XI. Obrót w symulatorze 3D ciał sztywnych

Zasadnicza różnica między cząstkami a bryłami sztywnymi polega na tym, że nie możemy ignorować rotacji ciał sztywnych. Dotyczy to zarówno ciał sztywnych 2D, jak i 3D. W dwóch wymiarach dość łatwo jest wyrazić orientację sztywnego ciała; potrzebujesz tylko jednego skalaru do przedstawienia obrotu ciała wokół pojedynczej osi. Jednak w trzech wymiarach istnieją trzy główne osie współrzędnych, wokół których może obracać się sztywny korpus. Co więcej, sztywny korpus w trzech wymiarach może obracać się wokół dowolnej osi, niekoniecznie jednej z osi współrzędnych. W dwóch wymiarach mówimy, że ciało sztywne ma tylko jeden rotacyjny stopień swobody, podczas gdy w trzech wymiarach mówimy, że ciało sztywne ma trzy obrotowe stopnie swobody. Może to prowadzić do wniosku, że w trzech wymiarach, musisz mieć trzy wielkości skalarne, które reprezentują obrót ciała. Rzeczywiście, jest to minimalne wymaganie i prawdopodobnie już znasz zestaw kątów, które reprezentują orientację sztywnego ciała w 3D - a mianowicie, trzy kąty Eulera (rzut, pochylenie i odchylenie), o których będziemy rozmawiać w części 15. Te trzy kąty - roll, pitch i odchylenie - są bardzo intuicyjne i łatwe do wizualizacji. Na przykład, w samolocie nos skacze w górę lub w dół, samolot toczy się (lub brzegi) w lewo lub w prawo, a odchylenie (lub kurs) zmienia się w lewo lub w prawo. Niestety, istnieje problem z użyciem tych trzech kątów Eulera w symulacjach ciała sztywnego. Problem jest numeryczny, który pojawia się, gdy kąt nachylenia osiąga plus minus 90 stopni (π / 2). Kiedy tak się dzieje, przechylenie i odchylenie stają się niejednoznaczne. Co gorsza, równania kątowe ruchu zapisane w kategoriach kątów Eulera zawierają terminy obejmujące cosinus kąta pochylenia w mianowniku, co oznacza, że gdy kąt nachylenia wynosi plus lub minus 90 stopni, równania stają się pojedyncze (tj. 0). Jeśli tak się stanie w twojej symulacji, wyniki będą co najmniej nieprzewidywalne. Biorąc pod uwagę ten problem z kątami Eulera, musisz użyć innych metod śledzenia orientacji w swojej symulacji. Omówimy dwa takie środki w tym rozdziale - w szczególności macierze rotacyjne i kwaternionów. Praktycznie każda książka komputerowa, którą przeczytaliśmy, zawiera rozdział lub rozdział dotyczący używania matryc obrotowych. Znacznie mniej dyskutuje o kwaterach, ale jeśli znasz kwadratury, to prawdopodobnie w tym samym kontekście, co macierzy obrotu - czyli w jaki sposób są używane do obracania punktów 3D, obiektów, scen i punktów widzenia. W symulacji trzeba jednak uzyskać nieco więcej macierzy lub kwater kwaterowych z rotacji i wykorzystywać je w innym kontekście niż do tego, do czego jesteś przyzwyczajony. W szczególności musisz śledzić orientację ciała w przestrzeni, a ponadto zmianę orientacji w czasie. W tym świetle omówimy macierze rotacyjne i kwaternionów.

Matryce obrotowe

Macierz rotacji jest macierzą 3 × 3, która po pomnożeniu przez punkt lub wektor daje wyniki w obrocie tego punktu wokół osi, co daje nowy zestaw współrzędnych. Możesz obracać punkty wokół osi w jednym układzie współrzędnych lub możesz użyć macierzy obrotu do zamiany punktów z jednego układu współrzędnych na inny, gdzie jeden jest obracany względem drugiego. Obracanie wektora przez macierz obrotu jest zwykle zapisywane w następujący sposób: jeśli v jest wektorem, a R jest macierzą obrotu, to v 'jest v obrócone przez R zgodnie ze wzorem:

$$v' = R\ v$$

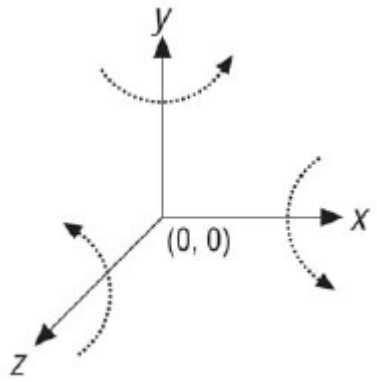
Możesz łączyć wiele macierzy rotacji odzwierciedlających wielokrotne rotacje w jedną macierz obrotu przy użyciu zwykłego mnożenia macierzy. Jeśli macierze obrotu są wyrażone w postaci stałych, globalnych współrzędnych, wówczas są one łączone w następujący sposób:

$$R_c = R_1 R_2$$

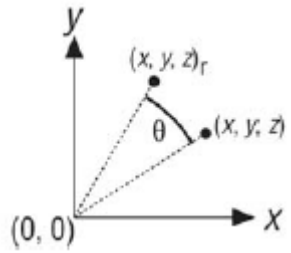
Tutaj R_c jest połączoną macierzą rotacji odzwierciedlającą obrót najpierw przez R1, a następnie przez R2. Jeśli macierze obrotu są wyrażone w postaci obrotowych, ustalonych na stałe współrzędnych, wówczas są one łączone w odwrotnej kolejności w następujący sposób:

$R_c = R_2R_1$

Nie przejdziemy do dowodu tej zależności, ale powodem, dla którego różni się ona w zależności od tego, jak zdefiniowaliście macierze rotacyjne, jest to, że macierze rotacji określone w ustalonych współrzędnych nie są dotknięte samym obrotem, ponieważ osie współrzędnych pozostają niezmienione. Z drugiej strony, jeśli macierze obrotu są zdefiniowane względem układu współrzędnych, który obraca się z powodu sekwencyjnego nakładania macierzy obrotu, to wszystkie macierze obrotu po pierwszym będą podlegały wpływowi, ponieważ zostały one najpierw zdefiniowane względem pierwotnego stanu współrzędnych system - to znaczy zanim została zastosowana pierwsza macierz obrotu. Oznacza to, że kolejne macierze rotacji muszą zostać skorygowane, aby odzwierciedlały nowy system, na który ma wpływ poprzedni obrót, zanim zostaną prawidłowo zastosowane. Innymi słowy, musisz obrócić R_2 o R_1, aby uzyskać nowy R_2 przed zastosowaniem go. Wszystko to dzieje się w taki sposób, aby odwrócić kolejność mnożenia macierzy obrotu, gdy są one zdefiniowane w obrotowym układzie współrzędnych. Rysunek 11-1 przedstawia praworęczny układ współrzędnych ilustrujący kierunki dodatniej rotacji wokół każdej osi współrzędnych.



Rozważmy obrót wokół osi Z, gdzie punkt pokazany na rysunku 11-2 jest obrócony o kąt θ.



Współrzędne punktu przed rotacją to (x, y, z) i po obrocie współrzędnymi są (xr, yr, zr). Obrócone współrzędne odnoszą się do oryginalnych współrzędnych i kąta obrotu o:

$xr = x \cos \theta - y \sin \theta$

$y_r = x \sin \theta + y \cos \theta$

$z_r = z$

Zauważ, że ponieważ punkt obraca się wokół osi Z, jego współrzędna z pozostaje niezmieniona. Aby zapisać to w notacji wektorowej, v'= R v, niech v = [x y z] i niech R będzie macierzą:

$$\begin{vmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Tutaj v'będzie nowym, obróconym wektorem, v' = [x_r r_r z_r]. Obrót wokół osi X i Y jest podobny do osi Z; jednakże w tych przypadkach współrzędne X i Y pozostają niezmienne podczas obrotów odpowiednio wokół każdej osi. Patrząc na rotację wokół każdej osi osobno, otrzymasz trzy macierze obrotu podobne do tego, które właśnie pokazaliśmy dla rotacji wokół osi Z. W przypadku obrotu wokół osi X macierz jest:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{vmatrix}$$

