XII. Symulatorze 3D ciała sztywnego

W tej części pokażemy, jak wykonać skok z 2D na 3D poprzez implementację a

Symulacja ciała sztywnego samolotu. W szczególności jest to symulacja hipotetycznego modelu samolotu, który omówimy obszernie w części 15. Ten samolot ma typową konfigurację z dużymi skrzydłami do przodu, pionowymi elewatorami, pojedynczym pionowym ogonem i płaskimi klapami zamontowanymi na skrzydłach. Podobnie jak w przypadku symulatora 2D w poprzednich rozdziałach, skoncentrujemy się na kodzie implementującym część fizyki symulatora, a nie na aspektach GUI związanych z platformą. Podobnie jak w 2D, istnieją cztery główne elementy tej symulacji 3D - model, integrator, dane wejściowe użytkownika i rendering. Pamiętaj, że model odnosi się do twojej idealizacji rzeczy - w tym przypadku samolotu - którą próbujesz symulować, podczas gdy integrator odwołuje się do metody, za pomocą której integrujesz równania różniczkowe ruchu. Te dwa elementy zajmują się większością fizyki symulacji. Elementy wejściowe i renderujące użytkownika odnoszą się do tego, w jaki sposób użytkownik może wchodzić w interakcje i wyświetlać symulację. W tej symulacji światowy układ współrzędnych ma dodatnią oś X skierowaną w stronę ekranu, jej dodatnią oś Y wskazuje na lewą stronę ekranu, a dodatnia oś Z skierowana jest w górę. Również lokalny lub ustalony przez ciało układ współrzędnych ma dodatnią oś X skierowaną w przód samolotu, jego dodatnia oś Y wskazuje na lewą stronę, a jej dodatnia oś Z skierowana jest w górę. Ponieważ jest to trójwymiarowa symulacja samolotu, kiedy już ją uruchomisz, będziesz mógł latać w dowolnym kierunku, zapętleniu, bankowości, nurkowaniu i wspinaniu się, lub wykonywać dowolny inny manewr akrobacyjny, jakiego pragniesz.

Model

Jednym z najważniejszych aspektów tej symulacji jest model lotu. Całą część 15 poświęcimy dyskusji na temat fizyki stojącej za tym modelem lotu, więc nie uwzględnimy tej dyskusji tutaj, poza wprowadzeniem kilku kluczowych bitów kodu. Aby zaimplementować model lotu, najpierw trzeba przygotować sztywną strukturę ciała, aby zamknąć wszystkie dane wymagane do całkowitego zdefiniowania stanu sztywnego ciała w dowolnej chwili podczas symulacji. W tym celu zdefiniowaliśmy strukturę o nazwie RigidBody:

typedef struct _RigidBody {

float fMass; // masa całkowita

Matrix3x3 mInertia; // masowy moment bezwładności

// we współrzędnych ciała

Matrix3x3 mInertiaWersja; // 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 we współrzędnych ciała

Vector vElertAngles; // Kąty Eulera we współrzędnych ciała

float fSpeed; // prędkość (wielkość prędkości)

Quaternion qOrientation; // orientacja we współrzędnych ziemi

Vector vForces; // całkowita siła na ciele

Vector vMoments; // całkowity moment (moment obrotowy) na ciele

} RigidBody, * pRigidBody;

Zauważysz, że jest bardzo podobny do struktury RigidBody2D, której używaliśmy w 2D symulacje poduszkowca. Jedną znaczącą różnicą jest jednak to, że w przypadku 2D orientacja była pojedynczą wartością zmiennoprzecinkową, a teraz w 3D jest kwaternionem typu Quaternion. Omówiliśmy zastosowanie kwaternionów do śledzenia orientacji ciała sztywnego w poprzedniej części. Następnym krokiem do zdefiniowania modelu lotu jest przygotowanie funkcji inicjalizacyjnej w celu zainicjowania samolotu na początku symulacji. W tym celu przygotowaliśmy funkcję o nazwie InitializeAirplane:

RigidBody Airplane; // globalna zmienna reprezentująca samolot

.

.

