Kolizje
(Contacts)

Contacts - obiekty stworzone przez Box2D do zarządzania kolizjami (zderzeniami) między dwoma fiksturami(różne fragmenty jednego ciała).

Istnieją różne rodzaje klas (pochodzących z b2Contact) do zarządzania kolizjami między różnego rodzaju fiksturami.

Dla przykładu: Jedna klasa kontaktów jest do zarządzania kolizjami wielokąt-wielokąt a inna do zarządzania kolizjami koło-koło.

Terminologia

contact point - punkt gdzie dwa ciała się dotykają
contact normal - wektor jednostkowy tych punktów z jednego kształtu do drugiego.
contact manifold - kontakt między dwoma wypukłymi wielokątami może generować do 2 “contact point”. Punkty te są grupowane w “contact manifold”, (w b2Manifold znajdziemy liczbę punktów kolizyjnych).

normal impulse - Normalna siła jest siłą przyłożoną w punkcie kontaktu aby ochronić kształty przed przenikaniem przez siebie. Dla wygody Box2D współpracuje z impulsami. Impuls jest to normalna siła pomnożona przez krok czasowy.

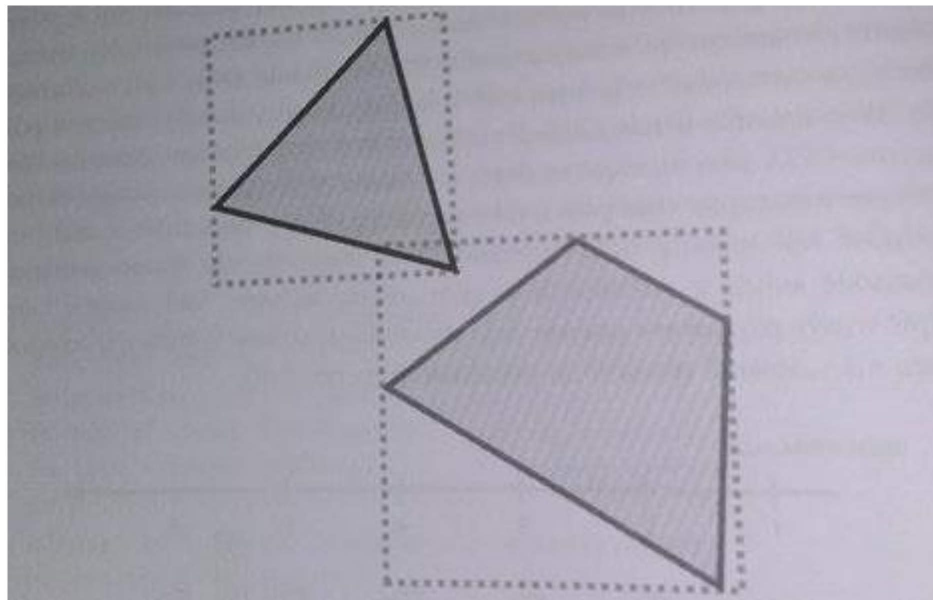
tangent impulse - Siła statyczna jest generowana w punkcie kontaktu aby symulować tarcie. Dla ułatwienia jest przechowywana jako impuls.

contact ids - identyfikatory kontaktów

Box2D wykorzystuje “contact ids” aby dopasować punkty kolizji do wszystkich kroków czasowych. Identyfikatory mają cechy które pomagają odróżnić jeden punkt od drugiego.

Kontakty są tworzone kiedy AABB kształtów nachodzą na siebie. Czasami filtrowanie kolizji będzie zapobiegało tworzeniu kontaktów. Kontakty są niszczone kiedy AABB przestają na siebie nachodzić.
AABB wykorzystuje się ze względu na szybkość i prostotę wykonania testów na nakładanie się prostokątów na siebie - jeśli dwa prostokąty nie nakładają się na siebie oznacza to że figury wpisane w te prostokąty też nie mogą na siebie nachodzić.

AABB - obrysy prostokątne ciał



Klasa Kontaktów

Klasa kontaktów jest tworzona i niszczona przez Box2D. Obiekty nie są tworzone przez użytkownika. Jednak jest on w stanie uzyskać dostęp do klasy kontaktów aby z nią współpracować.
Można pobrać z niej np. punkty kontaktu, normalną kontaktu.

