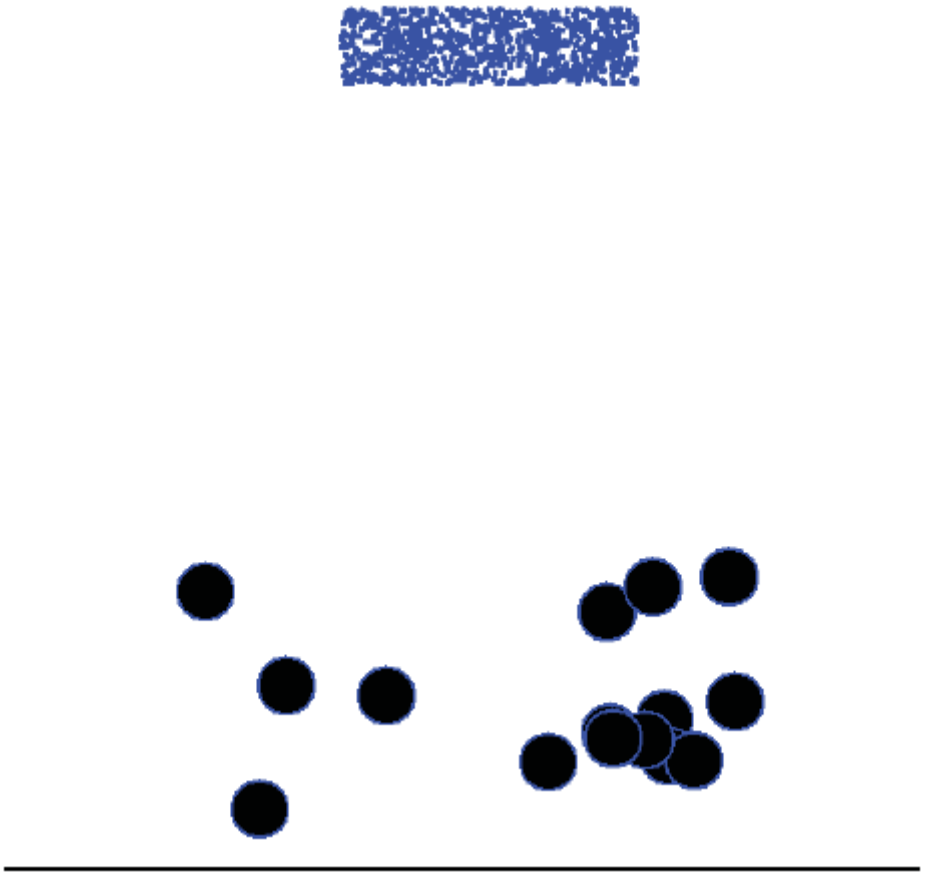
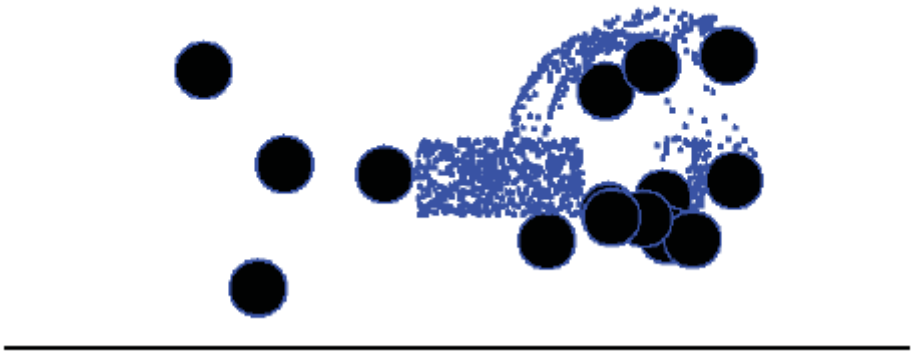
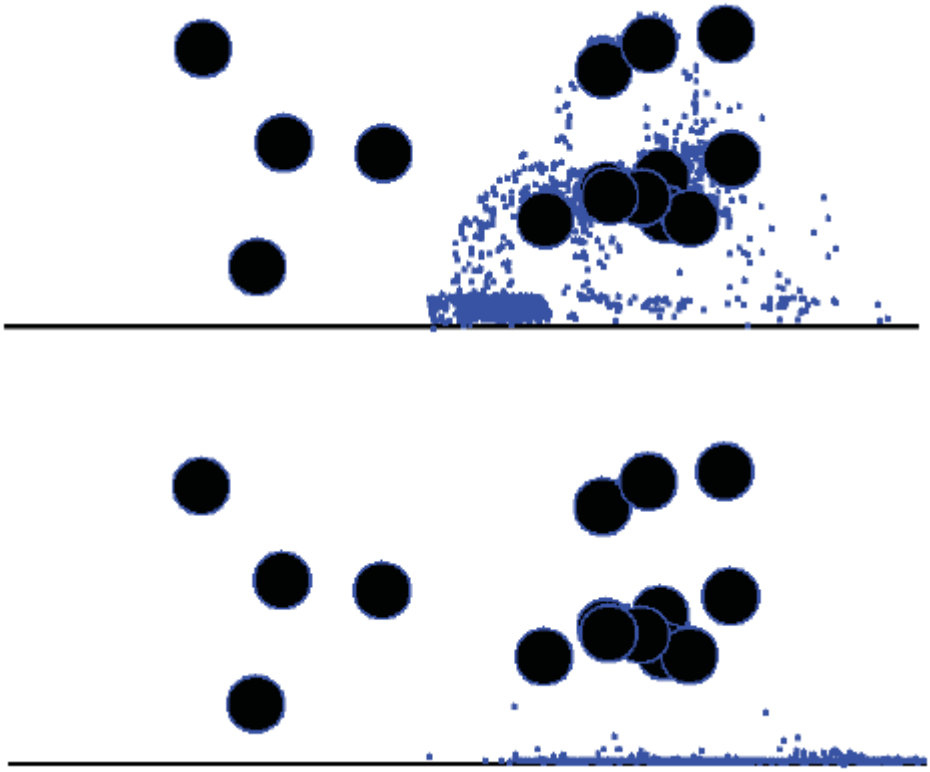
VIII. Cząsteczki

