X. Implementacja reakcji kolizji

W tej części pokażemy, jak dodać nieco podniecenia do przykładu poduszkowca omawianego w poprzednim rozdziale. W szczególności dodamy kolejny poduszkowiec i pokażemy, jak dodać reakcję kolizji, aby poduszkowiec mógł się zepsuć i odskoczyć jak kilka zderzaków. Jest to ważny element dla wielu rodzajów gier, dlatego ważne jest, abyś zrozumiał kod, który tutaj przedstawimy. Teraz byłby dobry czas, aby powrócić i przejrzeć Część 5, aby odświeżyć pamięć o podstawach reakcji kolizji w ciele sztywnym, ponieważ użyjemy zasad i formuł omawianych tam do opracowania algorytmów reakcji kolizji dla symulacji poduszkowca. W części 8 zobaczyłeś, jak wdrożyć liniową odpowiedź kolizji dla cząstek, a teraz pokażemy Ci, jak radzić sobie z efektami kątowymi. Na początek pokażemy najpierw, jak zaimplementować reakcję kolizji, tak jakby poduszkowiec był parą cząsteczek podobnych do tych w części 8. Podejście to wykorzystuje jedynie impuls liniowy i nie zawiera efektów kątowych, więc wyniki będą nieco nierealistyczne dla tych poduszkowców; jednak takie podejście ma zastosowanie do innych rodzajów problemów, które mogą Cię zainteresować (na przykład kolizje w kulkach bilardowych). Dodatkowo, przyjęcie tego podejścia pozwala nam bardzo wyraźnie pokazać rozróżnienie między efektami liniowymi i kątowymi. Włączenie efektów kątowych sprawi, że symulacja stanie się bardziej realistyczna; kiedy poduszkowiec uderzy się w siebie nawzajem, nie tylko odbiją się od siebie nawzajem, ale również się obrócą. Zanim zanurzymy się w kolizje, dodajmy kolejny poduszkowiec do przykładu, który zaczęliśmy w części 9. Przypomnijmy, że w tym przykładzie mieliśmy jedną jednostkę, którą można kontrolować za pomocą klawiatury. Teraz dodamy kolejny poduszkowiec, który po prostu porusza się pod stałym ciągiem naprzód. Później, gdy dodamy wykrywanie kolizji i odpowiedź, będzie można uruchomić nowy poduszkowiec, aby zmienić jego kurs. Wracając do przykładu z części 9, musimy dodać następną łódź w następujący sposób:

```
RigidBody2D Craft2;
```

Nazywamy nowy poduszkowiec, bardzo kreatywnie, Craft2. W funkcji Initialize musimy teraz dodać następujący kod:

```
bool Initialize (void)
{
.
.
.
Craft2.vPosition.x = _WINWIDTH / 2;
Craft2.vPosition.y = _WINHEIGHT / 2;
Craft2.fOrientation = 90;
.
.
.
}
```

Ten nowy przykład kodu ustawia drugi poduszkowiec na środku ekranu i wskazuje na spód. Jest również kilka wymaganych zmian w UpdateSimulation. Najpierw dodaj Craft2.Up dateBodyEuler (dt);

zaraz po linii Craft.UpdateBodyEuler (dt) ;. Następnie dodaj DrawCraft (Craft2, RGB (200, 200, 0)); po podobnej linii, która rysuje pierwsze Rzemiosło. Craft2 zostanie narysowany na żółto, aby odróżnić go od pierwszego Craft'a. Na końcu dodaj UpdateDimulation:

if (Craft2.vPosition.x> _WINWIDTH) Craft2.vPosition.x = 0;

if (Craft2.vPosition.x <0) Craft2.vPosition.x = _WINWIDTH;

if (Craft2.vPosition.y> _WINHEIGHT) Craft2.vPosition.y = 0;

if (Craft2.vPosition.y <0) Craft2.vPosition.y = _WINHEIGHT;

Teraz możemy dodać kod do obsługi wykrywania i reagowania na kolizje, co pozwala wbij poduszkowiec w nowy, który właśnie dodaliśmy.

