Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Przewijana gra akcji 2D w języku JavaScript z wykorzystaniem WebGL

Szymon Lewandowski
Nr indeksu: 208788

Praca magisterska napisana
pod kierunkiem
dr Marcina Kika

Politechnika
Wrocławska

Wrocław 2016

Spis treści

Rozdział 1

Wstęp

Cel pracy to stworzenie dwuwymiarowej gry komputerowej, która będzie mogła być umieszczona na stronie internetowej i odtworzona w przeglądarce. Zakresem pracy jest stworzenie gry z nastepującymi elementami:

- logika gry
- system wykrywania kolizji
- grafika i muzyka
- możliwość otworzenia gry w dowolnej nowoczesnej przeglądarce

Istnieje już wiele gier tego rodzaju, pierwszą z nich była Space Invaders z 1978 roku, a bardziej współczesne to na przykład takie tytuły jak Jets'N'Guns, Crimzon Clover albo najnowsze gry z serii Gradius.

W skład pracy wchodzi sześć rozdziałów:

- Rozdział pierwszy analizujący samą grę oraz problemy do rozwiązania
- Rozdział drugi opisujący logikę gry i sposoby obsługi obiektów w grze
- Rozdział trzeci przedstawiający różne systemy wykrywania kolizji i system wykrywania kolizji użyty w grze
- Rozdział czwarty mówiący o obsłudze grafiki w grze
- Rozdział piąty opisujący obsługę muzyki i efektów dźwiękowych w grze
- Rozdział szósty podsumowujący uzyskane wyniki

Rozdział 2

Analiza zagadnienia

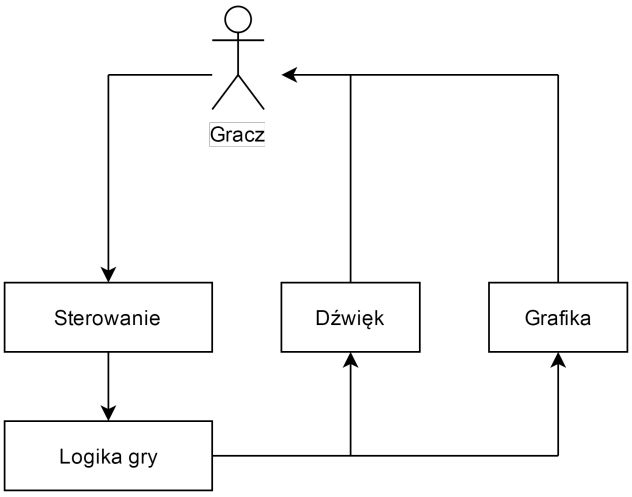
W tym rozdziale przedstawię założenia i oczekiwane efekty dla różnych aspektów gry.

Przede wszystkim, gra ma być skonstruowana tak aby działała na wszystkich nowszych i najczęściej używanych przeglądarkach, takich jak Google Chrome, Mozilla Firefox, Internet Explorer i Opera. Gra powinna także odpowiednio wyglądać i zachowywać się zgodnie z tym czego można by sie po niej spodziewać, na przykład jeśli dwa obiekty wyglądają jakby powinny się zderzyć, to rzeczywiście powinno zajść zderzenie.

Główny element gry to silnik gry. Zajmuje się on wykonywaniem odpowiednich funkcji na obiektach w grze. Zazwyczaj jest to pętla działająca tak długo jak działa gra, używająca timer-ów do odpowiednio częstego wywoływania funkcji aktualizujących obiekty w grze. W grze ważny jest także silnik graficzny, odpowiadający za wyświetlanie obiektów i efektów na ekranie. Jednym z problemów do rozwiązania jest odpowiednie zsynchronizowanie obu tych elementów z czasem i sobą nawzajem, aby uniknąć sytuacji takich jak:

- silnik gry zabierający cały czas procesora dla siebie, co powoduje rzadkie odwśieżanie zawartości ekranu i niegrywalność
- silnik graficzny zabierający dużo czasu na obliczenia, przez co gra spowalnia i jest dużo mniej grywalna
- obliczenia silnika graficznego i silnika gry pokrywające się, przez co niektóre obiekty zostają narysowane przed, a niektóre po aktualizacji stanu, co powoduje błędne wyświetanie pozycji niektórych obiektów i może wprowadzić gracza w błąd

Innym ważnym problemem jest odpowiednie wykrywanie kolizji w grze, tak aby nie było sytuacji, w której pokrywające się obiekty nie zderzają się ze sobą, albo obiekty będące daleko od siebie mają ze sobą kolizję. Ze względu na to, że to jeden z najważniejszych elemtentów w tego typu grze, potrzebny był algorytm, który będzie mógł szybko i dokładnie to sprawdzić. Zostało to rozwiązane przez podzielenie wykrywania kolizji na fazy oraz zastosowanie odpowiednich algorytmów wykrywania kolizji opisanych w rozdziale 4.



Rysunek 2.1: diagram przedstawiający interakcję gracza z grą

Interakcja gracza z grą jest bardzo prosta.
Gracz używa klawiatury do sterowania grą.
Jako odpowiedź otrzymuje obraz i dźwięk informujące o tym co aktualnie dzieje się w grze.
Do każdego z tych zadań istnieje odpowiednia klasa zajmująca się nim.

Klasy odpowiedzialne za sterowanie oraz logikę gry zostały opisane w rozdziale 3.

Sposób wyświetlania na ekranie aktualnej sytuacji, na co składa się narysowanie wszystkich obiektów w grze oraz dodanie efektów graficznych, został opisany w rozdziale 5.

Dźwięk został opisany w rozdziale 6.

Jednym z mniejszych problemów jest dopasowanie do siebie wszystkich elementów gry, czyli przygotowanie gry tak, by podobała się odbiorcy. Przeciwnicy powinni być odpowiednio dopasowani do poziomu zaawansowania gracza, a także nie powinno być zbyt prosto lub zbyt łatwo.

Rozdział 3

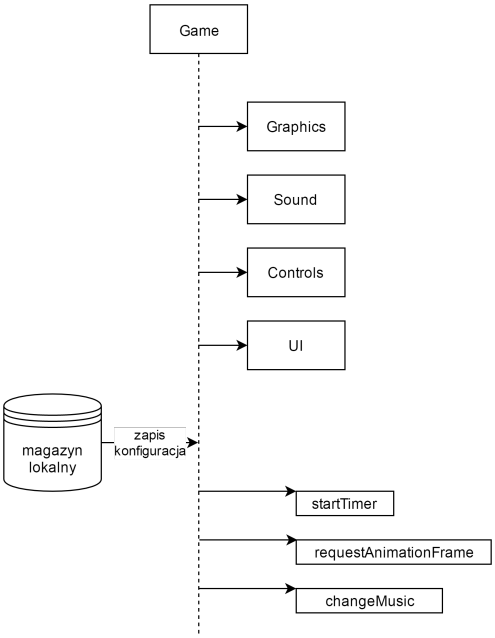
Logika gry

Inicjalizacja gry

W tym rozdziale omówię w jaki sposób działa silnik gry i mechanizmy umożliwiające poprawne działanie wszystkich jej aspektów.

Głównym obiektem w grze jest obiekt klasy Game, inicjalizujący wszystkie inne obiekty oraz synchronizujący ich działanie.

Inicjalizacja zaczyna się od utworzenia obiektów odpowiedzialnych za poszczególne elementy gry, czyli logiki, grafiki, muzyki i sterowania, opisanych w dalszej części pracy.



Rysunek 3.1: diagram przedstawiający interakcję gracza z grą

Następny krok to wczytanie ustawień, takich jak głośność i ustawienia graficzne, a także stanu gry - odblokowanych poziomów, zdobytych punktów, uzyskanych osiągnięć i kilka innych statystyk. Ustawienia te są wczytywane z udostępnianego przez przeglądarkę magazynu lokalnego, dostępnego w języku HTML5. Magazyn lokalny jest podobny do używanych na większości stron ciasteczek, ale jest większy i łatwiejszy w użyciu, ponieważ jest obiektem działającym jak słownik z języka JavaScript, zawierającym tylko klucze i ciągi znaków do nich przypisane. Dzięki temu można łatwo zapisać w nim obiekt z danymi po przetłumaczeniu go na format JSON. Przy wczytaniu danej typu string z magazynu lokalnego, wystarczy zamienić dane w formie JSON-a na zwykły obiekt JavaScript.

Dane przechowywane w magazynie lokalnym sa przypisane do strony i są w nim przechowywane dopóki ich nie usuniemy lub użytkownik nie wyczyści magazynu, inaczej niż w ciasteczkach, gdzie trzeba było podać datę przydatności, po której dany obiekt był usuwany. Ciasteczka mogły też pomieścić dużo mniej danych.

Istnieje odmiana magazynu lokalnego, magazyn sesji, w którym dane są obecne dopóki nie zamkniemy danej zakładki.

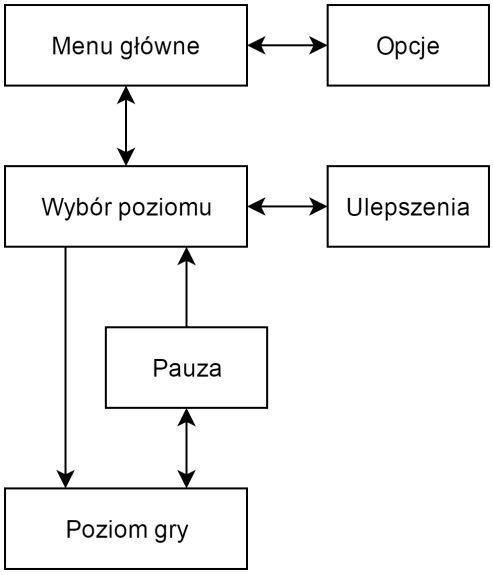
W przypadku braku danych w magazynie lokalnym używane są dane domyślne.

Po wczytaniu danych, następuje rozpoczęcie gry. Ustawiany jest timer, co 20 ms, czyli 50 razy na sekundę, uruchamiający funkcję obliczającą jeden cykl silnika gry. Następnie do przeglądarki jest wysyłane żądanie uruchomienia funkcji renderującej. Argument funkcji requestAnimationFrame jest uruchamiany wtedy przed następnym odświeżeniem rysowanych elementów (takich jak element Canvas, na którym jest umieszczany obraz z gry). Na koniec uruchamiana jest muzyka z głównego menu gry.

Po skończeniu inicjalizacji gry, okresowo wywołują się funkcje odpowiedzialne za logikę gry, opisaną poniżej, oraz za grafikę, opisaną w rozdziale 5.

Menu gry

Pierwsze co gracz widzi po włączeniu gry, to ekran głównego menu, zawierający opcje ”Graj” oraz ”Opcje”. Sterowanie w menu to klawisze strzałek, enter i escape, służące do zmiany podświetlonej opcji, wejścia do daneg menu oraz cofnięcia się z niego. Wybranie pozycji ”Opcje” przeniesie nas do menu z różnymi ustawieniami gry, takimi jak ilość efektów, głośność czy też możliwość zresetowania postępu w grze. Druga opcja natomiast przenosi nas do menu wyboru poziomu, będzie tam też można za zdobyte w grze punkty ulepszyć statek gracza. Na początku odblokowany jest tylko jeden poziom, ale z postępami w grze gracz odblokowuje kolejne poziomy, a także możliwości rozwoju statku. Po przejściu do któregoś z poziomów zaczyna się właściwa gra, na dole ekranu pojawia się statek gracza, a z góry zaczynają po chwili nadlatywać przeciwnicy. Wciśnięcie spacji powoduje strzelanie z głównych działek, a gdy trafimy w przeciwnika wystarczająco dużo razy, zastaje on pokonany a gracz dostaje punkty. Aby przejść poziom wystarczy dolecieć do jego końca, omijając lub pokonując spotkanych po drodze przeciwników. Aby zatrzymać grę należy wcisnąć przycisk escape. Menu pauzy pozwala przejść do panelu opcji, wrócić do gry lub skończyć poziom.