Dostęp do kolizji

Można uzyskać dostęp do kolizji na kilka sposobów:

- Bezpośrednio w Świecie i na danym “ciele”.(Ciało - kilka fikstur połączonych razem, Świat - Klasa świata jest klasą niezbędną do stworzenia symulacji. Świat może zawierać ciała i wiązania.) Możemy “odpytywać” dane ciało czy uczestniczyło w kolizji. Każde ciało ma listę, w której trzyma obiekty będące w stanie kolizji z odpytywanym ciałem. Listy te są aktualizowane w każdym kroku symulacji.

    - Wdrażając “contact listener” (*nasłuchiwacz kolizji* - wywołuje określoną metodę, gdy występuje kolizja)

Contact Listener

Można uzyskać dane o kontaktach przez wdrożenie Nasłuchiwacza Kolizji (b2ContactListener). Obsługuje on kilka wydarzeń: początek, koniec, poprzednie rozwiązanie, następne rozwiązanie.

```
class MyContactListener : public b2ContactListener
{
public:
  void BeginContact(b2Contact* contact)
  { /* handle begin event */ }

  void EndContact(b2Contact* contact)
  { /* handle end event */ }

  void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
  { /* handle pre-solve event */ }

  void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
  { /* handle post-solve event */ }
};
```

Wydarzenie: Początek

Wywołuje się kiedy dwie fikstury zaczynają się pokrywać. Wydarzenie może występować tylko wewnątrz punktu czasowego.

Wydarzenie: Koniec

Następuje kiedy dwie fikstury przestają na siebie nachodzić. Może być wywołany kiedy ciała zostają zniszczone więc musi nastąpić poza krokiem czasowym.

Wydarzenie: Poprzednie rozwiązanie

Wywoływane przed obliczeniami związanymi z reakcją ciał na kontakt(prędkością, odbiciem, tarciem itp.) Możemy w tym momencie ustawić różne parametry kolizji, niezależnie od cech jakie mają kolidujące ciała. (Możemy ustawić np. współczynnik odbicia, tarcia)
Poprzednie rozwiązanie może być wywołane kilka razy w kroku czasowym.

Wydarzenie: Następne rozwiązanie

Wywoływane po obliczeniach związanych z kontaktem ciał. Pozwala nam regulować jak ciało zachowa się po kolizji (np. możemy regulować wartość impulsu.) Jest to miejsce gdzie można zebrać wyniki z kolizji impulsowych.

Korzystając z obiektu b2ContactListener mamy pełną kontrolę nad każdą fazą kolizji - od jej zasygnalizowania, przez jej trwanie, aż po jej zakończenie gdy ciała przestają być w kontakcie.

Implementując własną klasę ContactListener trzeba pamiętać, aby w żadnej z metod nie niszczyć żadnych obiektów b2Body, gdyż prowadzi to do błędów w odwoływaniu się do pamięci zwolnionych już obiektów, które zostają jeszcze na listach kolizyjnych.

Filtrowanie Kolizji

Jeśli nie chcemy, by niektóre fikstury ze sobą kolidowały, możemy użyć *filtrowania kolizji*.
Każda fikstura może przynależeć do jednej lub kilku kategorii (maksymalnie 16).
Jeśli mamy ustawioną kategorię, możemy ustawić, czy dana fikstura ma kolidować z fiksturami z danej grupy czy nie.

Jeśli na przykład chcemy, żeby A nie kolidowało z B, ale kolidowało z C; natomiast B powinno kolidować z C:

C/C++

```cpp
#define CATEGORY(num) (1 << (num)) // kategoria x to włączony bit x w słowie (16-bitowym)

ADef.filter.categoryBits = CATEGORY( 0 ); // A ma kategorię 0
ADef.filter.maskBits = CATEGORY( 2 ); // i może kolidować z kategorią 2 (C)

BDef.filter.categoryBits = CATEGORY( 1 ); // B ma kategorię 1
BDef.filter.maskBits = CATEGORY( 2 ); // i może kolidować z kategorią 2 (C)

CDef.filter.categoryBits = CATEGORY( 2 ); // C ma kategorię 2
CDef.filter.maskBits = CATEGORY( 1 ) | CATEGORY( 0 ); // i może kolidować z kategoriami 1 (B) oraz 0 (A)
```

Często w grze nie chcesz, aby wszystkie obiekty się zderzały. Dla przykładu mogą być to drzwi przez które tylko niektóre postacie mogą przechodzić. To się nazywa filtrowaniem kolizji ponieważ niektóre interakcje są “odfiltrowywane”.