.

void InitializeAirplane (void)

{

float iRoll, iPitch, iYaw;

// Ustaw pozycję początkową

Airplane.vPosition.x = -5000,0f;

Airplane.vPosition.y = 0,0f;

Airplane.vPosition.z = 2000.0f;

// Ustaw prędkość początkową

Airplane.vVelocity.x = 60.0f;

Airplane.vVelocity.y = 0.0f;

Airplane.vVelocity.z = 0.0f;

Airplane.fSpeed = 60.0f;

// Ustaw początkową prędkość kątową

Airplane.vAngularVelocity.x = 0.0f;

Airplane.vAngularVelocity.y = 0.0f;

Airplane.vAngularVelocity.z = 0.0f;

// Ustaw początkowy ciąg, siły i momenty

Airplane.vForces.x = 500.0f;

Airplane.vForces.y = 0.0f;

Airplane.vForces.z = 0.0f;

ThrustForce = 500.0;

```
Airplane.vMoments.x = 0.0f;

Airplane.vMoments.y = 0,0f;

Airplane.vMoments.z = 0.0f;

// Zeruj prędkość we współrzędnych przestrzeni ciała

Airplane.vVelocityBody.x = 0.0f;

Airplane.vVelocityBody.y = 0.0f;

Airplane.vVelocityBody.z = 0.0f;

// Najpierw ustaw na false,

// możesz kontrolować później za pomocą klawiatury

Stalling = false;

Klapy = false;

// Ustaw początkową orientację

iRoll = 0,0f;

iPitch = 0.0f;

iYaw = 0.0f;

Airplane.qOrientation = MakeQFromEulerAngles (iRoll, iPitch, iYaw);

// Teraz idź dalej i obliczyć właściwości masy samolotu

CalcAirplaneMassProperties ();

}
```

Ta funkcja ustawia początkową lokalizację, prędkość, położenie i siłę ciągu samolotu i oblicza jego właściwości masy, wykonując połączenie z CalcAirplaneMassProper . Znacznie więcej tej funkcji zobaczysz w rozdziale 15, więc nie pokażemy całości rzecz tutaj. Chcemy wskazać fragment kodu, który wyraźnie różni się od tego, co robisz w symulacji 2D, i to jest obliczenie momentu tensora bezwładności:

```
void CalcAirplaneMassProperties (void)

{

.

.

.

// Teraz oblicz momenty i produkty bezwładności dla

// połączone elementy.

// (Ta macierz bezwładności (tensor) ma współrzędne ciała)
```

```
Ixx = 0; Iyy = 0; Izz = 0;
Ixy = 0; Ixz = 0; Iyz = 0;
dla (i = 0; i <8; i ++)
{
Ixx + = Element [i] .vLocalInertia.x + Element [i] .fMass *
(Element [i] .vCGCoords.y * Element [i] .vCGCoords.y +
Element [i] .vCGCoords.z * Element [i] .vCGCoords.z);
Iyy + = Element [i] .vLocalInertia.y + Element [i] .fMass *
(Element [i] .vCGCoords.z * Element [i] .vCGCoords.z +
Element [i] .vCGCoords.x * Element [i] .vCGCoords.x);
Izz + = Element [i] .vLocalInertia.z + Element [i] .fMass *
(Element [i] .vCGCoords.x * Element [i] .vCGCoords.x +
Element [i] .vCGCoords.y * Element [i] .vCGCoords.y);
Ixy + = Element [i] .fMass * (Element [i] .vCGCoords.x *
Element [i] .vCGCoords.y);
Ixz + = Element [i] .fMass * (Element [i] .vCGCoords.x *
Element [i] .vCGCoords.z);
Iyz + = Element [i] .fMass * (Element [i] .vCGCoords.y *
Element [i] .vCGCoords.z);
}
// W końcu ustaw masę samolotu i jego macierz bezwładności i weź
// odwrotność macierzy bezwładności
Airplane.fMass = mass;
Airplane.mInertia.e11 = Ixx;
Airplane.mInertia.e12 = -Ixy;
Airplane.mInertia.e13 = -Ixz;
Airplane.mInertia.e21 = -Ixy;
Airplane.mInertia.e22 = Iyy;
Airplane.mInertia.e23 = -Iyz;
Airplane.mInertia.e31 = -Ixz;
Airplane.mInertia.e32 = -Iyz;
```

```
Airplane.mInertia.e33 = Izz;

Airplane.mInertiaInverse = Airplane.mInertia.Inverse ();

}
```