Rysunek 3.2: Przejścia między menu

Główny silnik gry

Główny silnik gry działa poprzez aktualizowanie stanu gry co każde 20 milisekund. Jeden taki cykl polega zazwyczaj na uaktualnieniu wszystkich aktywnych obiektów. Gdy gracz jest w głównym menu gry, nie ma zbyt wiele obiektów do aktualizacji, dopiero gdy włączy się któryś z poziomów pojawiają się większe, bardziej skomplikowane obiekty. Na początku jest aktualizowany obiekt klasy UI, który zajmuje się obsługiwaniem aktualnego menu (poziom w grze to także rodzaj menu). Następny jest obiekt osługujący grafikę (tutaj aktualizacja polega zazwyczaj na wyczyszczeniu buforów), a potem wszystkie nowe dane do rysowania są przekazywane do obiektu obsługującego grafikę. Na końcu raz na dwie sekundy aktualne statystyki i opcje gry są zapisywane w magazynie oraz zostaje zwiększony licznik czasu.

Większość klas w projekcie ma funkcje ”update” oraz ”draw”, służące odpowiednio do aktualizacji oraz rysowania obiektu.

Niezależnie od aktualizacji działa sterowanie w grze. Jest ono przekazywane na dwa sposoby: jako tablica wciśniętych klawiszy przechowywana w odpowiednim obiekcie oraz zdarzenie wysyłane do aktualnego menu. To drugie służy tylko do sterowania w menu, i nie jest używane we właściwej grze poza opcją włączenia menu pauzy. Tablica klawiszy służy natomiast podczas aktualizacji w samej grze, i zawiera informację typu boolean dla każdego klawisza, mówiącą czy jest on wciśnięty.

Aktualizacja w trakcie poziomu

Aktualizacja w trakcie poziomu zaczyna się od aktualizacji obiektu danego poziomu, zazwyczaj powoduje to pojawienie się przeciwników oraz kończenie misji przy spełnieniu odpowiedniego warunku. Każdy poziom ma własna klasę reprezentującą go.
Każda klasa poziomu zawiera przypisane do niego:

- listę zawierającą dane o typie, czasie pojawienia się i zachowaniu przeciwników
- długość poziomu (gdy poziom się kończy, gracz wygrywa)
- informacje o specjalnym przeciwniku na końcu poziomu (lub jego braku)

Następny krok to aktualizacja wszystkich efektów, najczęściej przez zmianę ich pozycji. Efekty są opisane dokładniej w rozdziale 5.

Następnie aktualizują się przeciwnicy, czyli:

- zmieniają swoją pozycję
- używają danych im broni
- aktywują umiejętności specjalne

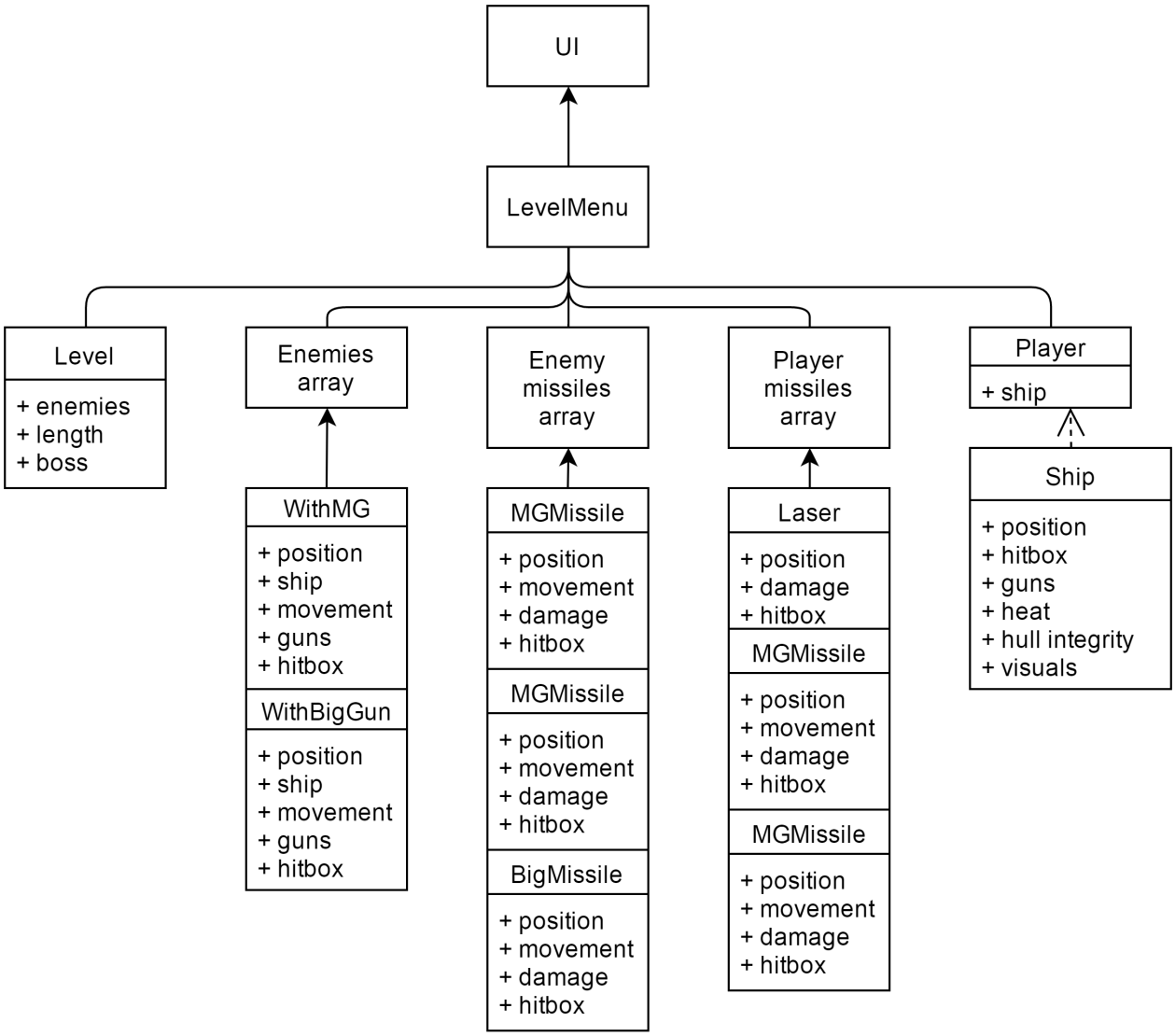
Każdy przeciwnik ma przypisane do siebie swoją pozycję, zachowanie, model statku oraz zazwyczaj broń lub bronie. Każda klasa typu przeciwnik ma także punkty uzyskane za pokonanie danego typu przeciwnika, jego maksymalną wytrzymałość oraz domyślny model używany przy wykrywaniu kolizji.

Kolejny krok aktualizuje pociski przeciwników, czyli zmienia ich położenie według zaprogramowanego ruchu, a także sprawdza czy nie zderzyły sie one z graczem. System kolizji jest opisany w rozdziale 4. Potem to samo jest powtarzane dla pocisków wystrzelonych przez gracza.

Obiekty pocisków zawierają zazwyczaj informacje o położeniu, obrażeniach, sposobie ruchu i modelu używanym do zderzeń.

Ostatnim krokiem jest aktualizacja statku gracza. Sprawdzane jest tam, czy gracz wcisnął klawisze służące do przemieszczania statku (strzałki). Na końcu aktualizowany jest statek gracza, w tym to czy gracz używał broni (klawisz spacji), czy chłodnica statku się nie przegrzała oraz czy statkowi nie skończyły sie punkty wytrzymałości, co wcześnie kończy grę. Obiekt typu Player jest jednym z bardziej skomplikowanych. Oprócz zwykłych danych takich jak wygląd, pozycja oraz model, zawiera także dokładniejsze informacje o broni. W przeciwieństwie do przeciwników jest też możliwość zmienić jego statek i broń, a na jego zachowanie oprócz samego sposobu poruszania się ma wpływ także sterowanie gracza. Ze względu na to że gracz widzi go cały czas jest jedną z najważniejszych części gry.

Gdy wszystkie kroki zostały wykonane, jeden cykl silnika gry się kończy i gra czeka na następny. W międzyczasie między aktualizacjami obraz jest także przerysowywany, co jest opisane w rozdziale 5.



Rysunek 3.3: przykład obiektu UI, pokazujący strukturę obiektów

Rozdział 4

System wykrywania kolizji

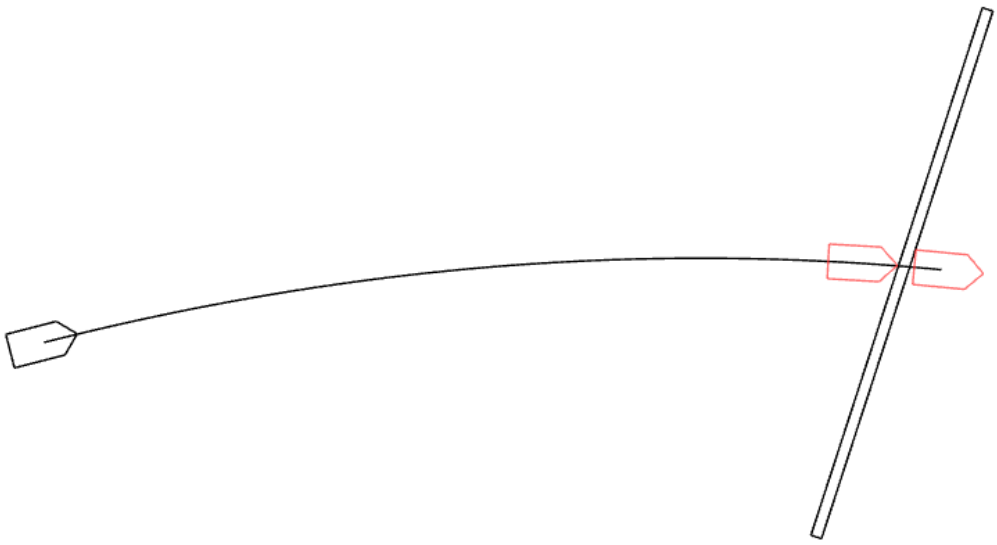
W tym rodziale opiszę jak działają różne systemy wykrywania kolizji i w jaki sposób niektóre z nich zostały użyte w projekcie.

