XIX. Sporty

Tematyka sportu jest prawie tak obszerna, jak wszystkie tematy, które razem omówiliśmy. Dla każdego jest sport, a sport wykorzystuje każdy model fizyczny, o którym mówiliśmy do tej pory. Temat obejmuje gry pełne akcesoriów, takich jak golf lub polo, do biegania, gdzie wszystko czego potrzebujesz to własne dwie stopy. Jednym z najbardziej atrakcyjnych aspektów sportu dla programisty gry jest to, że odbywa się on w ograniczonej przestrzeni fizycznej według projektu. W przeciwieństwie do strzelanki pierwszoosobowej, w której gracz ostatecznie osiągnie sztuczną granicę, w grze sportowej gracz nie spodziewa się, że będzie mógł wyjść z boiska. Niemal wszystkie sporty mają zdefiniowane wymiary, które są stosunkowo łatwe do modelowania. Tabela 19-1 wymienia kilka dyscyplin sportowych i ich profesjonalne wymiary pola.

Rozmiar boiska sportowego

Piłka nożna (piłka nożna) o długości	90-120 mi szerokości 45-90 m
Piłka nożna (w tym strefy końcowe) ma	109,7 m długości i szerokość 48,8 m
Baseball	27,4 m między podstawami; 18,39 z kopca z miotacza do bazy domowej; outfield jest różny
Koszykówka (międzynarodowa)	28 m na 15 m
Hokej na lodzie (międzynarodowy)	61 m na 30 m

Jak widać, poza baseballem - gdzie kształt pola outfield zmienia się w zależności od tego, na jakim stadionie się znajdujesz, modelowanie tych pól jest raczej prostym ćwiczeniem. Poza tym jedyne, co łączy sport, to fakt, że mają ludzkiego aktora. W tej części omówimy, w jaki sposób można symulować działanie ludzkie jako dane wejściowe dla innych omawianych symulacji fizycznych. W szczególności pokażemy wam przykład, w jaki sposób modelować osobę kołyszącą się w klubie golfowym za pomocą dokładnych modeli fizjologicznych. Nazywa się to biomechaniką. Zanim do tego dojdziemy, inną ważną rzeczą do zrozumienia, kiedy modelujesz sport, są granice ludzkiego ciała. Chociaż rekordy są łamane podczas każdej olimpiady, żadna istota ludzka nie jest w stanie skoczyć 10 stóp pionowo w powietrze. Dopóki celowo nie przełamiecie granic biomechaniki, spowoduje to zmniejszenie realizmu waszej gry. Statystyki biomechaniczne tego, co można uznać za wybitnego sportowca, podano w tabeli 19-2.

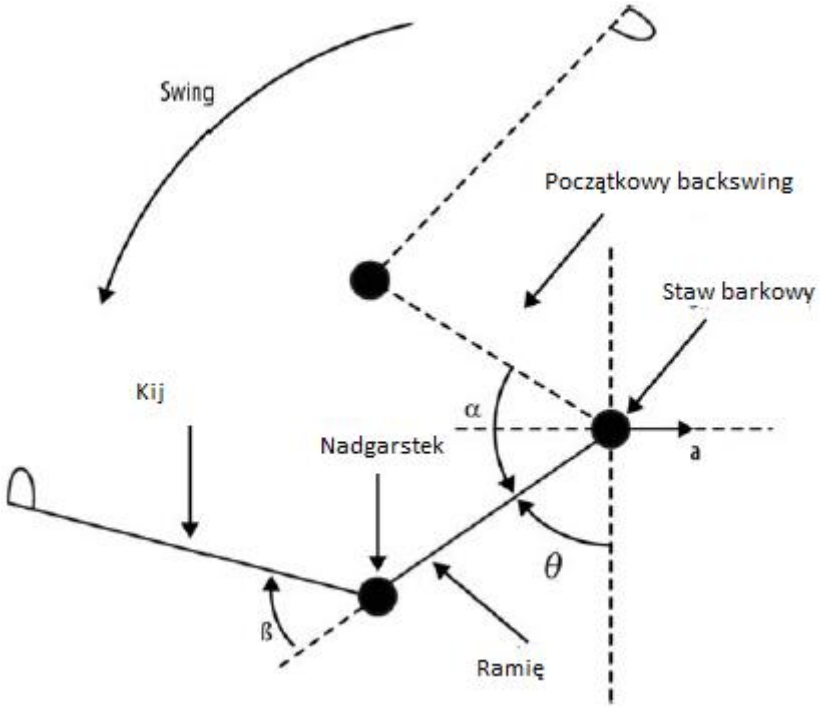
Atrybut fizyczny	Wartość średnia	Wartość rekordu
Skacz z postoju (pionowo)	81 cm	155 cm
Wysokość skoku w biegu (skocznia wysoka)	1,83 m	2,45 m
Odległość skokowa	5,0 m	8,95 m
Prędkość wyrzucania	24,5 m / s	46,0 m / s
Prędkość obrotowa powyżej 100 m	7,5 m / s	10 m / s
Prędkość obrotowa powyżej 10 000 m	3,7 m / s	6,3 m / s

Prawie wszystkie rekordy sportowe są dostępne gdzieś w Internecie, więc tabela 19-2 nie jest wyczerpująca. Warto jednak wykorzystać te wartości, aby ograniczyć symulacje ludzkich działań w grach wideo. Oczywiście, częścią podniecenia grania w gry wideo jest możliwość skakania wyżej i biegania szybciej niż w przeciwnym razie, ale dobra ankieta z biomechaniką da ci przynajmniej znać, co

jest niezwykłe, a co nie. Teraz rzućmy okiem na to, jak będziemy modelować ludzkiego aktora w grze sportowej.

Modelowanie swingu golfowego

Załóżmy, że piszesz grę w golfa i chcesz dodać trochę realizmu. To oczywiste ,że ważnym elementem gry jest swing golfowy. Innym jest uderzenie kijem, a jeszcze inna jest trajektorią piłki w locie. Możesz użyć techniki modelowania ruchu pocisku omówione wcześniej w Części 2, Część 4 i Część 6 w celu modelowania lotu piłki oraz techniki reakcji kolizji w części 5 w celu modelowania uderzenia piłki golfowej. Ale co ze swingiem golfowym? Zanim pokażemy ci jeden sposób modelowania zamachu golfowego, porozmawiajmy o tym, dlaczego chcesz to zrobić w pierwszej kolejności. Aby modelować uderzenie piłki kijem, musisz znać prędkość uderzenia piłki w momencie uderzenia. Ta prędkość jest funkcją swingu Golfista podnosi kij za plecy, skręca ciało i przygina łeb kijem, jednocześnie przykładając moment z nadgarstkami. Gdy kij się odchyla, moment obrotowy nadgarstka ulega odwróceniu, a kij uderza w dół, aż głowa kija zderzy się z piłką. (Lub, w naszym przypadku, zderza się z ziemią!) Teraz mamy wiele subtelnych szczegółów, które pominęliśmy tutaj w odniesieniu do techniki i fizyki, ale masz pomysł. W każdym razie swing określa prędkość główki kija w chwili uderzenia, która z kolei decyduje o prędkości kuli po uderzeniu. Jeśli pisałeś grę na Wii lub inną platformę, która może uchwycić ruch gracza, możesz powiązać ruch wahadłowy gracza z początkowym momentem obrotowym zastosowanym w wirtualnym klubie golfowym, określając w ten sposób, za pomocą jakiegoś modelu, dynamikę wahań i wynikająca prędkość głowy klubu. Golfiści poważnie traktują technikę swing, podobnie jak naukowcy badający dynamikę swinga. Aby zrozumieć, co sprawia, że dobry huśtawka lub jak poprawić swing, jest wielu naukowców aktywnie badających fizykę swingów golfowych. W rezultacie istnieje wiele modeli matematycznych o różnym stopniu realizmu i złożoności, które mają na celu zbadanie zamachu golfowego. Jednym z przykładów jest tak zwany model dwubiegunowy opisany w książce Theodore P. Jorgensena "The Physics of Golf". W swojej książce dr Jorgensen opisuje szczegółowo model dwubiegunowy, w tym założenia i uproszczenia, oraz dostarcza uzyskane równania, które należy rozwiązać, aby zasymulować swing golfowy oparty na tym modelu. Dostarcza nawet danych empirycznych służących do sprawdzania poprawności wyników modelu matematycznego. Jak pokazano na rysunku 19-1,



model dwubiegunowy zakłada, że ramię golfisty to jeden pręt, który rozciąga się od ramion do nadgarstków. To jest pręt ramienia. Klub jest reprezentowany przez inny pręt, który rozciąga się od końca nadgarstka pręta ramienia do głowy klubu. Ten model jest zasadniczo podwójnym wahadłem. Dokładniej, jest to napędzane podwójne wahadło, ponieważ model zakłada moment obrotowy przyłożony na końcu ramienia drążka ramienia, a kolejny moment obrotowy przykładany jest na przegubie łączącym ramię drążka z pręcikiem. Nie powtórzymy rozwoju dr Jorgensena modelu tutaj; zamiast tego pokażemy ci jak rozwiązać uzyskane równania:.