Samolot jest modelowany przez wiele elementów, z których każdy reprezentuje inną część konstrukcji samolotu - na przykład ogon steru, windy, skrzydła i kadłub. Określony tutaj kod przejmuje właściwości masy każdego elementu i łączy je, używając technik opisanych w Części 7, aby wymyślić połączony tensor bezwładności dla całego samolotu. Ważna różnica między tymi obliczeniami w symulacji 3D i symulacji 2D polega na tym, że bezwładność jest tensorem, a w 2D jest pojedynczym skalarem. InitializeAirplane jest wywoływana na samym początku programu. Stwierdziliśmy, że wygodnie jest nawiązać połączenie zaraz po utworzeniu głównego okna aplikacji. Ostatnia część modelu lotu ma związek z obliczaniem sił i momentów, które działają na samolot w danej chwili w czasie symulacji. Podobnie jak w symulacji poduszkowca 2D, bez tego rodzaju funkcji, samolot nic nie zrobi. W tym celu zdefiniowaliśmy funkcję o nazwie CalcAirplaneLoads, która jest wywoływana na każdym etapie symulacji. Ta funkcja opiera się na kilku innych funkcjach - LiftCoefficient, DragCoefficient, RudderLiftCoefficient i Rudder DragCoefficient. W większości przypadków kod zawarty w CalcAirplaneLoads jest podobny do kodu, który widziałeś w funkcji CalcLoads symulacji poduszkowca. CalcAirplanLoads jest nieco bardziej zaangażowany, ponieważ samolot jest modelowany przez szereg elementów, które przyczyniają się do całkowitego podniesienia i oporu samolotu. Jest jeszcze inna różnica, o której tutaj mówiliśmy:

```
void CalcAirplaneLoads (void)

{

.

.

.

// Konwertuj siły z przestrzeni modelu na przestrzeń Ziemi

Airplane.vForces = QVRotate (Airplane.qOrientation, Fb);

// Zastosuj grawitację (g jest zdefiniowane jako -32.174 ft / s ^ 2)

Airplane.vForces.z + = g * Airplane.fMass;

.

.

.

}
```

Prawie wszystkie siły działające na samolot są najpierw obliczane w ustalonych na stałe współrzędnych, a następnie przekształcane w ustalone na ziemi współrzędne przed przyłożeniem siły grawitacji. Konwersja współrzędnych odbywa się za pomocą funkcji QVRotate, która obraca wektor siły w oparciu o aktualną orientację samolotu, reprezentowaną przez kwaternion.

Całkowanie

Teraz, gdy kod służący do definiowania, inicjowania i obliczania obciążeń w samolocie jest kompletny, musisz opracować kod tak, aby faktycznie zintegrować równania ruchu, aby symulacja mogła rozwijać się w czasie. Pierwszą rzeczą, którą musisz zrobić, to zdecydować się na schemat integracji, z którego chcesz skorzystać. W tym przykładzie zdecydowaliśmy się na podstawową metodę Eulera. Omówiliśmy już kilka lepszych metod w części 7. Przechodzimy tutaj z metodą Eulera, ponieważ jest to proste i nie chcieliśmy, aby kod był zbyt skomplikowany, grzebiąc klucz, który musimy wskazać. W praktyce lepiej jest użyć jednej z innych metod, które omawiamy w części 7 zamiast metody Eulera. Mając to na uwadze, przygotowaliśmy funkcję o nazwie StepSimulation, która obsługuje całą integrację niezbędną do rzeczywistej propagacji symulacji:

```
void StepSimulation (float dt)

{

// Najpierw zajmij się tłumaczeniem:

// (Jeśli to ciało było cząstką, to wszystko, co musisz zrobić.)

Wektor Ae;

// oblicz wszystkie siły i momenty na samolocie:

CalcAirplaneLoads ();

// oblicz przyspieszenie samolotu w przestrzeni Ziemi:

Ae = Airplane.vForces / Airplane.fMass;

// obliczyć prędkość samolotu w przestrzeni Ziemi:

Airplane.vVelocity + = Ae * dt;

// obliczyć pozycję samolotu w przestrzeni Ziemi:

Airplane.vPosition + = Airplane.vVelocity * dt;

// Teraz zajmij się obrotami:

spławik mag;

// oblicz prędkość kątową samolotu w przestrzeni ciała:

Airplane.vAngularVelocity + = Airplane.mInertiaInverse *

(Airplane.vMoments -

(Airplane.vAngularVelocity ^

(Samolot.mInertia *

Airplane.vAngularVelocity)))

* dt;

// obliczyć nowy kwaternion rotacyjny:

Airplane.qOrientation + = (Airplane.qOrientation *

Airplane.vAngularVelocity) *
```