Są dwa modele wykrywania kolizji:

- ciągły, w którym ustalamy dokładny moment i miejsce zderzenia
- dyskretny, w którym to czy obiekty się zderzyły sprawdzamy co jakiś czas

Wykrywanie kolizji w modelu ciągłym

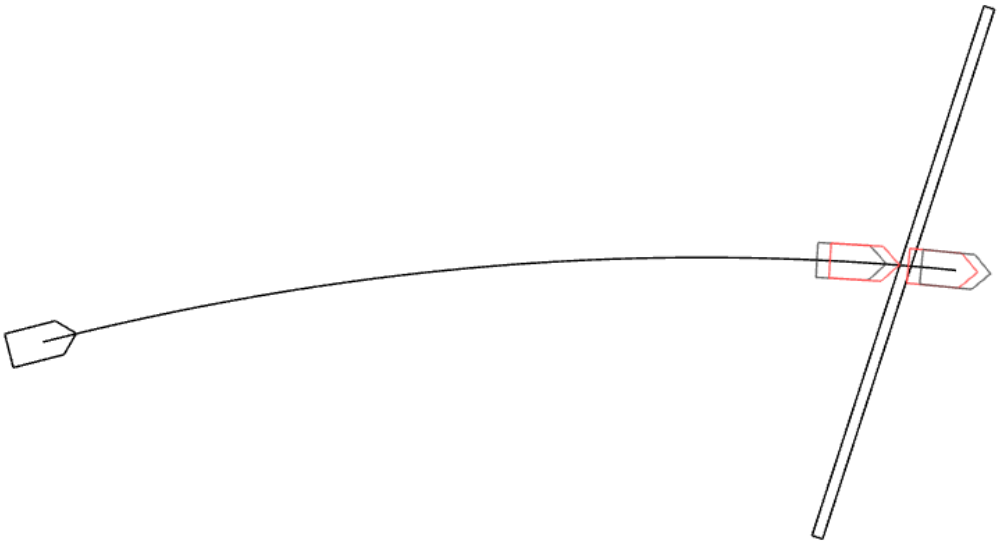
Ten model wykrywania kolizji jest zazwyczaj używany tam, gdzie jest wymagana bardzo duża dokładność w obliczaniu miejsca i czasu zderzenia. W obliczeniach związanych z tym modelem często używa się interpolacji i metod numerycznych, przez co jest zbyt czasochłonny aby użyć go w progamach działających w czasie rzeczywistym, jeśli trzeba takie obliczenia wykonać na wielu obiektach. Najczęściej używa się go w symulacjach i modelach fizycznych, a bardzo rzadko w grach. Często model taki musi uwzględniać informacje o parametrach fizycznych obiektów takich jak ich elastyczność, oraz to jak obiekt będzie się poruszał według zasad fizyki.



Rysunek 4.1: Wykrywanie kolizji w modelu ciągłym, punkt wejścia dla t=60.947, punkt wyjścia dla t=66.563

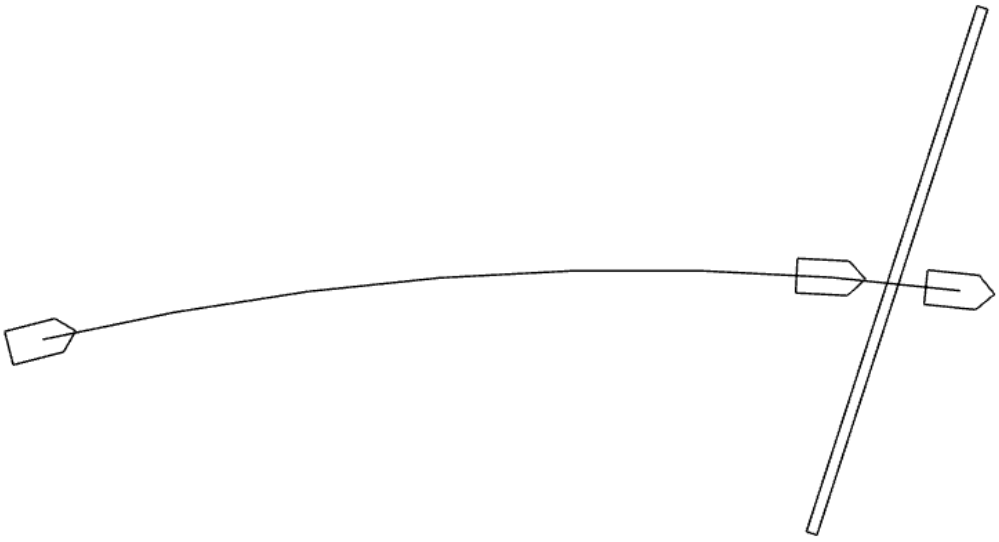
Wykrywanie kolizji w modelu dyskretnym

Ten model wykrywania kolizji jest używany wszędzie, gdzie nie jest wymagana taka dokładność jak przy trybie ciągłym. Polega on na sprawdzaniu czy obiekty kolidują co jakiś okres czasu, zazwyczaj co cykl obliczeń zmieniających położenie obiektów. Jest dużo prostszy i wymaga dużo mniej obliczeń, dlatego mozna go używać w czasie rzeczywistym na wielu obiektach naraz. Najczęściej jest on używany w grach albo prostych symulacjach, niewymagających dużej dokładności. Do algorytmów używających tego modelu nie trzeba też wysyłać tylu informacji, wystarczy podać mu listę modeli, na których chcemy sprawdzić, czy zaszły zderzenia.



Rysunek 4.2: Wykrywanie kolizji w trybie dyskretnym, zderzenie następuje między klatkami 61 i 66

Jego minusem jest jednak to że prędkości obiektów muszą być dopasowane do ich wielkości, jeśli mały obiekt porusza się bardzo szybko to zderzenie może nie zajść, ponieważ w jednym cyklu obliczeń obiekt będzie całkowicie przed, a w drugim całkowicie za przeszkodą.



Rysunek 4.3: Problem z wykrywaniem kolizji w trybie dyskretnym, w takim przypadku zderzenie między obiektami nie zostaje zarejestrowane

W moim projekcie zastosowałem model dyskretny, ponieważ nie jest tu wymagana taka duża dokładność i pasuje on do sposobu w jaki działa silnik gry (wykonywanie aktualizacji co pewien okres).

Wykrywanie kolizji dzieli się na dwie fazy:

- Broad phase
- Narrow phase

Broad Phase

W tej fazie wykrywa się jak najwięcej par obiektów, które na pewno się ze sobą nie zderzą, ponieważ np. są daleko od siebie. Obliczenia w tej fazie powinny być jak najprostsze, aby szybko i skutecznie odrzucić dużo par obiektów.

Algorytmy najczęściej używane podczas Broad phase dzielą się na te, które dzielą lub układają przestrzeń i te, które po prostu porównują każdy obiekt z każdym poprzez otoczenie go jakimś kształtem.
Algorytmy, które opiszę, to:

- AABB (Axis-Aligned Bounding Boxes)
- Bounding Boxes
- Bounding Circles
- Sweep and Prune
- Quadtree
- Octree
- Bounding Volume Hierarchy
- Bins

Czasami można w tej fazie użyć tymczasowej koherencji, czyli skorzystać z faktu, iż niektóre obiekty nie przesunęły się względem siebie od czasu ostatniego wykrywania kolizji, lub przesunęły się o względnie małą odległość, dzięki czemu od razu można je wykluczyć.

Bounding Volumes

Metody z grupy Bounding Volumes polegają na otoczeniu modelu jakimś prostym kształtem, i sprawdzaniu zderzeń tylko na tych kształtach. Są zazwyczej dość proste do zaimplementowania i bardzo dobre jeśli obiektów jest mało.

Space Partitioning

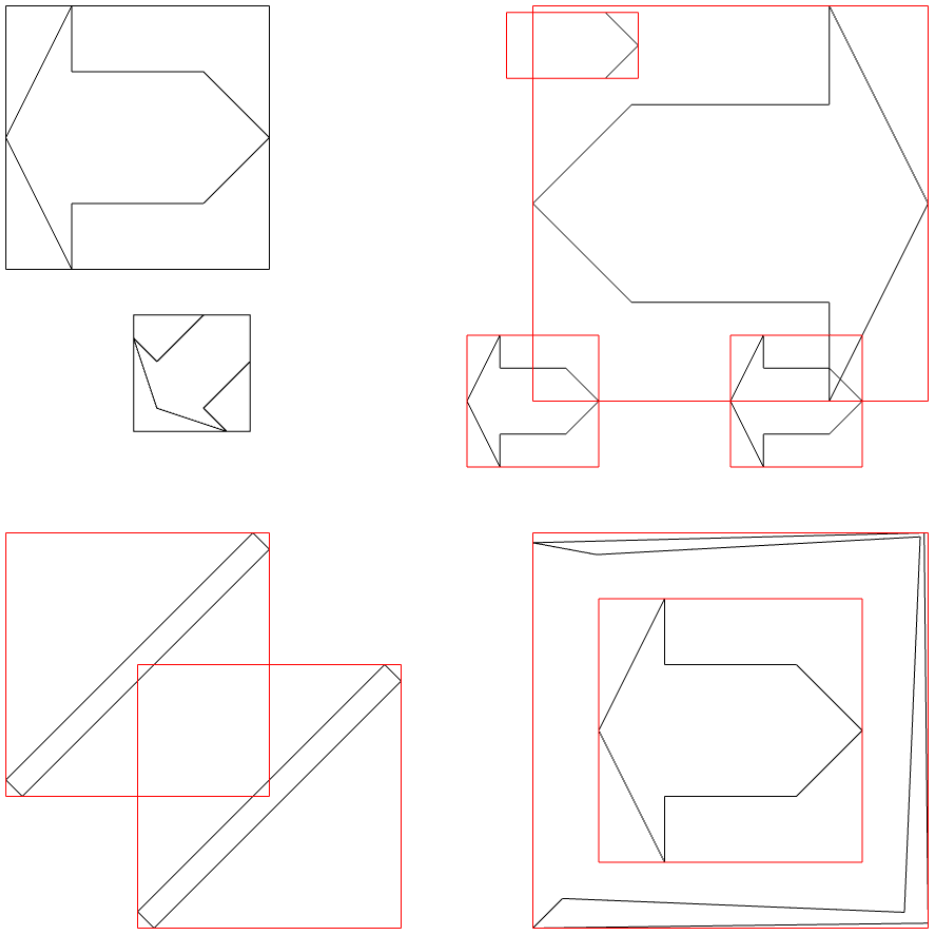
Druga grupa metod, wykorzystują one podział pola gry na obszary, zazwyczaj ułożone w strukturę drzewa. W każdym z pól każda para obiektów dla danego pola zostaje przetestowana czy została zderzona. Podział na obszary można zaimplementować na wiele sposobów.

AABB (Axis-Aligned Bounding Boxes)

Metoda AABB polega na otoczeniu każdego obiektu prostokątem i sprawdzeniu które prostokąty się pokrywają.

Zaletami tej metody są łatwość implementacji, a także dokładność dla obiektów przypominających kształtem prostokąty wyrównane do osi.

Wadami tej metody są czas działania, czyli $O(n^2)$ dla n elementów, oraz fakt że dla pewnych obiektów jest ona bardzo niedokładna. Dla obracających lub zmieniających rozmiar obiektów trzeba też zmienić rozmiar prostokąta, co czasami może wymagać dużej ilości obliczeń, jeśli model ma wiele wierzchołków.
Metody tej najlepiej używać, gdy obiektów jest niewiele i są one prostokątne (na przykład w platformówkach).