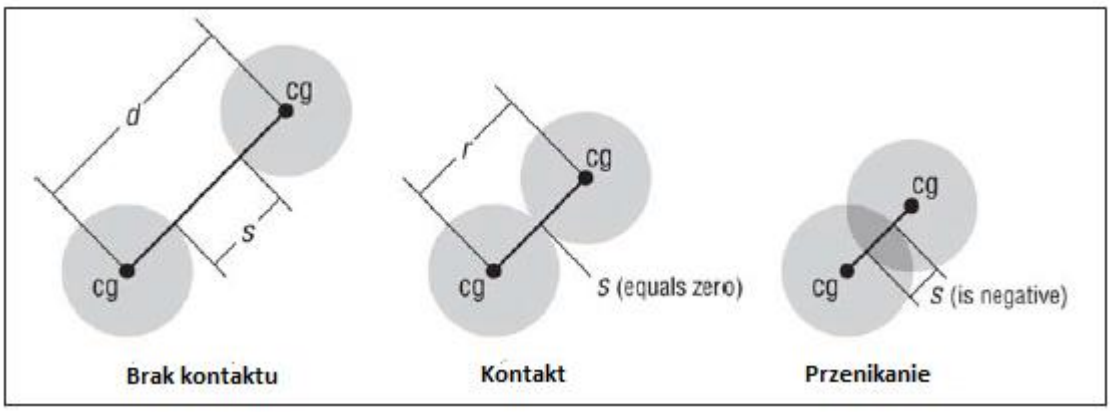
Liniowa reakcja kolizji

W tej sekcji pokażemy, jak wdrożyć prostą odpowiedź na kolizję, zakładając że dwa poduszkowce są cząstkami. Zamierzamy wdrożyć tylko minimum wykrywanie kolizji w tej symulacji; jednak niezależnie od poziomu zaawansowania swoich procedur wykrywania kolizji, są bardzo konkretne informacje, które musisz zebrać od swoich procedur wykrywania kolizji, aby działały procedury reagowania na kolizje oparte na fizyce. Aby zrewidować przykład poduszkowca z poprzedniego rozdziału, aby objąć prostą odpowiedź kolizji, musisz zmodyfikować funkcję UpdateSimulation i dodać kilka dodatkowych funkcji: CheckForCollision i ApplyImpulse. Przed wyświetleniem CheckForCollision, chcemy wyjaśnić, co musi zrobić funkcja wykrywania kolizji. Po pierwsze, musi dać znać, czy występuje kolizja między poduszkowcem. Po drugie, musi Cię poinformować, czy poduszkowiec przenika się nawzajem. Po trzecie, jeżeli poduszkowiec zderza się, musi powiedzieć, jaki jest normalny wektor kolizji i jaka jest względna prędkość pomiędzy zderzającym poduszkowcem. Aby ustalić, czy występuje kolizja, należy wziąć pod uwagę dwa czynniki:

• To, czy obiekty są dostatecznie blisko, w granicach tolerancji numerycznych rozpatrywane przy zderzaniu kontaktów

• Jaka jest względna prędkość normalna między obiektami

Jeśli obiekty nie są blisko siebie, oczywiście nie zderzyły się. Jeśli są w granicach tolerancji dla kontaktu, mogą się zderzać; i jeśli się dotykają i zachodzące na siebie tak, że poruszają się w sobie nawzajem, są przenikające, jak zilustrowane na rysunku 10-1.



Jeśli twoja procedura wykrywania kolizji wykryje, że oba obiekty są rzeczywiście wystarczająco blisko, aby zderzać się z kontaktem, musisz wykonać kolejne sprawdzenie względnej prędkości normalnej, aby

zobaczyć, czy odsuwają się od siebie nawzajem, czy też nie. Kolizja występuje, gdy obiekty są w kontakcie, a punkty styku poruszają się w kierunku do siebie. Penetracja jest ważna, ponieważ jeśli twoje obiekty nakładają się podczas symulacji, wyniki nie będą wyglądać realistycznie - będziesz mieć jeden poduszkowiec poruszający się wewnątrz drugiego. Musisz tylko wykryć ten stan przenikania, a następnie wykonać kopię zapasową symulacji, zmniejszyć czas i spróbować ponownie. Robisz to dopóki nie przestaną się penetrować lub są w granicach tolerancji, aby zostać uznanym za zderzający. Musisz wyznaczyć wektor normalnej prędkości kolizji, aby obliczyć impuls kolizji, który zostanie użyty do symulacji odpowiedzi na kolizję. W prostych przypadkach określenie tego normalnego wektora jest dość proste. W przypadku cząstek lub kulek normalne zderzenie jest po prostu wzdłuż linii, która łączy środki ciężkości każdego zderzającego się obiektu; jest to centralne oddziaływanie, o którym mowa w części 5, i jest takie samo jak w przypadku przykładu dotyczącego cząstek w części 8. Przyjrzyj się teraz funkcji przygotowanej dla tej symulacji, aby sprawdzić kolizje:

```
int CheckForCollision (pRigidBody2D body1, pRigidBody2D body2)

{

Vector d;

float r;

int retval = 0;

float;

Vector v1, v2;

float Vrn;

r = body1-> ColRadius + body2-> ColRadius;

d = body1-> vPosition - body2-> vPosition;

s = d.Magnitude () - r;

d.Normalize ();

vCollisionNormal = d;

v1 = body1-> vVelocity;

v2 = body2-> vVelocity;

vRelativeVelocity = v1 - v2;

Vrn = vRelativeVelocity * vCollisionNormal;

if ((fabs (s) <= ctol) && (Vrn <0.0))

{

retval = 1; // kolizja;

CollisionBody1 = body1;

CollisionBody2 = body2;

} else if (s <-ctol)
```

{

retval = -1; // interpenetrowanie

} else

retval = 0; // brak kolizji

powrót zwrotu;

}

Ta funkcja używa prostego sprawdzania okręgu granicznego, aby określić, czy poduszkowce się zderzają. Pierwszą rzeczą, którą robi, jest obliczenie odległości, r, która reprezentuje absolutną minimalną odległość pomiędzy tymi poduszkowcami, gdy są w kontakcie. ColRadius jest promieniem koła wyznaczającego poduszkowiec. Musimy obliczyć go dla każdego poduszkowca po inicjalizacji w następujący sposób:

-> ColRadius = SQRT (fDługość * fDługość + fWidth * fWidth);

