Kolizje w grach 2D, część 2 (sheadovas/poradniki/goto/kolizje/kolizje-w-grach-2d-czesc-2/)

Lip 26, 2015 / Kolizje (sheadovas/category/poradniki/goto/kolizje/)

W tym wpisie zajmiemy się „poważniejszymi” algorytmami służącymi do wykrywania kolizji: koło-czworokąt oraz pomiędzy wielokątami wypukłymi.

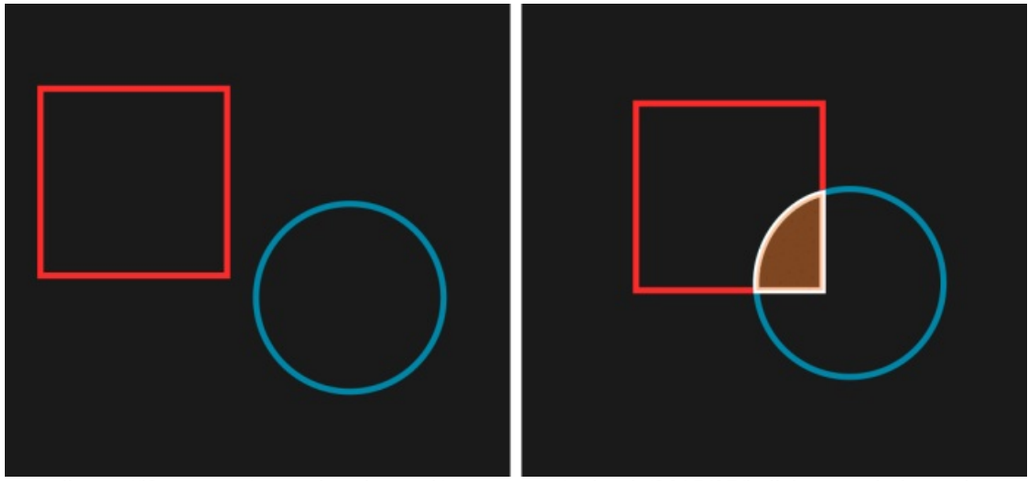
Jeszcze we wstępie chciałbym zauważyć, że te typy kolizji wymagają zrozumienia pewnych rzeczy na wyższym poziomie, poziom tej wiedzy oscyluje na poziomie liceum i 1 roku studiów (SAT). Nie opowiemy sobie też o wykrywaniu kolizji figur wklęsłych, bo nie jest to za często używane (wymaga to sporej ilości obliczeń, wtedy najlepiej jest zastosować uproszczenie do np. koła czy czworokąta).

Kolizja czworokąt-koło

W dalszej części tego paragrafu będę używał zamiennie słów: okrąg/koło w znaczeniu koła, oraz czworokąt/kwadrat jako dowolny prostokąt, czyli czworokąt.

Intuicja

Spróbujmy w sposób naiwny rozważyć wykrycie kolizji, pomiędzy tymi dwoma obiektami matematycznymi na podstawie rysunku poglądowego:



(https://i2.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/07/kolizja-rect-circle.png)

Brak kolizji | Kolizja

Na podstawie tego rysunku jesteśmy wysnuć dwa wnioski:

1. Kolizja nie zachodzi, gdy nie istnieje taki punkt, że należy do okręgu i czworokąta.
2. Kolizja zachodzi gdy istnieje przynajmniej jeden punkt wspólny pomiędzy czworokątem i kołem.

Wnioski są oczywiste. Pobawmy się tymi wnioskami i sprawdźmy co jesteśmy w stanie z nich wycisnąć.

Interesuje nas sytuacja gdy kolizja zachodzi i ją w realnym świecie będzie łatwiej użyć zatem sprawdźmy co jeszcze skrywa w sobie *Wniosek 2*.

Wiemy z niego, że do wykrycia kolizji wystarczy tylko jeden punkt wspólny, czyli całość w ostatecznej wersji sprowadzi się do wykrycia kolizji punkt-okrąg, co już potrafimy robić (część 1 tego artykułu).