(0,5f * dt);

// teraz znormalizuj orientacyjne kwaternienie:

mag = Airplane.qOrientation.Magnitude ();

if (mag! = 0)

Airplane.qOrientation / = mag;

// oblicz prędkość w przestrzeni ciała:

// (będziemy potrzebować tego do obliczenia siły podnoszenia i przeciągania)

Airplane.vVelocityBody = QVRotate (~ Airplane.qOrientation,

Airplane.vVelocity);

// oblicz prędkość powietrza:

Airplane.fSpeed = Airplane.vVelocity.Magnitude ();

// zdobądź kąty Eulera dla naszych informacji

Wektor u;

u = MakeEulerAnglesFromQ (Airplane.qOrientation);

Airplane.vEulerAngles.x = u.x; // rolka

Airplane.vEulerAngles.y = u.y; // wysokość

Airplane.vEulerAngles.z = u.z; // odchylenie

}

Pierwszą rzeczą, którą robi StepSimulation, jest wywołanie kalkulatora CalcAirplaneLoads obciążenia działającego na samolot w chwili obecnej w czasie. Następnie StepSimulation oblicza liniowe przyspieszenie samolotu na podstawie obciążeń prądowych. Następnie funkcja integruje, używając metody Eulera, jeden raz, aby obliczyć liniową prędkość samolotu, a następnie drugi raz, aby obliczyć położenie samolotu. Jak już skomentowaliśmy w kodzie, jeśli symulujesz cząstkę, to wszystko, co musiałbyś zrobić; jednak ponieważ nie jest to cząstka, musisz obsługiwać ruch kątowy. Pierwszym krokiem w obsłudze ruchu kątowego jest obliczenie nowej prędkości kątowej w tym momencie, z wykorzystaniem integracji Eulera, w oparciu o wcześniej obliczone momenty działające na samolot i jego właściwości masowe. Robimy to we współrzędnych ciała, stosując następujące równanie ruchu kątowego, ale przepisane w celu rozwiązania dla dω:

$$\sum \mathbf{M}_{cg} = d\mathbf{H}_{cg}/dt = \mathbf{I}\,(d\boldsymbol{\omega}/dt) + (\boldsymbol{\omega} \times (\mathbf{I}\,\boldsymbol{\omega}))$$

Następnym krokiem jest ponowne zintegrowanie, aby zaktualizować orientację samolotu, która jest wyrażona jako kwaternion. Tutaj musisz użyć równania różniczkowego dotyczącego orientacyjnego kwaternionu do prędkości kątowej, którą pokazaliśmy w części 11:

dq / dt = (1/2) ω q

Następnie, aby wymusić ograniczenie, że owa orientacja kwaterunkowa jest kwaternionem jednostkowym, funkcja idzie naprzód i normalizuje dwukierunnik orientacji. Ponieważ prędkość

liniowa została wcześniej obliczona w globalnych współrzędnych (stały układ współrzędnych), a ponieważ CalcAirplaneLoads potrzebuje prędkości w ustalonym (obrotowym) układzie współrzędnych, funkcja idzie naprzód i obraca wektor prędkości, przechowując wektor ustalony przez ciało w elemencie vVelocityBody struktury RigidBody. Jest to wykonywane dla wygody i wykorzystuje QVRotate dla quaternion funkcji obrotu, aby obrócić wektor w oparciu o aktualną orientację samolotu. Zauważ, że używamy koniugatu orientacyjnego kwaternionu, ponieważ teraz obracamy się od współrzędnych globalnych do współrzędnych ciała. Jako kolejną wygodę obliczamy prędkość powietrza, która jest po prostu wielkością wektora prędkości liniowej. Służy do zgłaszania prędkości powietrza na pasku tytułu głównego okna. Na koniec trzy rzuty, odstęp i odchylenie Eulera są pobierane z kwadratu orientacji, aby można je było również zgłosić na pasku tytułu okna głównego. Nie zapominaj, że StepSimulation musi być wywołany raz na cykl symulacji.