Następnie dystans dzielący poduszkowiec w czasie wywoływania tej funkcji jest określany i zapisywany w zmiennej d. Ponieważ zakładamy, że te poduszkowce są cząstkami, określenie d jest po prostu kwestią obliczenia odległości między współrzędnymi środka ciężkości każdego statku. Pod względem wektorów jest to po prostu wektor pozycji jednego statku, a nie wektor pozycji drugiego. Gdy funkcja ma d i r, musi określić rzeczywistą ilość przestrzeni, s, oddzielając okręgi ograniczające poduszkowiec. Po ustaleniu tego rozdziału funkcja normalizuje wektor d. Ponieważ wektor d znajduje się wzdłuż linii, która oddziela środki ciężkości poduszkowca, normalizacja powoduje powstanie kolizyjnego wektora normalnego, którego potrzebujemy do obliczeń reakcji kolizji. Wektor normalny kolizji zapisywany jest w zmiennej globalnej vCollisionNormal. Po obliczeniu normalnej kolizji funkcja ta określa względną prędkość między poduszkowcem. W formie wektorowej jest to po prostu różnica między wektorami prędkości każdego statku. Zwróć uwagę, że użyte tutaj wektory prędkości muszą być współrzędnymi globalnymi, a nie ustalonymi na stałe (lokalne). Ponieważ to, co jest naprawdę potrzebne do określenia, czy kolizja jest dokonywana, to względna prędkość normalna, funkcja przechodzi do pobrania wektora iloczynu względnej prędkości i normalnych wektorów kolizji, zapisując wynik w zmiennej Vrn. W tym momencie wszystkie obliczenia są zakończone, a jedyne co trzeba zrobić, to dokonać odpowiednich kontroli w celu ustalenia, czy w ogóle występuje kolizja, penetracja lub brak kolizji. Pierwszym sprawdzianem jest sprawdzenie, czy poduszkowiec się koliduje. Ustalamy to przez porównanie bezwzględnej wartości rozdziału poduszkowców, z tolerancją odległości, ctol. Jeśli bezwzględna wartość s jest mniejsza niż ctol, może wystąpić kolizja. Drugim wymaganiem jest, aby względna prędkość normalna była ujemna, co oznacza, że punkty uderzenia w poduszkowiec zbliżają się do siebie. Jeśli występuje kolizja, funkcja zwraca wartość 1, aby wskazać, że reakcja kolizji jest konieczna. Jeśli poduszkowiec nie będzie miał kolizji, wykonamy drugi test, aby sprawdzić, czy poruszają się tak blisko siebie, że przenikają się nawzajem. W tym przypadku, jeśli s jest mniejsze niż -cal, poduszkowiec penetruje, a funkcja zwraca wartość -1. Jeśli poduszkowiec nie koliduje i nie penetruje, funkcja po prostu zwraca wartość 0, co oznacza, że nie jest wymagane żadne dalsze działanie . Zanim przejdziemy dalej, powiedzmy słowo lub dwa o ctol - odległość tolerancji kolizji. Ta wartość podlega strojeniu. Nie ma jednej wartości, która działa dobrze we wszystkich przypadkach. Musisz wziąć pod uwagę ogólne rozmiary potencjalnie kolizyjnych obiektów, rozmiar kroku, którego używasz, oraz odległość, na jaką zderzające się obiekty pochodzą od przeglądarki podczas renderowania (tj. Ich skali). Zasadniczo powinieneś wybrać wartość, która sprawia, że kolizje wyglądają

poprawnie, tak, że z jednej strony obiekty nie przenikają się nawzajem, az drugiej strony nie raportujesz kolizji, gdy obiekty nie wydają się w ogóle dotykać .Spójrz teraz na inną nową funkcję, ApplyImpulse:

```
void ApplyImpulse (pRigidBody2D body1, pRigidBody2D body2)

{

float j;

j = (- (1 + fCr) * (vRelativeVelocity * vCollisionNormal)) /

((vCollisionNormal * vCollisionNormal) *

(1 / body1-> fMass + 1 / body2-> fMass));

body1-> vVelocity + = (j * vCollisionNormal) / body1-> fMass;

body2-> vVelocity - = (j * vCollisionNormal) / body2-> fMass;

}
```

Jest to prosta, ale istotna funkcja reakcji kolizji. To, co robi, to obliczenie liniowego impulsu kolizyjnego jako funkcji względnej prędkości normalnej, masy i współczynnika restytucji zderzaka poduszkowca, używając wzoru, który pokazaliśmy w części 5. Ponadto, stosuje ten impuls do każdego poduszkowca, skutecznie zmieniając ich prędkości w odpowiedzi na zderzenie. Zauważ, że impuls jest stosowany do jednego poduszkowca, a następnie do drugiego impulsu ujemnego. Po ukończeniu tych dwóch nowych funkcji nadszedł czas na zaktualizowanie narzędzia UpdateSimulation do obsługi wykrywania kolizji i reagowania, gdy symulacja postępuje w czasie. Oto, jak wygląda nowa funkcja UpdateSimulation:

```
void UpdateSimulation (float dt)

{

float dtime = dt;

bool tryAgain = true;

int check = 0;

RigidBody2D craft1Copy, craft2Copy;

bool didPen = false;

int count = 0;

Craft.SetThrusters (false, false);

if (IsKeyDown (VK_UP))

Craft.ModulateThrust (true);

if (IsKeyDown (VK_DOWN))

Craft.ModulateThrust (false);

if (IsKeyDown (VK_RIGHT))

Craft.SetThrusters (true, false);

if (IsKeyDown (VK_LEFT))
```

```
Craft.SetThrusters (false, true);
while (tryAgain i& dtime> tol)
{
tryAgain = false;
memcpy (& craft1Copy & Craft, sizeof (RigidBody2D));
memcpy (& craft2Copy, & Craft2, sizeof (RigidBody2D));
Craft.UpdateBodyEuler (dtime);
Craft2.UpdateBodyEuler (dtime);
CollisionBody1 = 0;
CollisionBody2 = 0;
check = CheckForCollision (& craft1Copy i craft2Copy);
if (sprawdź == PENETRATING)
{
dtime = dtime / 2;
tryAgain = true;
didPen = true;
} else if (sprawdź == COLLISION)
{
if (CollisionBody1! = 0 && CollisionBody2! = 0)
ApplyImpulse (CollisionBody1, CollisionBody2);
}
}
if (! didPen)
{
memcpy (& Craft, i craft1Copy, sizeof (RigidBody2D));
memcpy (& Craft2, & craft2Copy, sizeof (RigidBody2D));
}
}
```

Oczywiście ta wersja jest bardziej skomplikowana niż wersja oryginalna. Jest jeden główny powód: penetracja może wystąpić, ponieważ poduszkowiec może poruszać się wystarczająco daleko w ciągu jednego kroku, aby się pokryć. Wizualnie ta sytuacja jest nieatrakcyjna i nierealistyczna, więc

powinieneś spróbować jej zapobiec. Pierwszą rzeczą, którą wykonuje ta funkcja, jest wprowadzenie pętli while:

```
while (tryAgain i& dtime> tol)

{

.

.

.

}
```

Ta pętla służy do tworzenia kopii zapasowej symulacji, jeśli penetracja wystąpiła w początkowym kroku czasowym. Co się dzieje: funkcja najpierw próbuje zaktualizować poduszkowiec, a następnie sprawdza, czy nie ma kolizji. Jeśli dojdzie do kolizji, zostanie ona potraktowana przez zastosowanie impulsu. Jeśli jednak jest penetracja, to wiesz, że krok czasu był zbyt duży i musisz spróbować ponownie. Gdy tak się stanie, tryAgain jest ustawiony na true, krok czasu jest przecięty na pół i podejmowana jest kolejna próba. Funkcja pozostaje w tej pętli tak długo, jak długo występuje penetracja lub do momentu, w którym czas został zredukowany do wielkości wystarczająco małej, aby wymusić wyjście do pętli. Celem tej pętli jest znalezienie największej wielkości kroku, mniejszej lub równej dt, która może być wzięta i nadal unikać penetracji. Albo chcesz kolizję, albo nie ma kolizji. Możesz zadać sobie pytanie, kiedy małe stają się zbyt małe pod względem czasu? Zbyt mały jest oczywiście wtedy, gdy krok czasu zbliża się do 0, a cała twoja symulacja zmierza do zatrzymania. Dlatego możesz chcieć wprowadzić pewne kryteria, aby wyjść z tej pętli, zanim coś zbyt mocno zwolni. Wszystko to zależy od strojenia, a także od wartości ustawionej dla ctol. Nie możemy wystarczająco podkreślić znaczenia dostrojenia tych parametrów. Zasadniczo musisz dążyć do realizmu wizualnego, zachowując przy tym liczbę klatek na sekundę do wymaganych poziomów. Zaglądanie do tej pętli while pokazuje, co się dzieje. Po pierwsze, tryAgain jest ustawiony na false, optymistycznie zakładając, że nie będzie penetracji, a my wykonamy kopie stanów poduszkowca, odzwierciedlając ostatnie udane wezwanie do UpdateSimulation. Następnie wykonujemy zwykłe wywołanie funkcji UpdateBody dla każdej kopii poduszkowca. Następnie wywołuje funkcję wykrywania kolizji CheckForCollision, aby sprawdzić, czy Craft koliduje z Craft2, czy też nie. Jeśli jest penetracja, wtedy tryAgain ma wartość true, dtime jest przecięte na pół, didPen jest ustawione na true, a funkcja wykonuje kolejne okrążenie przez pętlę while. didPen jest flagą, która informuje nas, że wystąpił warunek penetracji. Jeśli wystąpiła kolizja, funkcja obsługuje ją poprzez zastosowanie odpowiedniego impulsu:

```
if (CollisionBody1! = 0 && CollisionBody2! = 0)

ApplyImpulse (CollisionBody1, CollisionBody2);
```

Po przejściu przez pętlę while zaktualizowane stany poduszkowca zostają zapisane, a UpdateSimulation zakończona. Ostatni fragment kodu, który należy dodać, zawiera kilka nowych zmiennych globalnych i definiuje:

```
#define LINEARDRAGCOEFFICIENT 0.25f

#define COEFFICIENTOFRESTITUTION 0.5f

#define COLLISIONTOLERANCE 2.0f

Vector vCollisionNormal;
```

Vector vRelativeVelocity;

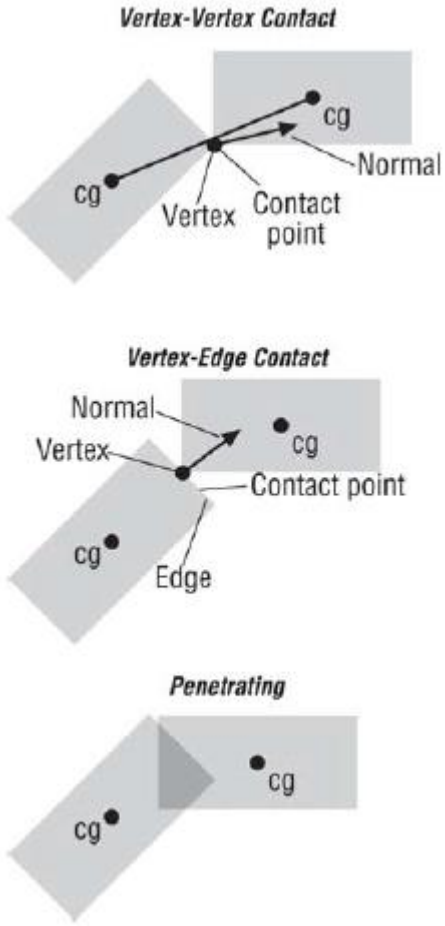
float fCr = COEFFICIENTOFRESTITUTION;

float const ctol = COLLISIONTOLERANCE;

Jedynym, o którym dotąd nie wspominaliśmy, chociaż widzieliśmy go w ApplyImpulse, jest fCr, współczynnik restytucji. Tutaj mamy ustawiony na 0.5, co oznacza, że zderzenia są w połowie drogi między idealnie elastyczną i doskonale nieelastyczną (odwołuj się do naszych wcześniejszych dyskusji na temat współczynników restytucji w Części 5, jeśli zapomniałeś tych o warunkach). Jest to jeden z tych parametrów, które będziesz musiał dostroić, aby uzyskać pożądane zachowanie. Podczas gdy jesteśmy na temat strojenia, powinniśmy wspomnieć, że będziesz musiał także grać z liniowym współczynnikiem oporu używanym do obliczania siły oporu na poduszkowcu. Współczynnik ten jest stosowany do symulowania oporu dynamicznego cieczy, ale odgrywa również ważną rolę w zakresie stabilności numerycznej. Trzeba trochę tłumienia w swojej symulacji, aby integrator nie wysadził się - to znaczy, że tłumienie pomaga utrzymać stabilność symulacji. To tyle, jeśli chodzi o implementację podstawowej reakcji kolizji. Jeśli uruchomisz ten przykład, będziesz mógł sterować poduszkowcem i odskoczyć odpowiednio. Możesz rozkoszować się masą każdego poduszkowca i współczynnikiem restytucji, aby zobaczyć, jak zachowuje się statek, gdy jest on masywniejszy od drugiego, lub gdy kolizja jest gdzieś pomiędzy idealnie elastyczną i doskonale nieelastyczną. Możesz zauważyć, że reakcja kolizji w tym przykładzie czasami wygląda trochę dziwnie. Należy pamiętać, że to dlatego, że ten algorytm reagowania na zderzenia, jak dotąd, zakłada, że poduszkowiec jest okrągły, podczas gdy w rzeczywistości są one prostokątne. Takie podejście będzie działać dobrze dla okrągłych obiektów, takich jak kule bilardowe, ale aby uzyskać poziom realizmu wymagany dla nieokrągłych ciał sztywnych, należy uwzględnić efekty kątowe. Pokażemy Ci, jak to zrobić w następnej sekcji.

Efekty kątowe

W tym efekty kątowe przyniosą bardziej realistyczne reakcje kolizji dla tych sztywnych ciał, poduszkowiec. Aby to zadziałało, musisz wprowadzić kilka zmian /ApplyImpulse i CheckForCollision ;. UpdateSimulation pozostanie niezmieniony. Bardziej rozległe zmiany są w CheckForCollision, więc omówimy je najpierw. Nowa wersja CheckForCollision wykona więcej niż proste sprawdzenie obwodu granicznego. Tutaj każdy poduszkowiec będzie reprezentowany przez wielokąt z czterema krawędziami i czterema wierzchołkami, a typy kontaktu, które będą sprawdzane, to: kontakt werteks-werteks i werteks.



Oprócz zadań omówionych w ostatniej sekcji, ta nowa wersja CheckForCollision musi również określić dokładny punkt kontaktu poduszkowca. To jest bardzo ważne rozróżnienie między tą nową wersją a ostatnią. Musisz wiedzieć punkt kontaktu, ponieważ aby wpłynąć na prędkość kątową, należy zastosować impuls w punkcie kontaktu. W ostatniej sekcji normalne do punktu kontaktowego zawsze przechodziło przez środek ciężkości poduszkowca, ponieważ zakładaliśmy, że są to kule; tak nie jest w tym przypadku. Teraz pojawia się wyzwanie znalezienia kolizji normalnej. Należy rozważyć dwa przypadki. W zderzeniach krawędzi i wierzchołków normalny jest zawsze prostopadły do krawędzi, która jest zaangażowana w kolizję. W zderzeniach werteksów i wierzchołków normalność jest jednak niejednoznaczna, więc zdecydowaliśmy się na normalną równoległość do linii łączącej środki ciężkości poduszkowca. Wszystkie te rozważania sprawiają, że CheckForCollisions jest bardziej zaangażowany niż w poprzedniej sekcji. Poniższa lista kodów pokazuje, co mamy na myśli:

```
int CheckForCollision(pRigidBody2D body1, pRigidBody2D body2)

{

Vector d;

float r;

int retval = 0;

float s;

Vector vList1[4], vList2[4];

float wd, lg;
```

```
int i,j;
bool haveNodeNode = false;
bool interpenetrating = false;
bool haveNodeEdge = false;
Vector v1, v2, u;
Vector edge, p, proj;
float dist, dot;
float Vrn;
// First check to see if the bounding circles are colliding
r = body1->fLength/2 + body2->fLength/2;
d = body1->vPosition - body2->vPosition;
s = d.Magnitude() - r;
if(s <= ctol)
{ // We have a possible collision, check further
// build vertex lists for each hovercraft
wd = body1->fWidth;
lg = body1->fLength;
vList1[0].y = wd/2; vList1[0].x = lg/2;
vList1[1].y = -wd/2; vList1[1].x = lg/2;
vList1[2].y = -wd/2; vList1[2].x = -lg/2;
vList1[3].y = wd/2; vList1[3].x = -lg/2;
for(i=0; i<4; i++)
{
VRotate2D(body1->fOrientation, vList1[i]);
vList1[i] = vList1[i] + body1->vPosition;
}
wd = body2->fWidth;
lg = body2->fLength;
vList2[0].y = wd/2; vList2[0].x = lg/2;
vList2[1].y = -wd/2; vList2[1].x = lg/2;
vList2[2].y = -wd/2; vList2[2].x = -lg/2;
```

```
vList2[3].y = wd/2; vList2[3].x = -lg/2;
for(i=0; i<4; i++)
{
VRotate2D(body2->fOrientation, vList2[i]);
vList2[i] = vList2[i] + body2->vPosition;
}
// Check for vertex-vertex collision
for(i=0; i<4 && !haveNodeNode; i++)
{
for(j=0; j<4 && !haveNodeNode; j++)
{
vCollisionPoint = vList1[i];
body1->vCollisionPoint = vCollisionPoint −
body1->vPosition;
body2->vCollisionPoint = vCollisionPoint −
body2->vPosition;
vCollisionNormal = body1->vPosition −
body2->vPosition;
vCollisionNormal.Normalize();
v1 = body1->vVelocityBody +
(body1->vAngularVelocity^body1->vCollisionPoint);
v2 = body2->vVelocityBody +
(body2->vAngularVelocity^body2->vCollisionPoint);
v1 = VRotate2D(body1->fOrientation, v1);
v2 = VRotate2D(body2->fOrientation, v2);
vRelativeVelocity = v1 - v2;
Vrn = vRelativeVelocity * vCollisionNormal;
if( ArePointsEqual(vList1[i],
vList2[j]) &&
(Vrn < 0.0) )
haveNodeNode = true;
```

```
}
}
// Check for vertex-edge collision
if(!haveNodeNode)
{
for(i=0; i<4 && !haveNodeEdge; i++)
{
for(j=0; j<3 && !haveNodeEdge; j++)
{
if(j==2)
edge = vList2[0] - vList2[j];
else
edge = vList2[j+1] - vList2[j];
u = edge;
u.Normalize();
p = vList1[i] - vList2[j];
proj = (p * u) * u;
d = p^u;
dist = d.Magnitude();
vCollisionPoint = vList1[i];
body1->vCollisionPoint = vCollisionPoint −
body1->vPosition;
body2->vCollisionPoint = vCollisionPoint −
body2->vPosition;
vCollisionNormal = ((u^p)^u);
vCollisionNormal.Normalize();
v1 = body1->vVelocityBody +
(body1->vAngularVelocity ^
body1->vCollisionPoint);
v2 = body2->vVelocityBody +
(body2->vAngularVelocity ^
```

```
body2->vCollisionPoint);
v1 = VRotate2D(body1->fOrientation, v1);
v2 = VRotate2D(body2->fOrientation, v2);
vRelativeVelocity = (v1 - v2);
Vrn = vRelativeVelocity * vCollisionNormal;
if( (proj.Magnitude() > 0.0f) &&
(proj.Magnitude() <= edge.Magnitude()) &&
(dist <= ctol) &&
(Vrn < 0.0) )
haveNodeEdge = true;
}
}
}
// Check for penetration
if(!haveNodeNode && !haveNodeEdge)
{
for(i=0; i<4 && !interpenetrating; i++)
{
for(j=0; j<4 && !interpenetrating; j++)
{
if(j==3)
edge = vList2[0] - vList2[j];
else
edge = vList2[j+1] - vList2[j];
p = vList1[i] - vList2[j];
dot = p * edge;
if(dot < 0)
{
interpenetrating = true;
}
}
```

```
}

}

if(interpenetrating)

{

retval = −1;

} else if(haveNodeNode || haveNodeEdge)

{

retval = 1;

} else

retval = 0;

} else

{

retval = 0;

}

return retval;

}
```

Pierwszą rzeczą, którą robi CheckForCollision, jest szybkie sprawdzenie obwodu aby sprawdzić, czy istnieje możliwość kolizji. Jeśli nie wykryje kolizji, funkcja po prostu się kończy, return 0. To jest ta sama kontrola okręgu kroczącego wykonana we wcześniejszej wersji:

```
r = body1-> fLength / 2 + body2-> fLength / 2;

d = body1-> vPosition - body2-> vPosition;

s = d.Magnitude () - r;

if (s <= ctol)

{

.

.

.

} else

retval = 0;

}
```

Jeśli kontrola okręgu obwodowego wskazuje na możliwość kolizji, to CheckForCollision postępuje, ustawiając dla każdego kilka poligonów, reprezentowanych przez listy wierzchołków poduszkowca:

```
wd = body1-> fWidth;

lg = body1-> fDługość;

vList1 [0] .y = wd / 2; vList1 [0] .x = lg / 2;

vList1 [1] .y = -wd / 2; vList1 [1] .x = lg / 2;

vList1 [2] .y = -wd / 2; vList1 [2] .x = -lg / 2;

vList1 [3] .y = wd / 2; vList1 [3] .x = -lg / 2;

dla (i = 0; i <4; i ++)

{

VRotate2D (body1-> fOrientation, vList1 [i]);

vList1 [i] = vList1 [i] + body1-> vPosition;

}

wd = body2-> fWidth;

lg = body2-> fDługość;

vList2 [0] .y = wd / 2; vList2 [0] .x = lg / 2;

vList2 [1] .y = -wd / 2; vList2 [1] .x = lg / 2;

vList2 [2] .y = -wd / 2; vList2 [2] .x = -lg / 2;

vList2 [3] .y = wd / 2; vList2 [3] .x = -lg / 2;

dla (i = 0; i <4; i ++)

{

VRotate2D (body2-> fOrientation, vList2 [i]);

vList2 [i] = vList2 [i] + body2-> vPosition;

}
```

Listy wierzchołków są inicjowane w nieobjętych ustalonymi bryłami (lokalnymi) w oparciu o długość i szerokość poduszkowca. Wierzchołki są następnie obracane, aby odzwierciedlić orientację każdego poduszkowca. Następnie do każdego wierzchołka dodawana jest pozycja każdego poduszkowca w celu konwersji z lokalnych współrzędnych na globalne współrzędne. Najpierw sprawdzanie kolizji werteks-wierzchołek, funkcja iteruje przez każdy wierzchołek na liście, porównując go z każdym wierzchołkiem na drugiej liście, aby zobaczyć jeśli punkty są zbieżne.

```
// Sprawdź kolizję wierzchołków-wierzchołków

for (i = 0; i <4 &&! haveNodeNode; i ++)

{

dla (j = 0; j <4 &&! haveNodeNode; j ++)

{
```

```
vCollisionPoint = vList1 [i];
body1-> vCollisionPoint = vCollisionPoint -
body1-> vPosition;
body2-> vCollisionPoint = vCollisionPoint -
body2-> vPosition;
vCollisionNormal = body1-> vPosition -
body2-> vPosition;
vCollisionNormal.Normalize ();
v1 = body1-> vVelocityBody +
(body1-> vAngularVelocity ^ body1-> vCollisionPoint);
v2 = body2-> vVelocityBody +
(body2-> vAngularVelocity ^ body2-> vCollisionPoint);
v1 = VRotate2D (body1-> fOrientation, v1);
v2 = VRotate2D (body2-> fOrientation, v2);
vRelativeVelocity = v1 - v2;
Vrn = vRelativeVelocity * vCollisionNormal;
if (ArePointsEqual (vList1 [i],
vList2 [j]) &&
(Vrn <0,0))
haveNodeNode = true;
}
}
```

To porównanie powoduje wywołanie kolejnej nowej funkcji, ArePointsEqual:

```
if (ArePointsEqual (vList1 [i],
vList2 [j]) &&
(Vrn <0,0))
haveNodeNode = true;
```

ArePointsEqual po prostu sprawdza, czy punkty znajdują się w określonej odległości od nawzajem, jak pokazano tutaj:

```
bool ArePointsEqual (wektor p1, wektor p2)
{
```

```
// Punkty są równe, jeśli każdy składnik jest w obrębie ctol od siebie

if ((fabs (p1.x - p2.x) <= ctol) &&

(fabs (p1.y - p2.y) <= ctol) &&

(fabs (p1.z - p2.z) <= ctol))

return true;

else

return false;

}
```

W zagnieżdżonych pętlach sprawdzania wierzchołków-wierzchołków wykonujemy szereg ważnych obliczeń w celu określenia wektora normalnej kolizji i prędkości względnej, które są wymagane do reakcji kolizji. Najpierw obliczamy punkt kolizji, który jest po prostu współrzędnymi wierzchołka, który bierze udział w kolizji. Zauważ, że ten punkt będzie w globalnych współrzędnych, więc będzie musiał zostać przekonwertowany na lokalne współrzędne dla każdego poduszkowca, aby był przydatny do reakcji kolizji. Oto jak to zrobić:

```
vCollisionPoint = vList1 [i];

body1-> vCollisionPoint = vCollisionPoint -

body1-> vPosition;

body2-> vCollisionPoint = vCollisionPoint -

body2-> vPosition;
```

Drugie obliczenie ma na celu określenie wektora normalnego kolizji, który dla zderzenia wierzchołków-wierzchołków, które założyliśmy, są wzdłuż linii łączącej środki ciężkości każdego poduszkowca. Obliczenia są takie same jak w wcześniejszej wersji CheckForCollision:

```
vCollisionNormal = body1-> vPosition -

body2-> vPosition;

vCollisionNormal.Normalize ();
```

Trzecie i ostatnie obliczenie ma na celu określenie prędkości względnej między punktami uderzenia. Jest to ważne odróżnienie od wcześniejszej wersji, ponieważ prędkości punktów uderzenia na każdym ciele są funkcjami liniowymi i kątowymi prędkości poduszkowca:

```
v1 = body1-> vVelocityBody +

(body1-> vAngularVelocity ^ body1-> vCollisionPoint);

v2 = body2-> vVelocityBody +

(body2-> vAngularVelocity ^ body2-> vCollisionPoint);

v1 = VRotate2D (body1-> fOrientation, v1);

v2 = VRotate2D (body2-> fOrientation, v2);
```

vRelativeVelocity = v1 - v2;

Vrn = vRelativeVelocity * vCollisionNormal;

Tutaj, v1 i v2 reprezentują prędkości punktów zderzenia względem każdego poduszkowca w lokalnych współrzędnych, które następnie są przekształcane na globalne współrzędne. Po uzyskaniu prędkości względnej vRelativeVelocity uzyskujemy względną prędkość normalną Vrn, przyjmując iloczyn punktowy prędkości względnej z normalnym wektorem kolizji. Jeśli nie ma kolizji wierzchołków-wierzchołków, CheckForCollision przechodzi do sprawdzenia kolizji werteksów:

// Sprawdź kolizję wierzchołków

if (! haveNodeNode)

{

for (i = 0; i <4 &&! haveNodeEdge; i ++)

{

dla (j = 0; j <3 &&! haveNodeEdge; j ++)

{

jeśli (j == 3)

edge = vList2 [0] - vList2 [j];

jeszcze

edge = vList2 [j + 1] - vList2 [j];

u = krawędź;

u.Normalize ();

p = vList1 [i] - vList2 [j];

proj = (p * u) * u;

d = p ^ u;

dist = d.Magnitude ();

vCollisionPoint = vList1 [i];

body1-> vCollisionPoint = vCollisionPoint -

body1-> vPosition;

body2-> vCollisionPoint = vCollisionPoint -

body2-> vPosition;

vCollisionNormal = ((u ^ p) ^ u);

vCollisionNormal.Normalize ();

v1 = body1-> vVelocityBody +

```
(body1-> vAngularVelocity ^

body1-> vCollisionPoint);

v2 = body2-> vVelocityBody +

(body2-> vAngularVelocity ^

body2-> vCollisionPoint);

v1 = VRotate2D (body1-> fOrientation, v1);

v2 = VRotate2D (body2-> fOrientation, v2);

vRelativeVelocity = (v1 - v2);

Vrn = vRelativeVelocity * vCollisionNormal;

if ((proj.Magnitude ()> 0.0f) &&

(proj.Magnitude () <= edge.Magnitude ()) &&

(dist <= ctol) &&

(Vrn <0,0))

haveNodeEdge = true;

}

}

}
```

Tutaj zagnieżdżone pętle sprawdzają każdy wierzchołek na jednej liście, aby sprawdzić, czy jest on w kontakcie z każdą krawędzią zbudowaną z wierzchołków na drugiej liście. Po zbudowaniu rozważanej krawędzi, zapisujemy i normalizujemy jej kopię, aby reprezentować wektor jednostki wskazujący wzdłuż krawędzi:

```
jeśli (j == 3)

edge = vList2 [0] - vList2 [j];

jeszcze

edge = vList2 [j + 1] - vList2 [j];

u = krawędź;

u.Normalize ();
```

Zmienna u reprezentuje ten wektor jednostkowy i będzie używana w kolejnych obliczeniach. Następny zestaw obliczeń określa położenie rzutu rozpatrywanego wierzchołka na rozpatrywaną krawędź, a także minimalną odległość od wierzchołka do krawędzi:

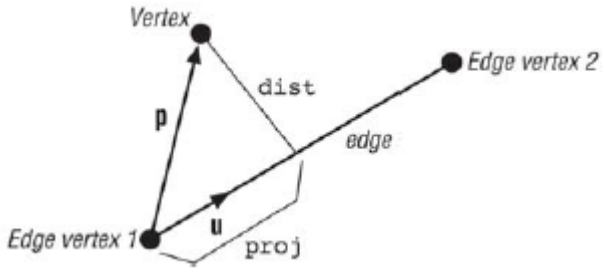
```
p = vList1 [i] - vList2 [j];

proj = (p * u) * u;

d = p ^ u;
```

dist = d.Magnitude ();

Zmienna p jest wektorem od pierwszego wierzchołka na krawędzi do rozpatrywanego wierzchołka, a proj to odległość od wierzchołka pierwszej krawędzi, wzdłuż krawędzi, do punktu, w którym powstaje wierzchołek. dist to minimalna odległość od wierzchołka do krawędzi. Rysunek 10-3 ilustruje tę geometrię.



W przypadku kolizji globalne położenie punktu uderzenia jest równe wierzchołkowi pod uwagę, które musimy przekonwertować na lokalne współrzędne dla każdego poduszkowca, jak pokazano tutaj:

vCollisionPoint = vList1 [i];

body1-> vCollisionPoint = vCollisionPoint -

body1-> vPosition;

body2-> vCollisionPoint = vCollisionPoint -

body2-> vPosition;

Ponieważ w tym rodzaju kolizji normalny wektor kolizji jest prostopadły do krawędzi, możesz to ustalić, biorąc wynik iloczynu u i p i przecinając go z u w następujący sposób:

vCollisionNormal = ((u ^ p) ^ u);

vCollisionNormal.Normalize ();

Te obliczenia dają wektor długości jednostki na płaszczyźnie wektorów ui p i prostopadle do krawędzi. Następnie określa się względną prędkość między punktami uderzenia na każdym poduszkowcu, tak jak w sprawdzaniu kolizji werteks-wierzchołek:

v1 = body1-> vVelocityBody +

(body1-> vAngularVelocity ^

body1-> vCollisionPoint);

v2 = body2-> vVelocityBody +

(body2-> vAngularVelocity ^

body2-> vCollisionPoint);

v1 = VRotate2D (body1-> fOrientation, v1);

v2 = VRotate2D (body2-> fOrientation, v2);

vRelativeVelocity = (v1 - v2);

Vrn = vRelativeVelocity * vCollisionNormal;

Przy ustalaniu, czy rozważany wierzchołek faktycznie koliduje z krawędź, musisz sprawdzić, czy odległość od wierzchołka mieści się w granicach tolerancji kolizji, a także upewnić się, że wierzchołek faktycznie wystaje na krawędź (to znaczy, że nie wystaje poza punkty końcowe krawędzi) . Dodatkowo musisz upewnić się, że względna normalna prędkość wskazuje, że punkty kontaktu zbliżają się do siebie. Oto, jak wygląda to sprawdzenie:

if ((proj.Magnitude ()> 0.0f) &&

(proj.Magnitude () <= edge.Magnitude ()) &&

(dist <= ctol) &&

(Vrn <0,0))

haveNodeEdge = true;

Po sprawdzeniu CheckForCollision w przypadku kolizji między wierzchołkiem a wierzchołkiem i krawędzią na wierzchołku aby sprawdzić penetrację:

if (! haveNodeNode &&! haveNodeEdge)

{

for (i = 0; i <4 & interferencja; i ++)

{

dla (j = 0; j <4 &&! interpenetrating; j ++)

{

jeśli (j == 3)

edge = vList2 [0] - vList2 [j];

jeszcze

edge = vList2 [j + 1] - vList2 [j];

p = vList1 [i] - vList2 [j];

dot = p * edge;

if (kropka <0)

{

interpenetrating = true;

}

}

}

}

Ta kontrola jest standardową metodą sprawdzania wielokątów za pomocą wektora kropek, aby określić, czy dowolny wierzchołek jednego wielokąta leży w granicach drugiego wielokąta. Po tym sprawdzeniu funkcja po prostu zwraca odpowiedni wynik. Tutaj znowu 0 oznacza brak kolizji lub penetracji, 1 oznacza kolizję, a -1 oznacza penetrację. Z CheckForCollision na uboczu, zwróć swoją uwagę na ApplyImpulse, który również musi zostać zmodyfikowany, aby uwzględnić efekty kątowe. W szczególności musisz użyć impulsu formuła, która obejmuje efekty kątowe i liniowe (patrz rozdział 5), a także masz zastosować impuls do kątowych prędkości poduszkowca, a nie liniowo prędkości. Oto jak wygląda nowa funkcja ApplyImpulse:

```
void ApplyImpulse (pRigidBody2D body1, pRigidBody2D body2)

{

float j;

j = (- (1 + fCr) * (vRelativeVelocity * vCollisionNormal)) /

((1 / body1-> fMass + 1 / body2-> fMass) +

(vCollisionNormal * (((body1-> vCollisionPoint ^

vCollisionNormal) / body1-> fInertia) ^ body1-> vCollisionPoint)) +

(vCollisionNormal * (((body2-> vCocusision ^

vCollisionNormal) / body2-> fInertia) ^ body2-> vCollisionPoint))

);

body1-> vVelocity + = (j * vCollisionNormal) / body1-> fMass;

body1-> vAngularVelocity + = (body1-> vCollisionPoint ^

(j * vCollisionNormal)) /

body1-> fInertia;

body2-> vVelocity - = (j * vCollisionNormal) / body2-> fMass;

body2-> vAngularVelocity - = (body2-> vCollisionPoint ^

(j * vCollisionNormal)) /

body2-> fInertia;

}
```

Pamiętaj, że impuls jest stosowany do jednego poduszkowca, podczas gdy jego ujemny jest stosowany do innego. Czyni to dla tej nowej wersji symulacji poduszkowca. Jeśli uruchomisz program teraz zobaczysz, że możesz rozbić poduszkowiec na siebie nawzajem i odbijają się one odpowiednio obrócić. To czyni bardziej realistyczną symulację niż prostą, liniowe podejście do kolizji w ostatniej sekcji. Tutaj znowu możesz grać z masa każdego poduszkowca i współczynnik restytucji, aby zobaczyć, jak te parametry wpływają na reakcję kolizji poduszkowca.