W tej części pokażemy Ci, jak zastosować to, czego nauczyłeś się w części 7 w prostym symulatorze cząstek. Zanim przejdziemy do specyfiki przedstawionego przykładu, rozważmy ogólnie cząstki. Cząstki to proste idealizacje, które można wykorzystać do symulacji wszelkiego rodzaju zjawisk lub efektów specjalnych w grze. Na przykład symulacje cząstek są często używane do symulacji dymu, ognia i wybuchów. Mogą być również wykorzystywane do symulacji wody, chmur pyłu i roju owadów, a także wielu innych rzeczy. Doprawdy, twoja wyobraźnia jest jedynym ograniczeniem. Cząstki nadają się do symulacji zarówno dyskretnych obiektów, takich jak odbijanie piłek, jak i ciągłych jak woda. Dodatkowo możesz łatwo przypisać szereg atrybutów do cząstek w zależności od tego, co modelujesz. Załóżmy na przykład, że modelujesz ogień za pomocą cząsteczek. Każda cząstka wzrośnie w powietrzu, a gdy się ochłodzi, jej kolor zmieni się, aż zniknie. Możesz powiązać kolor cząstek z ich temperaturą, która jest modelowana za pomocą termodynamiki. Atrybut, który chcesz śledzić, to temperatura cząstek. W poprzedniej pracy, AI for Game Programmers (O'Reilly), współautor tej książki David M. Bourg użył cząsteczek do przedstawienia rojów owadów, które roiły się, trzodę, ganiały i unikały w zależności od sztucznej inteligencji (AI). AI kontrolowało ich zachowanie, które następnie zostało zaimplementowane jako system cząstek przy użyciu zasad bardzo podobnych do tego, co zobaczysz w tym rozdziale. Cząstki nie są ograniczone do zbiorów niezależnych obiektów. W dalszej części tej książki dowiesz się, jak łączyć ze sobą cząsteczki za pomocą sprężyn, aby tworzyć odkształcalne obiekty, takie jak materiał. Cząsteczki są niezwykle wszechstronne i dobrze się uczysz, aby nauczyć się wykorzystywać swoją prostotę. Możesz użyć cząsteczek do modelowania piasku w prostej aplikacji telefonicznej symulującej klepsydrę. Powiązaj ten model piaskowy z technikami przyspieszeniomierza, o których dowiesz się w części 21, a będziesz mógł wykonać przepływ piasku, obracając telefon. Możesz łatwo wykorzystać cząstki do symulacji pocisków wylatujących z pistoletu. Wyobraź sobie pistolet Gatling wypuszczający grad ołowiu, wszystkie symulowane za pomocą prostych cząstek. A skoro mowa o wypluwaniu, to w jaki sposób wykorzystać cząstki do symulowania szczątków wyrzucanych z wybuchającego wulkanu jako specjalnego efektu w twojej grze przygodowej osadzonej w czasach prehistorycznych? Pamiętasz zabawkę Wooly Willy? Aby cząstki stały się bezpośrednią częścią gry, należy rozważyć zastosowanie dywersyjne, w którym można przeciągnąć stosy wirtualnych cząstek magnetycznych wokół zdjęcia portretowego, nadając komuś uroczą brodę lub wąsy podobne do Wooly Willy. Mam nadzieję, że teraz myślisz o twórczych sposobach wykorzystania cząstek w twoich grach. Tak, zajmijmy się wdrożeniem. Istnieją dwa podstawowe składniki do wdrożenia symulatora cząstek: model cząsteczkowy i integrator. (Cóż, możesz twierdzić, że trzecim podstawowym składnikiem jest renderer, w którym faktycznie rysujesz cząstki, ale to więcej grafiki niż fizyki, a my skupiamy się na modelowaniu i integracji w tej książce). Model bardzo prosto opisuje atrybuty cząstek w symulacji wraz z ich regułami zachowania. Mamy na myśli to w sensie fizycznym, a nie w sensie sztucznej inteligencji w tej książce, chociaż ogólnie rzecz biorąc, zaimplementowany przez ciebie model może bardzo dobrze uwzględniać odpowiednie reguły sztucznej inteligencji. Teraz integrator jest odpowiedzialny za aktualizację stanu cząsteczek podczas symulacji. W tym rozdziale stany cząstek będą opisywane przez ich pozycję i prędkość w danym momencie. Integrator zaktualizuje stan każdej cząstki pod wpływem wielu zewnętrznych sił bodźców, takich jak grawitacja, opór aerodynamiczny i kolizje. W dalszej części tego rozdziału przedstawimy szczegóły prostej symulacji cząstek w sposób przyrostowy. Pierwszym zadaniem będzie zasymulowanie zbioru cząsteczek pod wpływem samej grawitacji. Chociaż brzmi to elementarnie, taki przykład obejmuje wszystkie podstawowe składniki wspomniane wcześniej. Kiedy grawitacja jest pod kontrolą, pokażemy ci, jak zaimplementować opór powietrza i siły wiatru, aby wpłynąć na ruch cząstek. Następnie sprawimy, że rzeczy będą bardziej interesujące, pokazując, jak wdrożyć reakcję kolizji między cząstkami a płaszczyzną podłoża plus losowe przeszkody. Ten materiał kolizyjny będzie czerpał z materiału przedstawionego w Części 5, więc koniecznie przeczytaj ten rozdział pierwszy, jeśli jeszcze

tego nie zrobiłeś. Na rysunkach od 8-1 do 8-4 pokazano kilka ramek tej przykładowej symulacji wraz z przeszkodami i kolizjami. Użyj swojej wyobraźni, aby zwizualizować cząstki spadające pod wpływem grawitacji, aż uderzą o okrągłe obiekty, w którym to czasie odbijają się i ostatecznie osiadają na ziemi.







Podczas pracy w tej części pamiętaj o wszystkim, czego się tutaj uczysz będzie miało bezpośrednie zastosowanie do symulacji 2D i 3D. Części następujące po tym będą opierać się na omawianym tutaj materiale. Skupimy się na dwóch wymiarach w tej części, a później w książce pokażemy, jak rozszerzyć symulację na 3D. W rzeczywistości, w przypadku symulacji cząstek, prawie trywialne jest przejście od 2D do 3D. Zaufaj nam teraz.

Prosty model cząstek

Model cząstek, od którego zaczynamy, jest bardzo prosty. Wszystko, co chcemy osiągnąć na początku, polega na tym, że cząstki spadają pod wpływem grawitacji. Cząstki zostaną zainicjowane na wysokości powyżej płaszczyzny podłoża. Po rozpoczęciu symulacji, grawitacja będzie oddziaływać na każdą cząstkę, powodując ciągłe przyspieszanie w kierunku płaszczyzny podłoża, osiągając przy tym prędkość. Wyobraź sobie, że trzymasz garść małych kamieni wysoko, a następnie wypuszczasz je. Proste, co? Istnieje kilka atrybutów cząstek, które musimy wziąć pod uwagę nawet w przypadku tego prostego przykładu. Nasz model zakłada, że każda cząstka ma masę, a ustalona średnica (zakładamy, że nasze cząstki są kołami w 2D lub sferach w 3D), zajmuje pewną pozycję w przestrzeni i porusza się z pewną prędkością. Dodatkowo, każda cząstka działa na jakąś siatkową siłę zewnętrzną, która jest skupiskiem wszystkich sił działających na cząstkę. Te siły będą same w sobie grawitacją, ale ostatecznie będą zawierać siły oporu i uderzenia. Tworzymy klasę Particle, aby hermetyzować te atrybuty w następujący sposób:

```
class Particle {

public:

float fMass; // Total mass

Vector vPosition; // Position
```

```
Vector vVelocity; // Velocity

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

Vector vForces; // Total force acting on the particle

float fRadius; // Particle radius used for collision detection

Vector vGravity; // Gravity force vector

Particle(void); // Constructor

void CalcLoads(void); // Aggregates forces acting on the particle

void UpdateBodyEuler(double dt); // Integrates one time step

void Draw(void); // Draws the particle

}
```

Większość z tych atrybutów jest oczywista z uwagi na komentarze, które uwzględniliśmy. Zauważ, że kilka z tych atrybutów to typy wektorowe. Wektory te są zdefiniowane w niestandardowej bibliotece matematycznej. Ten typ sprawia, że zarządzanie wektorami i wykonywanie operacji arytmetycznych przy nich jest proste. Przypomnimy Ci tylko strukturę danych, której używa wektor: trzy skalary zwane x, y i z reprezentujące trzy wymiary lokalizacji lub ruchu w pewnym kierunku. Składnik z zawsze będzie ustawiony na 0 w przykładach z tego rozdziału. Powinieneś był zauważyć właściwość fSpeed w klasie Particle. Ta właściwość przechowuje wartość wektora prędkości, prędkość cząstki. Wykorzystamy to później, obliczając aerodynamiczne siły oporu. Zawarliśmy również właściwość typu Vector o nazwie vGravity, która przechowuje wektor siły grawitacji określający wielkość i kierunek działania siły grawitacji. To naprawdę nie jest konieczne, ponieważ można zakodować wektor siły grawitacji lub użyć zmiennej globalnej; jednak uwzględniliśmy to tutaj, aby zilustrować pewną kreatywną elastyczność. Na przykład możesz ponownie zdefiniować wektor grawitacji w grze, która wykorzystuje wejście przyspieszeniomierza do określenia kierunku grawitacji w odniesieniu do orientacji określonego urządzenia. I możesz mieć grę, w której niektóre artykuły reagują na różne wartości grawitacji w zależności od ich typu, które mogą być z twojej własnej mikstury. Oprócz własności, zauważysz kilka metod w klasie Cząstka. Konstruktor jest trywialny. Ustawia wszystko na 0, za wyjątkiem masy cząstki, promienia i wektora siły grawitacji. Poniższy kod ilustruje sposób zainicjowania wszystkiego:

```
Particle::Particle(void)

{

fMass = 1.0;

vPosition.x = 0.0;

vPosition.y = 0.0;

vPosition.z = 0.0;

vVelocity.x = 0.0;

vVelocity.y = 0.0;

vVelocity.z = 0.0;

fSpeed = 0.0;
```

```
vForces.x = 0.0;

vForces.y = 0.0;

vForces.z = 0.0;

fRadius = 0.1;

vGravity.x = 0;

vGravity.y = fMass * _GRAVITYACCELERATION;

}
```

Teraz jest chyba dobry czas na wyjaśnienie układu współrzędnych, który przyjęliśmy. Nasze pochodzenie świata znajduje się w lewym dolnym rogu okna programu przykładowego, z dodatnim x skierowanym w prawo, a dodatnim skierowanym w górę. Przyspieszenie spowodowane grawitacją działa w dół (to jest w ujemnym kierunku y). Używamy systemu jednostek SI i zdefiniowaliśmy przyspieszenie grawitacyjne w następujący sposób:

```
#define _GRAVITYACCELERATION -9.8f
```