Box2D pozwala osiągnąć filtrowanie kolizji poprzez zastosowanie klasy b2ContactFilter. Ta klasa wymaga implementacji funkcji ShouldCollide, która pobiera dwa punkty b2Shape. Twoja funkcja zwróci true jeżeli kształty powininny kolidować.

Klasa świata (World Class)

Klasa b2World jest jedną z podstawowych i najważniejszych klas Box2D. Jest ona kontenerem dla obiektów, łączeń i detektorów kolizji, oraz miejscem, w którym możliwe jest zarządzanie takimi obiektami.
Dzięki działaniom na obiekcie tej klasy możemy również sterować globalnymi parametrami naszej symulacji, np:

b2World::SetGravity(const b2Vec2& gravity);

oraz pobierać ich wartości/uzyskać dostęp, np:

b2Vec2 b2World::GetGravity();

Tworzenie świata

Klasa b2World posiada między innymi takie, podstawowe konstruktory: kopiujący

```
b2World(const b2World&)
```

z argumentem będącym wektorem grawitacji. Obiekt takiej klasy możemy stworzyć wykorzystując standardowy dla C++ sposób z operatorem new:

```
b2Vec2 gravity(0, -9.81f);
b2World *world = new b2World(gravity);
```

Metoda Step();

Po stworzeniu obiektu świata Box2D, można już przejść do wywoływania jej najważniejszej metody:

b2World::Step(float timeStep, int velocityIteration, int positionIteration);

Wewnątrz metody wykonywana jest cała symulacja fizyczna dla elementów świata. Dobrą praktyką umożliwiającą obserwowanie zadowalających efektów symulacji jest utrzymywanie wartości kroku czasowego na jednakowym poziomie:

float32 timeStep = 1.0f / 60.0f;

Iteratory odpowiadają za liczbę operacji wykonywanych podczas jednego kroku czasowego, kolejno dla ostatecznych prędkości oraz pozycji ciał w danym momencie.
Domyślnie argumenty te przyjmują wartości 3 i 8 iteracji.

Zwiększenie tych wartości może skutkować lepszymi wynikami samej symulacji, jednak w razie problemów ze stabilnością/płynnością, w pierwszej kolejności zawsze warto najpierw zmniejszyć krok czasowy, a dopiero później zwiększać wartości iteracji.

Lepiej zastosować krok czasowy 1/60 i 10 iteracji, niż 1/30 i 20.

Usuwanie obiektu świata odbywa się w analogiczny, standardowy dla języka C++ sposób, czyli poprzez zastosowanie operatora delete:

delete world;

Operacja ta wraz z samym obiektem niszczy również wszystkie zawarte w świecie elementy, tj: ciała, wiązania, kontakty, itp., ponieważ sam obiekt pełni między innymi funkcję kontenera dla nich.

Tworzenie i niszczenie obiektów klasy b2World jest jedyną operacją korzystającą z operatorów new i delete w C++. Wszystkie pozostałe obiekty są tworzone poprzez specjalne metody obiektów wobec nich nadrzędnych. Dzięki temu ułatwione jest zarządzanie zależnościami na przykład między obiektami składowymi świata.

Parametry świata

Z poziomu metod klasy b2World mamy dostęp do kilku paremetrów świata, które wedle uznania możemy zmienić.
Zmiana tych ustawień w powinna mieć charakter eksperymentalny, ponieważ większość symulacji będzie wymagała wartości domyślnych, aby poprawnie funkcjonować.
Wszystkie parametry zaprezentujemy podczas pracy na demie biblioteki po prezentacji.

Sleep

wartość domyślna - true

określa czy obiekty podczas symulacji będą mogły “zasnąć”, tj. czy obiekty nie przejawiające żadnej aktywności oraz takie, które nie wchodzą w interakcje z pozostałymi obiektami, będą mogły zostać pominięte w procesie obliczeń. Zastosowanie tego rozwiązania może przyspieszyć działanie programu/przebieg symulacji.