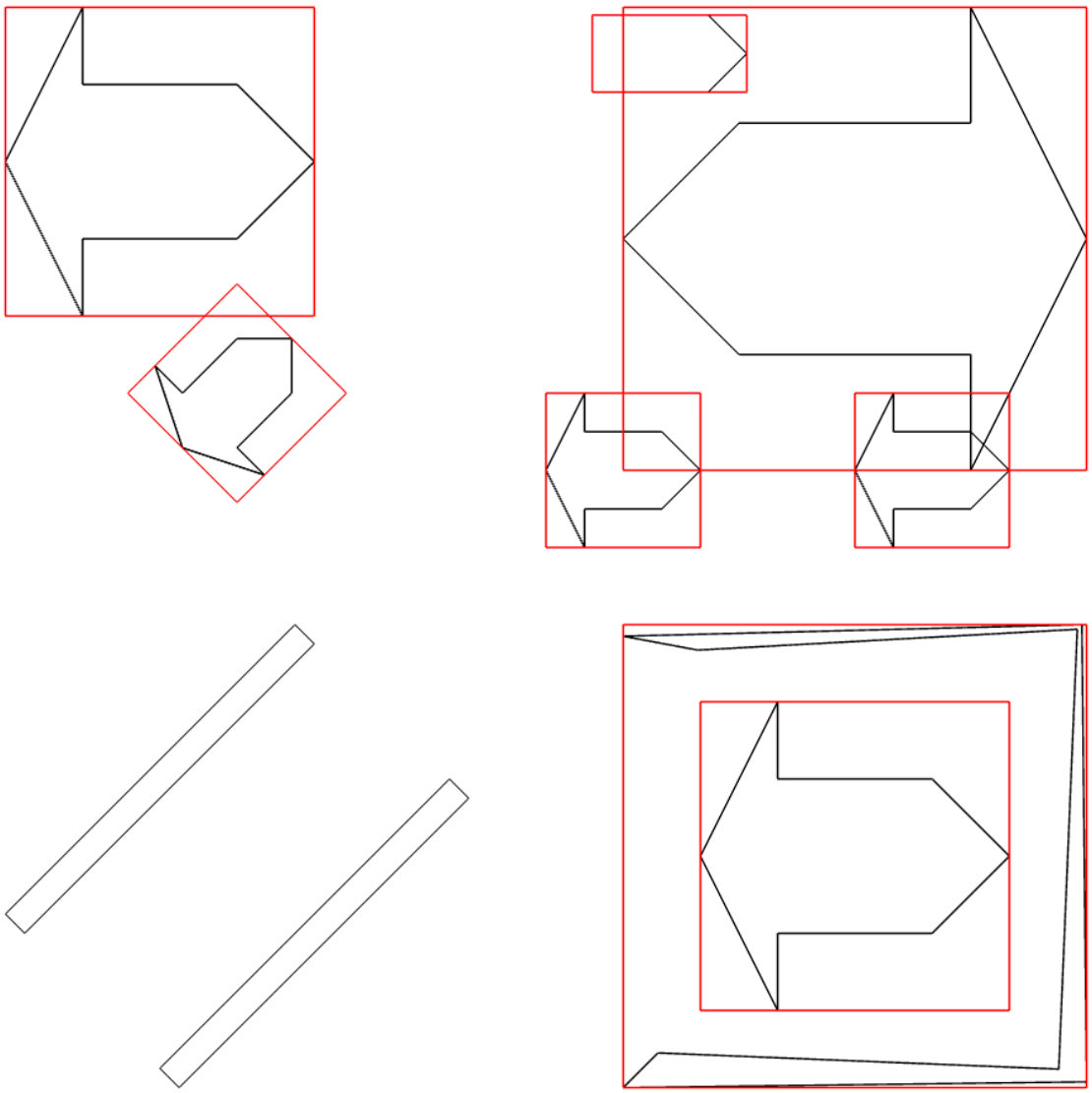
Rysunek 4.4: Przykład wykrywania kolizji metodą AABB, kolidujące obiekty są oznaczone na czerwono

Jak widać na obrazku, większość obiektów z czymś koliduje, ale pary obiektów będących daleko od siebie są odrzucone, co zdecydowanie zmniejsza liczbę obliczeń.

Bounding Boxes

Ta metoda polega na otoczeniu każdego obiektu prostokątem, który nie musi być wyrównany do osi, w innych aspektach jest podobna do poprzedniej. Jest ona odrobinę trudniejsza do zaimplementowania, ale dzięki możliwsoci obracania może być także dokładniejsza w niektórych przypadkach, a przy obrocie obiektu wystarczy obrócić odpowiadający mu prostokąt.

Wady tej metody to nieco większa złożoność obliczeń, ciągle przy $O(n^2)$ porównań.



Rysunek 4.5: Przykład wykrywania kolizji metodą Bounding Boxes

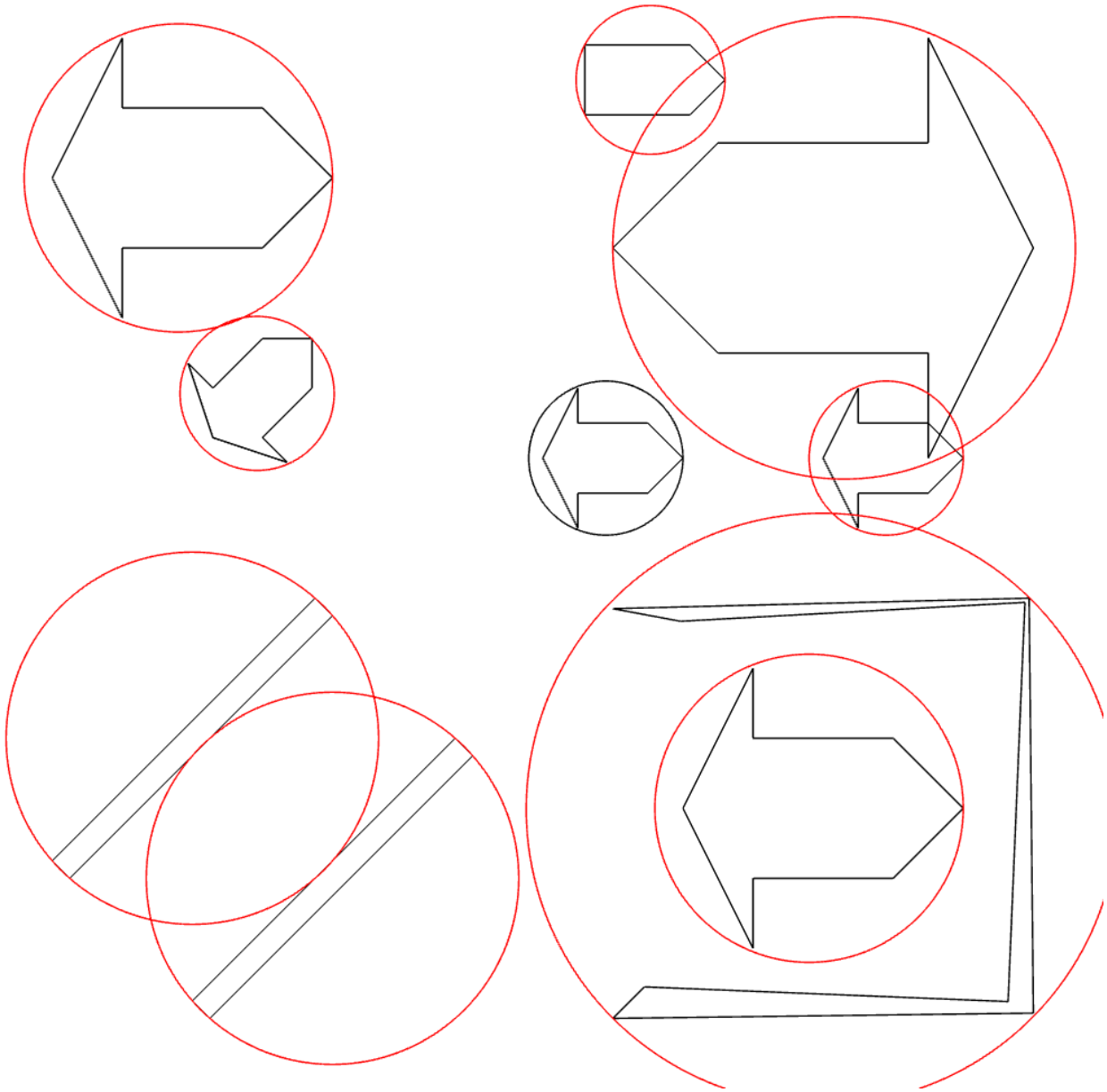
Patrząc na obrazek, w lewym dolnym rogu prostokąty odpowiadają teraz modelowi obiektu, ale za to w lewym górnym rogu widać, że tym razem program wykrył kolizję, której nie było w poprzednim przykładzie.

Bounding Circles

Ta metoda jest podobna do dwóch poprzednich, ale tym razem używamy koła do sprawdzania możliwych zderzeń.

Plusami tej metody są łatwość jej zaimplementowania i bardzo mała ilość wymaganych danych oraz obliczeń: wystarczy tylko obliczyć odległośc między dwoma obiektami i porównać z sumą ich promieni. Dodatkowo obrót obiektów i ich skalowanie nie wymusza obliczania danych na nowo, jesli używamy tej metody.

Wady tej metody to kiepska wydajność dla obiektów które są podłużne.

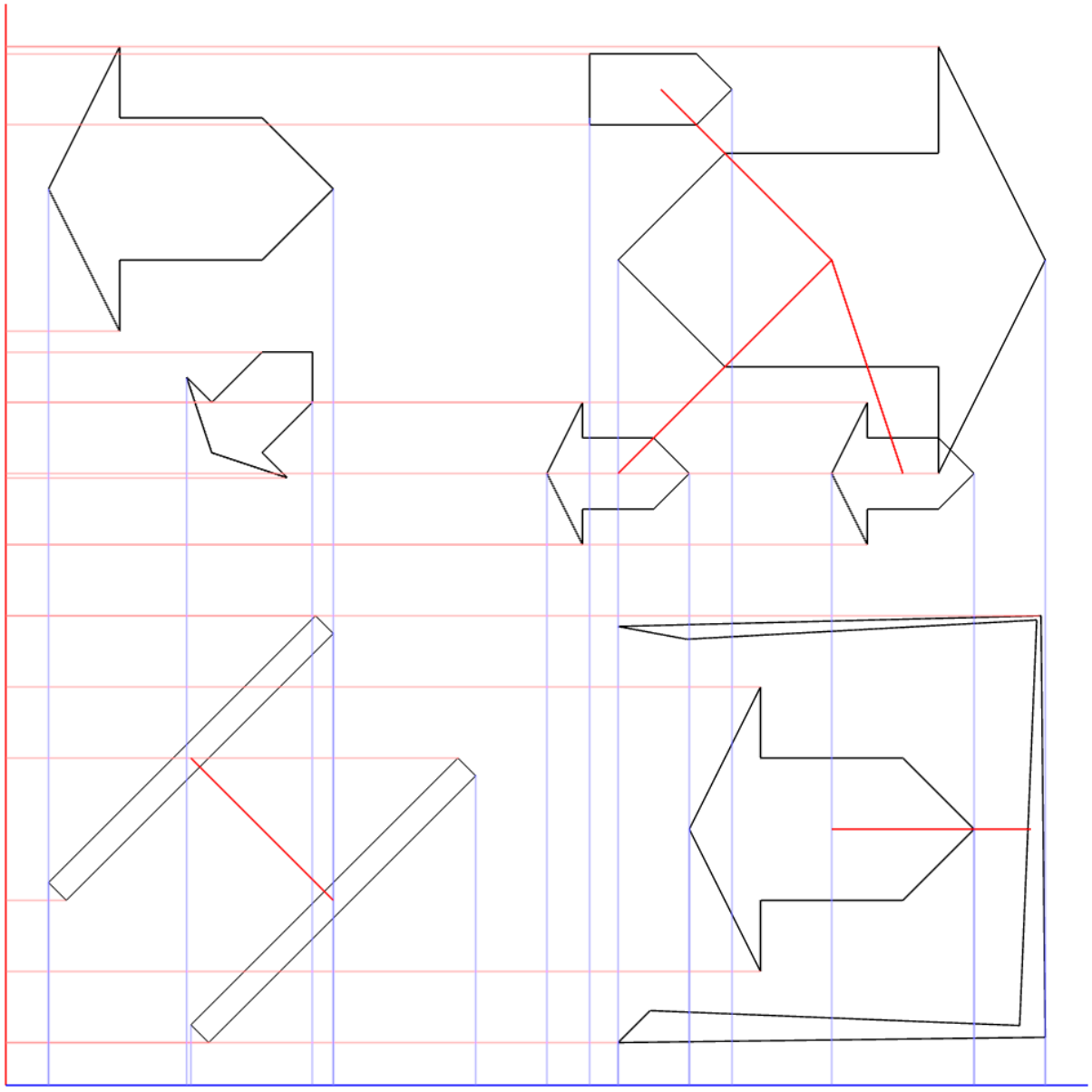


Rysunek 4.6: Przykład wykrywania kolizji metodą Bounding Circles

Sweep and Prune

Metoda polegająca na posortowaniu obiektów względem położenia na osiach tych ich punktów, które są najbliżej i najdalej początku osi. Potem jeśli początki i końce danej pary nakładają się na wszystkich osiach, to ta para przechodzi dany etap.