Kontrola lotów

W tym momencie symulacja nadal nie zadziała zbyt dobrze, ponieważ nie wdrożyłeś sterowania lotem. Sterowanie lotem pozwala na interakcję z różnymi powierzchniami sterującymi samolotu, aby faktycznie latać samolotem. Będziemy używać klawiatury jako głównego urządzenia wejściowego do sterowania lotem. Pamiętaj, że w symulacji opartej na fizyce, takiej jak ta, nie kontrolujesz bezpośrednio ruchu samolotu; kontrolujesz tylko, jak różne siły są przykładane do samolotu, które następnie, poprzez integrację w czasie, wpływają na ruch samolotu. W tej symulacji za pomocą klawiszy strzałek symuluje się drążek lotu. Strzałka w dół pociąga za kij, unosząc nos; strzałka w górę popycha drążek do przodu, powodując zanurkowanie nosa; lewa strzałka przesuwa płaszczyznę w lewo (strona portu); a prawa strzałka przesuwa płaszczyznę w prawo (po prawej stronie). Klawisz X stosuje lewe działanie steru, aby narysować płaszczyznę w kierunku lewej strony, podczas gdy klawisz C stosuje prawą akcję steru, aby spowodować odchylenie nosa w prawo. Ciąg jest sterowany za pomocą klawiszy A i Z. Klawisz A zwiększa siłę ciągu śruby o 100 funtów, a klawisz Z zmniejsza ciąg przez 100 funtów. Minimalny ciąg wynosi 0, a maksymalna dostępna siła ciągu 3000 funtów. Klawisz F aktywuje klapy do lądowania, aby zwiększyć siłę nośną przy niskiej prędkości, a klawisz D dezaktywuje klapy. Kontrolujemy skok przez odchylanie klap na tylnych windach; na przykład, aby pochylić dziób w górę, odbijamy tylne klapy windy do góry (to jest, tylna krawędź dźwigu jest uniesiona względem krawędzi natarcia). Kontrolujemy przechył w tej symulacji, stosując różnie klapy; na przykład, aby przeturlać się w prawo, odbijamy prawą klapę w górę, a lewą klapę w dół. Na koniec kontrolujemy zbaczanie, odchylając pionową ster ogonową; na przykład, aby odchylić w lewo, odbijamy tylną krawędź steru ogonowego w lewo. Przygotowaliśmy kilka funkcji do obsługi kontrolek lotu, które powinny być wywoływane za każdym razem, gdy użytkownik naciska jeden z klawiszy sterowania lotem. Istnieją dwie funkcje ciągu śruby:

```
void IncThrust (void)

{

ThrustForce + = _DTHRUST;

if (ThrustForce> _MAXTHRUST)

ThrustForce = _MAXTHRUST;

}

void DecThrust (void)

{
```

```
ThrustForce - = _DTHRUST;

if (ThrustForce <0)

ThrustForce = 0;

}
```

IncThrust po prostu zwiększa siłę ciągu przez _DTHRUST, sprawdzając, czy nie przekracza _MAXTHRUST. Zdefiniowaliśmy _DTHRUST i _MAXTHRUST w następujący sposób:

```
#define _DTHRUST 100.0f

#define _MAXTHRUST 3000.0f
```

Z drugiej strony DecThrust zmniejsza siłę ciągu, sprawdzając, czy nie spada ona poniżej 0. Aby kontrolować zbaczanie, przygotowaliśmy trzy funkcje, które manipulują sterem:

```
void LeftRudder (void)

{

Element [6] .fIncidence = 16;

}

void RightRudder (void)

{

Element [6] .fIncidence = -16;

}

void ZeroRudder (void)

{

Element [6] .fIncidence = 0;

}
```

LeftRudder zmienia kąt padania elementu [6], pionowego steru ogonowego, na 16 stopni, natomiast RightRudder zmienia kąt padania na -16 stopni. ZeroRudder centruje ster na 0 stopni. Lotki lub klapy manipulują tymi funkcjami, aby sterować rolką:

```
void RollLeft (void)

{

Element [0] .iFlap = 1;

Element [3] .iFlap = -1;

}

void RollRight (void)

{
```

```
Element [0] .iFlap = -1;

Element [3] .iFlap = 1;

}

void ZeroAilerons (void)

{

Element [0] .iFlap = 0;

Element [3] .iFlap = 0;

}
```