Metody:

void b2World::SetAllowSleeping(bool);

bool b2World::GetAllowSleeping();

Warm Starting

wartość domyślna - true

określa czy kolejne kroki powinny wykorzystywać dane uzyskane podczas poprzednich kroków symulacji. W efekcie może to przełożyć się na dokładniejszy przebieg symulacji oraz wyniki.

Metody:

void b2World::SetWarmStarting(bool);

bool b2World::GetWarmStarting();

Time of impact

wartość domyślna - true

określa czy symulacja powinna wykorzystywać ciągłą detekcję kolizji (CCD - Continuous Collision Detection) czy też dyskretną. Dyskretna detekcja kolizji może sprawiać, że szybko poruszające się ciała będą się całkowicie mijać, a same kolizje mogą być niewykrywalne.

Metody:

```
void b2World::SetContinuousPhysics(bool);
bool b2World::GetContinuousPhysics();
```

Przykładowe działania na obiektach świata

Mając stworzony obiekt świata możemy rozpocząć pracę nad jej własnościami oraz samą zawartością. Jako że b2World jest kontenerem dla innych obiektów, jednym z naszych celów może być wykonanie mniej lub bardziej skomplikowanego działania na każdym z podrzędnych elementów świata.
Na przykład:

```
for (b2Body*  b = myWorld->GetBodyList(); b; b->GetNext())
{
        b->SetAwake(true);
}
```

Innym przykładem takiego działania jest wysyłanie zapytań dotyczących kolizji obiektów lub też ich AABB. Dzięki implementacji poniższego kodu możemy znaleźć wszystkie fikstury, które potencjalnie nakładają się z tymi, które ustaliliśmy na sztywno oraz wybudzić obiekty, których są częścią.

```
class MyQueryCallback : public b2QueryCallback
{
        public: bool ReportFixture(b2Fixture* fixture)
        {
                b2Body* body = fixture->GetBody();
                body->SetAwake(true);
                // Return true to continue the query. return true;
        }
};
```

… dalszy ciąg:

```
MyQueryCallback callback;

b2AABB aabb;
aabb.lowerBound.Set(-1.0f, -1.0f);
aabb.upperBound.Set(1.0f, 1.0f);
myWorld->Query(&callback, aabb);
```

Dane użytkownika

W pracy nad projektami w Box2D przydatne może okazać się przekazanie wskaźników do pól klas takich jak b2Fixture, b2Body czy b2Joint. Dzięki analizie tych informacji, możemy dotrzeć do tego jak konkretne obiekty mają oddziaływać z innymi.

Na przykład wskaźnik do struktury zawarty w userData jakiejś fikstury może prowadzić nas do informacji o materiale danej powierzchni albo efektach jakim ma ulegać w danych przypadkach.

Konkretny przykład: możemy chcieć podłączyć wskaźnik do obiektu typu GameActor do jakiegoś ciała wewnątrz świata gry.

```
GameActor* actor = GameCreateActor();
b2BodyDef bodyDef;
bodyDef.userData = actor;
actor->body = box2Dworld->CreateBody(&bodyDef);
```

Tego typu operacje mogą okazać się przydatne kiedy:

- “actor” ma odnieść obrażenia po obsłużeniu kolizji z innym obiektem
- chcemy oskryptować zdarzenie w konkretnym miejscu świata gry, np. tuż po wejściu postaci do pomieszczenia (czyli wtedy, kiedy jego AABB zacznie kolidować z AABB pomieszczenia/triggera)

Ograniczenia Box2D

W oficjalnej dokumentacji dotyczącej biblioteki można przeczytać o kilku ograniczeniach, z którymi musi liczyć się twórca oprogramowania opartego o Box2D.

- brakuje stabilnej obsługi kontaktów wśród obiektów o dużej różnicy wagi - kiedy waga jednego jest 10-krotnie większa od drugiego
- łańcuch łączeń (joints) może zostać wydłużony, jeżeli obsługuje ciała znacząco różniące się od siebie masą
- kolizje *'shape versus shape'* mogą nie być dokładne i pozostawiać luki o wielkości 0,5 cm
- tryb ciągłych kolizji nie wspiera łączeń, dlatego można zaobserwować wydłużanie się ich pod wpływem prędkości poruszających się obiektów