Zalety tej metody to szybszy czas działania, $O(nlogn)$ w porównianiu do $O(n^2)$ poprzedniej metody.

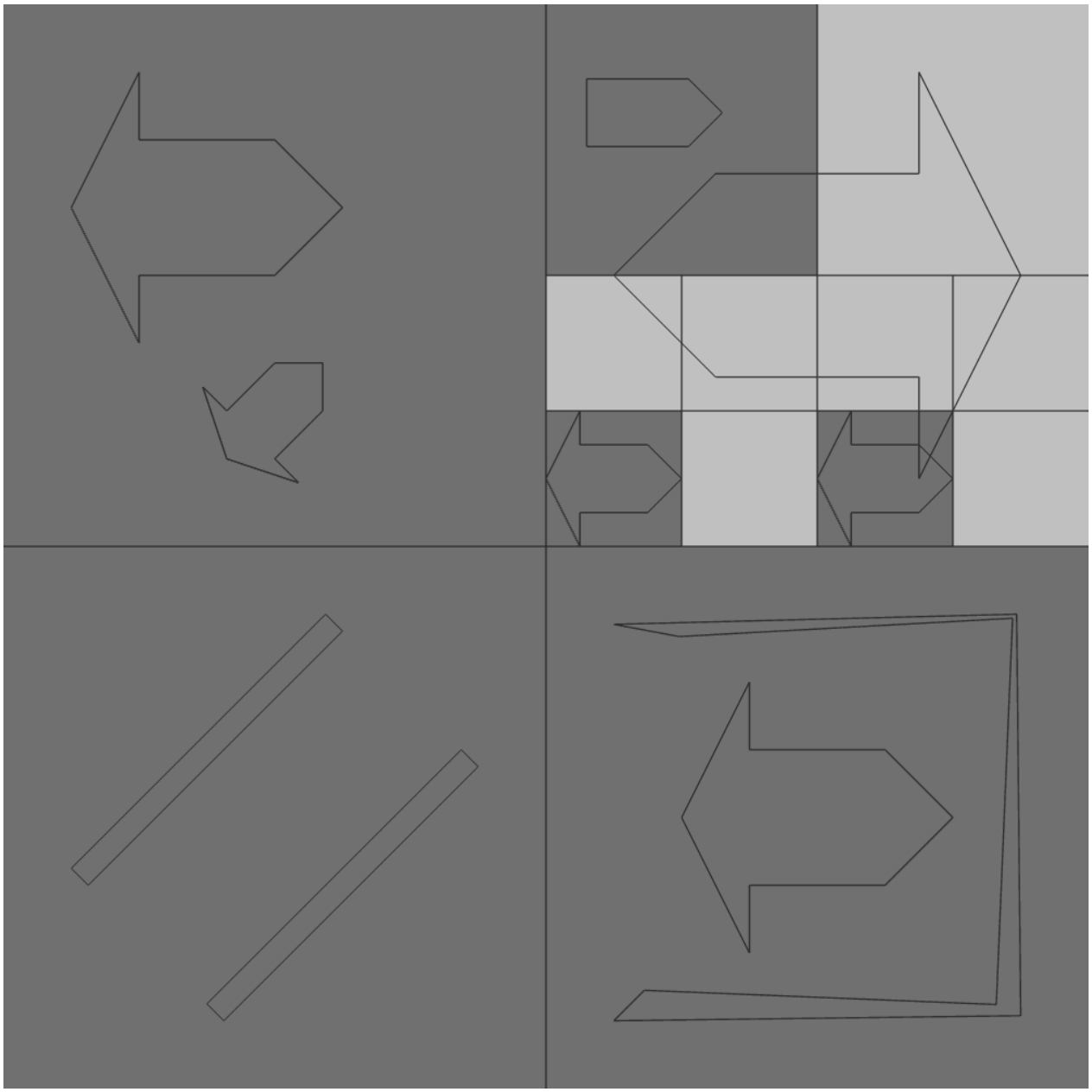


Rysunek 4.7: Przykład wykrywania kolizji metodą Sweep and Prune, kolizje są oznaczone czerwoną linią

Jak widać na obrazku, Wyniki są dokładnie takie same jak dla metody AABB, jednak czas działania tej metody jest dużo krótszy jeśli niewiele elementów ze sobą koliduje.

Quadtree

Każdy obszar jest kwadratem i może być podzielony na 4 podobszary, jeśli zawiera odpowiednio małe elementy. Wykrywanie kolizji zachodzi gdy w jednym obszarze znajduje się więcej niż jeden obiekt.



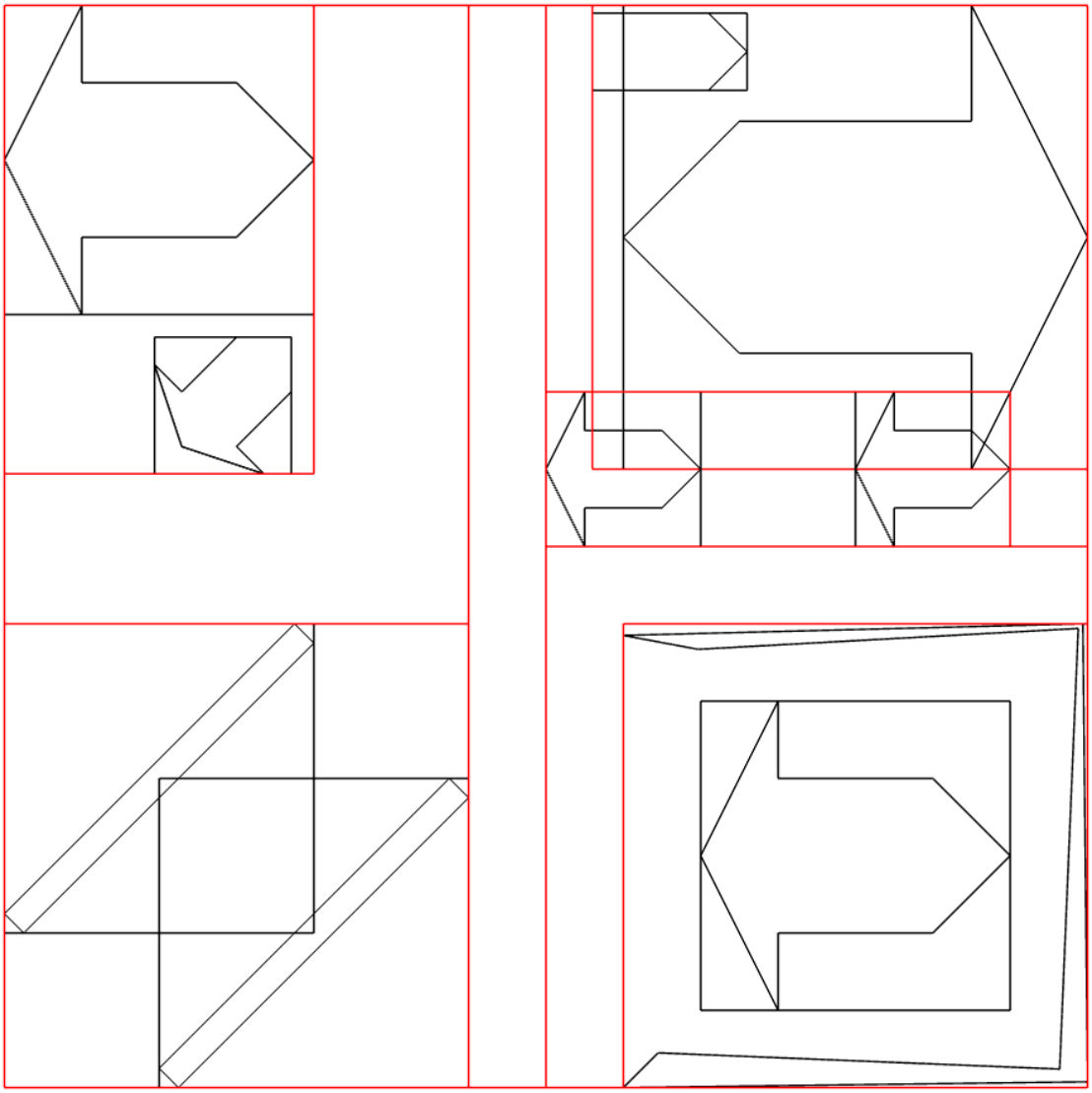
Rysunek 4.8: Przykład użycia metody Quadtree

Octree

Podobnie jak w Quadtree, Octree dzieli obszar na równej wielkości podobszary, z tym że tutaj podział następuje w przestrzeni trójwymiarowej, sześcian dzieli się na 8 mniejszych sześcianów.

Bounding Volume Hierarchy

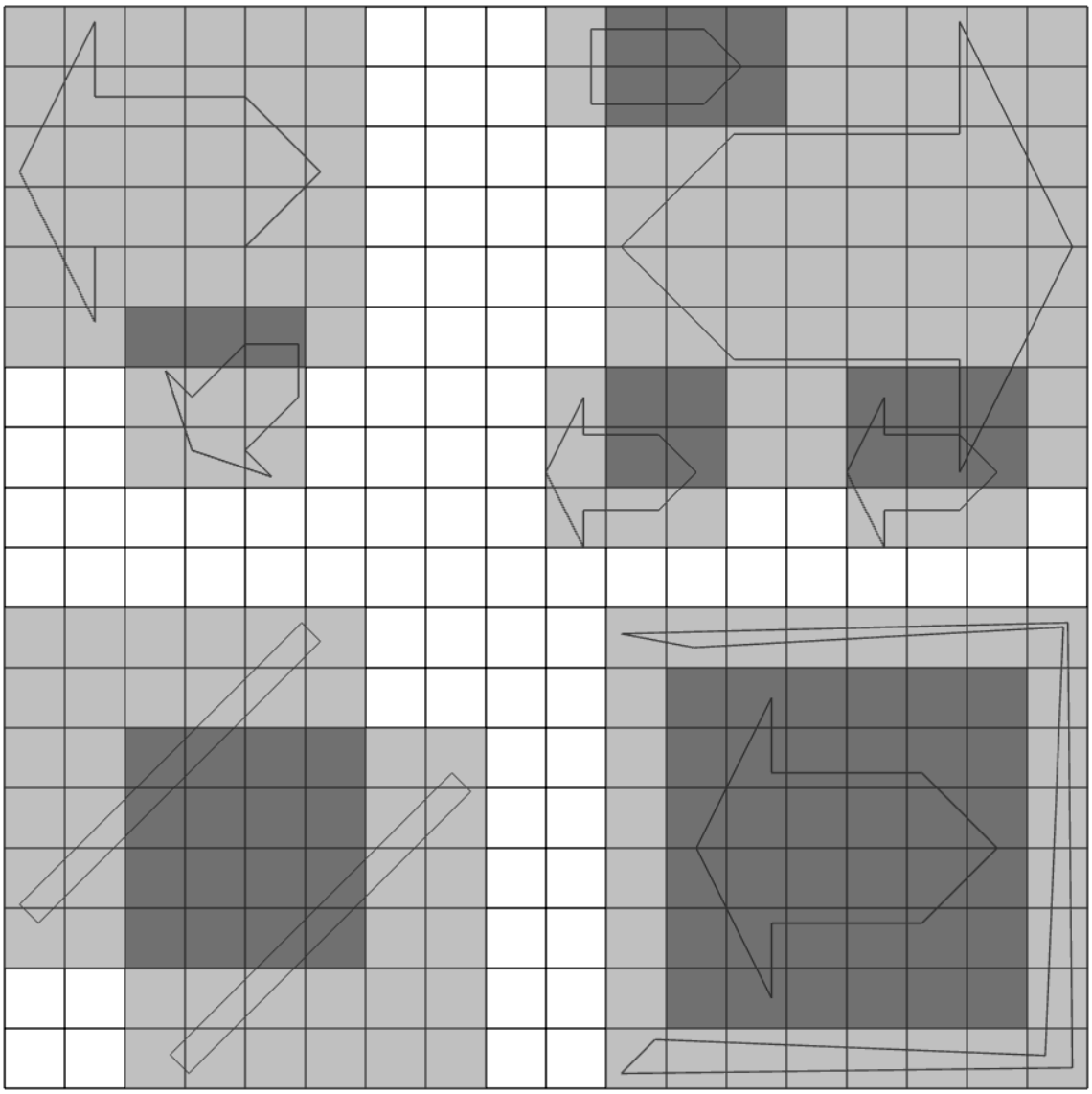
Ta metoda tworzy drzewo przez połączenie dwóch drzew będących blisko siebie w jedno większe drzewo, którego będą poddrzewami. Obydwa poddrzewaw całości mieszczą się w obszarze będącym korzeniem drzewa. Tutaj kolizje zachodzą, jeśli dwa poddrzewa się na siebie nakładają.