Zatem problem sprowadza się do znalezienia punktu, który leży jednocześnie wewnątrz okręgu i kwadratu.

> Niech:
>
> A – zbiór wszystkich punktów należących do koła,
>
> B – zbiór wszystkich punktów należących do czworokąta.

Wtedy:

Istnieje punkt wspólny pomiędzy A i B, gdy dla dowolnego punktu b ze zbioru B min(d(O, b)) <= r.

Gdzie:

- min(x, y) to funkcja wyszukująca wartość najmniejszą;
- d(x,y) to odległość pomiędzy punktami x i y.

Mniej bezdusznie mówiąc: kolizja zachodzi gdy odległość punktu najbliżej odległego od środka okręgu w punkcie O o promieniu r jest odległością mniejszą lub równą długości promienia.

Wszystko co zrobiliśmy to rozpisaliśmy nasz wniosek wykorzystując parę podstawowych definicji. Mam nadzieję, że to rozumiemy i darujemy sobie formalny dowód tego lematu.

Cel: znalezienie najbliższego punktu od środka okręgu

Kolejnym krokiem jest efektywne wyszukanie takiego punktu, najbardziej efektywnym rozwiązaniem (lecz nie działającym zawsze) jest branie punktów leżących jedynie na krawędzi czworokąta, w większości sytuacji to będzie najlepsze rozwiązanie i je zastosujemy.

O ile twierdzenie w czystej postaci działa zawsze, to gdy bierzemy jedynie punkty należące do krawędzi czworokąta, to w przypadku gdy koło znajdzie się wewnątrz większego czworokąta (do którego się w całości zmieści) to mimo istnienia kolizji, nie zostanie ona wykryta.

Jednakże to są już skrajne sytuacje i jeżeli nie dopuszczamy do nagłego pojawiania się okręgów wewnątrz czworokątów to nie mamy o co się martwić.

(https://i0.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/07/kolizja-rect-circle-2.png)
Błędne wykrywanie kolizji

Jeżeli spojrzysz na pierwszy rysunek, to zauważysz, że będziemy rozważali wykrycie kolizje pomiędzy dwoma odcinkami, tutaj znowu rozbijamy nasz problem na 2 mniejsze, już elementarne problemy (dziel i zwyciężaj!):

- sprawdzenie kolizji pomiędzy wierzchołkami, a okręgiem,
- sprawdzenie kolizji pomiędzy prostymi.

Jak widzimy z pozoru dość trudny problem sprowadziliśmy do naprawdę łatwego rozwiązania problemu, *how cool is that*.

Algorytm

Przejdźmy do algorytmu wykrywania kolizji, wykorzystamy to co stworzyliśmy w poprzedniej lekcji.

Przy okazji powiem, że nie testowałem go i ręki sobie uciąć nie dam czy jest w 100% poprawny, jednakże powinien być ok, można go też jeszcze zoptymalizować.

Kolizja box-circle C++

```cpp
#include <cmath>

class Vector2
{
public:
 Vector2()
 {
 x = y = 0;
 }

 Vector2(float _x, float _y)
 {
 x = _x;
 y = _y;
 }

 float x;
 float y;

 Vector2 operator+(Vector2 &w)
 {
 Vector2 r;
 r.x = this->x + w.x;
 r.y = this->y + w.y;

 return r;
 }
};


class Box
{
public:
```