Równanie 1:

$$(J+I+M_C R^2+2RS\cdot\cos(\beta))\ddot{\alpha}-(I+RS\cdot\cos(\beta))\ddot{\beta}+$$
$$+(\dot{\beta}^2-2\dot{\alpha}\dot{\beta})RS\sin(\beta)-S[g\sin(\theta+\beta)-a\cdot\cos(\theta+\beta)]-$$
$$-(S_A+M_c\cdot R)(g\cdot\sin(\theta)-a\cdot\cos(\theta))=Q_\alpha$$

Równanie 2:

$$I\ddot{\beta}-[I+RS\cdot\cos(\beta)]\ddot{a}+\dot{a}^2RS\cdot\sin(\beta)+S[g\cdot\sin(\theta+\beta)-a\cdot\cos(\theta+\beta)]=Q_\beta$$

Tabela 19-3 wyjaśnia, co reprezentuje każdy symbol

Symbol	Znaczenie
J	Masowy moment bezwładności pręta reprezentujący ramię. Jednostki to kg-m2.
I	Masowy moment bezwładności pręta reprezentującego kij. Jednostki to kg-m2.
M_c	Masa kija Jednostki to kg.
R	Długość pręta przedstawiającego ramię. Jednostki to m.
S	Pierwsza chwila pręta przedstawiająca kij wokół osi nadgarstka (gdzie pręt kija łączy się z prętem ramienia). Jednostki to kg-m.
α	Kąt pochwycony przez pręt ramienia od początkowej pozycji cofania. Jednostki są radianami.
β	Kąt nadgarstka. Jednostki są radianami.
g	Przyspieszenie z powodu grawitacji. Stała 9,8 m/s^2.
Θ	Kąt między prętem ramienia a osią pionową. Jednostki są radianami.
a	Poziome przyspieszenie barku. Jednostki to m / s2.
S_A	Pierwszy moment drążka ramienia wokół osi barku. Jednostki są w kg-m.
Q_α	Moment obrotowy nałożony na ramię na drążek ramienia. Jednostki to N-m.
Q_β	Moment obrotowy nałożone na stawie nadgarstkowym na pręt klubowy. Jednostki to N-m.

Równania te reprezentują sprzężony układ nieliniowych równań różniczkowych. One są sprzężone, ponieważ oba zależą od nieznanych wielkości α i β. One są wyraźnie równaniami różniczkowymi, ponieważ oba zawierają pochodne czasowe nieznanych wielkości. Są one nieliniowe, ponieważ zawierają sinusy i cosinusy jednego z niewiadomych oraz pochodne drugiej nieznanej wzbudzonej do pewnej mocy większej niż 1. A więc, jak rozwiązujemy te równania? Cóż, nie możemy tego zrobić w formie zamkniętej i musimy uciekać się do metod numerycznych. Istnieje wiele sposobów postępowania, ale podejście, które zastosujemy, to najpierw rozwiązać równanie 2 dla β¨ i podstawić wynik do równania 1. Następnie liczbowo zintegrujemy wynik za pomocą czwartego rzędu Runge-Kutta schemat, jak opisano w części 7. Dokładniej, dla każdego kroku czasowego Równanie 1 z β¨ zastąpione przez wyrażenie wyprowadzone z równania 1 zostanie rozwiązane dla α¨. Po znalezieniu α¨ możemy znaleźć β¨ używając drugiego równania, uprzednio rozwiązanego dla β¨. Następnie możemy zintegrować α¨ i β¨, aby znaleźć α i β. Ten proces powtarza się dla każdego kroku czasowego. Ponownie, jest to tylko jedna metoda rozwiązywania tych równań. Normalnie, w obliczu układu równań, praktycy używają schematów macierzy do równoczesnego rozwiązywania równań. Jest to prawie konieczne w przypadku układów równań, które obejmują więcej niż dwa równania. Jednakże, mając tylko dwa równania, jak to mamy tutaj, możemy uniknąć kosztownego obliczania inwersji macierzy za pomocą techniki, którą właśnie opisaliśmy.

Rozwiązywanie równań wahadłowych golfa

Teraz pokażemy, jak wdrożyć rozwiązanie opisane w prostej konsoli podania. Przykład rozwiązuje dwa równania rządzące α i β w czasie, których wyniki można następnie wykorzystać do wyznaczenia prędkości głowicy maczugowej w dowolnym momencie za pomocą równań kinematycznych, jak opisano w części 2. Alternatywnie można użyć następującego równania, które dr Jorgensen podaje dla prędkości głowy maczugi w swoim tekście:

$$V^2 = \left[R^2 + L^2 + 2RL \cdot \cos(\beta)\right]\dot{\alpha}^2 + L^2\dot{\beta}^2 - 2\left[L^2 + RL \cdot \cos(\beta)\right]\dot{\alpha}\dot{\beta}$$

W tym przykładzie użyjemy równania Jorgensena. Ponieważ kąty zainteresowania są obliczane w jednostkach radianów, ale chcemy je zgłosić w jednostkach stopni, najpierw tworzymy kilka definicji, aby dokonać konwersji za nas:

```
#define RADIANS (d) (d / 180.0 * 3.14159)

#define DEGREES (r) (r * 180.0 / 3.14159)
```

Następnie deklarujemy i inicjalizujemy wszystkie zmienne. Stosowane tu wartości początkowe są typowymi wartościami, które przyjęliśmy. Możesz zmienić te wartości, aby symulować różne huśtawki:

```
// Variables

double alpha = 0.0;

double alpha_dot = 0.0;

double alpha_dotdot = 0.0;

double beta = RADIANS(120.0);

double beta_dot = 0.0;

double beta_dotdot = 0.0;
```

```
double J = 1.15; // kg m^2
double I = 0.08; // kg m^2
double Mc = 0.4; // kg
double R = 0.62; // m
double L = 1.1; // m
double S = 0.4*1.1*0.75; // kg m
double g = 9.8; // m/s^2
double gamma = RADIANS(135.0);
double theta = gamma - alpha;
double SA = 7.3*0.62*0.5; // kg m
double Qalpha = 100; // N m
double Qbeta = −10; // N m
double a = 0.1*g; // m/s^2
double dt = 0.0025; // s
double time = 0; // s
double Vc = 0;
```

Następnie definiujemy dwie funkcje, które wykorzystamy do obliczenia pochodnych po raz drugi α i β (tj. $\ddot{\alpha}$ i $\ddot{\beta}$). Funkcje te wykorzystują po prostu równania 1 i 2 rozwiązane odpowiednio dla $\ddot{\alpha}$ i $\ddot{\beta}$.

ComputeAlphaDotDot, który rozwiązuje dla α¨, pokazano tutaj:

```
double ComputeAlphaDotDot(void)
{
double A, B, C, D, F, G;
double num, denom;
A = (J + I + Mc * R * R + 2 * R * S * cos(beta));
B = -(I + R * S * cos(beta));
F = Qalpha - (beta_dot * beta_dot - 2 * alpha_dot * beta_dot) * R * S *
sin(beta) + S * (g * sin(theta + beta) - a * cos(theta + beta))
+ (SA + Mc * R) * (g * sin(theta) - a * cos(theta));
C = B;
D = I;
G = Qbeta - alpha_dot * alpha_dot * R * S * sin(beta) −
```

```
S * (g * sin(theta + beta) - a * cos(theta + beta));

num = (F - (B * G / D));

denom = (A-(B*C/D));

return (F - (B * G / D)) / (A-(B*C/D));

}
```

Zmienne lokalne A, B, C, D, F i G są zmiennymi wygody używanymi do organizowania terminów w równaniu 1. Ta funkcja zwraca drugą pochodną α. ComputeBetaDotDot, pokazany obok, jest bardzo podobny do ComputeAlphaDotDot, ale rozwiązuje Równanie 2 zamiast. Ta funkcja zwraca drugą pochodną β:

```
double ComputeBetaDotDot(void)

{

double C, D, G;

C = -(I + R * S * cos(beta));

D = I;

G = Qbeta - alpha_dot * alpha_dot * R * S * sin(beta) −

S * (g * sin(theta + beta) - a * cos(theta + beta));

return (G - C * alpha_dotdot) / D;

}
```

Rozwiązanie równań 1 i 2 jest zgodne ze schematem Runge-Kutta, który pokazano w części 7. Dla każdego kroku czasowego są podejmowane cztery pośrednie kroki. Wielkość kroku czasu jest kontrolowana przez dt, który ustawiliśmy na 0,0025 s. Jeśli po prostu użyjesz metody Eulera, musisz znacznie zmniejszyć ten rozmiar kroku, aby uzyskać stabilne rozwiązanie. Wdrożyliśmy rozwiązanie w głównej funkcji naszego przykładu konsolowego. Kod jest następujący:

```
int _tmain(int argc, _TCHAR* argv[])

{

double a, at;

double b, bt;

int i;

FILE* fp;

double phi;

double Vc2;

double ak1, ak2, ak3, ak4;

double bk1, bk2, bk3, bk4;

FILE* fdebug;
```

```
fp = fopen("results.txt", "w");
fdebug = fopen("debug.txt", "w");
for(i = 0; i<200; i++)
{
time += dt;
if(time>=0.1)
{
Qbeta = 0;
}
// save results of previous time step
a = alpha;
b = beta;
at = alpha_dot;
bt = beta_dot;
// integrate alpha'' and beta''
// The K1 Step:
alpha_dotdot = ComputeAlphaDotDot();
beta_dotdot = ComputeBetaDotDot();
ak1 = alpha_dotdot * dt;
bk1 = beta_dotdot * dt;
alpha_dot = at + ak1/2;
beta_dot = bt + bk1/2;
// The K2 Step:
alpha_dotdot = ComputeAlphaDotDot();
beta_dotdot = ComputeBetaDotDot();
ak2 = alpha_dotdot * dt;
bk2 = beta_dotdot * dt;
alpha_dot = at + ak2/2;
beta_dot = bt + bk2/2;
// The K3 Step:
alpha_dotdot = ComputeAlphaDotDot();
```

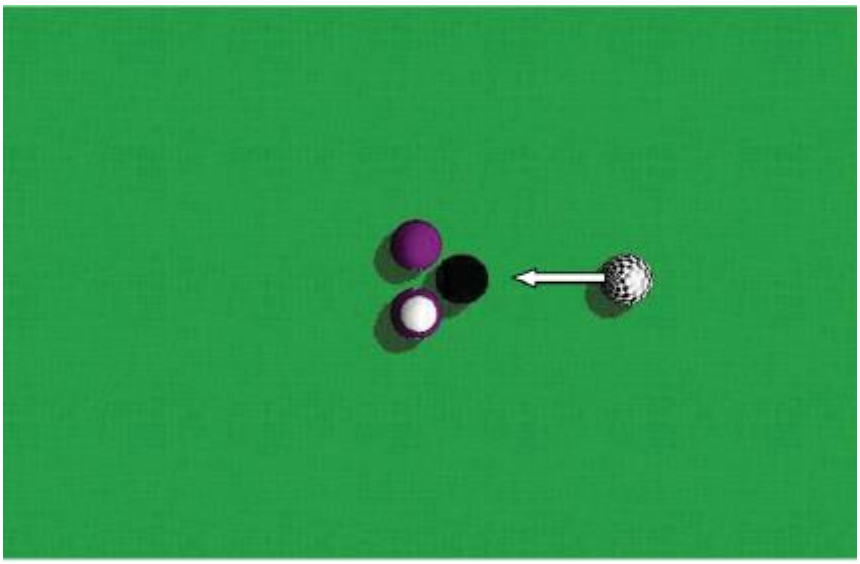
```
beta_dotdot = ComputeBetaDotDot();
ak3 = alpha_dotdot * dt;
bk3 = beta_dotdot * dt;
alpha_dot = at + ak3;
beta_dot = bt + bk3;
// The K3 Step:
alpha_dotdot = ComputeAlphaDotDot();
beta_dotdot = ComputeBetaDotDot();
ak4 = alpha_dotdot * dt;
bk4 = beta_dotdot * dt;
alpha_dot = at + (ak1 + 2*ak2 + 2*ak3 + ak4) / 6;
beta_dot = bt + (bk1 + 2*bk2 + 2*bk3 + bk4) / 6;
alpha = a + alpha_dot * dt;
beta = b + beta_dot * dt;
theta = gamma - alpha;
Vc2 = (R*R + L*L + 2 * R * L * cos(beta)) * ( alpha_dot * alpha_dot)
+ L*L * beta_dot * beta_dot
- 2 * (L*L + R * L * cos(beta)) * alpha_dot * beta_dot;
Vc = sqrt(Vc2);
phi = theta + beta;
fprintf(fp, "%f, %f, %f, %f, %f, %f\n", time, DEGREES(theta),
DEGREES(alpha), DEGREES(beta), DEGREES(phi), Vc);
fprintf(fdebug, "%f, %f, %f, %f, %f, %f, %f\n", time, DEGREES(alpha),
alpha_dot, alpha_dotdot, DEGREES(beta), beta_dot, beta_dotdot);
}
fclose(fp);
fclose(fdebug);
return 0;
}
```

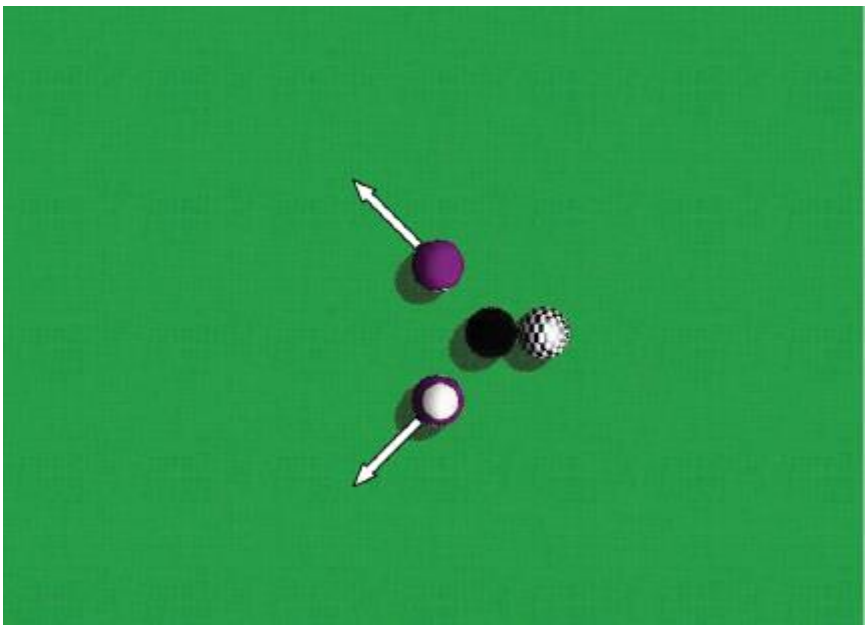
Zmienne lokalne a, at, b i bt są używane do tymczasowego zapisywania wyników poprzedniego kroku czasowego dla α i β i ich pierwszych pochodnych. i jest zmienną licznika. fp jest wskaźnikiem pliku, którego użyjemy do zapisania wyników do pliku tekstowego. phi służy do przechowywania sumy θ + β.

A Vc2 jest kwadratem prędkości głowy maczugi obliczonej zgodnie z równaniem Jorgensena. Zmienne od ak1 do ak4 oraz od bk1 do bk4 są używane do przechowywania wyników pośrednich schematu integracji Runge-Kutta. fdebug to wskaźnik pliku do pliku, którego używaliśmy do zapisywania informacji debugujących. Po otwarciu plików wyjściowych i debugowania funkcja wchodzi w pętlę, aby wykonać integrację w ciągu 200 kroków. Możesz zmienić liczbę stopni czasowych, jakie uznasz za odpowiednie dla aplikacji. Należy pamiętać, że wydarzenie swing, od początku do uderzenia piłki, ma miejsce w bardzo krótkim okresie czasu - tylko ułamki sekund długich. Po wejściu do pętli zobaczysz kod, który sprawdzi, ile czasu minęło; jeśli czas ten jest większy niż 0,1 s, moment obrotowy nadgarstka, Qbeta, jest ustawiony na 0. Jest to prosty model tego, w jaki sposób początkowo stosowany moment obrotowy nadgarstka jest zwolniony, pozwalając, aby klub przechodził obok ramienia. W zależności od swingu, który modelujesz, ten moment obrotowy może odwrócić kierunek, zmuszając klub do przekroczenia ramienia jeszcze bardziej. Książka doktora Jorgensena wyjaśnia wszystko szczegółowo, dając nawet wyniki eksperymentalne. Następnie wyniki poprzedniego kroku czasu zapisywane są w zmiennych a, b, at i bt. Pierwszy krok po prostu przechowuje wartości początkowe. Teraz integracja rozpoczyna się w pierwszym kroku, k1. Każdy z tych kroków obejmuje obliczanie α¨ i β¨ przy użyciu funkcji ComputeAlphaDotDot i ComputeBetaDotDot. Wyniki k1 są następnie obliczane i wykorzystywane do obliczania wyników pośrednich po raz pierwszy pochodnych α i β. Wszystkie cztery pośrednie etapy są przeprowadzane w podobny sposób. Na koniec obliczane są bieżące wyniki kroku czasowego dla alpha_dot i beta_dot, wraz z alfa i beta. Również kwadrat prędkości głowy maczugi, Vc2, jest obliczany za pomocą równania Jorgensena pokazanego wcześniej; a prędkość głowy maczugi, Vc, wynika z pierwiastka kwadratowego Vc2. Wyniki zainteresowania są zapisywane do plików wyjściowych i debugowania. Tak, to prawie tyle. Po zakończeniu pętli pliki zostaną zamknięte, a aplikacja zostanie zakończona. Gdyby to była prawdziwa gra, używałbyś wyników z prędkością głowy klubu wraz z metodą kolizji, którą pokazaliśmy w części 3, aby określić trajektorię piłki golfowej. Możesz modelować tor lotu piłki golfowej, korzystając z metod pokazanych w części 6.