Rysunek 4.9: Przykład użycia metody Bounding Volume Hierarchy

Bins

Najprostsza z metod opartych na dzieleniu obszaru. Metoda ta dzieli obszar na wiele równych części i do każdej z nich dodaje wszystkie obiekty które się z nią chociaż częściowo pokrywają. W danym obszarze kolizje liczone są w parach obiektów każdy z każdym.



Rysunek 4.10: Przykład użycia metody , im pole ciemniejsze, tym więcej obiektów je pokrywa

W projekcie została użyta metoda Bounding Circles, ze względu na to, że obiekty w grze często się obracają i mają w wielu przypadkach kształt zbliżony do koła. Aby zmniejszyć liczbę obliczeń, obiekty zostały podzielone na kategorie.

Obiekty podzielone są na następujące kategorie:

- przeciwnicy
- pociski przeciwników
- gracz
- pociski gracza
- teren i tło
- inne

Dzięki takiemu podziałowi można uniknąć obliczeń między obiektami które nie mają się ze sobą zderzać ani wchodzić w interakcję, a także wybierać z obiektami jakich kategorii dany obiekt może wchodzić w interakcje. Po przejściu przez etap Broad Phase, pozostałe pary obiektów zostają następnie przeniesione do drugiego etapu, którym jest Narrow Phase.

Narrow Phase

Ta faza polega na dokładnym wykryciu kolizji pomiędzy obiektami i pewną odpowiedzią czy obiekty kolidują czy też nie. Obliczenia w tej fazie powinny być tak dokładne jak to możliwe.

Wykrywanie kolizji metodą Pixel Perfect Collission Detection

Ta metoda jest używana zazwyczaj w grach dwuwymiarowych, gdzie model można przedstawić w postaci siatki pikseli, na której każdy piksel ma dwa kolory, jeden przedstawiający zawieranie się danego piksela w modelu obiektu i drugi w przeciwnym przypadku, zazwyczaj czarny.

Testowanie kolizji dwóch obiektów polega na nałożeniu odpowiednio przesuniętych siatek obu obiektów na siebie i sprawdzeniu, czy istnieją piksele w których kolor obu modeli się pokrywa.

Obliczenia te można dość łatwo wykonać na wielu wątkach jednocześnie, czasami były one też wspomagane sprzętowo. W dzisiejszych czasach to rozwiązanie jest jednak dość rzadko spotykane.

Można zasymulować tą metodę przy pomocy odpowiednio napisanej funkcji rysującej modele na buforze, a następnie sprawdzeniu czy bufor zawiera piksel w danym kolorze. Możliwe jest wtedy użycie do rysowania np. karty graficznej, co znacznie przyspieszyłoby obliczenia.

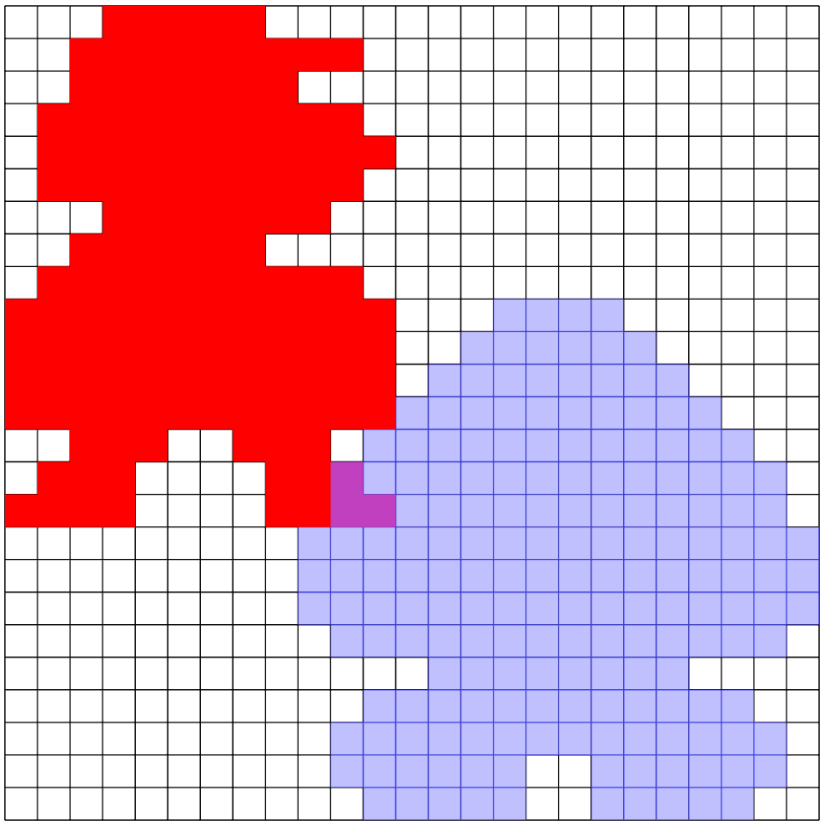
Poniżej jest napisany w pseudokodzie przykładowy algorytm sprawdzania zderzenia między dwoma obiektami.
Siatki składają się z pikseli, zawierających swoje położenie oraz wartość, 1 jeśli piksel należy, a 0 jeśli nie należy do modelu.
Na początku algorytmu tworzony jest odpowiednio duży bufor wypełniony zerami.
Następnie bufor jest wypełniany jedynkami tam, gdzie pierwsza siatka ma wypełnione piksele.
To samo zostaje powtórzone dla drugiej siatki, z tym, że teraz zamiast ustawiać wartość na 1, wykonujemy operację AND, dzięki czemu piksele będą miały wartość 1 tylko tam, gdzie modele się zderzają, piksele drugiej siatki są też przesunięte.
Na koniec zostaje sprawdzone, czy którykolwiek z pikseli bufora ma wartość 1, jeśli tak, to zostaje zwrócona wartość true, w przeciwnym wypadku zostaje zwrócona wartość false.

Obrazek 4.11 ilustruje przykład użycia tej metody, pierwszy obiekt jest narysowany na czerwono, a drugi na niebiesko, miejsce zderzenia jest oznaczone przez fioletowe pola. W przypadku, gdyby nie było takich pól, oznaczałoby to, że kolizja nie zaszła.

Algorithm 4.1: Algorytm wykrywający kolizję dwóch modeli metodą Pixel Perfect Collission Detection

Input: m_1 - siatka pierwszego obiektu
m_2 - siatka drugiego obiektu
x, y - przesunięcie drugiego obiektu względem pierwszego
Output: wartość typu boolean, mówiąca czy obiekty się zderzyły czy też nie

```
1  begin
2      xMin ← min(0, x);
3      xMax ← max(m1.width-1, m2.width+x-1);
4      yMin ← min(0, y);
5      xMax ← max(m1.height-1, m2.height+y-1);
6      test ← new ArrayOfZeroes[xMin..xMax][[yMin..yMax];
7      foreach pixel in m1 do
8          if pixel.val == 1 then
9              test[pixel.x][pixel.y] ← 1;
10     foreach pixel in m2 do
11         test[pixel.x][pixel.y] ← test[x+pixel.x][y+pixel.y] AND pixel.val;
12     foreach pixel in test do
13         if pixel == 1 then
14             return true;
15     return false;
```



Rysunek 4.11: Przykład Pixel Perfect Collission Detection

Wykrywanie kolizji metodą Separating Planes

Metoda Płaszczyzn Oddzielających polega na znalezieniu płaszczyzny, która oddziela od siebie dwa obiekty. Płaszczyzna taka istnieje wtedy i tylko wtedy, gdy są spełnione następujące warunki:

- Zbiory punktów należących do obydwu obiektów są zbiorami wypukłymi
- Obiekty się nie pokrywają

Zbiór wypukły to taki zbiór, w którym odcinek łączący dowolne dwa punkty ze zbioru, także znajduje się w całości w tym zbiorze.
Otoczka wypukła to najmniejszy zbiór wypukły, który zawiera cały dany zbiór, może być równa temu zbiorowi.
Aby zbiór utworzony z powierzchni danego modelu mógł być zbiorem wypukłym (czyli aby model był równy swojej otoczce wypukłej), model taki musi być wielokątem wypukłym, czyli kąt wewnętrzny między dowolnymi dwiema sąsiadującymi krawędziami musi być mniejszy lub równy Π radianów.
Jeśli modele obu zderzanych obiektów są otoczkami wypukłymi, to obiekty nie zderzają się ze sobą wtedy i tylko wtedy, gdy istnieje płaszczyzna oddzielająca te dwa modele.
Jednak co zrobić, gdy model nie jest równy swojej otoczce wypukłej?
Są na to dwa rozwiązania:

- stworzyć dla modelu jego otoczkę wypukłą i uzywać jej do obliczeń
- podzielić model tak, aby każda jego część była otoczką wypukłą

Minusem pierwszego rozwiązania jest niedokładność, wynikająca z dodania do pola modelu dodatkowej przestrzeni aby zrobić z niego zbiór wypukły. Plusem jest to, że liczba wierzchołków do przetworzenia się zmniejsza, ponieważ trzeba tylko pousuwać te wierzchołki, w których kąt wewnętrzny jest większy niż 2Π radianów.

Minusem drugiego rozwiązania jest zwiększenie mocy obliczeniowej potrzebnej do policzenia nowego modelu oraz wykrycie zderzenia na większej ilości wierzchołków. Plusem jest dokładność, ta metoda jest tak dokładna jak oryginalny model.

Jest wiele algorytmów tworzących z modelu jego otoczkę wypukłą. W projekcie został zaimplementowany algorytm skanu Grahama, przedstawiony poniżej.

Algorithm 4.2: Algorytm tworzący otoczkę wypukłą z danych punktów

Input: m - lista punktów należących do modelu
Output: lista punktów tworzących otoczkę wypukłą danego modelu

```
1  begin
2      p ← m[0];
3      foreach point in m do
4          if point.y<p.y OR (point.y == p.y AND point.x<p.x) then
5              p ← point;
6      pointList ← sortByPolarAngle(m, p);
7      insertInFront(pointList, p);
8      i ← 0;
9      while i <pointList.size do
10         p1 ← pointList[i];
11         p2 ← pointList[mod(i+1, pointList.size)];
12         p3 ← pointList[mod(i+2, pointList.size)];
13         if polarAngle(p1, p2)<polarAngle(p2, p3) then
14             i++;
15         else
16             deleteElementFromList(pointList, mod(i+1, pointList.size));
17     return pointList;
```

Powyższy algorytm przyjmuje na wejściu listę punktów tworzących model.

Pierwszy etap algorytmu to znalezienie punktu położonego najniżej na osi Y, a jeśli jest takich kilka to także przesuniętego najbardziej w lewo na osi X. Punkt ten będzie pierwszym punktem należącym do otoczki, nazwijmy go p.

Następnie algorytm sortuje wszystkie inne punkty według ich kąta polarnego względem punktu p. Po posortowaniu punkt p jest dodawany na początek listy.