A dla obrotu wokół osi Y macierz jest:

$$\begin{vmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{vmatrix}$$

Są to macierze rotacyjne, które zazwyczaj widzisz w tekstach graficznych komputera w kontekście przekształceń macierzowych, takich jak tłumaczenie, skalowanie i obracanie. Możesz połączyć wszystkie trzy macierze w jedną macierz rotacji, aby przedstawić kombinacje rotacji wokół każdej osi współrzędnych, używając mnożenia macierzy, jak wspomniano wcześniej. W symulacjach ciała sztywnego można użyć macierzy obrotu, aby przedstawić orientację sztywnego ciała. Innym sposobem, aby o tym pomyśleć, jest matryca obrotu, która po nałożeniu na niezablokowany sztywny korpus wyrównany ze stałym globalnym układem współrzędnych, obróci współrzędne sztywnego ciała tak, aby przypominały aktualną orientację ciała w danym momencie. Prowadzi to do innego ważnego rozważenia przy stosowaniu macierzy obrotowych do śledzenia orientacji w symulacjach ciała sztywnego: fakt, że macierz obrotu będzie funkcją czasu. Po ustawieniu początkowej macierzy obrotu dla sztywnego ciała, nigdy nie będziesz bezpośrednio obliczał jej ponownie z kątów orientacji; zamiast tego siły i momenty przyłożone do sztywnego ciała zmienią kątową prędkość ciała, podobnie powodując niewielkie zmiany orientacji w każdym kroku czasowym podczas symulacji. W ten sposób można zauważyć, że konieczne jest powiązanie macierzy obrotu z prędkością kątową, aby można było odpowiednio zaktualizować orientację. Potrzebna formuła jest następująca:

$dR/dt = \Omega R$

Tutaj Ω jest symetryczną macierzą ukośną zbudowaną z elementów wektora prędkości kątowej w następujący sposób:

$$\Omega = \begin{vmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{vmatrix}$$

Niezależnie od rygorystycznego dowodu tego związku, łatwo jest dostrzec jego piękno, co oznacza, że można rozróżnić macierz obrotu, po prostu macierz mnożąc ją przez prędkość kątową (w postaci Ω). W symulacji poznasz początkową macierz rotacji i obliczysz prędkość kątową na każdym kroku; w ten sposób można łatwo rozwijać lub integrować macierz rotacji. Powinieneś być w stanie zauważyć, że skoro tylko jednoznacznie obliczysz macierz rotacji i zaktualizujesz ją macierzą mnożąc, nie będziesz musiał używać kosztownych obliczeniowych funkcji trygonometrycznych podczas każdego kroku czasowego. Ponadto unikasz problemu osobliwości wymienionego we wstępie do tej części. Powinno być również oczywiste, że zyskujesz te świadczenia za pewną cenę. Najpierw musisz zmierzyć się z dziewięcioma parametrami w macierzy obrotu (każdy element w macierzy obrotu 3 × 3), aby przedstawić trzy kątowe stopnie swobody. Po drugie, aby to zrobić, trzeba nałożyć ograniczenia na macierz rotacji; konkretnie, musisz wymusić ograniczenie, że macierz jest ortogonalna z wyznacznikiem 1, tak aby spełniała następujące warunki (każda kolumna w macierzy reprezentuje wektor jednostkowy i wszystkie są prostopadłe do siebie):

$R^TR = I$

Tutaj R^T jest transpozycją R, a ja jest matrycą tożsamości. Z powodu błędów numerycznych, takich jak zaokrąglenie i obcięcie, będziesz musiał bardzo często wymuszać to ograniczenie w swojej symulacji. W przeciwnym razie twoja macierz rotacji zrobi więcej niż obrót twoich obiektów, może też przeskalować je lub przetłumaczyć. Zamiast zajmować się dziewięcioma parametrami i próbować ograniczyć sześć stopni swobody, tak aby można było reprezentować tylko te trzy, można zastosować alternatywne podejście, które pozwala zachować zalety macierzy rotacyjnych, ale za niższą cenę. Ta alternatywa, kwaternion, jest tematem następnej sekcji.