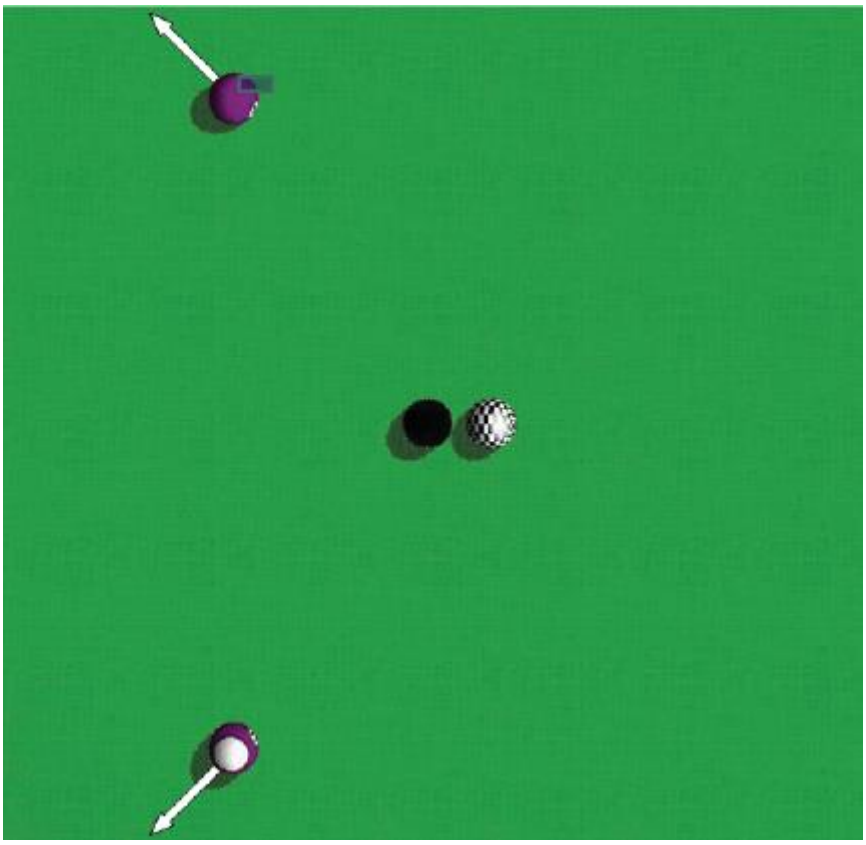
Bilard

Teraz rzućmy okiem na inny przykład. Nie możesz myśleć o bilardzie jako o sporcie, ale jest uznawany na arenie międzynarodowej jako sport wskazujący. Cue sports to rodzina sportów, w skład której wchodzą bilard, bilard, snooker i inne powiązane odmiany. Dla uproszczenia będziemy trzymać się terminu bilard, chociaż przedstawione tematy odnoszą się do wszystkich dyscyplin sportowych. Bilard jest dobrym przykładem aktywności, która ma miejsce w ograniczonej przestrzeni fizycznej. Tak więc, pisząc grę komputerową w bilard, musisz zajmować się jedynie skończoną przestrzenią złożoną z ugruntowanej geometrii. Stoły bilardowe mają zwykle 1,37 m × 2,74 m (4,5 stopy x 9 ft), niektóre dłuższe, a niektóre mniejsze w zależności od gry, stylu i dostępnej przestrzeni. Stoły są zazwyczaj pokrytymi tkaniną łupkami. Kule różnią się rozmiarem między grami i regionami, z kulami basenowymi w stylu amerykańskim o średnicy około 57 mm (2,25 cala). Piłki były wykonane z drewna, gliny lub kości słoniowej, ale obecnie są plastikowe. Wszystkie te cechy są ważnymi drobnymi szczegółami, które musisz wziąć pod uwagę, jeśli zamierzasz stworzyć realistyczną grę wideo w bilard. Łupek i twarde kulki z tworzywa sztucznego mają określone właściwości udarowe. Pokryty tkaniną stolik zapewnia odporność na walcowanie. Zderzaki boczne nie są tak twarde, jak stół łupkowy, co daje różne właściwości udarowe. Na szczęście dane dotyczące stołów bilardowych i piłek są łatwo dostępne w Internecie. Symulowanie bilarda w grze wideo jest dość proste. Bilard stanowi interesujący przykład, ponieważ kolizje są sercem gry, a taki przykład daje nam również możliwość wykazania się kontaktami tocznymi. Rysunek 19-2 i Rysunek 19-3 ilustrują przykład, na którym się skupimy.





Mamy trzy piłki obiektowe (te, które zostaną uderzone bilą) ustawione pośrodku stołu w konfiguracji z luźnym trójkątem. Bila biała pojawia się z prawej na ustaloną prędkość (patrz Ryc. 19-2), a następnie uderzenia ośmiu piłek (patrz Ryc. 19-3). Po początkowym uderzeniu piłki bilardowej i ósemki, ósma piłka przesuwa się w lewo i uderza o dwie kolejne piłki. Te kule następnie strzelają ukośnie. Większość energii z ósmej piłki jest przenoszona na dwie pozostałe piłki, więc ósma piłka szybko zatrzymuje się, gdy jest całowana przez białą bilę. Pozostałe dwie kule nadal toczą się ukośnie (19.4)



W tym przykładzie pokażemy, jak radzić sobie z zderzeniami kulkowymi, zderzeniami z kulkami, stolikiem kulowym, oporem aerodynamicznym na piłce, oporem toczenia, tarciem między piłkami w momencie uderzenia i tarciem między piłkami i tabela.

Realizacja

Jeśli zapoznałeś się i zapoznałeś z przykładami przedstawionymi w części 7 do części 13, wtedy przykład tego bilarda będzie ci bardzo dobrze znany; używamy tego samego podstawowego podejścia. Podczas każdego kroku czasu symulacji obliczamy wszystkie siły działające na każdą piłkę; zintegrować równania ruchu, aktualizując pozycję i prędkość każdej kulki; a następnie sprawdź i radzić sobie z kolizjami. Klasa sztywnego nadwozia użyta w tym przykładzie jest bardzo podobna do tej stosowanej w przykładzie samolotu w części 15. Mimo że kule są zwarte i okrągłe, i kuszące jest traktowanie ich jako cząstek, należy traktować je jako sztywne ciała 3D w aby uchwycić walcowanie i spinning, które są ważnymi elementami dynamiki kulek bilardowych. Klasa rigidbody przyjęta dla tego przykładu bilardu jest następująca.

typedef struct _RigidBody {

float fMass; // Total mass (constant)

Matrix3x3 mInertia; // Mass moment of inertia in body coordinates

Matrix3x3 mInertiaInverse;// Inverse of mass moment of inertia matrix

Vector vPosition; // Position in earth coordinates

Vector vVelocity; // Velocity in earth coordinates

```
Vector vVelocityBody; // Velocity in body coordinates

Vector vAcceleration; // Acceleration of cg in earth space

Vector vAngularAcceleration; //Angular acceleration in body coordinates

Vector vAngularAccelerationGlobal; // Angular acceleration

// in global coordinates

Vector vAngularVelocity; // Angular velocity in body coordinates

Vector vAngularVelocityGlobal; // Angular velocity in global coordinates

Vector vEulerAngles; // Euler angles in body coordinates

float fSpeed; // Speed (magnitude of the velocity)

Quaternion qOrientation; // Orientation in earth coordinates

Vector vForces; // Total force on body

Vector vMoments; // Total moment (torque) on body

Matrix3x3 mIeInverse; // Inverse of moment of inertia in earth coordinates

float fRadius; // Ball radius

} RigidBody, *pRigidBody;
```

Jak widać, ta klasa wygląda bardzo podobnie do klas sztywnego ciała, których używaliśmy w tym tekście, w szczególności w przykładzie samolotu. Wszyscy zwykli podejrzani są tutaj, a komentarze w tym przykładowym kodzie wskazują, co reprezentuje każdy członek klasy. Jedną szczególną własnością, której jeszcze nie widziałeś, jest fRadius - jest to po prostu promień kuli bilardowej, który jest używany, gdy sprawdzamy kolizje i obliczamy siły oporu. Jak omówiliśmy w części 14, ponieważ w symulacji, która może zderzać się z wieloma obiektami, będziemy przeprowadzać iteracje przez wszystkie obiekty, sprawdzając kolizje podczas przechowywania danych kolizji. Ponieważ w tej symulacji nie ma zbyt wielu obiektów, tak naprawdę nie musimy dzielić przestrzeni gry, aby zoptymalizować testy wykrywania kolizji. Dane, które musimy przechowywać dla każdej kolizji, znajdują się w następującej strukturze kolizji:

```
typedef struct _Collision {

int body1;

int body2;

Vector vCollisionNormal;

Vector vCollisionPoint;

Vector vRelativeVelocity;

Vector vRelativeAcceleration;

Vector vCollisionTangent;

} Collision, *pCollision;
```

Dwie pierwsze właściwości są wskaźnikami dla dwóch ciał biorących udział w kolizji. Następna właściwość, vCollisionNormal, przechowuje normalny wektor w punkcie kontaktu kolizji z wektorem skierowanym na zewnątrz ciała2. Następna właściwość, vCollision Point, przechowuje współrzędne punktu kontaktu w globalnych współrzędnych. Ponieważ mamy do czynienia z kulami (kulami bilardowymi), kolizja kolizyjna zawsze składa się z pojedynczego punktu dla każdej zderzenia kulki lub kulki. Następne dwie właściwości przechowują względną prędkość i przyspieszenie między dwoma ciałami w punkcie kolizji. Dane są przechowywane odpowiednio w vRelativeVelocity i vRelativeAcceleration. Aby uchwycić tarcie w punkcie kontaktu, musimy znać wektor styczny do ciał w miejscu kontaktu. Ta styczna jest przechowywana w vCollisionTangent. Przygotowaliśmy kilka globalnych definicji do przechowywania kluczowych danych, co pozwala nam łatwo dostroić symulację:

#define BALLDIAMETER 0.05715f

#define BALLWEIGHT 1.612f

#define GRAVITY -9.87f

#define LINEARDRAGCOEFFICIENT 0.5f

#define ANGULARDRAGCOEFFICIENT 0.05f

#define FRICTIONFACTOR 0.5f

#define WSPÓŁCZESNOŚĆFREZENTACJA 0.8f

#define COEFFICIENTOFRESTITUTIONGROUND 0.1f

#define FRICTIONCOEFFICIENTBALLS 0.1f

#define FRICTIONCOEFFICIENTGROUND 0.1f

#define ROLLINGRESISTANECEEFFICIENT 0.025f

Pierwsze trzy definicje przedstawiają średnicę kulek w metrach, masę kulki niutonami, a przyspieszenie spowodowane grawitacją w m/s^2. Średnica i waga kulek są typowymi wartościami kulek bilardowych w stylu amerykańskim (to jest średnio 2,25 cala i 5,8 uncji). Pozostałe definicje są objaśniające i reprezentują współczynniki nieindukcyjne, takie jak współczynniki oporu i współczynniki restytucji. Wartości, które widzisz są tym, co wymyśliliśmy po dostrojeniu symulacji. Z pewnością je dostroisz, jeśli rozwiniesz własną grę bilardową. Do symulacji używamy trzech ważnych zmiennych globalnych, jak pokazano poniżej:

RigidBody Bodies [NUMBODIES];

Kolizje zderzeniowe [LICZBY * 8];

int NumCollisions = 0;

Ciała to tablica typów RigidBody i reprezentuje kolekcję kulek bilardowych. Tutaj zdefiniowaliśmy NUMBODIES jako 4, więc w tej symulacji są cztery kule bilardowe. Przyjęliśmy konwencję, że bila zawsze będzie Bodies [0]

Inicjalizacja

Na początku symulacji musimy zainicjować wszystkie cztery kule bilardowe. Używamy jednej funkcji, InitializeObjects, dla tego zadania. To długa funkcja, ale jest naprawdę prosta. Kod znajduje się na tej i następnych stronach. Bodies[0] to biała bila i tak właśnie jest ustawiono 50 średnic kulek wzdłuż

ujemnej osi X z dala od kulek obiektów. Dla tej liczby nie ma magii; wybieraliśmy to arbitralnie. Teraz celowo ustawiliśmy przesunięcie bili białej (wszystkich piłek, jeśli o to chodzi) do połowy średnicy, tak aby kulki dotykały właśnie stołu na początku symulacji. Aby rzut piłki bił od prawej do lewej, uzyskaliśmy początkową prędkość 7 m / s wzdłuż dodatniej osi X. Przy tej początkowej prędkości bila biała zacznie się ślizgać stół na krótką odległość, ponieważ również zaczyna się toczyć z powodu tarcia między piłką a stołem. W nadchodzącej próbce kodu można zobaczyć, że wszystkie inne właściwości kinematyczne są ustawione na 0 dla bili białej. W przypadku kulek obiektów wszystkie ich właściwości kinematyczne są ustawione na 0. Zachęcamy do eksperymentowania z różnymi wartościami początkowymi dla kinematycznych właściwości białej piłki. Na przykład, spróbuj ustawić prędkość kątową wokół dowolnej osi współrzędnych na wartość inną niż 0. Pozwoli to zobaczyć, jak obracająca się kula może poruszać się lekko w lewo lub w prawo w zależności od obrotu. Fajnie jest też zobaczyć, jak spin wpływa na kulki obiektów po zderzeniu z bilą. Oprócz ustawiania pozycji i kinematycznych właściwości kulek InitializeObjects inicjuje także właściwości masy. Wykorzystaliśmy zdefiniowaną wcześniej BALLWEIGHT podzieloną przez przyspieszenie grawitacyjne, aby określić masę kulki. void InitializeObjects (konfiguracja int)

```
{

float iRoll, iPitch, iYaw;

int i;

pływak Ixx, Iyy, Izz;

pływa;

////////////////////////////////////////////////////////////////////////////// ///

// Inicjalizacja bili białej:

// Ustaw pozycję początkową

Obiekty [0] .vPosition.x = -BALLDIAMETER * 50,0f;

Obiekty [0] .vPosition.y = 0,0f;

Obiekty [0] .vPosition.z = BALLDIAMETER / 2.0f;

// Ustaw prędkość początkową

s = 7,0;

Obiekty [0] .vVelocity.x = s;

Bodies [0] .vVelocity.y = 0.0f;

Bodies [0] .vVelocity.z = 0.0f;

Obiekty [0] .fSpeed = s;

// Ustaw początkową prędkość kątową

Obiekty [0] .vAngularVelocity.x = 0.0f; // obracaj wokół osi długiej

Bodies [0] .vAngularVelocity.y = 0.0f; // obracaj wokół osi poprzecznej

Ciała [0] .vAngularVelocity.z = 0.0f; // obrót wokół osi pionowej
```

```
Obiekty [0] .vAngularAcceleration.x = 0.0f;
Obiekty [0] .vAngularAcceleration.y = 0.0f;
Obiekty [0] .vAngularAcceleration.z = 0.0f;
Obiekty [0] .vAcceleration.x = 0.0f;
Obiekty [0] .vAcceleration.y = 0.0f;
Obiekty [0] .vAcceleration.z = 0.0f;
// Ustaw początkowe siły i momenty
Obiekty [0] .vForces.x = 0.0f;
Bodies [0] .vForces.y = 0.0f;
Obiekty [0] .vForces.z = 0.0f;
Bodies [0] .vMoments.x = 0.0f;
Bodies [0] .vMoments.y = 0.0f;
Bodies [0] .vMoments.z = 0.0f;
// Zeruj prędkość we współrzędnych przestrzeni ciała
Bodies [0] .vVelocityBody.x = 0.0f;
Bodies [0] .vVelocityBody.y = 0.0f;
Bodies [0] .vVelocityBody.z = 0.0f;
// Ustaw początkową orientację
iRoll = 0,0f;
iPitch = 0.0f;
iYaw = 0.0f;
Obiekty [0] .qOrientation = MakeQFromEulerAngles (iRoll, iPitch, iYaw);
// Ustaw właściwości masy
Ciała [0] .fMass = BALLWEIGHT / (- g);
Ixx = 2.0f * Bodies [0] .fMass / 5.0f * (BALLDIAMETER / 2 * BALLDIAMETER / 2);
Izz = Iyy = Ixx;
Obiekty [0] .mInertia.e11 = Ixx;
Obiekty [0] .mInertia.e12 = 0;
Ciała [0] .mInertia.e13 = 0;
Obiekty [0] .mInertia.e21 = 0;
Ciała [0] .mInertia.e22 = Iyy;
```

```
Obiekty [0] .mInertia.e23 = 0;
Obiekty [0] .mInertia.e31 = 0;
Obiekty [0] .mInertia.e32 = 0;
Ciała [0] .mInertia.e33 = Izz;
Bodies [0] .mInertiaInverse = Bodies [0] .mInertia.Inverse ();
Bodies[0].fRadius = BALLDIAMETER/2;
////////////////////////////////////////////////
// Initialize the other balls
for(i=1; i<NUMBODIES; i++)
{
// Set initial position
if(i==1)
{
Bodies[i].vPosition.x = 0.0;
Bodies[i].vPosition.y = -(BALLDIAMETER/2.0f+0.25*BALLDIAMETER);
Bodies[i].vPosition.z = BALLDIAMETER/2.0f;
} else if(i==2) {
Bodies[i].vPosition.x = 0.0;
Bodies[i].vPosition.y = BALLDIAMETER/2.0f+0.25*BALLDIAMETER;
Bodies[i].vPosition.z = BALLDIAMETER/2.0f;
} else {
Bodies[i].vPosition.x = -BALLDIAMETER;
Bodies[i].vPosition.y = 0.0f;
Bodies[i].vPosition.z = BALLDIAMETER/2.0f;
}
// Set initial velocity
Bodies[i].vVelocity.x = 0.0f;
Bodies[i].vVelocity.y = 0.0f;
Bodies[i].vVelocity.z = 0.0f;
Bodies[i].fSpeed = 0.0f;
// Set initial angular velocity
```

```
Bodies[i].vAngularVelocity.x = 0.0f;
Bodies[i].vAngularVelocity.y = 0.0f;
Bodies[i].vAngularVelocity.z = 0.0f;
Bodies[i].vAngularAcceleration.x = 0.0f;
Bodies[i].vAngularAcceleration.y = 0.0f;
Bodies[i].vAngularAcceleration.z = 0.0f;
Bodies[i].vAcceleration.x = 0.0f;
Bodies[i].vAcceleration.y = 0.0f;
Bodies[i].vAcceleration.z = 0.0f;
// Set the initial forces and moments
Bodies[i].vForces.x = 0.0f;
Bodies[i].vForces.y = 0.0f;
Bodies[i].vForces.z = 0.0f;
Bodies[i].vMoments.x = 0.0f;
Bodies[i].vMoments.y = 0.0f;
Bodies[i].vMoments.z = 0.0f;
// Zero the velocity in body space coordinates
Bodies[i].vVelocityBody.x = 0.0f;
Bodies[i].vVelocityBody.y = 0.0f;
Bodies[i].vVelocityBody.z = 0.0f;
// Set the initial orientation
iRoll = 0.0f;
iPitch = 0.0f;
iYaw = 0.0f;
Bodies[i].qOrientation = MakeQFromEulerAngles(iRoll, iPitch, iYaw);
// Set the mass properties
Bodies[i].fMass = BALLWEIGHT/(-g);
Ixx = 2.0f * Bodies[i].fMass / 5.0f * (BALLDIAMETER*BALLDIAMETER);
Izz = Iyy = Ixx;
Bodies[i].mInertia.e11 = Ixx;
Bodies[i].mInertia.e12 = 0;
```

```
Bodies[i].mInertia.e13 = 0;

Bodies[i].mInertia.e21 = 0;

Bodies[i].mInertia.e22 = Iyy;

Bodies[i].mInertia.e23 = 0;

Bodies[i].mInertia.e31 = 0;

Bodies[i].mInertia.e32 = 0;

Bodies[i].mInertia.e33 = Izz;

Bodies[i].mInertiaInverse = Bodies[i].mInertia.Inverse();

Bodies[i].fRadius = BALLDIAMETER/2;

}

}
```

Krok Symulacji

Podczas każdego kroku czasowego główna pętla symulacji wywołuje metodę StepSimulation. Ta funkcja, pokazana poniżej, jest niemal identyczna z funkcjami StepSimulation, które omówiliśmy w innych przykładach pokazanych w tej książce, więc naprawdę nie ma tutaj żadnych niespodzianek. StepSimulation najpierw wywołuje CalcObjectForces, o czym chwilę rozmawiamy, a następnie przystępuje do integracji równań ruchu dla każdej piłki. Używamy tutaj prostego schematu Eulera dla uproszczenia. Po integracji, StepSimulation wykonuje kilka wywołań funkcji, aby poradzić sobie z kolizjami. Wkrótce omówimy to.

```
void StepSimulation (float dtime)

{

Vectorr Ae;

int i;

float dt = dtime;

int check = NOCOLLISION;

int c = 0;

// Oblicz wszystkie siły i momenty na kulkach:

CalcObjectForces ();

// Integrate the equations of motion:

for(i=0; i<NUMBODIES; i++)

{

// Calculate the acceleration in earth space:

Ae = Bodies[i].vForces / Bodies[i].fMass;
```

```
Bodies[i].vAcceleration = Ae;
// Calculate the velocity in earth space:
Bodies[i].vVelocity += Ae * dt;
// Calculate the position in earth space:
Bodies[i].vPosition += Bodies[i].vVelocity * dt;
// Now handle the rotations:
float mag;
Bodies[i].vAngularAcceleration = Bodies[i].mInertiaInverse *
(Bodies[i].vMoments -
(Bodies[i].vAngularVelocity^
(Bodies[i].mInertia *
Bodies[i].vAngularVelocity)));
Bodies[i].vAngularVelocity += Bodies[i].vAngularAcceleration * dt;
// Calculate the new rotation quaternion:
Bodies[i].qOrientation += (Bodies[i].qOrientation *
Bodies[i].vAngularVelocity) *
(0.5f * dt);
// Now normalize the orientation quaternion:
mag = Bodies[i].qOrientation.Magnitude();
if (mag != 0)
Bodies[i].qOrientation /= mag;
// Calculate the velocity in body space:
Bodies[i].vVelocityBody = QVRotate(~Bodies[i].qOrientation,
Bodies[i].vVelocity);
// Get the angular velocity in global coords:
Bodies[i].vAngularVelocityGlobal = QVRotate(Bodies[i].qOrientation,
Bodies[i].vAngularVelocity);
// Get the angular acceleration in global coords:
Bodies[i].vAngularAccelerationGlobal = QVRotate(Bodies[i].qOrientation,
Bodies[i].vAngularAcceleration);
// Get the inverse intertia tensor in global coordinates
```

```
Matrix3x3 R, RT;

R = MakeMatrixFromQuaternion(Bodies[i].qOrientation);

RT = R.Transpose();

Bodies[i].mIeInverse = R * Bodies[i].mInertiaInverse * RT;

// Calculate the air speed:

Bodies[i].fSpeed = Bodies[i].vVelocity.Magnitude();

// Get the Euler angles for our information

Vector u;

u = MakeEulerAnglesFromQ(Bodies[i].qOrientation);

Bodies[i].vEulerAngles.x = u.x; // roll

Bodies[i].vEulerAngles.y = u.y; // pitch

Bodies[i].vEulerAngles.z = u.z; // yaw

}

// Handle Collisions :

check = CheckForCollisions();

if(check == COLLISION)

ResolveCollisions();

}
```

Obliczanie sił

Pierwsze wywołanie funkcji wykonane przez StepSimulation to wywołanie CalcObjectForces, które jest odpowiedzialne za obliczanie wszystkich sił na każdej z kul, z wyjątkiem sił kolizji. To jest to samo podejście stosowane w poprzednich przykładach. Cały kod źródłowy CalcObjectForces znajduje się tutaj:

```
void CalcObjectForces(void)

{

Vector Fb, Mb;

Vector vDragVector;

Vector vAngularDragVector;

int i, j;

Vector ContactForce;

Vector pt;

int check = NOCOLLISION;
```

```
pCollision pCollisionData;
Vector FrictionForce;
Vector fDir;
double speed;
Vector FRn, FRt;
for(i=0; i<NUMBODIES; i++)
{
// Reset forces and moments:
Bodies[i].vForces.x = 0.0f;
Bodies[i].vForces.y = 0.0f;
Bodies[i].vForces.z = 0.0f;
Bodies[i].vMoments.x = 0.0f;
Bodies[i].vMoments.y = 0.0f;
Bodies[i].vMoments.z = 0.0f;
Fb.x = 0.0f; Mb.x = 0.0f;
Fb.y = 0.0f; Mb.y = 0.0f;
Fb.z = 0.0f; Mb.z = 0.0f;
// Do drag force:
vDragVector = -Bodies[i].vVelocityBody;
vDragVector.Normalize();
speed = Bodies[i].vVelocityBody.Magnitude();
Fb += vDragVector * ((1.0f/2.0f)*speed * speed * rho *
LINEARDRAGCOEFFICIENT * pow(Bodies[i].fRadius,2) *
Bodies[i].fRadius*pi);
vAngularDragVector = -Bodies[i].vAngularVelocity;
vAngularDragVector.Normalize();
Mb += vAngularDragVector * (Bodies[i].vAngularVelocity.Magnitude() *
Bodies[i].vAngularVelocity.Magnitude() * rho * ANGULARDRAGCOEFFICIENT
* 4 * pow(Bodies[i].fRadius,2)*pi);
// Convert forces from model space to earth space:
Bodies[i].vForces = QVRotate(Bodies[i].qOrientation, Fb);
```

```
// Apply gravity:
Bodies[i].vForces.z += GRAVITY * Bodies[i].fMass;
// Save the moments:
Bodies[i].vMoments += Mb;
// Handle contacts with ground plane:
Bodies[i].vAcceleration = Bodies[i].vForces / Bodies[i].fMass;
Bodies[i].vAngularAcceleration = Bodies[i].mInertiaInverse *
(Bodies[i].vMoments -
(Bodies[i].vAngularVelocity^
(Bodies[i].mInertia *
Bodies[i].vAngularVelocity)));
// Resolve ground plane contacts:
FlushCollisionData();
pCollisionData = Collisions;
NumCollisions = 0;
if(DOCONTACT)
check = CheckGroundPlaneContacts(pCollisionData, i);
if((check == CONTACT) && DOCONTACT)
{ j = 0;
{
assert(NumCollisions <= 1);
ContactForce = (Bodies[i].fMass * (-Bodies[i].vAcceleration *
Collisions[j].vCollisionNormal)) *
Collisions[j].vCollisionNormal;
if(DOFRICTION)
{
double vt = fabs(Collisions[j].vRelativeVelocity *
Collisions[j].vCollisionTangent);
if(vt > VELOCITYTOLERANCE)
{
// Kinetic:
```

```
FrictionForce = (ContactForce.Magnitude() *
FRICTIONCOEFFICIENTGROUND) *
Collisions[j].vCollisionTangent;
} else {
// Static:
FrictionForce = (ContactForce.Magnitude() *
FRICTIONCOEFFICIENTGROUND * 2 *
vt/VELOCITYTOLERANCE) *
Collisions[j].vCollisionTangent;
}
} else
FrictionForce.x = FrictionForce.y = FrictionForce.z = 0;
// Do rolling resistance:
if(Bodies[i].vAngularVelocity.Magnitude() > VELOCITYTOLERANCE)
{
FRn = ContactForce.Magnitude() *
Collisions[j].vCollisionNormal;
Collisions[j].vCollisionTangent.Normalize();
Vector m = (Collisions[j].vCollisionTangent
*(ROLLINGRESISTANCECOEFFICIENT *
Bodies[i].fRadius))^FRn;
double mag = m.Magnitude();
Vector a = Bodies[i].vAngularVelocity;
a.Normalize();
Bodies[i].vMoments += -a * mag;
}
// accumlate contact and friction forces and moments
Bodies[i].vForces += ContactForce;
Bodies[i].vForces += FrictionForce;
ContactForce = QVRotate(~Bodies[i].qOrientation, ContactForce);
FrictionForce = QVRotate(~Bodies[i].qOrientation,
```

```
FrictionForce);

pt = Collisions[j].vCollisionPoint - Bodies[i].vPosition;

pt = QVRotate(~Bodies[i].qOrientation, pt);

Bodies[i].vMoments += pt^ContactForce;

Bodies[i].vMoments += pt^FrictionForce;

}

}

}

}
```

Jak widać, po wejściu do CalcObjectForces kod wchodzi do pętli, która przechodzi cyklicznie przez wszystkie obiekty kul bilardowych, obliczając siły działające na każdą z nich. Pierwsza obliczona siła to prosty opór aerodynamiczny. Obliczane są zarówno opory liniowe, jak i kątowe. Obliczamy wielkość liniowego oporu przez pomnożenie liniowego współczynnika oporu przez 1 / 2ρV2r2πr, gdzie ρ jest gęstością powietrza, V jest prędkością liniową kuli, a r jest promieniem kuli. Obliczamy wartość momentu oporu kątowego przez pomnożenie współczynnika oporu kątowego przez ωρ4r2π, gdzie ω jest prędkością kątową. Ponieważ ruch opóźniający przeciąganie, wektory liniowego oporu i kątowego oporu są po prostu przeciwne do liniowych i kątowych wektorów prędkości, odpowiednio. Normalizowanie tych wektorów, a następnie pomnożenie przez odpowiednie wartości oporu daje liniowy i kątowy wektor siły i momentu oporu. Następny zestaw sił obliczony w CalcObjectForces to siły kontaktowe między blatem a każdą kulką. Istnieją trzy siły kontaktowe. Jedną z nich jest siła pionowa, która utrzymuje kulki od upadku przez stół, druga to siła tarcia, która powstaje, gdy piłki przesuwają się wzdłuż stołu, a trzecia to opór toczenia. Siły te powstają tylko wtedy, gdy piłka styka się ze stołem. Porozmawiamy o tym, jak ustalić, czy piłka ma kontakt ze stołem w dalszej części tego rozdziału. Na razie zakładamy, że jest kontakt i jak obliczyć siły kontaktowe. Aby obliczyć siłę pionową wymaganą do utrzymania piłki przez stół, musimy najpierw obliczyć przyspieszenie liniowe piłki, które jest równe sumie sił (z wyłączeniem sił nacisku) działających na piłkę podzielonej przez masę piłki. Następnie pobieramy ujemny iloczyn punktowy tego przyspieszenia i wektora prostopadle do powierzchni stołu i mnożymy wynik przez masę kulki. Daje to wielkość siły kontaktu, a aby uzyskać wektor, pomnóżmy tę wielkość przez wektor jednostkowy prostopadle do powierzchni stołu. Poniższe dwa wiersze kodu wykonują następujące obliczenia:

```
Obiekty [i] .vAcceleration = Obiekty [i] .vForses / Bodies [i] .fMass;

ContactForce = (Obiekty [i] .fMass * (-Bodies [i] .vAcceleration *

Collision [j] .vCollisionNormal)) *

Collision [j] .vCollisionNormal;
```

Vector vCollisionNormal jest określany przez CheckGroundPlaneContacts, który omówimy później. Podobnie jak w przypadku kolizji, CheckGroundPlaneContacts wypełnia między innymi danymi strukturę danych zawierającą punkt kontaktu, prędkość względną pomiędzy piłką i stołem w punkcie kontaktu, a także styk wektorów normalnych i stycznych. Aby obliczyć ślizgową siłę tarcia, musimy najpierw określić styczny składnik względnej prędkości między piłką a stołem. Jeśli kulka ślizga się lub ślizga podczas toczenia, wówczas względna prędkość styczna będzie większa niż 0. Jeżeli piłka będzie

się toczyć bez przesuwania, wówczas prędkość względna będzie wynosić 0. W obu przypadkach będzie występować siła tarcia; w pierwszym przypadku użyjemy współczynnika tarcia kinetycznego, a w drugim wykorzystamy współczynnik tarcia statycznego. Siła tarcia obliczana jest w taki sam sposób jak pokazano w części 3. Poniższe wiersze kodu wykonują wszystkie te obliczenia:

```
ContactForce = (Bodies[i].fMass * (-Bodies[i].vAcceleration *

Collisions[j].vCollisionNormal)) *

Collisions[j].vCollisionNormal;

double vt = fabs(Collisions[j].vRelativeVelocity *

Collisions[j].vCollisionTangent);

if(vt > VELOCITYTOLERANCE)

{

// Kinetic:

FrictionForce = (ContactForce.Magnitude() *

FRICTIONCOEFFICIENTGROUND) *

Collisions[j].vCollisionTangent;

} else {

// Static:

FrictionForce = (ContactForce.Magnitude() *

FRICTIONCOEFFICIENTGROUND * 2 *

vt/VELOCITYTOLERANCE) *

Collisions[j].vCollisionTangent;

}
```

Pamiętaj, że siły te będą tworzyć momenty, jeśli nie będą działać przez środek ciężkości kulki. Tak więc, po obliczeniu i agregowaniu tych sił, musisz także rozwiązać wszystkie utworzone momenty i zagregować je za pomocą tych samych formuł, które pokazaliśmy w tym tekście. Poniższe linie kodu zajmują się tymi zadaniami:

```
// accumlate contact and friction forces and moments

Bodies[i].vForces += ContactForce;

Bodies[i].vForces += FrictionForce;

ContactForce = QVRotate(~Bodies[i].qOrientation, ContactForce);

FrictionForce = QVRotate(~Bodies[i].qOrientation,

FrictionForce);

pt = Collisions[j].vCollisionPoint - Bodies[i].vPosition;
```

```
pt = QVRotate(~Bodies[i].qOrientation, pt);

Bodies[i].vMoments += pt^ContactForce;

Bodies[i].vMoments += pt^FrictionForce;
```

Opór toczenia powstaje dzięki niewielkim odkształceniom w pokrytym tkaniną stole, tworząc małą przegrodę, którą piłka musi pokonać podczas jej toczenia. Ta dywizja przesuwa środek docisku tylko odrobinę w kierunku toczenia. To małe Przesunięcie powoduje moment pomnożony przez siłę nacisku. Wynikowy moment sprzeciwia się toczeniu; w przeciwnym razie, bez jakiegoś innego oporu, piłka nadal toczyłaby się nierealistycznie. Poniższy kod oblicza opór toczenia:

```
// Do rolling resistance:

if(Bodies[i].vAngularVelocity.Magnitude() > VELOCITYTOLERANCE)

{

FRn = ContactForce.Magnitude() *

Collisions[j].vCollisionNormal;

Collisions[j].vCollisionTangent.Normalize();

Vector m = (Collisions[j].vCollisionTangent

*(ROLLINGRESISTANCECOEFFICIENT *

Bodies[i].fRadius))^FRn;

double mag = m.Magnitude();

Vector a = Bodies[i].vAngularVelocity;

a.Normalize();

Bodies[i].vMoments += -a * mag;

}
```

Obsługa kolizji

Wcześniej widziałeś, gdzie StepSimulation wykonuje kilka wywołań funkcji, aby poradzić sobie z kontrolą kolizji i odpowiedzią. Zobaczysz także, gdzie CalcObjectForces wykonuje wywołanie funkcji, które sprawdza kontakty. Funkcje sprawdzające kolizje lub kontakty korzystają z tablicy Collisions, którą wcześniej pokazaliśmy. Ta tablica przechowuje wszystkie istotne informacje dotyczące zderzeń lub kontaktów - kolizji lub kolektora kontaktowego, wektorów normalnych i stycznych, prędkości względnej itd. Pierwszą funkcją, którą rozważymy, jest CheckForCollisions, która jest wywoływana pod koniec StepSimulation. CheckForCollisions sprawdza zderzenia kulki z piłką; mamy oddzielną funkcję do sprawdzania kolizji na stole, do którego dojdziemy później. CheckForCollisions opiera się na koncepcjach, które już omówiliśmy i pokazaliśmy w poprzednich rozdziałach, więc podsumujemy tutaj jego działanie. Zasadniczo, dwie kulki bilardowe kolidują, jeśli 1) zmierzają w kierunku siebie nawzajem, oraz 2) odległość dzieląca ich centra jest mniejsza niż lub równa sumie ich promieni. Jeśli oba te kryteria są spełnione, to kolizja jest zarejestrowana, a wszystkie istotne dane są przechowywane w tablicy Collisions:

```
int CheckForCollisions(void)
```

```
{
int status = NOCOLLISION;
int i, j;
Vector d;
pCollision pCollisionData;
int check = NOCOLLISION;
float r;
float s;
Vector tmp;
FlushCollisionData();
pCollisionData = Collisions;
NumCollisions = 0;
// check object collisions with each other
for(i=0; i<NUMBODIES; i++)
{
for(j=0; j<NUMBODIES; j++)
if((j!=i) && !CollisionRecordedAlready(i, j))
{
// do a bounding sphere check
d = Bodies[i].vPosition - Bodies[j].vPosition;
r = Bodies[i].fRadius + Bodies[j].fRadius;
s = d.Magnitude() - r;
if(s < COLLISIONTOLERANCE)
{// possible collision
Vector pt1, pt2, vel1, vel2, n, Vr;
float Vrn;
pt1 = (Bodies[i].vPosition + Bodies[j].vPosition)/2;
tmp = pt2 = pt1;
pt1 = pt1-Bodies[i].vPosition;
pt2 = pt2-Bodies[j].vPosition;
vel1 = Bodies[i].vVelocity +
```

```
(Bodies[i].vAngularVelocityGlobal^pt1);
vel2 = Bodies[j].vVelocity +
(Bodies[j].vAngularVelocityGlobal^pt2);
n = d;
n.Normalize();
Vr = (vel1 - vel2);
Vrn = Vr * n;
if(Vrn < -VELOCITYTOLERANCE)
{
// Have a collision so fill the data structure
assert(NumCollisions < (NUMBODIES*8));
if(NumCollisions < (NUMBODIES*8))
{
pCollisionData->body1 = i;
pCollisionData->body2 = j;
pCollisionData->vCollisionNormal = n;
pCollisionData->vCollisionPoint = tmp;
pCollisionData->vRelativeVelocity = Vr;
pCollisionData->vCollisionTangent = (n^Vr)^n;
pCollisionData->vCollisionTangent.Normalize();
pCollisionData++;
NumCollisions++;
status = COLLISION;
} }
}
}
}
for(i=0; i<NUMBODIES; i++)
{
check = NOCOLLISION;
assert(NumCollisions < (NUMBODIES*8));
```

```
check = CheckGroundPlaneCollisions(pCollisionData, i);

if(check == COLLISION)

{

status = COLLISION;

pCollisionData++;

NumCollisions++;

}

}

return status;

}
```

Ponieważ CheckForCollisions wykonuje pętle na wszystkich kulach, sprawdzając kolizje z każdą inną piłką, możliwe, że kolizja byłaby rejestrowana dwukrotnie. Na przykład, i-ta kula może być zderzająca się z j-tą piłką, a później j-ta kula również zostanie zderzona z i-tą piłką. Nie chcemy zapisywać tych informacji dwa razy, więc używamy poniższej funkcji, aby sprawdzić, czy kolizja między dwoma poszczególnymi kulkami jest już zarejestrowana. Jeśli tak, pomijamy ponowne zapisywanie danych:

```
bool CollisionRecordedAlready(int i, int j)

{

int k;

int b1, b2;

for(k=0; k<NumCollisions; k++)

{

b1 = Collisions[k].body1;

b2 = Collisions[k].body2;

if( ((b1 == i) && (b2 == j)) ||

((b1 == j) && (b2 == i)) )

return true;

}

return false;

}
```

Sprawdzanie kolizji z piłeczkami jest dość proste. Jeśli 1) okaże się, że piłka jest podszedł do stołu z prędkością większą niż 0 (lub niewielkim progiem), a 2) położenie pionowe kuli do jej środka jest mniejsze lub równe jej promieniu, a następnie rejestrujemy kolizję. CheckGroundPlaneCollisions obsługuje to dla nas:

```
int CheckGroundPlaneCollisions(pCollision CollisionData, int body1)
```

```
{
Vector tmp;
Vector vel1;
Vector pt1;
Vector Vr;
float Vrn;
Vector n;
int status = NOCOLLISION;
if(Bodies[body1].vPosition.z <= (Bodies[body1].fRadius))
{
pt1 = Bodies[body1].vPosition;
pt1.z = COLLISIONTOLERANCE;
tmp = pt1;
pt1 = pt1-Bodies[body1].vPosition;
vel1 = Bodies[body1].vVelocity/*Body*/ +
(Bodies[body1].vAngularVelocityGlobal^pt1);
n.x = 0;
n.y = 0;
n.z = 1;
Vr = vel1;
Vrn = Vr * n;
if(Vrn < -VELOCITYTOLERANCE)
{
// Have a collision so fill the data structure
assert(NumCollisions < (NUMBODIES*8));
if(NumCollisions < (NUMBODIES*8))
{
CollisionData->body1 = body1;
CollisionData->body2 = −1;
CollisionData->vCollisionNormal = n;
CollisionData->vCollisionPoint = tmp;
```

```
CollisionData->vRelativeVelocity = Vr;

CollisionData->vCollisionTangent = (n^Vr)^n;

CollisionData->vCollisionTangent.Reverse();

CollisionData->vCollisionTangent.Normalize();

status = COLLISION;

}

}

}

return status;

}
```

Rozwiązywanie kolizji, niezależnie od tego, czy zdarzają się kolizje kula-kula, czy kula-stół, wykorzystuje to samo podejście, które już wam pokazaliśmy. Dlatego też nie przejdziemy ponownie do kodu, a zamiast tego pokażemy tylko funkcję, która implementuje reakcję kolizji:

```
Bodies[b2].vAngularVelocity =

QVRotate(~Bodies[b2].qOrientation,

Bodies[b2].vAngularVelocityGlobal);

}

} else { // Ground plane:

fCr = COEFFICIENTOFRESTITUTIONGROUND;

pt1 = Collisions[i].vCollisionPoint - Bodies[b1].vPosition;

// Calculate impulse:

j = (-(1+fCr) * (Collisions[i].vRelativeVelocity *

Collisions[i].vCollisionNormal)) /

( (1/Bodies[b1].fMass) +

(Collisions[i].vCollisionNormal *

( ( (pt1 ^ Collisions[i].vCollisionNormal) *

Bodies[b1].mIeInverse )^pt1)));void ResolveCollisions(void)

{

int i;

double j;

Vector pt1, pt2, vB1V, vB2V, vB1AV, vB2AV;

float fCr = COEFFICIENTOFRESTITUTION;
```

```
int b1, b2;
float Vrt;
float muB = FRICTIONCOEFFICIENTBALLS;
float muG = FRICTIONCOEFFICIENTGROUND;
bool dofriction = DOFRICTION;
for(i=0; i<NumCollisions; i++)
{
b1 = Collisions[i].body1;
b2 = Collisions[i].body2;
if( (b1 != −1) && (b1 != b2) )
{
if(b2 != −1) // not ground plane
{
pt1 = Collisions[i].vCollisionPoint - Bodies[b1].vPosition;
pt2 = Collisions[i].vCollisionPoint - Bodies[b2].vPosition;
// Calculate impulse:
j = (-(1+fCr) * (Collisions[i].vRelativeVelocity *
Collisions[i].vCollisionNormal)) /
((1/Bodies[b1].fMass + 1/Bodies[b2].fMass) +
(Collisions[i].vCollisionNormal * ( ( (pt1 ^
Collisions[i].vCollisionNormal) *
Bodies[b1].mIeInverse )^pt1) ) +
(Collisions[i].vCollisionNormal * ( ( (pt2 ^
Collisions[i].vCollisionNormal) *
Bodies[b2].mIeInverse )^pt2) ) );
Vrt = Collisions[i].vRelativeVelocity *
Collisions[i].vCollisionTangent;
if(fabs(Vrt) > 0.0 && dofriction) {
Bodies[b1].vVelocity +=
((j * Collisions[i].vCollisionNormal) +
((muB * j) * Collisions[i].vCollisionTangent)) /
```

```
Bodies[b1].fMass;
Bodies[b1].vAngularVelocityGlobal +=
(pt1 ^ ((j * Collisions[i].vCollisionNormal) +
((muB * j) * Collisions[i].vCollisionTangent))) *
Bodies[b1].mIeInverse;
Bodies[b1].vAngularVelocity =
QVRotate(~Bodies[b1].qOrientation,
Bodies[b1].vAngularVelocityGlobal);
Bodies[b2].vVelocity +=
((-j * Collisions[i].vCollisionNormal) + ((muB *
j) * Collisions[i].vCollisionTangent)) /
Bodies[b2].fMass;
Bodies[b2].vAngularVelocityGlobal +=
(pt2 ^ ((-j * Collisions[i].vCollisionNormal) +
((muB * j) * Collisions[i].vCollisionTangent)))
* Bodies[b2].mIeInverse;
Bodies[b2].vAngularVelocity =
QVRotate(~Bodies[b2].qOrientation,
Bodies[b2].vAngularVelocityGlobal);

} else {
// Apply impulse:
Bodies[b1].vVelocity +=
(j * Collisions[i].vCollisionNormal) /
Bodies[b1].fMass;
Bodies[b1].vAngularVelocityGlobal +=
(pt1 ^ (j * Collisions[i].vCollisionNormal)) *
Bodies[b1].mIeInverse;
Bodies[b1].vAngularVelocity =
QVRotate(~Bodies[b1].qOrientation,
Bodies[b1].vAngularVelocityGlobal);
```

```
Bodies[b2].vVelocity -=
(j * Collisions[i].vCollisionNormal) /
Bodies[b2].fMass;
Bodies[b2].vAngularVelocityGlobal -=
(pt2 ^ (j * Collisions[i].vCollisionNormal)) *
Bodies[b2].mIeInverse;
Vrt = Collisions[i].vRelativeVelocity *
Collisions[i].vCollisionTangent;
if(fabs(Vrt) > 0.0 && dofriction) {
Bodies[b1].vVelocity +=
( (j * Collisions[i].vCollisionNormal) + ((muG *
j) * Collisions[i].vCollisionTangent) ) /
Bodies[b1].fMass;
Bodies[b1].vAngularVelocityGlobal +=
(pt1 ^ ((j * Collisions[i].vCollisionNormal) +
((muG * j) * Collisions[i].vCollisionTangent))) *
Bodies[b1].mIeInverse;
Bodies[b1].vAngularVelocity =
QVRotate(~Bodies[b1].qOrientation,
Bodies[b1].vAngularVelocityGlobal);
} else {
// Apply impulse:
Bodies[b1].vVelocity +=
(j * Collisions[i].vCollisionNormal) /
Bodies[b1].fMass;
Bodies[b1].vAngularVelocityGlobal +=
(pt1 ^ (j * Collisions[i].vCollisionNormal)) *
Bodies[b1].mIeInverse;
Bodies[b1].vAngularVelocity =
QVRotate(~Bodies[b1].qOrientation,
Bodies[b1].vAngularVelocityGlobal);
```

```
}

}

}

}

}
```

Ostatnią funkcją, którą musimy ci pokazać, jest CheckGroundPlaneContacts. Przypomnijmy, że ta funkcja jest wywoływana z CalcObjectForces w celu ustalenia, czy piłka znajduje się w spoczynkowym kontakcie ze stołem. Jeśli pionowa pozycja kuli jest mniejsza lub równa jej promieniu plus niewielka tolerancja, a prędkość pionowa kuli wynosi 0 (lub prawie tak, jak w przypadku niewielkiej tolerancji), wówczas uznajemy kulę w kontakcie ze stołem. Jeśli istnieje kontakt, odpowiednie dane są przechowywane w tablicy Collisions i używane do rozstrzygnięcia kontaktu, a nie kolizji, w CalcObjectForces:

```
int CheckGroundPlaneContacts(pCollision CollisionData, int body1)

{

Vector v1[8];

Vector tmp;

Vector u, v;

Vector f[4];

Vector vel1;

Vector pt1;

Vector Vr;

float Vrn;

Vector n;

int status = NOCOLLISION;

Vector Ar;

float Arn;

if(Bodies[body1].vPosition.z <= (Bodies[body1].fRadius + COLLISIONTOLERANCE))

{

pt1 = Bodies[body1].vPosition;

pt1.z = COLLISIONTOLERANCE;

tmp = pt1;

pt1 = pt1-Bodies[body1].vPosition;

vel1 = Bodies[body1].vVelocity/*Body*/ +
```

```
(Bodies[body1].vAngularVelocityGlobal^pt1);
n.x = 0;
n.y = 0;
n.z = 1;
Vr = vel1;
Vrn = Vr * n;
if(fabs(Vrn) <= VELOCITYTOLERANCE) // at rest
{
// Check the relative acceleration:
Ar = Bodies[body1].vAcceleration +
(Bodies[body1].vAngularVelocityGlobal ^
(Bodies[body1].vAngularVelocityGlobal^pt1)) +
(Bodies[body1].vAngularAccelerationGlobal ^ pt1);
Arn = Ar * n;
if(Arn <= 0.0f)
{
// We have a contact so fill the data structure
assert(NumCollisions < (NUMBODIES*8));
if(NumCollisions < (NUMBODIES*8))
{
CollisionData->body1 = body1;
CollisionData->body2 = −1;
CollisionData->vCollisionNormal = n;
CollisionData->vCollisionPoint = tmp;
CollisionData->vRelativeVelocity = Vr;
CollisionData->vRelativeAcceleration = Ar;
CollisionData->vCollisionTangent = (n^Vr)^n;
CollisionData->vCollisionTangent.Reverse();
CollisionData->vCollisionTangent.Normalize();
CollisionData++;
NumCollisions++;
```

```
status = CONTACT;
}
}
}
}
return status;
}
```

To wszystko do tego przykładu bilarda. Jak widać, użyliśmy zasadniczo tych samych metod przedstawionych w innych przykładach w tej książce, aby zaimplementować ten przykład. O nowych informacjach, które tutaj pokazaliśmy, to sposób obliczania oporu toczenia. Przy odrobinie wysiłku można połączyć materiał przedstawiony w tym przykładzie z materiałem ruchu pocisku przedstawionym w części 6, aby modelować wszystkie rodzaje piłek sportowych. Niezależnie od tego, czy modelujesz kulę bilardową odbijającą się od stołu, czy koszykówkę odbijającą się od tablicy, metody są takie same. Jedyne, co się zmieni, to współczynniki empiryczne, których używasz do modelowania każdej piłki i powierzchni. Baw się dobrze.