Ostatnim etapem algorytmu jest usunięcie punktów, które nie tworzą otoczki. Punkty takie można rozpoznać po tym, że jeśli przechodzimy listę, to idąc po krawędzi łączącej dany punkt z poprzednim musimy skręcić zgodnie z ruchem wskazówek zegara, aby przejść na krawędź łączącą dany punkt z następnym. Wszystkie takie punkty usuwamy z listy, pozostałe dla których skręcamy przeciwnie do ruchu wskazówek zegara, zostawiamy na liście.

Wynikiem jest lista punktów tworzących otoczkę wypukłą danego modelu.

Złożoność algorytmu dla n punktów to O(n) dla pierwszego etapu (wybierania początkowego punktu), O(nlogn) dla sortowania oraz O(n) dla przejścia po liście i usunięcia niepotrzebnych wierzchołków. Algorytm jest więc ograniczony przez czas sortowania, O(nlogn). Złożoność pamięciowa algorytmu to O(n).

Istnieją algorytmy rozwiązujące ten problem w czasie O(nlogh), gdzie h to ilość punktów w wyjściowym modelu, są to więc algorytmy, których czas działania jest zależny od wielkości wyjścia.

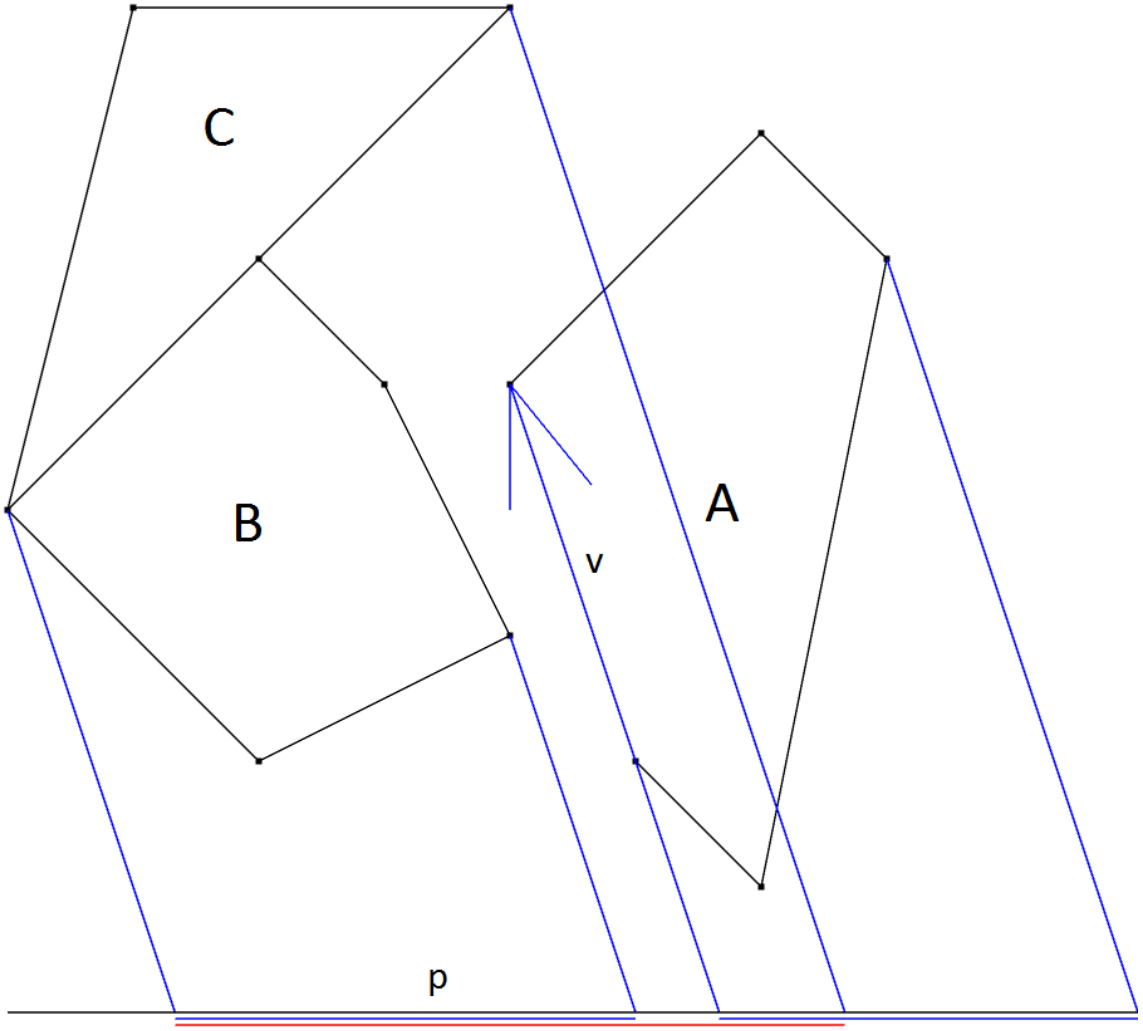
Użycie drugiego rozwiązania sprowadza się zazwyczaj do problemu triangulacji wielokąta, z czasem O(nlogn) dla najlepszych algorytmów. W projekcie zostało jednak użyte pierwsze rozwiązanie dla skomplikowanych modeli, a dla mniejszych modeli dane są tworzone ręcznie przy tworzeniu modelu.

Testowanie kolizji

Gdy obiekty są już w postaci zbiorów wypukłych, można przejść do sprawdzenia czy zaszło zderzenie.
Algorytm wyrażony w pseudokodzie wygląda następująco:

1. dla każdego punktu z pierwszego obiektu:
   (a) wyznacz wektor v od danego punktu do następnego punktu w modelu
   (b) weź dowolną prostą p, która nie jest równoległa do wektora v
   (c) wykonaj rzut prostopadły obu obiektów na prostą p, otrzymując dwa odcinki na tej prostej, m_1 i m_2
   (d) jeśli m_1 i m_2 się nie pokrywają, została znaleziona prosta oddzielająca oba obiekty (równoległa do v), następuje koniec algorytmu
2. powtórz obliczenia zamieniając obiekty ze sobą
3. jeśli nie została znaleziona płaszczyzna oddzielająca, obiekty się pokrywają, koniec algorytmu

Wynikiem tego algorytmu jest pewna odpowiedź czy obiekty kolidują, czy też nie.
Złożoność algorytmu, dla wielkości modeli wynoszących n_1 i n_2, wynosi O($n_1 n_2$).



Rysunek 4.12: Przykład wykrywania kolizji metodą płaszczyzn oddzielających,

Jak widać na obrazku 4.12, obiekty A i B mają płaszczyznę oddzielającą równoległą do v, ponieważ ich rzuty na prostą p nie pokrywają się. Dla A i C żadna płaszczyzna równoległa do v nie oddziela tych obiektów, w kolejnym cyklu pętli jednak zostałaby znaleziona taka prosta, równoległa do następnego boku wielokąta A.

Rozdział 5

Grafika

Ten rozdział traktuje o wyświetlaniu obiektów na ekranie i tworzeniu efektów specjalnych oraz cząsteczkowych.

Do wyświetlania grafiki w grze został użyty WebGL, rozwinięcie języka JavaScript, pozwalające wyświetlać obraz trójwymiarowy w przeglądarce. WebGL polega na używaniu shaderów, czyli programów wykonywanych na kartach graficznych. Język ten bazuje na języku OpenGL w wersji 2.0. WebGL jest tworzony przez Khronos Group, w skład którego wchodzą Mozilla, Apple, Google i Opera Software. W skład shaderów WebGL wchodzą dwa typy shaderów, vertex shader oraz fragment shader. Składnia jest podobna do języka C.
Typy danych dostępne w shaderach to:

- bool, int i float, zachowujące się jak w zwykłym języku programowania

bi vec2/vec3/vec4 - wektory wielkości 2, 3 oraz 4 dla poprzednich typów - odpowiednio b to bool, i to int a brak przedrostka to float

- mat2/mat3/mat4 - macierze kwadratowe 2x2, 3x3, 4x4 ze zmiennymi typu float
- sampler2D - zmienna umożliwiająca dostęp do tekstury 2D
- samplerCube - zmienna umożliwiająca dostęp do tekstury kostki

Można także używać struktur oraz tablicy zmiennych, jednak struktury nie mogą być używane z modyfikatorem varying. W shaderach można także używać wielu funkcji matematycznych i geometrycznych, są tu także dostępne proste operacje na macierzach i wektorach jak mnożenie lub dodawanie.

Dostępne modyfikatory to:

- const - podobnie jak w języku C, jest to stała, tylko do odczytu
- attribute - dostepne tylko w vertex shaderze - oznacza zmienną w której można umieścić dane przekazane do shadera (po podzieleniu)
- uniform - podobnie jak modyfikator attribute, ten modyfikator pozwala umieścić w zmiennej dane z zewnątrz, jednak tutaj są one wspólne dla wszystkich przetwarzanych wierzchołków, nie są dzielone
- varying - dane otrzymane z vertex shadera, które zostaną w przetworzone w rasteryzatorze i podane dalej do fragment shadera (można w nich umieścić np. pozycje wierzchołków na teksturze, a zostanie otrzymana pozycja dla każdego z pikseli w otrzymanym kształcie)

Zazwyczaj jako zmienne z modyfikatorem uniform przekazywane są macierze przekształceń oraz światła, z modyfikatorem attribute przekazywane są dane do shadera, a modyfikatora varying używa się dla danych przekazywanych z vertex shadera do pixel shadera, na przykład pozycja tekstury. Specyfikacja języka WebGL jest opisana na stronie Khronos Group[1]. Bardzo przydatny podczas pisania programów jest też krótki dokument opisujący w skrócie język WebGL[2].

Vertex shader

Vertex shader zajmuje się przetwarzaniem wierzchołków. Dane są przekazywane w całości, i dzielone na poszczególne wierzchołki automatycznie. Na przykład, jeśli do wektora składającego się z 4 pól podamy ciąg długości 12, to zostanie on podzielony na dane dla 3 wierzchołków. Następnie procesor wydaje polecenia mówiące o tym które wierzchołki mają zostać przetworzone i jaki kształt ma zostać użyty. O tym które wierzchołki będą przetworzone decydujemy, wybierając pierwszy wierzchołek i ich ilość. Dostępne kształty to:

- punkty (każdy wierzchołek utworzy jeden punkt o zadanej wielkości)
- linie, w tym:
  - pomiędzy parami wierzchołków (każda para tworzy osobną linię)
  - łamana otwarta (linia jest utworzona między danym wierzchołkiem i następnym dla każdego z wybranych wierzchołków)
  - łamana zamknięta (tak jak poprzednio, ale łączone są też pierwszy i ostatni wierzchołek)
- trójkąty, w tym:
  - każda trójka wierzchołków osobno
  - wachlarz (dla każdego wierzchołka poza pierwszym bierze się dany wierzchołek, następny wierzchołek oraz pierwszy wierzchołek)
  - pas (brane są dany wierzchołek oraz dwa następne)

Przykładowo można narysować dwa trójkąty podając dane dla 6 wierzchołków, ale jeśli te trójkąty mają wspólny bok to mozemy zmniejszyć ilość wierzchołków podając jedynie 4 rogi i rysując je jako wachlarz. Można w ten sposób zmniejszyć ilość danych wysyłanych do karty graficznej co przyspiesza program.

W pixel shaderze istnieje specjalna zmienna gl_Position, do której należy zapisać pozycję wierzchołka po przetworzeniu, a także zmienna gl_PointSize, służąca do ustawienia wielkości punktu, jeśli rysujemy punkty.

