel : public Model
Klasa definiująca konstruktor pozwalający stworzyć statyczny obiekt w box2d czyli obiekt który nie porusza się i nie działają na niego prawa fizyki, “posiada nieskończoną masę”. Definiowany jest podobnie jak w DynamicModel za pomocą fixtury, jedynie fixtura to kształ “łańcuchowy”, jest to fixtura która można tylko przypisać do ciał statycznych, za to jej zaletą jest że może mieć bardzo duże wymiary co w przypadku ciał dynamicznych może być ograniczeniem. Klasa głównie ma za zadanie dobre zdefiniowanie podłoża i mapy.

Klasa Spring
Klasa umożliwiająca w łatwy sposób przypisanie danego dynamicznego ciałą do myszki i poruszanie tym ciałem zgodnie z ruchami myszki. Funkcja była potrzeba do debugowania, nie została użyta w ostatecznej realizacji.

Pliki silnie współpracujące z biblioteką box2d:
1. worm.h / .cpp
2. bullet.cpp
3. wormContact.cpp

Powyższe pliki zawierają implementacje klas (wszystkie zdeklarowane w worm.h): Worm (worm.cpp, wormContact.cpp), Bullet (bullet.cpp), Baseball (bullet.cpp), Granade (bullet.cpp)

Klasa Worm : public GR::DynamicAnimatedObject

Wstęp:

Dzięki dziedziczeniu klasa posiada już wskaźnik na obiekt odpowiadający za fizykę jak i sama w sobie jest obiektem odpowiadającym za wyświetlanie. Głównym jej celem było zdefiniowanie jej metod odpowiadających z płynne poruszanie się w świecie oraz za przechowywanie aktualnie posiadanej broni, a także za odpowiednie wykrywanie kolizji oraz współpracy między klasami kolidującymi, co okazało się jednym z większych wyzwań.

Opis działania:

[shot() - strzelanie]
Przy strzelaniu tworzymy obiekt klasy Bullet lub pochodny i musimy obsługiwać jego kolizje. Dzięki zaprzyjaźnieniu tych klas klasa Worm może efektywnie obsługiwać kolizje oraz zachowaliśmy bezpieczeństwo atrybutów klasy.

[funkcja update - fizyka]
Generalnym zadaniem funkcji update jest zaktualizowanie pozycji do narysowania robaka ale przy okazji wywołuje ona funkcje odpowiedzialne za kolizje i tzw. cooldowny)

[sprawdzanie kolizji - plik wormContact]

contactHandler()

Z powodu na dostęp do kolizji z box2d jedynie z poziomy klas już zdefiniowanych w box2d wywołujemy listę kolizji jednego ciała i sprawdzamy parametry drugiego (istnieje drugie rozwiązanie z userData ale na poziomie klasy b2Body nie jest ono dostępne w aktualnej wersji biblioteki, pomimo że jest zawarta w dokumentacji, jest jedynie dostępne na poziomie definicji ciała, co okazało się nam zniszczyć całą hierarchię klas w programie więc musieliśmy zrobić to w inny sposób.). Dzięki naszemu rozwiązaniu mamy dostęp do jednej z naszych klas w tym przypadku Worm. Z tego powodu musimy opierając się na parametrach fizycznych drugiego ciała które są dostępne na klasie b2Body**.** W przypadku dotknięcie statycznego ciała pod odpowiednim kątem (od góry), ustawiamy możliwość wskoczenia 2 razy naszemu rodakowi. W przypadku jak dotknie nas inny obiekt dynamiczny i jest ustawiony jako “bullet” to zadajemy naszemu rodakowi obrażenia, identyfikując rodzaj pocisku za pomocą jego fizycznych parametrów, oraz nadajemy naszemu rodakowi odpowiednie siły. Ustawiamy też odpowiednie cooldowny. Jest to ciekawe rozwiązanie ponieważ udało nam się w ten sposób ominąć brak referencji na obiekt drugiej klasy i dokładne sprawdzenie jakiego typu jest ten obiekt.

bulletContactHandler()

Funkcja odpowiadająca jedynie za obsługę pocisku, jest wywoływane podczas gdy pocisk istnieje i odpowiedzialna za jego destrukcję. W takim przypadku dalej nie posiadamy referencji na obiekt którego dotknie pocisk ale nie potrzebujemy tego mieć ponieważ wszystkie wrażliwe elementy (Worm) już same wiedza jak zareagować na odpowiednią kolizje za pomoca własnych contactHandlerów. Pocisk po dotknięciu jakiegokolwiek ciała sam się niszczy, dzięki sprytnemu wskaźnikowi nie musimy martwić się o dealokację pamięci. Jednym z problemów tutaj był brak możliwości usunięcia ciałą podczas wykrycia kolizji ponieważ, biblioteka liczyła nasze pozycje ciał, więc trzeba było to zrealizować za pomocą wskaźników mówiących czy ciało ma zostać zniszczone w następnym obrocie pętli czy nie.