To 9,8 m/s^2 w ujemnym kierunku y. Ustawiliśmy domyślnie masę każdej cząstki na 1 kg, co oznacza, że siła grawitacji wynosi 1 kg razy 9,8 m/s^2, czyli 9,8 newtonów siły. Ustawiliśmy promień każdej cząstki na jedną dziesiątą metra. Te masy i promienie są naprawdę arbitralne; możesz ustawić je na wszystko, co pasuje do tego, co modelujesz. Metoda CalcLoads jest odpowiedzialna za obliczanie wszystkich obciążeń-sił działających na cząstkę, z wyjątkiem sił uderzenia (poradzimy sobie z nimi później). Na razie jedyną siłą działającą na cząstki jest grawitacja lub po prostu waga każdej cząstki. CalcLoads jest bardzo prosty w tym momencie:

```
void Particle :: CalcLoads (void)

{

// Resetuj siły:

vForces.x = 0.0f;

vForces.y = 0.0f;

// Siły agregujące:

vForces + = vGravity;

}
```

Pierwszym zadaniem jest zresetowanie wektora vForces. vForces to wektor zawierający siłę netto działającą na cząstkę. Wszystkie te siły są agregowane w CalcLoads, co pokazuje linia vForces + = vGravity. Ponownie, jak dotąd, jedyną siłą do agregacji jest siła grawitacji.

Całkowanie

Metoda UpdateBodyEuler integruje równania ruchu dla każdej cząstki. Ponieważ mamy do czynienia z cząsteczkami, jedynym równaniem ruchu, z którym musimy się zmagać, jest to, że do tłumaczenia; obrót nie ma znaczenia dla cząstek (przynajmniej nie dla nas tutaj). Poniższy przykładowy kod pokazuje UpdateBodyEuler.

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

{

Vector a;

Vector dv;

Vector ds;

// Integrate equation of motion:

a = vForces / fMass;

dv = a * dt;

vVelocity += dv;

ds = vVelocity * dt;

vPosition += ds;

// Misc. calculations:

fSpeed = vVelocity.Magnitude();

}
```

Jak wskazuje nazwa tej metody, wdrożyliśmy metodę całkowania Eulera jak opisano w części 7. Stosując tę metodę, po prostu musimy podzielić agregat siły działające na cząstkę przez masę cząstki, aby uzyskać przyspieszenie cząstki. Linia kodu a = vForces / fMass właśnie to robi. Zauważ tutaj, że a jest Vector, podobnie jak vForces. fMass jest skalarem, a operator / zdefiniowany w klasie Vector zajmuje się dzieleniem każdego składnika wektora vForces przez fMass i ustawianiem odpowiednich komponentów w a. Zmiana prędkości, dv, jest równa przyspieszeniom razy zmiana czasu, dt. Nowa prędkość cząstki jest następnie obliczana przez linię vVelocity + = dv. Tutaj znowu vVelocity i dv są wektorami, a operator + = dba o arytmetykę wektorową. To jest pierwsze faktyczne całkowanie. Druga integracja ma miejsce w następnych kilku liniach, gdzie wyznaczamy przesunięcie cząstki i nową pozycję poprzez integrację jej prędkości. Linia ds = vVelocity * dt określa przesunięcie lub zmianę położenia cząstki, a linia vPosition + = ds oblicza nową pozycję, dodając przesunięcie do starej pozycji cząstki. Ostatnia linia w UpdateBodyEuler oblicza prędkość cząstki, przyjmując wartość wektora prędkości. Dla celów demonstracyjnych używanie metody Eulera jest w porządku. W rzeczywistej grze zaleca się bardziej niezawodną metodę opisaną w części 7.

Rendering

W tym przykładzie renderowanie cząstek jest dość trywialne. My tylko rysujemy kółka używając wywołań Windows API owiniętych w nasze własne funkcje, aby ukryć część specyficznego dla Windowsa kodu. Poniższy fragment kodu jest wszystkim, czego potrzebujemy do renderowania cząstek.

```
void Particle::Draw(void)

{

RECT r;

float drawRadius = max(2, fRadius);
```

```
SetRect(&r, vPosition.x − drawRadius,

_WINHEIGHT − (vPosition.y − drawRadius),

vPosition.x + drawRadius,

_WINHEIGHT − (vPosition.y + drawRadius));

DrawEllipse(&r, 2, RGB(0,0,0));

}
```

Oczywiście możesz tu użyć własnego kodu renderowania, a wszystko, co naprawdę musisz zwrócić, to konwersja ze współrzędnych globalnych na współrzędne okna. Pamiętaj, założyliśmy, że nasz globalny układ współrzędnych znajduje się w lewym dolnym rogu okna, natomiast układ współrzędnych rysowania okien ma swoje źródło w lewym górnym rogu okna. Aby przekształcić współrzędne w tym przykładzie, wystarczy odjąć pozycję y cząsteczki od wysokości okna.

Podstawowy symulator

Sercem tej symulacji zajmuje się klasa Cząstek opisana wcześniej. Jednak musimy pokazać, w jaki sposób ta klasa jest używana w kontekście głównego programu. Najpierw definiujemy kilka globalnych zmiennych w następujący sposób:

```
// Global Variables:

int FrameCounter = 0;

Particle Units[_MAX_NUM_UNITS];
```

FrameCounter zlicza liczbę kroków czasowych zintegrowanych przed aktualizacją ekranu graficznego. Liczba kroków, które możesz włączyć do symulacji, zanim zaktualizujesz wyświetlacz, to kwestia strojenia. Zobaczysz, jak to jest używane chwilowo podczas omawiania funkcji UpdateSimulation. Jednostki to tablica rodzajów cząstek. Będą one przedstawiać poruszające się cząstki w symulacji - te, które spadają z góry i odbijają się od okrągłych obiektów, które dodamy później. W większości przypadków każda jednostka jest inicjowana zgodnie z pokazanym wcześniej konstruktorem Particle. Jednak wszystkie ich pozycje są u źródła, dlatego wywołujemy następującą funkcję Initialize, aby losowo rozmieścić cząstki w górnej środkowej części ekranu w prostokącie o szerokości _SPAWN_AREA_R * 4 i wysokości _SPAWN_AREA_R, gdzie _SPAWN_AREA_R jest tylko globalnym określeniem, które wymyśliliśmy.

```
bool Initialize(void)

{

int i;

GetRandomNumber(0, _WINWIDTH, true);

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

{

Units[i].vPosition.x = GetRandomNumber(_WINWIDTH/2-_SPAWN_AREA_R*2,

_WINWIDTH/2+_SPAWN_AREA_R*2, false);
```

```
Units[i].vPosition.y = _WINHEIGHT −
GetRandomNumber(_WINHEIGHT/2-_SPAWN_AREA_R,
_WINHEIGHT/2, false);
}
return true;
}
```

OK, teraz rozważmy UpdateSimulation, jak pokazano w poniższym fragmencie kodu. Ta funkcja jest wywoływana w każdym cyklu za pośrednictwem głównej pętli komunikatów programu i odpowiada za przechodzenie przez wszystkie jednostki, wykonywanie odpowiednich wywołań funkcji w celu aktualizacji ich pozycji i renderowania sceny.

```
{
double dt = _TIMESTEP;
int i;
// initialize the back buffer
if(FrameCounter >= _RENDER_FRAME_COUNT)
{
ClearBackBuffer();
}
// update the particles (Units)
for(i=0; i<_MAX_NUM_UNITS; i++)
{
Units[i].CalcLoads();
Units[i].UpdateBodyEuler(dt);
if(FrameCounter >= _RENDER_FRAME_COUNT)
{
Units[i].Draw();
}
if(Units[i].vPosition.x > _WINWIDTH) Units[i].vPosition.x = 0;
if(Units[i].vPosition.x < 0) Units[i].vPosition.x = _WINWIDTH;
if(Units[i].vPosition.y > _WINHEIGHT) Units[i].vPosition.y = 0;
if(Units[i].vPosition.y < 0) Units[i].vPosition.y = _WINHEIGHT;
}
```

```
// Render the scene if required

if(FrameCounter >= _RENDER_FRAME_COUNT) {

CopyBackBufferToWindow();

FrameCounter = 0;

} else

FrameCounter++;

}
```

Dwie zmienne lokalne w UpdateSimulation to dt i i. i jest trywialne i służy jako zmienna licznika. dt reprezentuje małą, ale skończoną ilość czasu, w sekundach którym każdy krok integracji jest podejmowany. Globalny define_TIMESTEP przechowuje krok czasu, który ustawiliśmy na 0,1 sekundy. Następny segment kodu sprawdza wartość licznika klatek, a jeśli licznik ramek osiągnął określoną liczbę klatek, przechowywany w _RENDER_FRAME_COUNT, a następnie bufor tylny jest wyczyszczony, aby przygotować go do rysowania i ostatecznie kopiowania na ekran. Następna sekcja kodu pod komentarzem aktualizuje cząstki właśnie poprzez wywołanie metod CalcLoads i UpdateBodyEuler każdej Jednostki. Te dwie linie odpowiadają za aktualizację wszystkich sił działających na każdą cząstkę, a następnie integrują równanie ruchu dla każdej cząstki. Następne kilka linii w pętli for narysuje każdą cząstkę, jeśli jest to wymagane, i zawija każdą pozycję cząsteczki wokół zakresu okna, jeśli będą one rozwijać się poza krawędź okna. Zwróć uwagę, że w tym przykładzie używamy współrzędnych okna.

Wdrażanie sił zewnętrznych

Na początek dodamy kilka prostych sił zewnętrznych - ciągłe opory powietrza i siłę wiatru. Użyjemy formuł przedstawionych w części 3 do przybliżenia tych sił, traktując je w podobny sposób. Przypomnijmy, że nadal opór powietrza jest aerodynamiczną siłą oporu działającą na obiekt poruszający się z pewną prędkością przez nieruchome powietrze. Przeciąganie zawsze działa, aby przeciwstawić się ruchowi. Chociaż użyjemy tych samych formuł do obliczenia siły wiatru, pamiętajmy, że siła wiatru niekoniecznie musi powstrzymywać ruch. Możesz mieć tylny wiatr popychający obiekt, lub wiatr może pochodzić z dowolnego kierunku z elementami, które popychają obiekt na boki. W tym przykładzie przyjmujemy wiatr boczny od lewej do prawej, działając tak, aby popychać cząstki w bok, z ciągłym oporem powietrza, który opiera się ich opadaniu. Kiedy później dodamy kolizje, ta sama formuła oporu będzie działać, aby przeciwstawić się ruchowi cząstek w dowolnym kierunku, w którym podróżują, gdy się odbijają. Formuła, której użyjemy do modelowania ciągłego oporu powietrza, to:

$$F_d = ½\,\rho V^2 A C_d$$

Tutaj Fd jest wielkością siły oporu. Jego kierunek jest wprost przeciwny do prędkości poruszającej się cząstki. ρ jest gęstością powietrza, przez którą porusza się cząstka, V jest wielkością prędkości cząstki (jej prędkością), A jest rzutowanym obszarem cząstki tak, jakby była kulą, a Cd jest współczynnikiem oporu. Możemy użyć tej samej formuły do oszacowania siły wiatru popychającej cząsteczkę w bok. Jedyna różnica polega na tym, że V jest prędkością wiatru, a kierunek uzyskanej siły wiatru pochodzi z założonego kierunku od lewej do prawej. Aby dodać te dwie siły do naszej symulacji, musimy wprowadzić kilka dodatków do metody CalcLoads klasy Particle. Poniższy kod pokazuje, jak wygląda teraz CalcLoads. Pamiętajcie, że wszystko, co mieliśmy tutaj pierwotnie, to pierwsze trzy linie wykonywalnego kodu pokazane poniżej - kod, który inicjuje zagregowany wektor siły, a następnie linia

kodu, która dodaje siłę ze względu na grawitację do siły skupienia. Cała reszta kodu w tej metodzie jest nowa.

```
void Particle::CalcLoads(void)

{

// Reset forces:

vForces.x = 0.0f;

vForces.y = 0.0f;

// Aggregate forces:

// Gravity

vForces += vGravity;

// Still air drag

Vector vDrag;

Float fDrag;

vDrag-=vVelocity;

vDrag.Normalize();

fDrag = 0.5 * _AIRDENSITY * fSpeed * fSpeed *

(3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;

vDrag*=fDrag;

vForces += vDrag;

// Wind

Vector vWind;

vWind.x = 0.5 * _AIRDENSITY * _WINDSPEED *

_WINDSPEED * (3.14159 * fRadius * fRadius) *

_DRAGCOEFFICIENT;

vForces += vWind;

}
```

Po dodaniu siły do grawitacji do agregatu deklarowane są dwie nowe zmienne lokalne. vDrag to wektor, który będzie reprezentował siłę ciągnącego powietrza. fDrag to wielkość tej siły oporu. Ponieważ wiemy, że wektor siły oporu jest dokładnie przeciwny do wektora prędkości cząstki, możemy zrównać vDrag z ujemnym vVelocity, a następnie znormalizować vDrag, aby uzyskać wektor jednostkowy wskazujący w kierunku przeciwnym do prędkości cząstki. Następnie obliczamy wielkość siły oporu za pomocą wzoru pokazanego wcześniej. Linia ta obsługuje to:

```
fDrag = 0,5 * _AIRDENSITY * fSpeed * fSpeed *
```

(3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;

Tutaj _AIRDENSITY jest globalną definicją reprezentującą gęstość powietrza, którą ustawiliśmy na 1,23 kg / m3 (standardowe powietrze w 15 ° C). fSpeed to prędkość cząstki: wielkość jej prędkości. Linia 3.14159 * fRadius * fRadius reprezentuje rzutowany obszar cząstki, zakładając, że cząstka jest kulą. I wreszcie, _DRAGCOEFFICIENT jest współczynnikiem oporu, który ustawiliśmy na 0,6. Wybraliśmy tę wartość z wykresu współczynnika oporu dla gładkiej kuli w porównaniu do liczby Reynoldsa pokazanej w rozdziale 6. Po prostu sprawdziliśmy wartość w zakresie liczb Reynoldsa od 1e4 do 1e5. Masz tutaj wybór strojenia wartości współczynnika oporu w celu uzyskania pożądanego efektu, lub możesz utworzyć dopasowanie krzywej lub tabelę odnośników, aby wybrać współczynnik oporu odpowiadający liczbie Reynoldsa ruchomej cząstki. Teraz, gdy mamy siłę oporu, po prostu pomnożymy tę siłę przez wektor przeciągania jednostki, aby uzyskać końcowy wektor siły przeciągania. Ten wektor jest następnie agregowany w vForces. Obsługujemy siłę wiatru w podobny sposób, z kilkoma różnicami w szczegółach. Po pierwsze, ponieważ wiemy, że wektor siły wiatru jednostkowego znajduje się w dodatnim kierunku x (tzn. Działa od lewej do prawej), możemy po prostu ustawić składową x wektora siły wiatru, vWind, do wielkości siły wiatru . Obliczyliśmy tę siłę wiatru przy użyciu tej samej formuły, którą używaliśmy do ciągłego oporu powietrza, z wyjątkiem użycia prędkości wiatru zamiast prędkości cząstki. Użyliśmy _WINDSPEED, globalnego definiującego, aby reprezentować prędkość wiatru, którą ustawiliśmy na 10 m / s (około 20 węzłów). Wreszcie siły wiatru są agregowane w vForces. Na tym etapie cząstki będą podlegały wpływowi grawitacji, ale nie tak szybko, jak by miały bez siły oporu. A teraz będą również dryfować w prawo ze względu na siłę wiatru. Po dodaniu siły do grawitacji do agregatu deklarowane są dwie nowe zmienne lokalne. vDrag to wektor, który będzie reprezentował siłę ciągnącego powietrza. fDrag to wielkość tej siły oporu. Ponieważ wiemy, że wektor siły oporu jest dokładnie przeciwny do wektora prędkości cząstki, możemy zrównać vDrag z ujemnym vVelocity, a następnie znormalizować vDrag, aby uzyskać wektor jednostkowy wskazujący w kierunku przeciwnym do prędkości cząstki. Następnie obliczamy wielkość siły oporu za pomocą wzoru pokazanego wcześniej. Linia ta obsługuje to:

fDrag = 0,5 * _AIRDENSITY * fSpeed * fSpeed *

(3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;

Tutaj _AIRDENSITY jest globalną definicją reprezentującą gęstość powietrza, którą ustawiliśmy na 1,23 kg/m^3 (standardowe powietrze w 15°C). fSpeed to prędkość cząstki: wielkość jej prędkości. Linia 3.14159 * fRadius * fRadius reprezentuje rzutowany obszar cząstki, zakładając, że cząstka jest kulą. I wreszcie, _DRAGCOEFFICIENT jest współczynnikiem oporu, który ustawiliśmy na 0,6. Wybraliśmy tę wartość z wykresu współczynnika oporu dla gładkiej kuli w porównaniu do liczby Reynoldsa pokazanej w rozdziale 6. Po prostu sprawdziliśmy wartość w zakresie liczb Reynoldsa od 1e4 do 1e5. Masz tutaj wybór strojenia wartości współczynnika oporu w celu uzyskania pożądanego efektu, lub możesz utworzyć dopasowanie krzywej lub tabelę odnośników, aby wybrać współczynnik oporu odpowiadający liczbie Reynoldsa ruchomej cząstki. Teraz, gdy mamy siłę oporu, po prostu pomnożymy tę siłę przez wektor przeciągania jednostki, aby uzyskać końcowy wektor siły przeciągania. Ten wektor jest następnie agregowany w vForces. Obsługujemy siłę wiatru w podobny sposób, z kilkoma różnicami w szczegółach. Po pierwsze, ponieważ wiemy, że wektor siły wiatru jednostkowego znajduje się w dodatnim kierunku x (tzn. Działa od lewej do prawej), możemy po prostu ustawić składową x wektora siły wiatru, vWind, do wielkości siły wiatru . Obliczyliśmy tę siłę wiatru przy użyciu tej samej formuły, którą używaliśmy do ciągłego oporu powietrza, z wyjątkiem użycia prędkości wiatru zamiast prędkości cząstki. Użyliśmy _WINDSPEED, globalnego definiującego, aby reprezentować prędkość wiatru, którą ustawiliśmy na 10 m/s (około 20 węzłów). Wreszcie siły wiatru są agregowane w vForces. Na tym etapie

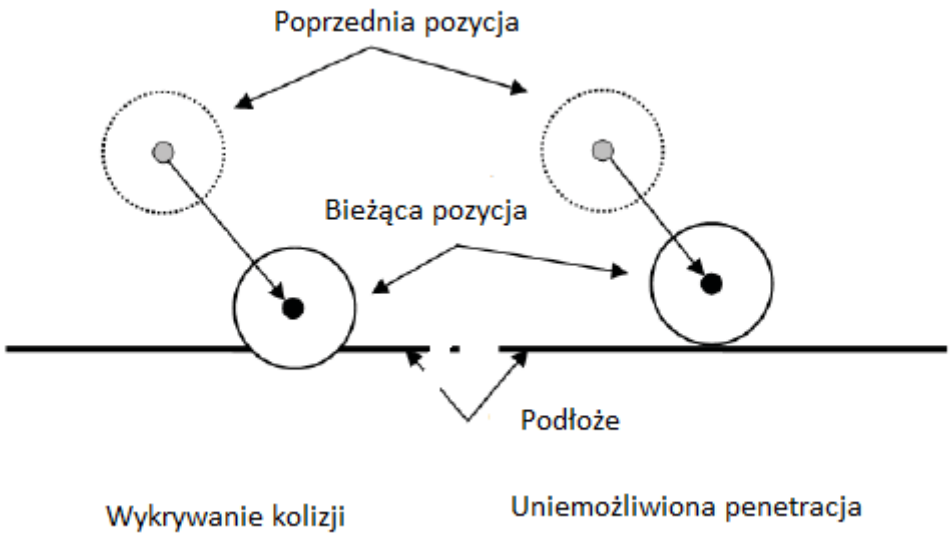
cząstki będą podlegały wpływowi grawitacji, ale nie tak szybko, jak by miały bez siły oporu. A teraz będą również dryfować w prawo ze względu na siłę wiatru.

Wdrażanie kolizji

Dodanie sił zewnętrznych sprawiło, że symulacja stała się nieco bardziej interesująca. Jednak, aby naprawdę sprawić, że się pojawią, dodamy kolizje. W szczególności zajmujemy się kolizjami między cząstkami a kolizjami między cząstkami. Jeśli jeszcze nie przeczytałeś Części 5, która dotyczy kolizji, powinieneś, ponieważ w tym przykładzie zastosujemy zasady opisane w tej części. W tym przykładzie zastosujemy wystarczającą obsługę kolizji, aby umożliwić cząstkom odbijanie się od ziemi i obiektów okrągłych, a wrócimy do obsługi kolizji bardziej szczegółowo w części 10. Materiały w tym rozdziale powinny zaostrzyć apetyt. Zaczniemy od łatwiejszego przypadku kolizji między cząstkami.

Zderzenia cząstek z ziemią

Zasadniczo to, co zamierzamy osiągnąć za pomocą wykrywania kolizji między cząstkami aby zapobiec przedostawaniu się cząstek przez płaszczyznę podłoża określoną przy jakiejś współrzędnej y. Wyobraź sobie poziomą nieprzenikalną powierzchnię, której cząsteczki nie mogą przejść przez. Jest kilka rzeczy, które musimy zrobić, aby wykryć cząstkę rzeczywiście zderzając się z płaszczyzną podłoża. Jeśli tak, to musimy poradzić sobie z kolizją, sprawdzenie, czy cząstki reagują w odpowiedni sposób. Lewa strona rysunku 8-5 ilustruje scenariusz kolizji.



Łatwo ustalić, czy doszło do kolizji. W danym czasie trwania symulacji cząstka mogła przesunąć się z poprzedniej pozycji (pozycja w poprzednim punkcie czasowym) do aktualnej pozycji. Jeśli ta aktualna pozycja umieści współrzędną środka ciężkości cząstki w promieniu jednej cząstki płaszczyzny podłoża, może wystąpić kolizja. Mówimy, że może dlatego, że inne kryteria musimy sprawdzić, aby ustalić, czy zdarza się kolizja, niezależnie od tego, czy cząstka porusza się w kierunku płaszczyzny podłoża. Jeśli cząstka porusza się w kierunku płaszczyzny podłoża i znajduje się w promieniu jednej płaszczyzny podłoża, następuje kolizja. Może się również zdarzyć, że cząstka przeszła całkowicie przez płaszczyznę podłoża, w którym to przypadku zakładamy kolizję. Aby zapobiec takiej penetracji płaszczyzny podłoża, musimy zrobić dwie rzeczy. Najpierw musimy zmienić położenie cząstki tak, aby dotykała ona płaszczyzny podstawy, jak pokazano po prawej stronie rysunku 8-5. Po drugie, musimy zastosować siłę uderzenia wynikającą z kolizji, aby zmusić cząstkę albo do zejścia w dół do płaszczyzny uziemienia, albo odejścia od płaszczyzny podłoża. Wszystkie te kroki tworzą wykrywanie kolizji i reakcję. Jest kilka zmian

i dodatków, które musimy wprowadzić w kodzie przykładowym, aby zaimplementować wykrywanie kolizji i odpowiedź typu "ziemia-ziemia". Zacznijmy od klasy Particle. Do Particle dodaliśmy trzy nowe właściwości w następujący sposób:

```
class Particle {

.

.

.

Vector vPreviousPosition;

Vector vImpactForces;

bool bCollision;

.

.

.

};
```

vPreviousPosition służy do zapisania pozycji cząstki w poprzednim kroku to znaczy, w czasie t-dt. vImpactForces służy do agregowania wszystkich działających sił uderzenia na cząstce. Zobaczysz później, że możliwe jest zderzenie cząstek z więcej niż jeden przedmiot w tym samym czasie. bCollision jest po prostu flagą używaną do wskazania, czy kolizja została wykryta z cząsteczką w bieżącym kroku czasowym. Jest to ważne, ponieważ w momencie wystąpienia kolizji przyjmujemy, że jedynymi siłami działającymi na cząstkę są siły uderzenia; wszystkie inne siły - grawitacja, opór i wiatr - są ignorowane w tym momencie. Używamy bCollision w zaktualizowanej metodzie CalcLoads:

```
void Particle::CalcLoads(void)

{

// Reset forces:

vForces.x = 0.0f;

vForces.y = 0.0f;

// Aggregate forces:

if(bCollision) {

// Add Impact forces (if any)

vForces += vImpactForces;

} else {

// Gravity

vForces += vGravity;

// Still air drag
```

```
Vector vDrag;

float fDrag;

vDrag -= vVelocity;

vDrag.Normalize();

fDrag = 0.5 * _AIRDENSITY * fSpeed * fSpeed *

(3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;

vDrag *= fDrag;

vForces += vDrag;

// Wind

Vector vWind;

vWind.x = 0.5 * _AIRDENSITY * _WINDSPEED * _WINDSPEED *

(3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;

vForces += vWind;

}

}
```

Jedyną różnicą między tą wersją CalcLoads i poprzednią jest to, że dodaliśmy warunkowe, jeśli (bCollision) {...} else {...}. Jeśli bCollision jest prawdą, to mamy do czynienia z kolizją i jedynymi siłami, które się łączą, są siły uderzenia. Jeśli nie ma kolizji, jeśli bCollision jest fałszywy, to siły nie uderzenia są agregowane w zwykły sposób. Być może zauważyłeś, że agregujemy siły uderzenia w tym przykładzie. Jest to podejście alternatywne wobec tego pokazanego w części 5. Tam pokazaliśmy, jak obliczyć impuls i zmienić prędkość obiektu w odpowiedzi na kolizję, stosując zachowanie pędu. Nadal obliczamy impulsy, tak jak w Części 5; jednak w tym przykładzie obliczymy siłę uderzenia na podstawie tego impulsu i zastosujemy tę siłę do zderzających się obiektów. Pozwolimy, by integrator numeryczny zintegruje tę siłę, aby uzyskać nowe prędkości krążącej cząsteczki. Obie metody działają, a my pokażemy Ci przykład poprzedniej metody później. Pokazujemy tę ostatnią metodę tylko po to, by zilustrować niektóre alternatywy. Zaletą tej ostatniej metody jest to, że jest to łatwe obliczyć siły uderzenia z powodu wielu uderzeń i pozwolić integratorowi zająć się nimi wszystkimi naraz. Teraz, po wprowadzeniu tych zmian w Particle, musimy dodać linię kodu do Up DateSimulation, jak pokazano tutaj:

```
void UpdateSimulation (void)

{

.

.

.

// zaktualizuj jednostki sterowane komputerowo:

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

```
{
Units [i] .bCollision = CheckForCollisions (& (Jednostki [i]));
Units [i] .CalcLoads ();
Units [i] .UpdateBodyEuler (dt);
.
.
.
} // end i-loop
.
.
.
}
```

Nowa linia to Units[i] .bCollision = CheckForCollisions (& (Units [i]) ;. Check ForCollisions to nowa funkcja, która przyjmuje podaną jednostkę, której wskaźnik jest przekazywany jako argument i sprawdza, czy nie koliduje z niczym - w tym przypadku z ziemią. Po wykryciu kolizji funkcja CheckForCollisions oblicza także siłę uderzenia i zwraca wartość true, aby poinformować nas o kolizji. CheckForCollisions jest następująca:

```
bool CheckForCollisions (Particle * p)
{
Vector n;
Vector vr;
float vrn;
float J;
Vector Fi;
bool hasCollision = false;
// Zresetuj zagregowaną siłę uderzenia
p-> vImpactForces.x = 0;
p-> vImpactForces.y = 0;
// sprawdź kolizje z płaszczyzną uziemienia
if (p-> vPosition.y <= (_GROUND_PLANE + p-> fRadius)) {
n.x = 0;
n.y = 1;
```

```
vr = p-> vVelocity;

vrn = vr * n;

// sprawdź, czy cząstka porusza się w kierunku ziemi

if (vrn <0.0) {

J = - (vr * n) * (_RESTITUTION + 1) * p-> fMass;

Fi = n;

Fi * = J / _TIMESTEP;

p-> vImpactForces + = Fi;

p-> vPosition.y = _GROUND_PLANE + p-> fRadius;

p-> vPosition.x = ((_GROUND_PLANE + p-> fRadius -

p-> vPreviousPosition.y) /

(p-> vPosition.y - p-> vPreviousPosition.y) *

(p-> vPosition.x - p-> vPreviousPosition.x)) +

p-> vPreviousPosition.x;

hasCollision = true;

}

}

return hasCollision;

}
```

CheckForCollisions dokonuje dwóch kontroli: 1) sprawdza, czy cząstka nie jnawiązuje u lub przechodzi przez płaszczyznę ziemi; i 2) sprawdza, aby się upewnić czy cząstka faktycznie zbliża się do płaszczyzny podłoża. Pamiętaj, że cząstka może być w kontakcie z samolotem naziemnym zaraz po tym, jak kolizja została potraktowana z cząstka odrywająca się od ziemi. W takim przypadku nie chcemy rejestrować innego kolizja. Rozważmy szczegóły tej funkcji, zaczynając od zmiennych lokalnych. n jest wektorem reprezentującym wektor normalny jednostki, od płaszczyzny uziemienia do kolidującej z nim cząstki. W przypadku zderzenia z ziemią, w tym przykładzie wektor normalny jednostki jest zawsze w górę, ponieważ płaszczyzna podłoża jest płaska. Oznacza to, że wektor normalny jednostki zawsze będzie składał się z x, a jego składnik y będzie wynosił 1. Wektor vr jest wektorem prędkości względnej między cząstką a ziemią. Ponieważ ziemia nie porusza się, prędkość względna jest po prostu prędkością cząstki. vrn to skalar, który jest używany do przechowywania komponentu prędkości względnej w kierunku wektora normalnego jednostki kolizyjnej. Obliczamy vrn, pobierając iloczyn o wartości względnej z normalnym wektorem jednostkowym. J jest skalarem, który przechowuje impuls wynikający z kolizji. Fi jest wektorem, który przechowuje siłę uderzenia uzyskaną na podstawie impulsu J. Na koniec, parametr toCollision jest flagą ustawianą na podstawie tego, czy wykryto kolizję czy nie. Teraz przyjrzymy się szczegółom w CheckForCollisions. Pierwszym zadaniem jest zainicjowanie wektora siły uderzenia, vImpactForces, na 0. Następnie wykonujemy pierwszą kontrolę kolizji, określając, czy pozycja y cząstki jest mniejsza niż wysokość płaszczyzny podłoża plus promień cząstki. Jeśli tak, to

wiemy, że mogło dojść do kolizji. (_GROUND_PLANE reprezentuje współrzędną y płaszczyzny podłoża, którą ustawiliśmy na 100.) Jeśli zdarzyło się zderzenie, wykonujemy następną kontrolę - aby ustalić, czy cząstka porusza się w kierunku płaszczyzny podłoża. Aby wykonać tę drugą kontrolę, obliczamy wektor normalny wektora, względną prędkość i względną składową prędkości w normalnym bezpośrednim zderzeniu, jak opisano wcześniej. Jeśli względna prędkość w normalnym kierunku jest ujemna (tj., Jeśli vrn <0), to wystąpiła kolizja. Jeśli jedna z tych kontroli jest nieprawidłowa, kolizja nie wystąpiła i funkcja kończy działanie, zwracając wartość false. Ciekawe rzeczy się zdarzają, gdy przejdzie drugie badanie. W tym miejscu musimy określić siłę uderzenia, która spowoduje odbijanie się cząstki od płaszczyzny podłoża. Oto specjalny kod obliczający siłę uderzenia:

```
J = -(vr*n) * (_RESTITUTION + 1) * p->fMass;

Fi = n;

Fi *= J/_TIMESTEP;

p->vImpactForces += Fi;

p->vPosition.y = _GROUND_PLANE + p->fRadius;

p->vPosition.x = (_GROUND_PLANE + p->fRadius −

p->vPreviousPosition.y) /

(p->vPosition.y - p->vPreviousPosition.y) *

(p->vPosition.x - p->vPreviousPosition.x) +

p->vPreviousPosition.x;

hasCollision = true;
```

Obliczamy impuls za pomocą wzorów przedstawionych w części 5. J jest skalarem równym ujemnej względnej prędkości w normalnym kierunku pomnożonej przez współczynnik restytucja plus 1-krotność masy cząstek. Przypomnijmy, że współczynnik restytucji, _RESTYTUCJA, reguluje, jak elastyczna lub nieelastyczna jest kolizja, lub innymi słowy, ile energii jest przenoszone z powrotem do cząstki podczas uderzenia. Mamy tę wartość ustawioną na 0.6, ale można ją zmienić w zależności od tego, jaki efekt chcesz osiągnąć. Wartość 1 sprawia, że cząstki są bardzo sprężyste, podczas gdy wartość, powiedzmy, 0,1 powoduje, że przywierają do ziemi. Teraz, aby obliczyć siłę uderzenia, Fi, która będzie działała na cząstkę podczas następnego kroku, powodując, że odbija się ona od ziemi, ustawiamy Fi jako równy normalnemu wektorowi kolizji. Wielkość siły uderzenia jest równa impulsu J, podzielonego przez czas w sekundach. Linia Fi * = J / _TIMESTEP; dba o obliczenie ostatecznej siły uderzenia. Aby cząsteczka nie przeniknęła do ziemi, przesuwamy ją tak, aby po prostu spoczywała na ziemi. Pozycję y łatwo obliczyć jako wysokość płaszczyzny uziemienia plus promień cząstki. Obliczamy pozycję x przez liniową interpolację między poprzednią pozycją cząstki a jej aktualną pozycją przy użyciu nowo wyliczonej ypozycji. To skutecznie cofa cząsteczkę wzdłuż linii działania jej prędkości do punktu, w którym dotyka ona tylko płaszczyzny podłoża. Po uruchomieniu symulacji zobaczysz, że cząstki spadają, dryfując nieco od lewej do prawej, aż trafią w płaszczyznę podłoża. Gdy uderzą, odbiją się na ziemi, w końcu spoczną. Ich specyficzne zachowanie w tym zakresie zależy od tego, jaki współczynnik oporu używasz i jakiego współczynnika restytucji używasz. Jeśli masz wiatr, gdy cząstki opadną pionowo, powinny nadal dryfować w prawo, tak jakby ślizgały się po płaszczyźnie podłoża.

Zderzenia cząstek z przeszkodami

Aby było naprawdę interesująco, dodamy teraz okrągłe przeszkody, które zobaczyliśmy Rysunek 8-1 do Rysunek 8-4. Cząstki będą mogły uderzyć i odbić się lub nawet osiąść w szczelinach wykonanych przez nakładające się przeszkody. Przeszkody są po prostu statycznymi cząstkami. Zdefiniujemy je jako cząstki i zainicjujemy je, ale pomijamy je podczas integracji równań ruchu dynamicznych cząstek. Oto deklaracja dla tablicy Przeszkód:

```
Przeszkody cząstek [_NUM_OBSTACLES];
```

Inicjowanie przeszkód polega na przypisaniu im pozycji i wspólnego promienia i masę. Kilka linii kodu pokazanych obok zostało dodanych do głównej wersji programu Initialize do losowego ustawiania przeszkód w dolnej, środkowej części okna nad płaszczyzną podłoża. Rysunki od 8-1 do 8-4 ilustrują ich charakter roz[powszechniania

```
{
.
.
.
for(i=0; i<_NUM_OBSTACLES; i++)
{
Obstacles[i].vPosition.x = GetRandomNumber(_WINWIDTH/2 −
_OBSTACLE_RADIUS*10,
_WINWIDTH/2 +
_OBSTACLE_RADIUS*10, false);
Obstacles[i].vPosition.y = GetRandomNumber(_GROUND_PLANE +
_OBSTACLE_RADIUS, _WINHEIGHT/2 −
_OBSTACLE_RADIUS*4, false);
Obstacles[i].fRadius = _OBSTACLE_RADIUS;
Obstacles[i].fMass = 100;
}
.
.
.
}
```

Rysowanie przeszkód jest łatwe, ponieważ są to typy cząstek z metodą Draw już rysuje okrągłe kształty. Stworzyliśmy DrawObstacles do iteracji poprzez tablicę Przeszkód, wywołując metodę Draw każdej z przeszkód.

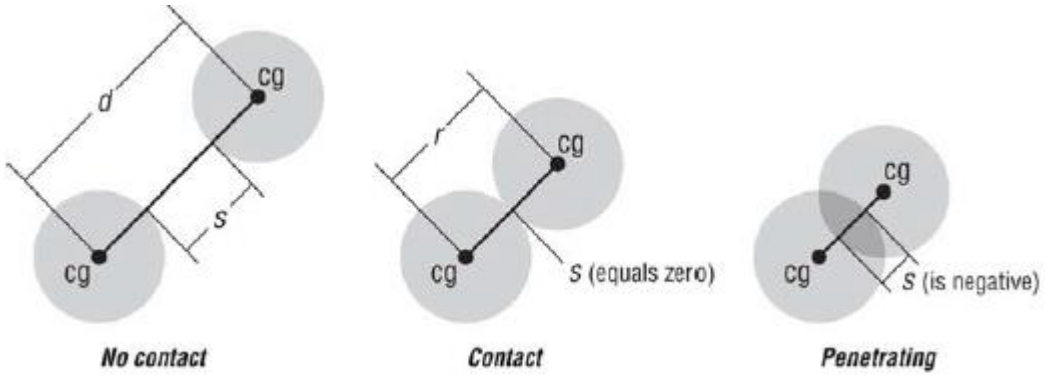
```
void DrawObstacles(void)
```

```
{
int i;
for(i=0; i<_NUM_OBSTACLES; i++)
{
Obstacles[i].Draw();
}
}
```

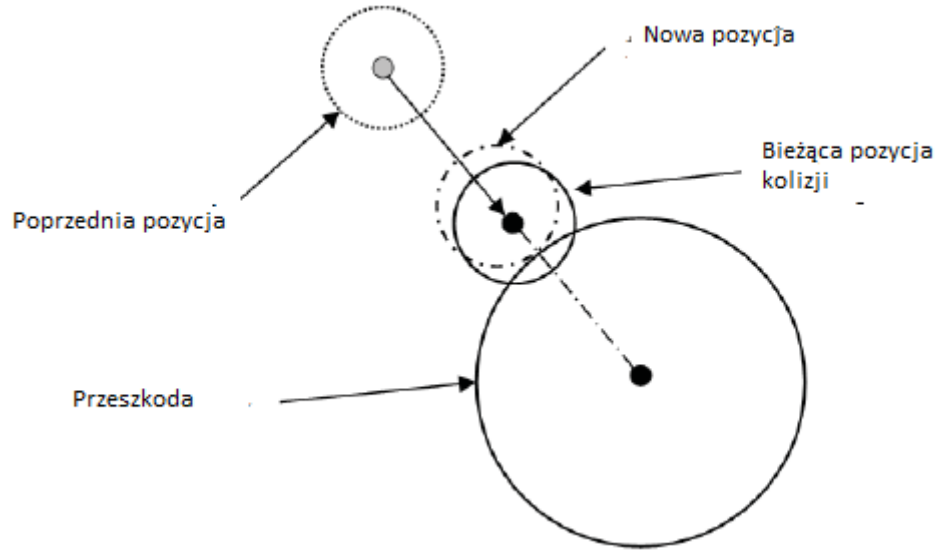
DrawObstacles is then called from UpdateSimulation:

```
void UpdateSimulation(void)
{
.
.
.
// initialize the back buffer
if(FrameCounter >= _RENDER_FRAME_COUNT)
{
ClearBackBuffer();
// Draw ground plane
DrawLine(0, _WINHEIGHT - _GROUND_PLANE,
_WINWIDTH, _WINHEIGHT - _GROUND_PLANE,
3, RGB(0,0,0));
DrawObstacles();
}
.
.
.
}
```

Ostatni fragment kodu, który musimy dodać, ma w pełni funkcjonujące kolizje z przeszkodami polega na dodaniu do modułu CheckForColli większej liczby kodów wykrywania i obsługi kolizji funkcja sions. Zanim przejrzymy CheckForCollisions, zastanówmy się nad kolizją kół ogólnie, aby lepiej zrozumieć, co zrobi nowy kod. Rysunek 8-6 ilustruje kolizję dwóch kółek.



Naszym celem jest wykrycie, czy kręgi te kolidują poprzez sprawdzenie odległości między ich centrami. Jeśli odległość między dwoma ośrodkami jest większa niż suma promieni okręgów, to cząsteczki nie kolidują. Najwyższa ilustracja na rysunku 8-6 pokazuje odległość, d, pomiędzy środkami i odległość, s, między krawędziami okręgów; s to różnica między nimi. Innym sposobem myślenia o tym jest to, że jeśli s jest dodatnie, to nie ma kolizji. Odnosząc się do środkowej ilustracji na rysunku 8-6, jeśli s jest równe 0, koła są w kontakcie. Jeśli s jest liczbą ujemną, jak pokazano na najniższej ilustracji, wówczas kręgi się przenikają. Zastosujemy te zasady do wykrywania kolizji kryminalnych w celu wykrycia kolizji między naszymi cząstkami i przeszkodami, ponieważ oba są okręgami. Rysunek 8-7 pokazuje, jak mogą wyglądać nasze kolizje między cząstkami.



Obliczymy s dla każdej cząstki dla każdej przeszkody, aby określić kontakt lub penetrację. Jeśli znajdziemy, to wykonamy kontrolę względnej prędkości w ten sam sposób, w jaki zrobiliśmy zderzenie cząstek z podłożem, aby sprawdzić, czy cząsteczka porusza się w kierunku przeszkody. Jeśli tak, to mamy kolizję i wspieramy cząstkę wzdłuż normalnej linii działania, która jest po prostu linią łączącą centra cząstki i przeszkody. Obliczymy siłę uderzenia tak, jak zrobiliśmy to wcześniej i pozwolimy integratorowi zająć się resztą. OK, teraz spójrzmy na nowy kod w CheckForCollisions:

```
bool CheckForCollisions(Particle* p)

{

.

.
```

```
.
// Check for collisions with obstacles
float r;
Vector d;
float s;
for(i=0; i<_NUM_OBSTACLES; i++)
{
r = p->fRadius + Obstacles[i].fRadius;
d = p->vPosition - Obstacles[i].vPosition;
s = d.Magnitude() - r;
if(s <= 0.0)
{
d.Normalize();
n = d;
vr = p->vVelocity - Obstacles[i].vVelocity;
vrn = vr*n;
if(vrn < 0.0)
{
J = -(vr*n) * (_RESTITUTION + 1) /
(1/p->fMass + 1/Obstacles[i].fMass);
Fi = n;
Fi *= J/_TIMESTEP;
p->vImpactForces += Fi;
p->vPosition -= n*s;
hasCollision = true;
}
}
}
.
.
.
```

}

Nowy kod jest prawie taki sam jak kod, który sprawdza i radzi sobie z kolizjami między czołami. Jedyne duże różnice dotyczą sposobu obliczania odległości między cząstką a przeszkodą oraz tego, w jaki sposób dostosowujemy położenie zderzającej się cząstki, aby zapobiec jej przeniknięciu przez przeszkodę, ponieważ normalny wektor jednostki może nie być prosto w górę, jak wcześniej. Reszta kodu jest taka sama, więc skupmy się na różnicach. Jak wyjaśniono wcześniej i zilustrowano na rysunku 8-6, musimy obliczyć separację s pomiędzy cząsteczką a przeszkodą. Aby uzyskać s, deklarujemy zmienną r i porównujemy ją do sumy promieni cząstki i przeszkody, przeciwko której sprawdzamy kolizję. Definiujemy d, a Vector, jako różnicę między pozycjami cząstki i przeszkody. Wielkość d minus r daje s. Jeśli s jest mniejsze od 0, to dokonujemy względnej kontroli prędkości. Teraz, w tym przypadku, wektor normalny kolizji znajduje się wzdłuż linii łączącej środki dwóch kółek reprezentujących cząsteczkę i przeszkodę. Cóż, to tylko wektor, który już obliczyliśmy. Aby uzyskać wektor normalny jednostki, po prostu normalizujemy d. Wektor prędkości względnej jest po prostu różnicą prędkości cząstek i przeszkody. Ponieważ przeszkody są statyczne, prędkość względna jest po prostu prędkością cząstki. Ale obliczyliśmy prędkość względną, przyjmując różnicę wektora vr = p-> vVelocity -Obstacles [i] .vVelocity, ponieważ w bardziej zaawansowanym scenariuszu możesz mieć ruchome przeszkody. Pobranie iloczynu punktowego wektora prędkości względnej, vr, za pomocą wektora normalnego jednostki, daje względną prędkość w normalnym kierunku kolizji. Jeśli ta względna prędkość jest mniejsza od 0, to cząstka i obiekt kolidują, a kod przechodzi do obliczania siły uderzenia w sposób podobny do opisanego wcześniej dla kolizji cząstek z ziemią. Jedyna różnica polega na tym, że masy cząsteczek i obiektów pojawiają się w formule impulsu. Wcześniej założyliśmy, że podłoże jest nieskończenie masywne w stosunku do masy cząstki, więc 1 m długości dla gruntu spadło do 0, w zasadzie zejście z równania. Wróć do części 5, aby przywołać formuły impulsów. Po obliczeniu siły uderzenia, kod tworzy kopię cząstki o odległość równą s, penetrację, wzdłuż linii oddziaływania normalnego wektora kolizji, dając nam to, czego pragniemy (jak pokazano na rysunku 8-7). Teraz, po uruchomieniu tej symulacji, zobaczysz spadające cząsteczki, odbijające się od przeszkód lub płynące wokół nich, w zależności od wartości, której używasz dla współczynnika restytucji, ostatecznie odbijającego się i spoczywającego na płaszczyźnie podłoża. Jeśli masz prędkość wiatru większą niż 0, cząsteczki nadal będą dryfować wzdłuż płaszczyzny podłoża od lewej do prawej.

Strojenie

Tuning jest ważną częścią tworzenia dowolnego symulatora. Strojenie oznacza różne rzeczy dla różnych ludzi. Dla niektórych dostrojenie polega na dostosowywaniu formuł i współczynników, aby dopasować symulację do określonej "właściwej odpowiedzi", podczas gdy inne dostrajają parametry, aby wygląd symulacji wyglądał i zachowywał się tak, jak chcesz, bez względu na to, czy jest to technicznie właściwa odpowiedź, czy nie. W końcu właściwą odpowiedzią na grę jest to, że jest zabawna i solidna. Mówiąc o solidności, inni ludzie patrzą na tuning w sensie dostosowywania parametrów, aby symulacja była stabilna. W rzeczywistości to wszystko jest tuningiem i powinieneś myśleć o tym, jako niezbędny element rozwoju symulacji. Jest to proces, w trakcie którego zmieniasz, przesuwasz i dostosowujesz rzeczy, aby symulacja wykonywała to, co chcesz. Na przykład można użyć tej samej przykładowej symulacji do modelowania bardzo sprężystych gumowych kulek. Aby to osiągnąć, prawdopodobnie dostosujesz współczynnik restytucji do wartości zbliżając się do 1 i być może obniżając współczynnik oporu. Cząstki będą odbijały się w każdym miejscu z dużą ilością energii. Jeśli, z drugiej strony, chcesz modelować coś wzdłuż linii błota, obniżysz współczynnik restytucji i zwiększysz współczynnik oporu. Nie ma prawidłowej lub złej kombinacji współczynnika restytucji lub współczynnika oporu do użycia, o ile jesteś zadowolony z wyników.

Kolejnym aspektem, który możesz dostroić, jest liczba ramek symulacji na ramkę renderującą. Obliczenia symulacyjne mogą być tak długie, że powstałe animacje są zbyt gwałtowne, ponieważ nie aktualizujesz wyświetlacza wystarczająco. Odwrotność może być prawdą w innych przypadkach. Ważnym parametrem, który się w to gra, jest wielkość kroku czasowego, jaką wykonujesz w każdej iteracji symulacji. Jeśli rozmiar kroku jest zbyt mały, wykonasz wiele kroków symulacyjnych, spowalniając animację. Z drugiej strony, mały krok czasowy może utrzymać symulację liczbowo stabilną. Twój wybrany integrator odgrywa tutaj ogromną rolę. Jeśli sprawisz, że twój czas będzie zbyt duży, symulacja może po prostu wysadzić się w powietrze i nie zadziałać. Będzie numerycznie niestabilny. Nawet jeśli nie wybuchnie, możesz zobaczyć dziwne wyniki. Na przykład, jeśli krok czasowy w przykładowej symulacji omawianej w tym rozdziale jest zbyt duży, cząstki mogą całkowicie przekroczyć przeszkody w jednym kroku czasowym, nie mając kolizji, która w przeciwnym razie by się zdarzyła. (Pokażemy ci w Części 10, jak poradzić sobie z tą sytuacją.) Ogólnie, strojenie jest niezbędną częścią opracowywania symulacji opartych na fizyce, a my zachęcamy do eksperymentowania - próbując różnych kombinacji parametrów, aby zobaczyć, jakie wyniki można osiągnąć. Powinieneś wypróbować kombinacje, które mogą złamać przykład w tej części, aby zobaczyć, co się dzieje i czego powinieneś unikać w wdrożonej grze.