RollLeft odchyla lotkę portu, znajdującą się na sekcji skrzydła portu (Element [0]), w górę, i prawą burtę, znajdującą się na prawej części skrzydła (Element [3]), w dół. RollRight robi dokładnie odwrotnie, a ZeroAilerons resetuje klapy z powrotem do ich niezdefiniowanych pozycji. Zdefiniowaliśmy jeszcze jeden zestaw funkcji do sterowania windami na rufie, aby kontrolować wysokość tonu:

```
void PitchUp (void)

{

Element [4] .iFlap = 1;

Element [5] .iFlap = 1;

}

void PitchDown (void)

{

Element [4] .iFlap = -1;

Element [5] .iFlap = -1;

}

void ZeroElevators (void)

{

Element [4] .iFlap = 0;

Element [5] .iFlap = 0;

}
```

Element [4] i Element [5] to windy. PitchUp odchyla ich klapy w górę, i PitchDown odchyla klapy w dół. ZeroElevators resetuje ich klapy z powrotem do swoich niezdefiniowane pozycje. Wreszcie, istnieją jeszcze dwie funkcje kontrolujące klapy lądowania:

```
void FlapsDown (void)

{

Element [1] .iFlap = -1;
```

```
Element [2] .iFlap = -1;

Klapy = true;

}

void ZeroFlaps (void)

{

Element [1] .iFlap = 0;

Element [2] .iFlap = 0;

Klapy = false;

}
```

Klapy do lądowania są zamontowane na wewnętrznych sekcjach skrzydła, lewej i prawej burcie, które są Element [1] i Element [2]. FlapsDown odchyla klapy w dół, podczas gdy zero Klapy przywracają je do nieodwróconej pozycji. Tak jak powiedzieliśmy, funkcje te powinny zostać wywołane, gdy użytkownik naciska klawisze sterowania lotem. Co więcej, należy je wywołać przed wywołaniem metody StepSimulation, tak aby można je było uwzględnić w obliczeniach siły i momentu kroku bieżącego. Sekwencja wywołań powinna wyglądać mniej więcej tak:

```
.

.

.

ZeroRudder ();

ZeroAilerons ();

ZeroElevators ();

// zniżka

if (IsKeyDown (VK_UP))

PitchDown ();

// Podnieś

if (IsKeyDown (VK_DOWN))

PitchUp ();

// przewiń w lewo

if (IsKeyDown (VK_LEFT))

RollLeft ();

// zwróć w prawo

if (IsKeyDown (VK_RIGHT))

RollRight ();
```

```
// Zwiększenie ciągu

if (IsKeyDown (0x41)) // A

IncThrust ();

// Zmniejsz ciąg

if (IsKeyDown (0x5A)) // Z

DecThrust ();

// Odejdź w lewo

if (IsKeyDown (0x58)) // x

LeftRudder ();

// zbaczaj w prawo

if (IsKeyDown (0x43)) // c

RightRudder ();

// klapy lądowania w dół

if (IsKeyDown (0x46)) // f

FlapsDown ();

// klapy lądujące w górę

if (IsKeyDown (0x44)) // d

ZeroFlaps ();

StepSimulation (dt);
```

. Przed wywołaniem funkcji StepSimulation sprawdzamy każdy z kluczy sterujących lotem, aby sprawdzić, czy jest naciśnięcie. Jeśli tak, wywoływana jest odpowiednia funkcja. Funkcja IsKeyDown, która sprawdza, czy dany klawisz jest wciśnięty, wygląda tak w implementacji Windows:

```
BOOL IsKeyDown (short KeyCode)

{

KRÓTKI powrót;

retval = GetAsyncKeyState (KeyCode);

if (HIBYTE (retval))

return TRUE;

return FALSE;

}
```

Ważne jest, aby pamiętać, że klucze są sprawdzane asynchronicznie ponieważ możliwe jest, że w danym momencie zostanie naciśnięty więcej niż jeden klawisz i muszą one być obsługiwane

jednocześnie zamiast pojedynczo (tak jak w przypadku standardowej funkcji przetwarzania komunikatów Windows). Dodanie kodu kontroli lotu w znacznym stopniu uzupełnia fizykalną część symulacji. Do tej pory masz ukończony model, integrator oraz elementy wejściowe użytkownika lub elementy sterowania lotem. Pozostaje tylko ustawić główne okno aplikacji i faktycznie rysować coś, co reprezentuje to, co symulujesz. Zostawimy tę część dla ciebie lub możesz spojrzeć na przykład, który zamieściliśmy na stronie książki, aby zobaczyć, co zrobiliśmy na komputerze z Windows.