Kolizja figur wypukłych

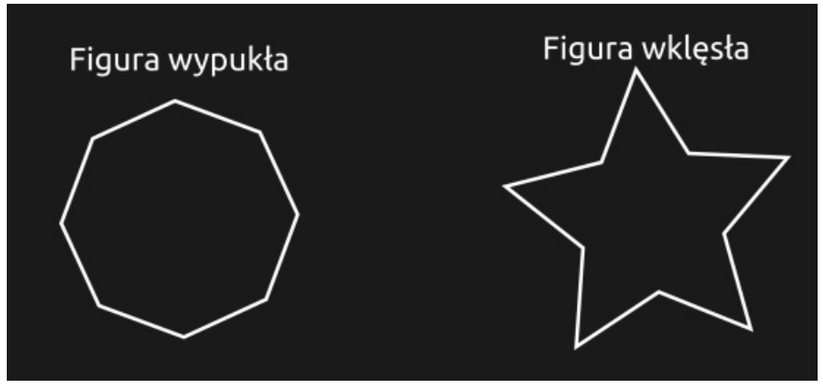
Teraz zajmiemy się algorytmem do wykrywania kolizji figur wypukłych. Tym algorytmem jest SAT (***S**eparating **A**xis **T**heorem*), czyli Twierdzenie Separacji Osi.

Jest to niezwykle potężne narzędzie, nie ograniczają go założenia jakie mieliśmy przy czworokątach, które mówiły że boki muszą być równoległe do osi OX i OY.

Przypomnienie

Przypomnijmy ze szkoły podstawowej kiedy figurę możemy nazwać wypukłą:

> Figura jest wypukła wtedy gdy dla dowolnych punktów A i B należących do tej figury, odcinek AB zawiera się w tej figurze.



(https://i0.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/07/kolizja-sat-figury-def.png)

SAT (wprowadzenie)

Ten algorytm korzysta z dość oczywistej zależności: jeżeli dwa obiekty nie kolidują ze sobą to pomiędzy nimi da się narysować prostą, która nie będzie przecinać żadnego z tych obiektów.

kolizja sat-overview

(https://i2.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/07/kolizja-sat-overview.png)

Jak łatwo się domyślić taka wersja algorytmu nie jest zbyt praktyczna i raczej trudna do implementacji, dlatego korzysta się ze zmodyfikowanej wersji:

> Jeżeli dwie figury nie kolidują ze sobą to istnieje oś, dla której rzuty (projekcje) obu figur nie nachodzą na siebie.

To podejście do problemu przez negację znamy już z poprzedniego artykułu, gdzie wykrywaliśmy kolizję AABB (box-box). Rzut figur wygląda następująco:

kolizja sat-projectiles

(https://i0.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/07/kolizja-sat-projectiles.png)

Jak łatwo się domyślić takich osi jest więcej niż 1 i algorytm wykonuje się dopóki nie natrafi na pierwszą oś bez punktów wspólnych lub sprawdzi wszystkie rzuty. Czyli do wykrycia do potwierdzenia informacji o kolizji musimy sprawdzić wszystkie osie, a do zdobycia informacji wystarczy w najlepszym razie tylko jedno sprawdzenie. Ma to niewątpliwie plus, ponieważ zazwyczaj będziemy spotykali się właśnie z tą sytuacją.

Wybierane osie do sprawdzania kolizji powinny być normalnymi krawędzi każdej krawędzi, czyli wektorami prostopadłymi do nich. Normalne mogą być znormalizowane jeżeli zależy nam na otrzymywaniu informacji w jaki sposób figury kolidują to jesteśmy zmuszeni do normalizacji.

Rzutów dokonujemy zgodnie z kierunkiem kierunkiem normalnej, następnie sprawdzamy, czy odległość od skrajnych punktów będących najbliżej siebie będzie większa od 0.

SAT

Ten algorytm jest zbyt obszerny jak na jeden (i tak już spory) artykuł. Przykładową implementację w kombinacji SFML c++ możesz znaleźć w kodzie źródłowym *Mechanized Techno Explorer* (kurs *Piszemy grę w SFML'u*).

Jeżeli wciąż chcecie więcej to w przystępny sposób opowiedział o nim Olivier Renault w swoim kursie Polycolly (http://www.shead.ayz.pl/wp-content/uploads/2015/07/Polycolly.zip).

Posłowie

Dzięki za przeczytanie, mam nadzieję że Wam się podobała mini-seria poradnika kolizji 2D. Dajcie znać co o niej sądzicie w komentarzach.

Code ON!