[Pociski - Bullet, Baseball, Grenade]

Klasa Bullet jest wyjściową klasą dla pozostałych pocisków. Definiuje ona kluczowy interfejs używany do rysowania oraz nadawania parametrów fizycznych obiektowi. Ponieważ był nam potrzebny uniwersalny wskaźnik który będzie tworzył odpowiednie różne pociski, virtualność pomogła nam rozwiązać problem. Przykładowo w przypadku Baseball'a nie chcemy rysować naszego pocisku bo on nie istnieje oraz nie chcemy mu nadawać prędkości w nieskończoność jak to robimy w przypadku zwyczajnych pocisków, więc wystarczyło napisać funkcję update oraz stworzyć odpowiedni konstruktor.
W przypadku Grenade sytuacja miała się podobnie tym razem chcieliśmy rysować pocisk ale musieliśmy mu nadać inne prędkości i tak samo w tym przypadku odpowiedni konstruktor i nadpisanie funkcji update się sprawdziło. Dodatkowo umożliwia to nam dalszy rozwój pocisków i w łatwy sposób dodawanie ich, przykładowo możemy dodać bardziej obszarowy wybuch dla naszego granatu bądź zrobić z niego granat odłamkowy..

[Pliki Worm]

Pliki te zawierają w sobie klasy postaci, broni, pocisków itd. . Dodatkowo zawierają metody umożliwiające bindowanie klawiszy do odpowiednich metod, takich jak ruch, skakanie, strzelanie, rzucanie granatem itp. Znajduje się w nich także część logiki gry odpowiadająca za podstawowe mechanizmy gry takie jak śmierć robaka/zmiana broni/ilość użyć broni na turę itd.

[Window]

Plik ten odpowiada za prawidłowe działanie okna gry, ilość klatek w animacjach, rysowanie i updatowanie obiektów.

[Game]

Logika gry taka jak tury, warunki zwycięstwa przerwy pomiędzy turami itd.

[Dynamic/Static Object]

Pliki zajmujące się wszelakimi obiektami w grze, dynamicznymi - takimi na które oddziaływuje fizyka, statycznymi obiektami tak zwanego tła,

oraz obiektami statycznymi fizycznymi - używanych np do tworzenia tereny czyli obiektu o nieskończenie dużej masie na który nie działa grawitacja.

[Animacje]

Animacje zostały wykonane w programie inkscape, animacje były elementem, którego wykonanie pochłaniało bardzo dużo czasu a jednocześnie bez nich ciężko było rozwijać dalej grę. Ostatecznie część animacji nie została zaimplementowana ze względu na zmianę koncepcji lub rezygnacji z implementacji niektórych funkcjonalność, z powodu ograniczenia czasowego projektu. Animacje były projektowane w taki sposób aby jak najlepiej ułatwić pracę z kodem tak aby to animacje dostosowywały się do możliwości naszego silnika a nie silnik do dostępnych animacji.

[poruszanie się]

Obiekty poruszają się dzięki ustawianiu natychmiastowej ich prędkości w odpowiednim kierunku. (Rozważaliśmy wariant nadania jemu prędkości w zależności w sposób sił fizycznych lecz ten sposób nie był idealny z powodu na ciągłe przyspieszanie ciała oraz jego niestałą prędkość) Problemem w tym przypadku było ograniczenie tej prędkości (niezależnie od wariantu z siłami czy ustawieniem prędkości) w zależności od terenu ponieważ funkcja była wywoływane w każdej chwili kiedy przycisk był przyciśnięty, co sprawiało że worm był wstanie pokonać prawie każdą lekko pochyłą ścianę ze zbyt szybką prędkością, z powodu fizycznego rozkładu sił. Nie mogliśmy również ograniczyć jego prędkości po prostu bo wtedy jego skok też byłby ograniczony i siły by się nie sumowały w odpowiedni sposób. Rozwiązaniem było sprawdzanie kolizji z ziemią (wormContact.cpp - w późniejszej części) i ustawianie odpowiedniej flagi, który również pomagał nam przy pozwalaniu robakowi skakać lub nie.