Kwaterniony

Kwaterniony są w pewnym sensie dziwactwem matematycznym. Opracowano je ponad 100 lata temu przez Williama Hamiltona poprzez jego pracę w złożonej (wyimaginowanej) matematyce, ale znalazły bardzo mało praktycznego zastosowania. Kwaternion to ilość, rodzaj jak wektor, ale składa się z czterech składników. Zazwyczaj jest napisane w formie:

$q = q_0 + q_x i + q_y j + q_z k$

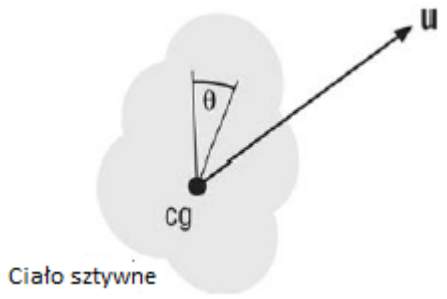
Kwaternion jest w rzeczywistości przestrzenią czterowymiarową w złożonej przestrzeni i niestety nie nadaje się do wizualizacji. Nie martw się jednak: nasze wykorzystanie kwaternionów do reprezentowania orientacji w trzech wymiarach pozwala nam przywiązać do nich fizyczne znaczenie, jak zobaczysz za chwilę. Szczególnie interesujące dla nas jest to, co jest znane jako jednostkowe kwaternionie spełniające następujące warunki:

$q_0^2 + q_x^2 + q_y^2 + q_z^2 = 1$

Jest to analogiczne do znormalizowanego lub jednostkowego wektora. Możesz również napisać kwaternion w postaci $q = [q_0, v]$, gdzie v jest wektorem, $q_x i + qy j + qz k$, a q_0 jest skalarem. W kontekście obrotu v oznacza kierunek, w którym znajduje się oś obrotu. Dla danego obrotu, θ, o arbitralnej osi reprezentowanej przez wektor jednostkowy u, reprezentacyjne kwaternionie można zapisać w następujący sposób:

$q = [\cos(\theta / 2), \sin(\theta / 2)\, u]$

Zostało to zilustrowane na rysunku 11-3 dla dowolnego sztywnego korpusu obracającego się wokół osi przechodzącej przez jej środek ciężkości.



Sztywne ciało obraca się o kąt θ od pokazanej w kolorze jasnoszarym do pozycji zaznaczonej ciemnoszarym. W tym przypadku wektor jednostkowy u jest wektorem v znormalizowanym do długości jednostki. Łatwo zauważyć, że kwaternion, gdy jest używany do reprezentowania rotacji lub orientacji, wymaga, abyś poradził sobie tylko z czterema parametrami zamiast z dziewięcioma, z zastrzeżeniem łatwo zadowalającego ograniczenia, że kwaternion jest jednostkowym kwaternionem. Użycie kwaternionów do odwzorowania orientacji jest podobne do użycia matryc obrotowych. Najpierw ustawiasz kwaternion, który reprezentuje początkową orientację sztywnego ciała w czasie 0 (jest to jedyny czas, w którym obliczysz kwaternion jawnie). Następnie aktualizujesz orientację, aby odzwierciedlić nową orientację w danej chwili w czasie, używając prędkości kątowych, które są obliczane dla tej chwili. Jak widać tutaj, równanie różniczkowe dotyczące orientacji kwaternionności do prędkości kątowej jest bardzo podobne do tego

dla macierzy obrotowych:

$dq / dt = (1/2)\, \omega\, q$

Tutaj prędkość kątowa jest zapisana w postaci kwaternionowej jako [0, ω] i wyraża się w ustalonych, globalnych współrzędnych. (ω jest nadal prędkością kątową, ale musisz ją umieścić w postaci kwaternionu zamiast formy wektorowej, gdy ją mnożysz przez kwaternion q). Jeśli ω wyraża się w obrotowych, ustalonych na stałe współrzędnych, to musisz użyć tego równania:

$dq / dt = (1/2)\, q\, \omega$

Podobnie jak w przypadku macierzy rotacyjnych, możesz użyć kwaternionów do obracania punktów lub wektorów. Jeśli v jest a

wektor, następnie v 'jest obróconym wektorem podlegającym kwaternionowi q:

$v' = qvq^*$

Tutaj q^* jest koniugatem kwaternionem q zdefiniowanym jako:

$q^* = q_0 - q_x\, i - q_y\, j - q_z\, k$

Można również użyć powyższej formuły do konwersji wektorów z jednego układu współrzędnych na inny, gdzie jeden jest obracany względem drugiego. Musisz to zrobić, na przykład, w twoich symulacjach, gdzie są konwersujące siły zdefiniowane w ustalonych, globalnych współrzędnych, może być konieczne przeliczenie prędkości ciała zdefiniowanej w globalnych współrzędnych na współrzędne ciała, abyś mógł użyć prędkości w obliczeniach sił.

Operacje kwaternionów

Podobnie jak w przypadku wektorów i macierzy, kwaternion ma swoje własne reguły dla różnych operacji, których potrzebujesz, takich jak mnożenie, dodawanie, odejmowanie i tak dalej. Aby ułatwić Ci pracę, dodaliśmy przykładowy kod do Dodatku C, który implementuje wszystkie potrzebne operacje kwaternionów; chcemy jednak podkreślić kilka ważniejszych tutaj. Klasa Quaternion jest zdefiniowana za pomocą składnika skalarnego, n, i komponentu wektorowego, v, gdzie v to wektor, x i + y j + zk. Klasa ma dwa konstruktory, z których jeden inicjuje kwaternion na 0, a drugi z nich inicjalizuje elementy do przekazanych do konstruktora:

```
class Quaternion {

public:

float n; // number (scalar) part

Vector v; // vector part: v.x, v.y, v.z

Quaternion(void);

Quaternion(float e0, float e1, float e2, float e3);

.

.

.

};.
```

Magnitude

Metoda Magnitude zwraca wielkość kwaternionu zgodnie z następującą formułą:

$$|\mathbf{q}| = \sqrt{n^2 + x^2 + y^2 + z^2}$$

Jest to podobne do obliczania wielkości wektora, z wyjątkiem tego, że w przypadku kwaternionów należy wziąć pod uwagę czwarty termin, iloczyn skalarny n. Oto kod, który oblicza wielkość dla naszej klasy Quaternion:

```
inline float Quaternion :: Magnitude (void)

{

return (float) sqrt (n * n + v.x * v.x + v.y * v.y + v.z * v.z);

}
```

Sprzężenie: operator ~

Sprzężenie iloczynu kwaternionów jest równy iloczynowi sprzężeń kwaternionu , ale w odwrotnej kolejności:

~ (qp) = (~ p) (~ q)

Oto kod, który oblicza koniugat dla naszej klasy Quaternion:

```
Quaternion operator ~ (void) const {return kwaternion (n, -v.x, -v.y, -v.z);}
```

QVRotate

Ta funkcja obraca wektor v o jednostkę kwaternionu q zgodnie z tą formułą:

p'= (q)(v)(~ q)

Tutaj ~q jest koniugatem jednostki kwaterunkowej, q:

```
inline Vector QVRotate (Quaternion q, wektor v)

{

Quaternion t;

t = q * v * (~ q);

return t.GetVector ();

}
```

Ten operator przyjmuje sprzężenie kwaternionu, ~ q, który jest po prostu ujemną częścią wektora. Jeśli q = [n, x i + y j + z k], to ~ q = [n, (-x) i + (-y) j + (-z) k].

Mnożenie kwaternionów : operator *

Ten operator wykonuje mnożenie kwaternionowe zgodnie z następującą formułą:

$q\ p = n_q\ n_p - v_q \bullet {}_{vp} + n_q\ v_p + n_p\ v_q + (v_q \times v_p)$

Tutaj $n_q n_p - v_q \bullet v_p$ jest skalarną częścią wyniku, podczas gdy $n_q\ v_p + n_p\ v_q + (v_q \times v_p)$ jest częścią wektorową. Zwróć też uwagę, że v_q i v_p są częściami wektora q i p, odpowiednio: • jest operatorem wektora iloczynu skalarnego, a × jest operatorem wektora iloczynu wektorowego. Mnożenie kwaternionowe jest asocjacyjne, ale nie przemienne, a więc:

q (ph) = (qp) h

$qp \neq pq$

Oto kod, który mnoży dwa Quaternions, q_1 i q_2:

```
inline Quaternion operator*(Quaternion q1, Quaternion q2)

{

return Quaternion(q1.n*q2.n - q1.v.x*q2.v.x

- q1.v.y*q2.v.y - q1.v.z*q2.v.z,

q1.n*q2.v.x + q1.v.x*q2.n

+ q1.v.y*q2.v.z - q1.v.z*q2.v.y,

q1.n*q2.v.y + q1.v.y*q2.n

+ q1.v.z*q2.v.x - q1.v.x*q2.v.z,

q1.n*q2.v.z + q1.v.z*q2.n

+ q1.v.x*q2.v.y - q1.v.y*q2.v.x);
```

}

Mnożenie wektorowe: operator *

Ten operator mnoży kwaternion, q, przez wektor v, jak gdyby wektor v był kwaternionem z jego składnikiem skalarnym równym 0. Istnieją dwie formy tego operatora w zależności od kolejności, w której napotykają kwaternion i wektor. Ponieważ przyjmuje się, że v jest kwaternionem, a jego część skalarna równa się 0, zasady mnożenia są zgodne z tymi, które omówiono wcześniej dla mnożenia kwaternionowego:

```
inline Quaternion operator*(Quaternion q, Vector v)

{

return Quaternion( -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),

q.n*v.x + q.v.y*v.z - q.v.z*v.y,

q.n*v.y + q.v.z*v.x - q.v.x*v.z,

q.n*v.z + q.v.x*v.y - q.v.y*v.x);

}

inline Quaternion operator*(Vector v, Quaternion q)

{

return Quaternion( -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),

q.n*v.x + q.v.z*v.y - q.v.y*v.z,

q.n*v.y + q.v.x*v.z - q.v.z*v.x,

q.n*v.z + q.v.y*v.x - q.v.x*v.y);

}
```

MakeQFromEulerAngles

Ta funkcja konstruuje kwaternion z zestawu kątów Eulera. Dla danego zestawu kątów Eulera, odchylenia (ψ), skoku (τ) i obrotu (ϕ), określając obrót wokół osi Z, następnie osi Y, a następnie osi X, można skonstruować reprezentację obrót kwaterunkowy. Robisz to, najpierw konstruując kwaternion dla każdego kąta Eulera a następnie pomnożenie trzech kwaterninów zgodnie z zasadami mnożenia kwaternionowego. Oto trzy kwaterunki reprezentujące każdy kąt obrotu Eulera:

$q_{roll} = [\cos(\phi / 2), (\sin(\phi / 2)) i + 0 j + 0 k]$

$q_{pitch} = [\cos(\tau / 2), 0 i + (\sin(\tau / 2)) j + 0 k]$

$q_{yaw} = [\cos(\psi / 2), 0 i + 0 j + (\sin(\psi / 2)) k]$

Każdy z tych kwaterninów jest długości jednostki. Teraz możesz pomnożyć te kwaterunki, aby uzyskać jeden, który reprezentuje obrót lub orientację zdefiniowaną przez trzy kąty Eulera:

$q = q_{yaw}\ q_{pitch}\ q_{roll}$

Wykonanie tego mnożenia daje:

q = [{cos (ϕ / 2) cos (τ / 2) cos (ψ / 2) + sin (ϕ / 2) sin (τ / 2) sin (ψ / 2)},

{sin (ϕ / 2) cos (τ / 2) cos (ψ / 2) - cos (ϕ / 2) sin (τ / 2) sin (ψ / 2)} i +

{cos (ϕ / 2) sin (τ / 2) cos (ψ / 2) + grzech (ϕ / 2) cos (τ / 2) sin (ψ / 2)} j +

{cos (ϕ / 2) cos (τ / 2) sin (ψ / 2) - sin (ϕ / 2) sin (τ / 2) cos (ψ / 2)} k]

Oto kod, który przyjmuje trzy kąty Eulera i zwraca kwaternion:

```
inline Quaternion MakeQFromEulerAngles(float x, float y, float z)
{
Quaternion q;
double roll = DegreesToRadians(x);
double pitch = DegreesToRadians(y);
double yaw = DegreesToRadians(z);
double cyaw, cpitch, croll, syaw, spitch, sroll;
double cyawcpitch, syawspitch, cyawspitch, syawcpitch;
cyaw = cos(0.5f * yaw);
cpitch = cos(0.5f * pitch);
croll = cos(0.5f * roll);
syaw = sin(0.5f * yaw);
spitch = sin(0.5f * pitch);
sroll = sin(0.5f * roll);
cyawcpitch = cyaw*cpitch;
syawspitch = syaw*spitch;
cyawspitch = cyaw*spitch;
syawcpitch = syaw*cpitch;
q.n = (float) (cyawcpitch * croll + syawspitch * sroll);
q.v.x = (float) (cyawcpitch * sroll - syawspitch * croll);
q.v.y = (float) (cyawspitch * croll + syawcpitch * sroll);
q.v.z = (float) (syawcpitch * croll - cyawspitch * sroll);
return q;
}
```

MakeEulerAnglesFromQ

Ta funkcja wydziela trzy kąty Eulera z danego kwaternionu. Możesz wyodrębnić trzy kąty Eulera z kwaternionu, najpierw przekształcając kwaterniony w macierz obrotu, a następnie wyodrębniając kąty Eulera z macierzy obrotu. Niech R będzie dziewięcioelementową macierzą rotacji:

$$\mathbf{R} = \begin{vmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{vmatrix}$$

i niech q będzie kwaternionem

$\mathbf{q} = [n, \ x\ \mathbf{i} + y\ \mathbf{j} + z\ \mathbf{k}]$

Następnie każdy element w R jest obliczany z q w następujący sposób:

$$\begin{aligned} r_{11} &= n^2 + x^2 - y^2 - z^2 \\ r_{21} &= 2xy + 2zn \\ r_{31} &= 2zx - 2yn \\ r_{12} &= 2xy - 2zn \\ r_{22} &= n^2 - x^2 + y^2 - z^2 \\ r_{32} &= 2zy + 2xn \\ r_{13} &= 2xz + 2yn \\ r_{23} &= 2yz - 2xn \\ r_{33} &= n^2 - x^2 - y^2 + z^2 \end{aligned}$$

Aby wyodrębnić kąty Eulera, odchylenie (ψ), skok (τ) i przechylenie (ϕ), od R, możesz użyć tych relacji:

$$\begin{aligned} \tan \psi &= r_{21} / r_{11} \\ \sin \tau &= -r_{31} \\ \tan \varphi &= r_{32} / r_{33} \end{aligned}$$

Oto kod, który wyodrębnia trzy kąty Eulera, zwrócone w postaci wektora, z danego kwaternionu:

```
inline Vector MakeEulerAnglesFromQ(Quaternion q)

{

double r11, r21, r31, r32, r33, r12, r13;

double q00, q11, q22, q33;

double tmp;

Vector u;

q00 = q.n * q.n;

q11 = q.v.x * q.v.x;

q22 = q.v.y * q.v.y;

q33 = q.v.z * q.v.z;
```

```
r11 = q00 + q11 - q22 - q33;

r21 = 2 * (q.v.x*q.v.y + q.n*q.v.z);

r31 = 2 * (q.v.x*q.v.z - q.n*q.v.y);

r32 = 2 * (q.v.y*q.v.z + q.n*q.v.x);

r33 = q00 - q11 - q22 + q33;

tmp = fabs(r31);

if(tmp > 0.999999)

{

r12 = 2 * (q.v.x*q.v.y - q.n*q.v.z);

r13 = 2 * (q.v.x*q.v.z + q.n*q.v.y);

u.x = RadiansToDegrees(0.0f); //roll

u.y = RadiansToDegrees((float) (-(pi/2) * r31/tmp)); // pitch

u.z = RadiansToDegrees((float) atan2(-r12, -r31*r13)); // yaw

return u;

}

u.x = RadiansToDegrees((float) atan2(r32, r33)); // roll

u.y = RadiansToDegrees((float) asin(-r31)); // pitch

u.z = RadiansToDegrees((float) atan2(r21, r11)); // yaw

return u;

}
```

Kwaterniony w symulatorach 3D

Przedstawione operacje kwaternionów są wymagane, gdy używasz kwaternionów do reprezentowania orientacji w symulacjach 3D. Wszystkie symulacje 3D omówione w tej książce wykorzystują te operacje kwaternionów, w tym rozdziale podkreślimy, gdzie są one używane w kontekście przykładu samolotu przedstawionego w części 15. Podczas inicjowania orientacji samolotu, musisz ustawić jej ułożenie kwaterunkowe w coś odpowiadającego kątom Eulera, których pragniesz. Robisz to w następujący sposób:

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

W tej próbce kodowej samolot jest klasą ciała sztywnego z właściwością qOrientation, który reprezentuje kwaternionię orientacyjną, która jest klasą kwaternionów. iRoll, iPitch i iYaw to trzy kąty Eulera opisujące orientację samolotu. Jeśli w dowolnym momencie chcesz zgłosić kąty Eulera - na przykład na ekranie heads-up, takim jak interfejs odtwarzacza gier, możesz użyć funkcji MakeEulerAnglesFromQ w następujący sposób:

```
// get the Euler angles for our information
```

```
Vector u;

u = MakeEulerAnglesFromQ(Airplane.qOrientation);

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

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

Airplane.vEulerAngles.z = u.z; // yaw
```

Bardzo często wygodniej jest obliczyć obciążenie obiektu, np. Samolotu współrzędne stałe. Na przykład podczas obliczania oporu aerodynamicznego na samolocie, będziesz chciał poznać względną prędkość powietrza nad samolotem w ustalonych współrzędnych. Wynikowa siła oporu będzie również w ustalonych współrzędnych. Jednak, gdy rozdzielasz wszystkie obciążenia na statku powietrznym, aby określić jego ruch w ustalonych współrzędnych ziemskich, będziesz chciał przekształcić te siły z ustalonych na stałe współrzędnych do ustalonych przez Ziemię współrzędnych. Możesz użyć QVRotate, aby obrócić dowolny wektor, taki jak wektor siły, z jednego układu współrzędnych na inny. Poniższy przykładowy kod pokazuje, w jaki sposób QVRotate jest używany do konwersji wektora siły we współrzędnych ustalonych przez ciało na równoważną siłę w ustalonych współrzędnych ziemnych.

```
void CalcAirplaneLoads (void)

{

.

.

.

// Konwertuj siły z przestrzeni modelu na przestrzeń Ziemi

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

.

.

.

}
```

Podczas symulacji będziesz musiał zaktualizować orientację samolotu poprzez integrację kątowych równań ruchu. Pierwszym krokiem w obsłudze ruchu kątowego jest obliczenie nowej prędkości kątowej w danym momencie na podstawie wcześniej obliczonych momentów działających na samolot i jego właściwości masowych. Robimy to we współrzędnych ciała za pomocą kątowego równania ruchu:

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

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, o której mówiliśmy wcześniej:

$dq / dt = (1/2)\ \omega\ q$

Następnie, aby wymusić ograniczenie, że ta orientacja kwaternionowa jest kwaternionem jednostkowym, musisz znormalizować orientację kwaternionów. Poniższy przykładowy kod ilustruje te kroki:

.

.

.

```
// 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);
```

.

.

.

Zwróć uwagę na ostatni wiersz kodu w poprzedniej próbce. Linia ta zamienia wektor prędkości samolotu z ustalonych przez Ziemię współrzędnych na współrzędne stałe z użyciem QVRotate. Przypomnij sobie, że wygodniej jest obliczyć siły ciała w ustalonych współrzędnych ciała. QVRotate umożliwia pracę z wektorami tam i z powrotem od ustalonych na stałe do ustalonych przez użytkownika współrzędnych.