Fragment shader

Fragment shader zajmuje się przetwarzaniem pojedynczych pikseli. Dane w tym shaderze otrzymuje się z rasteryzatora i są one osobne dla każdego piksela, który zostanie narysowany na ekranie. Po obliczeniu koloru danego fragmentu (lub odrzuceniu go), kolor zapisuje się w zmiennej gl_FragColor. W tym shaderze dostępna są także zmienne gl_FragCoord typu vec4, pozycja fragmentu na ekranie, oraz gl_FrontFacing typu boolean, mówiąca czy powierzchnia danego fragmentu jest zwrócona w stronę kamery.

We fragment shaderze zazwyczaj na podstawie pozycji z tekstury i innych danych, takich jak pozycja, kolor i typ światła, oblicza się kolor danego piksela.

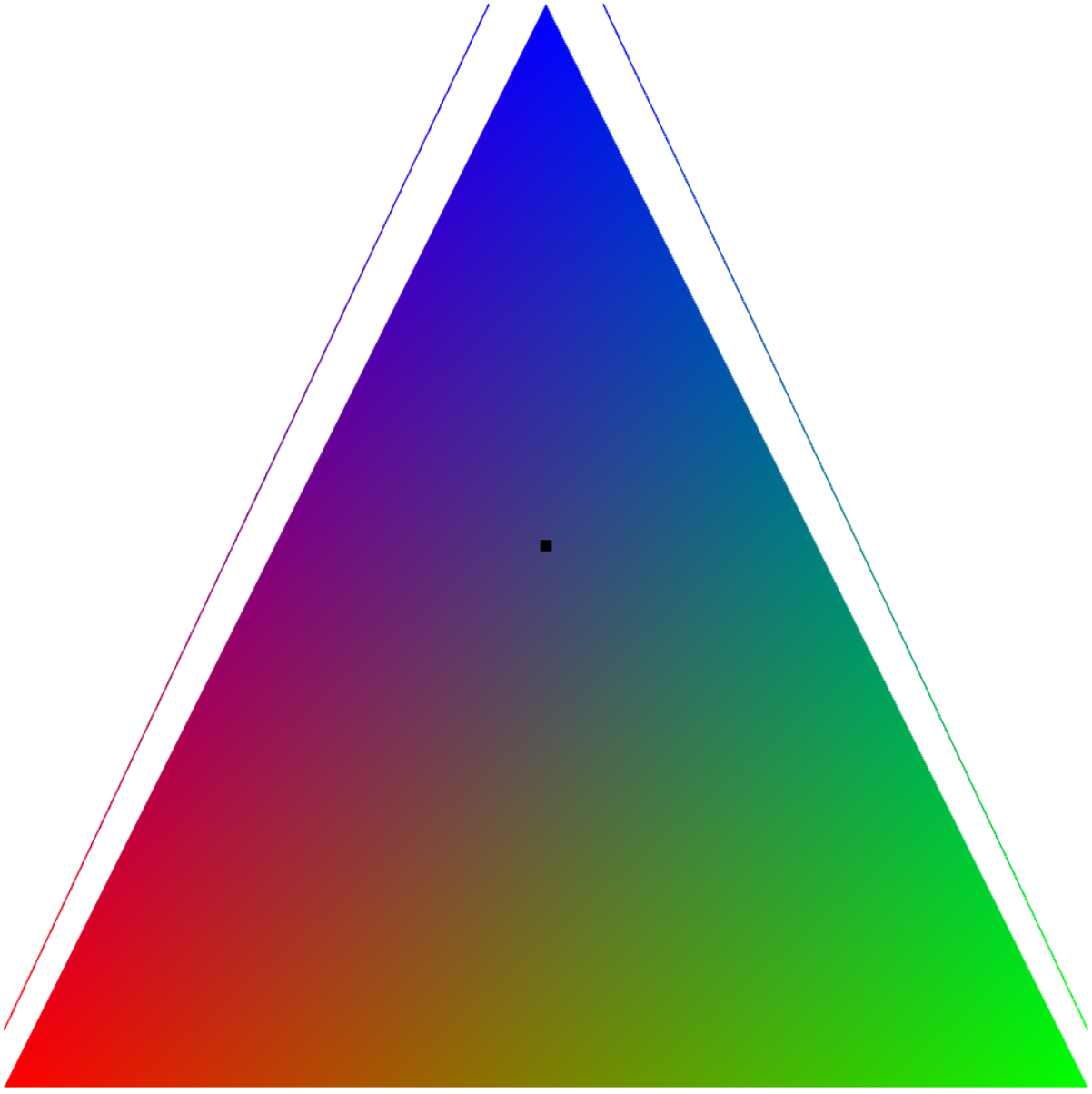
Przykładowy kod prostego programu shader:

Pixel shader

```
1 attribute vec2 position;
2 attribute vec3 color;
3
4 varying vec3 c;
5
6 void main(void) {
7     c = color;
8     gl_Position = vec4(position, 0.0, 1.0);
9     gl_PointSize = 10.0;
10 }
```

Fragment shader

```
1 precision mediump float;
2 #define defaultAlpha 1.0
3
4 varying vec3 c;
5
6 void main(void) {
7     gl_FragColor = vec4(c, defaultAlpha);
8 }
```



Rysunek 5.1: Wynik narysowania trójkąta, dwóch linii i punktu wielkości 10 w przykładowym shaderze (szerokość punktu w oryginale to 10 pikseli)

Przykładowe shadery z efektami

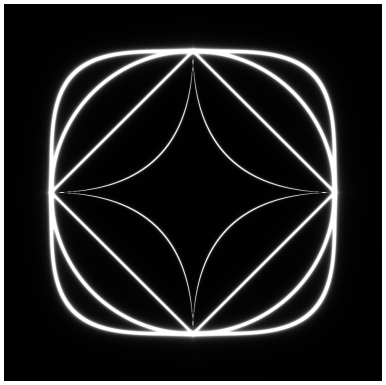
Zaprezentuję tutaj kilka shaderów tworzących ciekawe efekty. Działają one w ten sposób, że kolor danego piksela jest obliczany jedynie na podstawie jego pozycji, używając funkcji matematycznych. Ze względu na działanie podzieliłem je na kształty oraz kolory.

Kształty

Kształty dają jako wynik liczbę, przez którą mnożomy kolor dla danego piksela, aby wzmocnić lub osłabić w nim kolor.

Zaokrąglony kwadrat

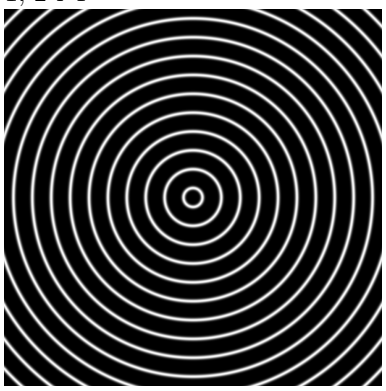
Uzyskuje się go przez obliczenie odległości za pomocą miary używającej innej potęgi (dla potęgi równej 2 uzyskuje się koło). Przy uzależnieniu stopnia potęgi od czasu można uzyskać ładny efekt pulsowania.



Rysunek 5.2: Przykład zaokrąglonego kwadratu dla potęg 0.5, 1, 2 i 4

Rozchodzące się koła

Ten efekt uzyskuje się przez pomnożenie odległości od środka przez jakąś stałą, dodanie aktualnego czasu, a następnie użycie na wyniku funkcji okresowej, takiej jak sinus. Aby uzyskać efekt perpektywy i tunelu można też użyć logarytmu z odległości.



Rysunek 5.3: Przykład efektu rozchodzących się kół

Rozdział 6

Muzyka i efekty dźwiękowe

Ten rozdział opisuje jak działa muzyka i efekty dźwiękowe w mojej grze.

Dźwięki

Wszystkie efekty dźwiękowe oraz muzyka przechowywane są w plikach skompresowanych za pomocą MPEG Audio Layer 3 czyli popularnym formacie mp3. Każdy dźwięk, który ma zostać odtworzony, musi najpierw być pobrany z serwera.

Do odtwarzania plików dźwiękowych używam klasy Audio, dostępnej w piątej wersji języka HTML. Przed tą wersją odtwarzanie dźwięku wymagało użycia wtyczki, ale dzięki rozwojowi języka HTML jest to teraz dostępne jako część standardu.

Efekty dźwiękowe

Wszystkie dźwięki w mojej grze są obsługiwane przez napisaną przeze mnie klasę Sound. Wszystkie nazwy dźwięków są przechowywane w słowniku w tej klasie. Funkcja służąca do odtwarzania efektów dźwiękowych wygląda tak:

```
1 play(name, volume) {
2     this.activeSound++;
3     var a = this.activeSound;
4     this.playing[a] = new Audio("sounds/" + this.sounds[name]);
5     this.playing[a].onended = this.defaultOnEnded(a);
6     this.playing[a].volume = volume*conf.overallVolume*conf.effectsVolume;
7     this.playing[a].play();
8 }
```

Jako argumenty funkcja przyjmuje nazwę efektu dźwiękowego i jego głośność.

Aby można było odtwarzać wiele dźwięków naraz, wszystkie dźwięki są umieszczane w tablicy. Dzięki użyciu tablicy wszystkich odtwarzanych dźwięków, można mieć dostęp i wyłączyć każdy z nich (na przykład przy resetowaniu poziomu, wracaniu do menu itp.). Aby nie było kolizji, przy każdym nowym dźwięku zwiększany jest licznik dźwięków.

Potem tworzony jest nowy dźwięk z pliku, którego nazwa jest wartością w słowniku dla klucza w zmiennej name. Funkcją włączana po zakończeniu danego dźwięku usuwa go. Końcowa głośność dźwięku jest iloczynem podanej głośności, całkowitej głośności gry oraz głośności efektów dźwiękowych. Na koniec dźwięk jest odtwarzany.

Muzyka

Muzyka zachowuje się nieco inaczej niż efekty dźwiękowe. Przede wszystkim, w dowolnym momencie jest odtwarzana tylko jedna ścieżka dźwiękowa dla muzyki, jest za to ona często zamieniana na inną, na przykład przy przejściu z głównego menu do któregoś poziomu.
Funkcja zmieniająca odtwarzaną muzykę prezentuje się następująco:

```
1  changeMusic(music, fadetime = 1) {
2      var thisvar = this;
3      if (this.changingMusic) {
4          return;
5      } else if (this.music == null) {
6          this.changingMusic = true;
7          this.playMusic(music);
8      } else {
9          this.changingMusic = true;
10         var musicVolume = this.musicVolume;
11         var fadetimeScaled = fadetime*100;
12         for (var i=0;i<fadetimeScaled;i++) {
13             window.setTimeout((function(volume) {
14                 return function() {
15                     thisvar.setMusicVolume(volume);
16                 }
17             })((1-i/fadetimeScaled)*musicVolume), 10*i);
18         }
19         window.setTimeout(function() {
20             thisvar.playMusic(music);
21             thisvar.setMusicVolume(1);
22         }, fadetimeScaled*10);
23     }
24 }
```

Dzięki zastosowaniu zmiennej typu boolean, podczas zmiany muzyki nie można przypadkowo zacząć zmieniać muzyki na jeszcze inną, co mogłoby powodować odtworzenie dwóch piosenek naraz.
Dodatkowo muzyka jest stopniowo wyciszana żeby uzyskać efekt Fade, po czym odtwarzany jest nowy utwór. Czas trwania efektu Fade jest modyfikowalny.

Rozdział 7

Podsumowanie

Podsumowując...

Bibliografia

[1] WebGL Specification by Khronos Group
https://www.khronos.org/registry/webgl/specs/1.0/

[2] WebGL Reference Card by Khronos Group
https://www.khronos.org/files/webgl/webgl-reference-card-1˙0.pdf

[3] WebGL Wiki
https://www.khronos.org/webgl/wiki/

[4] W3Schools JavaScript tutorial
http://www.w3schools.com/js/

[5] W3Schools JavaScript reference
http://www.w3schools.com/jsref/

[6] Mozilla Developer Network JavaScript reference
http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference