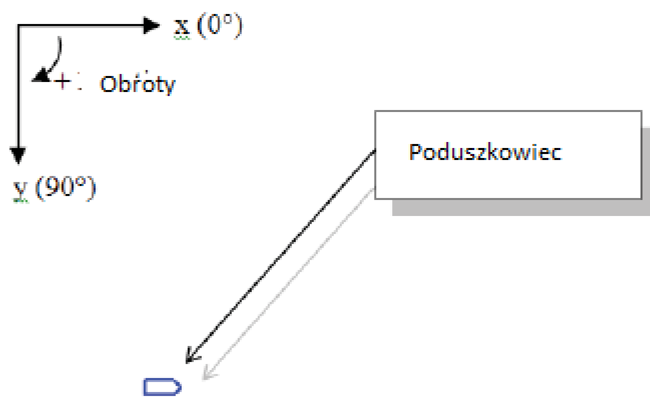
IX. Symulator bryły sztywnej 2D

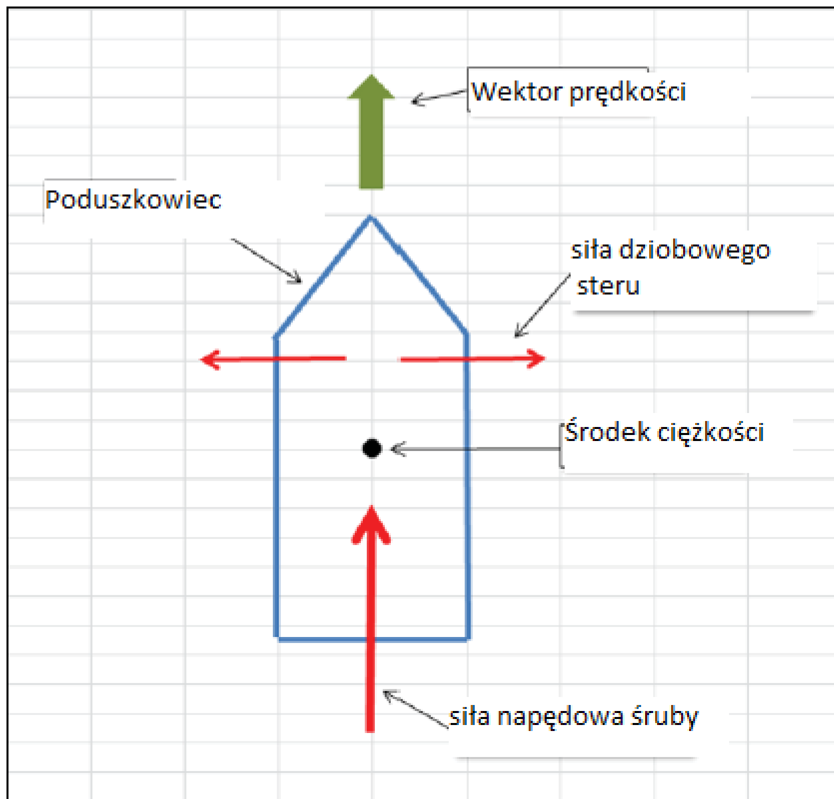
Po przeczytaniu części 8 nauczyłeś się podstawowych składników, które wchodzą w skład symulatora, w szczególności symulatora cząstek. W tym rozdziale spojrzymy poza cząstki sztywnych brył 2D. Główna różnica polega na tym, że ciała sztywne obracają się, a ty musisz poradzić sobie z dodatkowym równaniem ruchu - mianowicie kątowym równaniem ruchu, odnoszącym się do przyspieszenia kątowego sztywnego ciała i bezwładności do sumy wszystkich momentów (momentów obrotowych) działających na ciało sztywne . Podstawowe elementy symulatora - model, integrator, renderer itp. - są takie same jak wcześniej; po prostu musisz poradzić sobie z rotacją. W dwóch wymiarach obsługa obrotu jest prosta. Sprawy stają się nieco bardziej zaangażowane podczas przetwarzania obrotu w trzech wymiarach, a problem ten potraktujemy w części 11. Przykład, który przyjrzymy się bliżej w tym rozdziale, jest prosty z założenia. Chcemy skupić się na różnicach między symulatorem cząstek a symulatorem sztywnego ciała 2D. W części 10 omówimy prosty przykład radzenia sobie z wieloma ciałami sztywnymi i kolizjami. Właśnie tam rzeczy naprawdę się interesują. Na razie rozważymy pojedyncze sztywne ciało, wirtualny poduszkowiec, który porusza się po ekranie pod wpływem sił nacisku, które można kontrolować za pomocą klawiatury. Choć prosty, ten przykład obejmuje najbardziej podstawowe aspekty symulacji sztywnych brył 2D. Rysunek 9-1 pokazuje



nasz wirtualny poduszkowiec. Spiczasty koniec jest z przodu, a poduszkowiec zacznie przesuwać się z lewej strony ekranu w prawo. Za pomocą klawiszy strzałek można zwiększyć lub zmniejszyć prędkość i skręcić w lewo lub w prawo (port lub prawą burtę). W tej symulacji światowy układ współrzędnych ma oś x skierowaną w prawo, swoją oś Y skierowana w dół w kierunku dolnej części ekranu, a oś Z skierowana w ekran. Mimo że jest to przykład 2D, gdzie cały ruch jest ograniczony do płaszczyzny x-y, nadal potrzebujesz osi Z, wokół której obraca się poduszkowiec. Ponadto lokalny lub unieruchomiony układ współrzędnych ma oś X skierowaną w stronę przodu poduszkowca, jego oś Y wskazuje na prawą stronę, a oś Z na ekran. Lokalny układ współrzędnych jest przymocowany do sztywnego ciała w jego położeniu środka ciężkości.

Model

Poduszkowiec modelowany w tej symulacji jest uproszczoną wersją poduszkowca, którą modelujemy w rozdziale 17. Więcej informacji na temat tego modelu można znaleźć w rozdziale 17. Dla wygody powtarzamy tutaj niektóre podstawowe właściwości modelu. Rysunek 9-2 ilustruje główne cechy modelu.



Zakładamy, że ten poduszkowiec działa na gładkim lądzie i jest wyposażony w pojedyncze śmigło, umieszczone w kierunku końca rufy jednostki, które zapewnia ciąg do przodu. Aby sterować, jednostka wyposażona jest w dwa napędy dziobowe, jeden do portu i drugi do prawego steru. Te ster strumieniem służą do sterowania poduszkowcem. Używamy uproszczonego modelu przeciągania, w którym jedyny element oporu jest spowodowany oporem aerodynamicznym całego pojazdu ze stałym rzutowanym obszarem. Ten model jest podobny do tego zastosowanego w części 8 w przypadku oporu cząstek. Bardziej rygorystyczny model uwzględnia rzeczywisty rzutowany obszar jednostki w zależności od kierunku prędkości względnej, jak w przykładzie symulatora lotu omówionego w części 15, a także opór tarcia między spód spódnicy i ziemi. Zakładamy również, że środek oporu - punkt, przez który możemy przyjąć, że wektor siły oporu jest zastosowany - znajduje się w pewnej odległości od środka środka ciężkości, aby zapewnić niewielką stabilność kierunkową (czyli przeciwdziałanie obrotowi). Pełni to tę samą funkcję, co pionowe płetwy ogonowe w samolocie. Ponownie, bardziej rygorystyczny model zawierałby efekty rotacji na opór aerodynamiczny, ale ignorujemy to tutaj. W kodzie, pierwszą rzeczą, którą musisz zrobić, aby reprezentować ten pojazd, jest zdefiniowanie klasy ciała sztywnego, która zawiera wszystkie informacje potrzebne do jej śledzenia i obliczenia sił i momentów na niej działających. Ta klasa RigidBody2D jest bardzo podobna do klasy Particle z części 8, ale z pewnymi dodatkami dotyczącymi głównie rotacji. Oto, jak to zrobiliśmy:

```
class RigidBody2D {

public:

float fMass; // masa całkowita (stała)

float fInertia; // masowy moment bezwładności

float fInertiaInverse; // odwrotność masy bezwładności

Vector vPosition; // pozycja we współrzędnych ziemi
```

```
Vector vVelocity; // prędkość we współrzędnych ziemskich
Vector vVelocityBody; // prędkość w współrzędnych ciała
Vector vAngularVelocity; // prędkość kątowa w współrzędnych ciała
float fSpeed; // prędkość
float fOrientation; // orientacja
Vector vForces; // całkowita siła na ciele
Vector vMoment; // całkowity moment na ciele
float ThrustForce; // Wielkość siły ciągu
Vector Nękani, Stałość; // siły dziobowego steru
float fWidth; // wymiary ograniczające
float fDługość;
float fHeight;
Vector CD; // lokalizacja środka przeciągania we współrzędnych ciała
Vector CT; // lokalizacja środka śruby napędowej w korpusie.
Vector CPT; // lokalizacja portu dziobowego steru uderzenia w korpusy ciała.
Vector CST; // lokalizacja sterburty dziobowego steru dziobowego w ciele
// współrzędne
float ProjectedArea; // rzutowany obszar ciała
RigidBody2D (void);
void CalcLoads (void);
void UpdateBodyEuler (double dt);
void SetThrusters (bool p, bool s);
void ModulateThrust (bool up);
};
```

Komentarze do kodu krótko wyjaśniają każdą właściwość i do tej pory zobaczyłeś wszystkie te właściwości gdzieś w tej książce, więc nie wyjaśniamy ich tutaj ponownie. To powiedziawszy, zauważ, że kilka z tych właściwości jest takich samych jak te przedstawione w Klasie Cząstek z Części 8. Te właściwości obejmują fMass, vPosition, vVelocity, fSpeed, vForces i fRadius. Wszystkie pozostałe właściwości są nowe i wymagane do obsługi aspektów ruchu obrotowego ciał sztywnych. Konstruktor RigidBody2D jest prosty, jak pokazano poniżej, i po prostu inicjuje wszystkie właściwości do dowolnie wybranych wartości, które uznaliśmy za dobre. W części 17 dowiesz się, jak modelujemy bardziej realistyczny poduszkowiec.

```
RigidBody2D :: RigidBody2D (void)
{
```

```
fMass = 100;
fInertia = 500;
fInertiaInverse = 1 / fInertia;
vPosition.x = 0;
vPosition.y = 0;
fWidth = 10;
fDługość = 20;
fHeight = 5;
fOrientation = 0;
CD.x = -0,25 * fDługość;
CD.y = 0,0f;
CD.z = 0,0f;
CT.x = -0,5 * fDługość;
CT.y = 0,0f;
CT.z = 0,0f;
CPT.x = 0,5 * fDługość;
CPT.y = -0,5 * fWidth;
CPT.z = 0,0f;
CST.x = 0,5 * fDługość;
CST.y = 0,5 * fWidth;
CST.z = 0,0f;
ProjectedArea = (fLength + fWidth) / 2 * fHeight; // przybliżenie
ThrustForce = _THRUSTFORCE;
}
```

Podobnie jak w symulatorze cząstek z Części 8, zauważysz tutaj, że klasa Vector jest właściwie potrójny (to znaczy ma trzy komponenty - x, y i z). Ponieważ jest to przykład 2D, składnikami z zawsze będą 0, z wyjątkiem wektora prędkości kątowej, w którym będzie używany tylko składnik z (ponieważ obrót występuje tylko wokół osi Z). Klasa, której używamy w tym przykładzie, jest omówiona w Dodatku A. Powodem, dla którego nie napisaliśmy osobnej klasy wektorów 2D, z tylko komponentami x i y, jest to, że później rozszerzymy ten kod na 3D i chcemy uzyskać przyzwyczaiłeś się do używania klasy wektorowej 3D. Poza tym całkiem łatwo jest stworzyć klasę wektorową 2D z klasy 3D, po prostu usuwając komponent z. Podobnie jak w przykładzie cząstek z rozdziału 8, potrzebujemy metody CalcLoads dla ciała sztywnego. Tak jak poprzednio, metoda ta oblicza wszystkie obciążenia działające na sztywny korpus, ale teraz obciążenia te obejmują zarówno siły, jak i momenty, które spowodują obrót. CalcLoads wygląda następująco:

```
void RigidBody2D :: CalcLoads (void)

{

Vector Fb; // przechowuje sumę sił

Vector Mb; // przechowuje sumę momentów

Vector Thrust; // wektor oporowy

// wyzeruj siły i momenty:

vForces.x = 0.0f;

vForces.y = 0.0f;

vForces.z = 0.0f; // zawsze zero w 2D

vMoment.x = 0.0f; // zawsze zero w 2D

vMoment.y = 0.0f; // zawsze zero w 2D

vMoment.z = 0.0f;

Fb.x = 0,0f;

Fb.y = 0,0f;

Fb.z = 0,0f;

Mb.x = 0.0f;

Mb.y = 0,0f;

Mb.z = 0,0f;

// Zdefiniuj wektor ciągu, który działa poprzez CG jednostki

Thrust.x = 1,0f;

Thrust.y = 0,0f;

Thrust.z = 0.0f; // zero w 2D

Thrust * = ThrustForce;

// Oblicz siły i momenty w przestrzeni ciała:

Vector vLocalVelocity;

float fLocalSpeed;

Vector vDragVector;

float tmp;

Vector r vResultant;

Vector vtmp;

// Oblicz aerodynamiczną siłę oporu:
```

```
// Oblicz prędkość lokalną:
// Lokalna prędkość zawiera prędkość z powodu
// ruch liniowy jednostki,
// plus prędkość w każdym elemencie
// ze względu na rotację jednostki.
vtmp = vAngularVelocity ^ CD; // część obrotowa
vLocalVelocity = vVelocityBody + vtmp;
// Oblicz lokalną prędkość powietrza
fLocalSpeed = vLocalVelocity.Magnitude ();
// Znajdź kierunek, w którym będzie działać drag.
// Przeciąganie zawsze działa zgodnie z wartością względną
// prędkość, ale w przeciwnych kierunkach
if (fLocalSpeed> tol)
{
vLocalVelocity.Normalize ();
vDragVector = -vLocalVelocity;
// Określ siłę wypadkową na elemencie.
tmp = 0,5f * rho * fLocalSpeed * fLocalSpeed
* Przewidywany poziom odniesienia;
vResultant = vDragVector * _LINEARDRAGCOEFFICIENT * tmp;
// Zachowaj całkowitą sumę tych wypadkowych sił
Fb + = vResultant;
// Oblicz moment dotyczący CG
// i zachowaj pełną sumę tych chwil
vtmp = CD ^ vResultant;
Mb + = vtmp;
}
// Oblicz siły pędnika dziobowego w porcie i sterburcie:
// Zachowaj całkowitą sumę tych wypadkowych sił
Fb + = PThrust;
// Oblicz moment na temat CG siły tego elementu
```

```
// i zachowaj całkowitą liczbę tych momentów (łączny moment)

vtmp = CPT ^ PThrust;

Mb + = vtmp;

// Zachowaj całkowitą sumę tych wypadkowych sił (całkowita siła)

Fb + = Pofałdowanie;

// Oblicz moment na temat CG siły tego elementu

// i zachowaj całkowitą liczbę tych momentów (łączny moment)

vtmp = CST ^ SThrust;

Mb + = vtmp;

// Teraz dodaj siłę napędową

Fb + = Pchnięcie; // żadna chwila, ponieważ linia akcji przechodzi przez CG

// Konwertuj siły z przestrzeni modelu na przestrzeń Ziemi

vForces = VRotate2D (fOrientation, Fb);

vMoment + = Mb;

}
```

Pierwszą rzeczą, którą robi CalcLoads, jest zainicjowanie zmiennych siły i momentu, które będą zawierały wszystkie siły i momenty działające na jednostkę w dowolnym momencie. Tak jak musimy zsumować siły, musimy także zsumować momenty. Siły będą używane wraz z liniowym równaniem ruchu, aby obliczyć liniowe przesunięcie sztywnego ciała, podczas gdy momenty będą używane z kątowym równaniem ruchu, aby obliczyć orientację ciała. Funkcja następnie definiuje wektor przedstawiający ciąg śruby, Thrust. Wektor ciągu śmigła działa w dodatnim (lokalnym) kierunku x i ma wielkość zdefiniowaną przez ThrustForce, którą użytkownik ustawia za pomocą interfejsu klawiatury (dojdziemy do tego później). Zauważ, że jeśli ThrustForce ma wartość ujemną, to siła ciągu będzie faktycznie odwrotnym ciągiem zamiast naprzód. Po zdefiniowaniu wektora ciągu funkcja ta oblicza opór aerodynamiczny działający na poduszkowiec. Te obliczenia są bardzo podobne do tych omówionych w części 17. Pierwszą rzeczą do zrobienia jest określenie prędkości względnej w centrum oporu, biorąc pod uwagę zarówno ruch liniowy, jak i kątowy. Będziesz potrzebował wielkości wektora prędkości względnej podczas obliczania wielkości siły oporu, a będziesz potrzebował kierunku wektora prędkości względnej, aby określić kierunek siły oporu, ponieważ zawsze jest on przeciwny wektorowi prędkości. Wiersz vtmp = vAngularVelocity ^ CD oblicza prędkość liniową w środku ciągu, pobierając wektor iloczynu wektora prędkości kątowej z wektorem położenia centrum przeciągania, CD. Rezultat jest przechowywany w tymczasowym wektorze, vtmp, a następnie dodawany jest wektorowo do wektora prędkości ciała, vVelo cityBody. Wynikiem tego wektora jest wektor prędkości reprezentujący prędkość punktu zdefiniowanego przez CD, w tym wkłady ruchu liniowego i kątowego ciała. Obliczamy rzeczywistą siłę oporu, która działa w kierunku zgodnym z kierunkiem przeciwnym do wektora prędkości, w sposób podobny do tego w przypadku cząstek, używając prostej formuły odnoszącej siłę oporu do prędkości kwadratu, gęstości powietrza, przewidywanego obszaru, i współczynnik oporu. Poniższy kod wykonuje następujące obliczenia:

```
vLocalVelocity.Normalize ();
```

vDragVector = -vLocalVelocity;

// Określ siłę wypadkową na elemencie.

tmp = 0,5f * rho * fLocalSpeed * fLocalSpeed

* Przewidywany poziom odniesienia;

vResultant = vDragVector * _LINEARDRAGCOEFFICIENT * tmp;

Zwróć uwagę, że współczynnik przeciągania LINEARDRAGCOEFFICIENT jest zdefiniowany następująco:

#define LINEARDRAGCOEFFICIENT 1.25f

Po określeniu siły oporu, zostaje ona zagregowana w całkowitym wektorze siły w następujący sposób:

Fb + = vResultant;

Oprócz agregowania tej siły, musimy zsumować moment z powodu tej siły w całkowitym momencie wektora w następujący sposób:

vtmp = CD ^ vResultant;

Mb + = vtmp;

Pierwsza linia oblicza moment ze względu na siłę oporu, przyjmując krzyż wektorowy iloczyn wektora położenia, do środka oporu, za pomocą wektora siły przeciągania. Druga linia dodaje tę siłę do zmiennej, gromadząc te momenty. Po zakończeniu obliczania oporu, kalkulatory CalcLoads kontynuują obliczanie sił i momentów związanych z pędnikami dziobowymi, które mogą być aktywne lub nieaktywne w danym momencie.

Fb + = PThrust;

vtmp = CPT ^ PThrust;

Mb + = vtmp;

Pierwsza linia agreguje siłę dziobowego napędu strumieniowego do Fb. PThrust jest wektorem siły obliczane w metodzie SetThrusters w odpowiedzi na dane z klawiatury. Następne dwie linie obliczają i sumują moment z powodu siły pędnika. Podobny zestaw linii kodu następuje, obliczając siłę i moment ze względu na ster strumieniowy sterburtu. Następnie siła ciągu śmigła zostaje dodana do sumy sił. Pamiętaj, ponieważ siła ciągu śmigła działa przez środek ciężkości, nie ma powodu do zmartwień. Dlatego potrzebujemy tylko:

Fb + = Thrust // żadna chwila, ponieważ linia akcji przechodzi przez CG

Ostatecznie, całkowita siła jest przekształcana z lokalnych współrzędnych na współrzędne świata poprzez obrót wektora, biorąc pod uwagę orientację poduszkowca, a całkowite siły i momenty są przechowywane tak, że są dostępne, gdy przychodzi czas na integrację równań ruchu za każdym razem . Jak widać, obliczanie obciążeń na sztywnym korpusie jest nieco bardziej skomplikowane niż to, co widzieliśmy wcześniej, gdy mamy do czynienia z cząstkami. Wynika to oczywiście z tego, że ciała sztywne mogą się obracać. Jednak fajne jest to, że cała ta nowa złożoność jest zawarta w CalcLoads, a reszta symulatora jest prawie taka sama, jak w przypadku cząstek.

Przekształcanie współrzędnych

Porozmawiajmy o transformacji ze współrzędnych lokalnych do światowych trochę więcej, ponieważ zobaczysz tego rodzaju transformację ponownie w kilku miejscach. Obliczając siły działające na ciało sztywne, chcemy te siły w formie wektorowej względem współrzędnych, które są ustalone w odniesieniu do poduszkowca (np. Względem środka ciężkości ciała z osią x skierowaną w kierunku przedniej części ciała). ciało i oś y skierowane w stronę prawej burty). Upraszcza to nasze obliczenia sił i momentów. Jednakże, integrując równanie ruchu, aby zobaczyć, jak ciało tłumaczy we współrzędnych światowych, używamy równań ruchu we współrzędnych światowych, co wymaga od nas reprezentowania zagregowanej siły we współrzędnych światowych. Właśnie dlatego obróciliśmy siłę agregacji na końcu metody CalcLoads. W dwóch wymiarach transformacja współrzędnych wymaga małej trygonometrii, jak pokazano w następującej funkcji VRotate2D:

```
Wektor VRotate2D (kąt pływaka, wektor u)

{

float x, y;

x = u.x * cos (DegreesToRadians (-angle)) +

u.y * sin (DegreesToRadians (-angle));

y = -u.x * sin (DegreesToRadians (-angle)) +

u.y * cos (DegreesToRadians (-angle));

return Vector (x, y, 0);

}
```

Kąt tutaj przedstawia orientację lokalnego, ustalonego układu współrzędnych w odniesieniu do układu współrzędnych świata. Podczas konwersji z lokalnych współrzędnych na współrzędne światowe, użyj kąta dodatniego; używaj ujemnego kąta, gdy idziesz w drugą stronę. To tylko konwencja, którą przyjęliśmy, więc transformacje od lokalnych współrzędnych do współrzędnych globalnych są pozytywne. Widać, że faktycznie przyjmujemy ujemny parametr kąta, więc w rzeczywistości można by pozbyć się tego negatywu, a wtedy transformacje z lokalnych współrzędnych na współrzędne świata byłyby w rzeczywistości ujemne. To twoje preferencje. Ta funkcja będzie używana jeszcze kilka razy w różnych sytuacjach przed końcem tej części.

Całkowanie

Metoda UpdateBodyEuler faktycznie całkuje równania ruchu dla sztywnego ciało. Ponieważ mamy do czynienia ze sztywnym ciałem, w przeciwieństwie do cząstki, mamy dwa równania ruchu: jeden do tłumaczenia, a drugi do obrotu. Poniższy przykładowy kod pokazuje UpdateBodyEuler.

```
void RigidBody2D :: UpdateBodyEuler (double dt)

{

Vector a;

Vector dv;

Vectorr ds;

float aa;
```

```
float dav;

float dr;

// Oblicz siły i momenty:

CalcLoads ();

// Zintegruj liniowe równanie ruchu:

a = vForces / fMass;

dv = a * dt;

vVelocity + = dv;

ds = vVelocity * dt;

vPosition + = ds;

// Zintegruj kątowe równanie ruchu:

aa = vMoment.z / fInertia;

dav = aa * dt;

vAngularVelocity.z + = dav;

dr = RadiansToDegrees (vAngularVelocity.z * dt);

fOrientacja + = dr;

// Misc. obliczenia:

fSpeed = vVelocity.Magnitude ();

vVelocityBody = VRotate2D (-fOrientation, vVelocity);

}
```

Jak wskazuje nazwa tej metody, wdrożyliśmy metodę całkowania Eulera jak opisano w części 7. Integracja liniowego równania ruchu dla ciała sztywnego podąża dokładnie tymi samymi krokami, które wykorzystaliśmy do zintegrowania liniowego równania ruchu cząstek. Wszystko, czego potrzeba, to podzielenie sił agregujących działających na ciało przez masę ciała, aby uzyskać przyspieszenie ciała. Linia kodu a = vForces /fMass robi właśnie to. Zauważ tutaj, że a jest Vector, podobnie jak vForces. fMass jest skalarem, a operator / zdefiniowany w klasie Vector zajmuje się dzieleniem każdego składnika wektora vForces przez fMass i ustawianiem odpowiednich komponentów w a. Zmiana prędkości, dv, jest równa przyspieszeniom razy zmiana czasu, dt. Nowa prędkość ciała jest obliczana przez linię vVelocity + = dv. Tutaj znowu vVelocity i dv są wektorami, a operator + = dba o arytmetykę wektorową. Jest to pierwsza faktyczna integracja do tłumaczenia. Druga integracja odbywa się w następnych kilku liniach, gdzie określamy przemieszczenie ciała i nowe położenie poprzez integrację jego prędkości. Linia ds = vVelocity * dt określa przemieszczenie lub zmianę pozycji ciała, a linia vPosition + = ds oblicza nową pozycję, dodając przesunięcie do starej pozycji ciała. To jest to do tłumaczenia. Następnym zadaniem jest zintegrowanie kątowego równania ruchu, aby znaleźć nową orientację ciała. Linia aa = vMoment.z / fInertia; oblicza przyspieszenie kątowe ciała, dzieląc moment skupienia działający na ciało przez moment bezwładności masy. aa jest skalarem, podobnie jak fInertia, ponieważ jest to problem 2D. W 3D rzeczy są nieco bardziej skomplikowane, a do tego dojdziemy w

Części 11. Obliczamy zmianę prędkości kątowej, dav, skalar, przez pomnożenie aa przez wielkość kroku czasu, dt. Nowa prędkość kątowa to po prostu stara prędkość plus zmiana: vAngularVelocity.z + = dav. Zmiana orientacji jest równa nowej prędkości kątowej pomnożonej przez krok czasu: vAngularVelocity.z * dt. Zauważ, że konwertujemy plik zmiany orientacji z radianów na stopnie, ponieważ śledzimy orientację w stopniach. Tak naprawdę nie musisz, dopóki jesteś konsekwentny. Ostatnia linia w UpdateBodyEuler oblicza prędkość liniową ciała przez transformację wielkość wektora prędkości do lokalnych, współrzędne ciała. Przypomnij sobie w CalcLoads, że potrzebujemy prędkości ciała w ustalonych współrzędnych w celu obliczenia siły oporu na ciele.

Rendering

W tym prostym przykładzie, renderowanie wirtualnego poduszkowca jest tylko trochę bardziej zaangażowane niż renderowanie cząstek w przykładzie z części 8. Wszystko, co robimy, to narysować kilka połączonych linii za pomocą wywołań interfejsu Windows API owinięte w nasze własne funkcje, aby ukryć niektóre specyficznego dla systemu Windows kodu. Poniższy fragment kodu jest wszystkim, czego potrzebujemy do renderowania

hovercraft:

```
void DrawCraft(RigidBody2D craft, COLORREF clr)
{
Vector vList[5];
double wd, lg;
int i;
Vector v1;
wd = craft.fWidth;
lg = craft.fLength;
vList[0].x = lg/2; vList[0].y = wd/2;
vList[1].x = -lg/2; vList[1].y = wd/2;
vList[2].x = -lg/2; vList[2].y = -wd/2;
vList[3].x = lg/2; vList[3].y = -wd/2;
vList[4].x = lg/2*1.5; vList[4].y = 0;
for(i=0; i<5; i++)
{
v1 = VRotate2D(craft.fOrientation, vList[i]);
vList[i] = v1 + craft.vPosition;
}
DrawLine(vList[0].x, vList[0].y, vList[1].x, vList[1].y, 2, clr);
DrawLine(vList[1].x, vList[1].y, vList[2].x, vList[2].y, 2, clr);
```

```
DrawLine(vList[2].x, vList[2].y, vList[3].x, vList[3].y, 2, clr);

DrawLine(vList[3].x, vList[3].y, vList[4].x, vList[4].y, 2, clr);

DrawLine(vList[4].x, vList[4].y, vList[0].x, vList[0].y, 2, clr);

}
```

Możesz tutaj oczywiście użyć własnego kodu do renderowania i wszystkiego, co naprawdę trzeba zapłacić zwraca się szczególną uwagę na transformowanie współrzędnych konturu poduszkowca współrzędne ciała do świata. Obejmuje to obracanie współrzędnych wierzchołka z przestrzeni spiętrzonej za pomocą funkcji VRotate2D, a następnie dodanie położenia środka ciężkości poduszkowca do każdego przekształconego wierzchołka. Te linie zajmują się tą transformacją współrzędnych:

```
for(i=0; i<5; i++)

{

v1 = VRotate2D(craft.fOrientation, vList[i]);

vList[i] = v1 + craft.vPosition;

}
```

Podstawowy symulator

Sercem tej symulacji zajmuje się klasa RigidBody2D opisana wcześniej. Jednak musimy pokazać, w jaki sposób ta klasa jest używana w kontekście głównego programu. Ten symulator jest bardzo podobny do tego pokazanego w rozdziale 8, jeśli chodzi o cząstki, więc jeśli przeczytałeś ten rozdział już możesz przebijać się przez tę sekcję. Najpierw definiujemy kilka globalnych zmiennych w następujący sposób:

```
// Zmienne globalne:

int FrameCounter = 0;

RigidBody2D Craft;
```

FrameCounter zlicza liczbę kroków czasowych zintegrowanych przed aktualizacją ekranu graficznego. Liczba kroków, które możesz włączyć do symulacji, zanim zaktualizujesz wyświetlacz, to kwestia strojenia. Zobaczysz, jak to jest używane chwilowo podczas omawiania funkcji UpdateSimulation. Craft to typ RigidBody2D, który będzie reprezentował nasz wirtualny poduszkowiec. W większości przypadków Craft jest inicjowany zgodnie z pokazanym wcześniej konstruktorem RigidBody2D. Jednak jego pozycja jest początkowa, dlatego wywołujemy następującą funkcję Initialize, aby zlokalizować Craft na środku ekranu w pionie i po lewej stronie. Ustawiamy jego orientację na 0 stopni, tak aby wskazywała na prawą stronę ekranu:

```
bool Initialize (void)

{

Craft.vPosition.x = _WINWIDTH / 10;

Craft.vPosition.y = _WINHEIGHT / 2;

Craft.fOrientation = 0;
```

```
return true;

}
```

OK, teraz rozważmy UpdateSimulation, jak pokazano w poniższym fragmencie kodu. To funkcja jest wywoływana w każdym cyklu za pośrednictwem głównej pętli komunikatów programu i jest odpowiedzialna za wykonywanie odpowiednich wywołań funkcji w celu aktualizacji położenia poduszkowca i orientację, a także renderowanie sceny. Sprawdza również stany strzałki klawiatury i wykonuje odpowiednie wywołania funkcji:

```
void UpdateSimulation (void)

{

double dt = _TIMESTEP;

RECT r;

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);

// zaktualizuj symulację

Craft.UpdateBodyEuler (dt);

if (FrameCounter> = _RENDER_FRAME_COUNT)

{

// zaktualizuj wyświetlacz

ClearBackBuffer ();

DrawCraft (Craft, RGB (0,0,255));

CopyBackBufferToWindow ();

FrameCounter = 0;

} else

FrameCounter ++;

if (Craft.vPosition.x> _WINWIDTH) Craft.vPosition.x = 0;
```

```
if (Craft.vPosition.x <0) Craft.vPosition.x = _WINWIDTH;

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

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

}
```

Zmienna lokalna dt reprezentuje małą, ale skończoną ilość czasu, w sekundach którym każdy krok integracji jest podejmowany. Globalny zdefiniuj _TIMESTEP przechowuje krok czasu, który ustawiliśmy na 0,001 sekundy. Ta wartość podlega strojeniu. Pierwsza operacja, jaką wykonuje UpdateSimulation, polega na zresetowaniu stanów silników strzałowych do nieaktywnych przez wywołanie metody SetThrusters w następujący sposób:

```
Craft.SetThrusters (false, false);
```

Następnie klawiatura jest odpytywana za pomocą funkcji IsKeyDown. To jest funkcja opakowania stworzyliśmy do enkapsulacji niezbędnych wywołań interfejsu Windows API używanych do sprawdzania kluczowych stanów. Jeśli zostanie naciśnięty klawisz strzałki w górę, wówczas wywoływana jest metoda ModulateThrust RigidBody2D, jak pokazano tutaj:

```
Craft.Modulate Thrust (true);
```

Jeśli zostanie naciśnięty klawisz strzałki w dół, zostanie wywołane polecenie ModulateThrust, przekazując wartość false zamiast true

ModulateThrust wygląda następująco:

```
void RigidBody2D :: ModulateThrust (bool up)

{

double dT = up? _DTHRUST: -_ DTHRUST;

ThrustForce + = dT;

if (ThrustForce> _MAXTHRUST) ThrustForce = _MAXTHRUST;

if (ThrustForce <_MINTHRUST) ThrustForce = _MINTHRUST;

}
```

Wszystko, co robi, to zwiększanie siły ciągu śmigła o niewielką wartość, zwiększającą się lub zmniejsza go, w zależności od wartości parametru up. Wracając do UpdateSimulation, wykonujemy kilka kolejnych połączeń z IsKeyDown, sprawdzając stany lewej i prawej strzałki. Jeśli klawisz strzałki w lewo jest w dół, wywoływana jest metoda SetThrusters Rigid Body2D, przekazując wartość false jako pierwszy parametr i wartość true jako drugi parametr. Jeśli klawisz strzałki w prawo jest wyłączony, wartości

są wywrócony. SetThrusters wygląda następująco:

```
void RigidBody2D :: SetThrusters (bool p, bool s)

{

PThrust.x = 0;

PThrust.y = 0;
```

SThrust.x = 0;

SThrust.y = 0;

Jeżeli p)

PThrust.y = _STEERINGFORCE;

jeśli (s)

SThrust.y = -_STEERINGFORCE;

}

Resetuje ona wektory ciągu i steru dziobowego sterburty, a następnie ustawia je zgodnie z parametrami przekazywanymi w SetThrusters. Jeśli p jest prawdziwe, wówczas pożądany jest skręt w prawo i powstaje siła ciągu, PThrust, skierowana w stronę prawej burty. Wydaje się, że jest to przeciwieństwo tego, czego można się spodziewać, ale jest to wyrzutnia dziobowego drążka portowego, popychająca dziób poduszkowca w prawo (po prawej stronie). Podobnie, jeśli s jest prawdziwe, powstaje siła naporu, która popchnie dziób poduszkowca w lewo (port). Teraz z zarządzanymi siłami ciągu, UpdateSimulation wykonuje połączenie:

Craft.UpdateBodyEuler (dt)

UpdateBodyEuler integruje równania ruchu opisane wcześniej.

Następny segment kodu sprawdza wartość licznika klatek. Jeśli licznik ramek osiągnął określoną liczbę klatek (przechowywanych w _RENDER_FRAME_COUNT), następnie bufor tylny jest czyszczony, aby przygotować go do rysowania i ostatecznie kopiowania na ekran. W końcu ostatnie cztery linie kodu zawijają pozycję poduszkowca wokół krawędzi ekranu.

Strojenie

Prawdopodobnie będziesz chciał dostroić ten przykład, aby dobrze działał na twoim komputerze, ponieważ tego nie robiliśmy zaimplementować dowolne profilowanie dla szybkości procesora. Ponadto należy dostroić różne parametry określające zachowanie poduszkowca, aby zobaczyć, jak reaguje. Mamy to ustawione co sprawia, że poduszkowiec wykazuje miękką reakcję na skręcanie, to znaczy, po zastosowaniu siły skrętu, jednostka będzie dążyć do śledzenia w swoim oryginalnym nagłówku na chwilę, nawet gdy ziewnął. Nie zareaguje, jak by się odwrócił. Oczywiście możesz zmienić to zachowanie. Niektóre rzeczy, które proponujemy do grania, obejmują wielkość kroku czasu i różne stałe

zdefiniowaliśmy w następujący sposób:

#define _THRUSTFORCE 5.0f

#define _MAXTHRUST 10.0f

#define _MINTHRUST 0.0f

#define _DTHRUST 0.001f

#define _STEERINGFORCE 3.0f

#define _LINEARDRAGCOEFFICIENT 1.25f

_THRUSTFORCE jest początkową wielkością siły napędowej śmigła. _MAXTHRUST i _MINTHRUST ustawiają górną i dolną granicę tej siły, która jest modulowana przez użytkownika naciskając klawisze strzałek w górę i w dół. _DTHRUST to przyrostowa zmiana ciągu w odpowiedzi na naciśnięcie klawiszy strzałek w górę iw dół. _STEERINGFORCE to wielkość sił steru dziobowego. Powinieneś grać z tą wartością, aby zobaczyć, jak zmienia się zachowanie poduszkowca. Wreszcie _LINEARDRAGCOEFFICIENT to współczynnik oporu używany do obliczania oporu aerodynamicznego. Jest to kolejna dobra gra, aby zobaczyć, jak wpływa na zachowanie. Mówiąc o przeciągnięciu, położenie środka oporu zainicjowanego w konstruktorze RigidBody2D jest dobrym parametrem do zmiany, aby zrozumieć, w jaki sposób wpływa on na zachowanie poduszkowca. Wpływa na kierunkową stabilność łodzi, która wpływa na jej promień skrętu - szczególnie przy wyższych prędkościach.