XXI. Kopiąc Żółwia : Aproksymacja zwykłych i wolnych funkcji

Przez długi czas jednym z niewielu sposobów efektywnego wykorzystania funkcji takich jak sin / cos było zbudowanie stołu. Z procesorem pozbawionym FPU, a przy kilku cyklach do stracenia, funkcja taka jak sinus była całkowicie tabu w gra w czasie rzeczywistym. Stoły działały całkiem dobrze i były używane przez jakiś czas, dopóki FPU nie stało się wystarczająco szybkie, aby odczytanie tablicy stało się znacznie mniej atrakcyjne niż posiadanie dokładnej wartości. W końcu tablice miały ograniczoną liczbę próbek. Niestety, te funkcje są częścią najwolniejszej grupy instrukcji, jaką dzisiaj ma do zaoferowania PC. W rzeczywistości są one tak powolne, że aproksymacja oparta na oprogramowaniu zapewniająca taką samą dokładność może faktycznie przewyższać każdą złożoną funkcję. Oznacza to, że możesz przyspieszyć wiele interesujących funkcji. W części 12 dowiedziałeś się o wielu metodach przybliżania i sposobie ich wykorzystania do przybliżania różnych funkcji. Dobrze jest mieć dobre teoretyczne podstawy, ale nie jest to zbyt przydatne, jeśli nie ma żadnej aplikacji do tworzenia gier. Rozdział ten idzie o krok dalej w przybliżeniu, całkowicie ponownie wdrażając kosztowne funkcje FPU, aby były one szybsze. Najwyraźniej nie chciałbyś tego robić, gdyby nie było praktycznych zastosowań dla wszystkich tych funkcji, więc ten rozdział pokaże niesamowite rzeczy, które możesz zrobić z aproksymacją, kilkoma zgrabnymi sztuczkami i zdrowym, innowacyjnym umysłem. Dokładniej, ten rozdział obejmie następujące zagadnienia:

■ Przybliżanie funkcji przestępnych

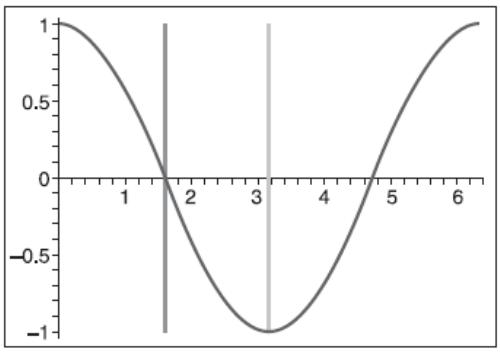
■ Symulacja fizyczna

Przybliżanie funkcji przestępnych

Nadzmysłowy. Jest słowo, które zakręci ci na głowie zakrzywiony znak zapytania. Najprostsza definicja funkcji transcendentalnych nie może być wyrażona w terminach algebry. Znanymi przykładami takich funkcji są zestaw funkcji trygonometrycznych sinus, cosinus ,tangens, ex i wiele innych. Może to być niespodzianką, ponieważ cosinus można obliczyć na komputerze. Prawda jest taka, że takie funkcje są aproksymowane i nie są obliczane dokładnie. Jeśli nie możesz wyrazić takich funkcji za pomocą algebry, najlepiej jest przybliżyć funkcje za pomocą algebry. Ponieważ takie funkcje nie są naturalne dla naszej matematyki, zwykle dają złożoną nieskończoną aproksymację algebraiczną iw konsekwencji dają bardzo powolne metody w porównaniu z innymi funkcjami algebraicznymi. Każdy, kto pracował z funkcjami trygonometrycznymi, może poświadczyć, jak powolne są te funkcje, a liczenie cykli Intela dla takich funkcji pokazuje, że są najwolniejsze z pakietu. To wstyd, ponieważ funkcje te są niezwykle użyteczne dla wszystkiego, co naturalne pod sferyczną koordynacją świata (np. Rotacje). W następnej sekcji przedstawimy kilka serii, aby dokładnie i skutecznie obliczyć takie funkcje.

Przybliżenie cosinusa i sinusoidy

Jeśli chcesz zbliżyć się do krzywej cyklicznej, takiej jak funkcje sinusowe lub cosinus, pierwszą rzeczą, którą powinieneś zrobić, jest poszukiwanie łatwych symetrii i rozważ tylko najprostszą sekcję. Patrząc na pełny okres fali cosinusowej, zilustrowany na rysunku 21.1,



można usunąć drugą połowę za pomocą symetrii.:

$$\cos(Angle) = \cos(2\pi - Angle)$$

Ponadto można usunąć drugi kwartał przez odbicie i odwrócenie:

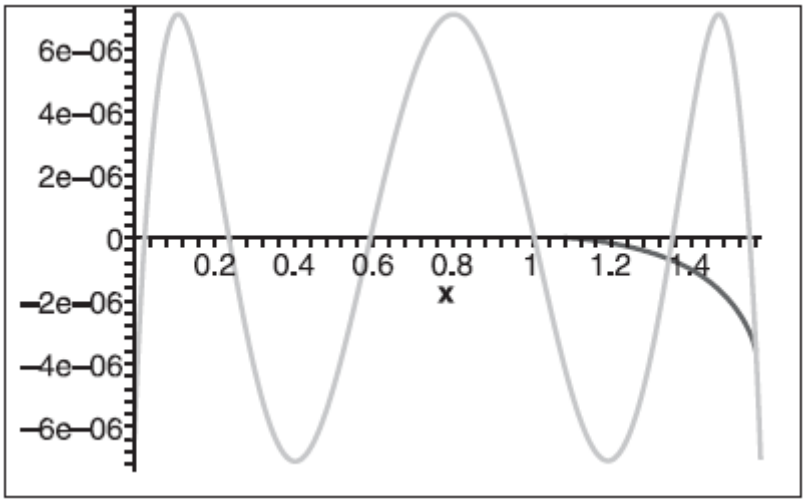
$$\cos(Angle) = -\cos(\pi - Angle)$$

Na koniec pozostaje ci przybliżona funkcja cosinusa od 0 do π/2, dla której ty może generować znacznie lepsze przybliżenie niż cosinus o pełnej długości. W rzeczywistości każde przybliżenie uwzględniające cały okres cosinusu będzie podatne na kary o wysokiej precyzji. Ten sam pomysł można zastosować do funkcji sinus. Szczególnie interesujący jest fakt, że szereg Taylora staje się lepsza od szeregu Minimax, pod względem szybkości obliczeniowej i precyzji, po około 16 bitach precyzji. Oczywiście, aby szereg Taylor była szybsza od szereg Minimax, będziesz musiał wyodrębnić x2. Kod dla obliczeń Minimax cosinusu jest następujący, podobnie jak tabele dla cosinusa i sinusa. Kod do ustawienia serii Taylora będzie nieco inny, ponieważ musi wstępnie obliczyć x^2, ale całkiem podobnie. Podobną konfigurację można zastosować do sinusa. Zwróć uwagę na sposób obsługi sprawy 90 stopni. Znak jest tylko odwrócony, a kod nadal działa poprawnie. Zwróć uwagę, że ta funkcja obsługuje tylko kąty od 0 do 2 π. Nie powinno być zbyt trudno upewnić się, że wszystkie argumenty przekazywane do tej funkcji pasują do tego zakresu, stale sprawdzając przepełnienie zakresu.

```
float cos (float X)

{

float Wynik, Kwadrat;

jeśli (X> PI)

X = 2.0f * PI - X;

// Obrót o 90 °

jeśli (X <= PI / 2.0f) {

Wynik = X0CoEff;

Kwadrat = X;

} else {

Kwadrat = X - PI; // Odwróć znak
```

```
Wynik = -X0CoEff;

X = -Square;

}

Wynik + = Kwadrat * XiCoEff;

Kwadrat * = X; // Dwie ostatnie linie powtarzają się dla każdego coeffa.

return Result;
}
```

Kolejnym pytaniem jest, o ile szybciej jest ten kod niż tradycyjne funkcje C / C ++? Wszystkie porównania w tym rozdziale są wykonywane przy użyciu kompilatora Microsoft Visual C ++ .NET, który wykorzystuje funkcje FPU. Szybkość kodu jest całkowicie zależna od kompilatora i testowanej maszyny. Jednak na podstawie przeprowadzonych testów każde przedstawione przybliżenie jest szybsze niż wersja biblioteki (fcos). W przypadku cosinusa, ostatnie zajmuje około 70% pierwotnego czasu. Kiedy osiągniesz 13 bitów precyzji, podwoisz prędkość. Zarówno dla kosinusów jak i sinusów pierwszy zestaw współczynników daje trzykrotną poprawę. W przypadku funkcji sinusowej najwolniejszy przypadek jest dwa razy szybszy niż pierwotny. W obu przypadkach osiągasz całkiem spory postęp w tej dziedzinie. Zwróć uwagę, że krzywa sinusoidalna daje lepsze przybliżenia niż wartości cosinusoidalne. Z tego powodu należy wziąć pod uwagę, że krzywa cosinusowa jest tylko krzywą sinusoidalną przesuniętą o 90 stopni i po prostu odjąć π/ 2 od wartości wejściowych. Istnieją przybliżenia Minimax, które są bardziej złożone niż przybliżenie 9 bitów. Ale oczywiście seria Taylora jest mniej lub bardziej złożona z większą precyzją, co ilustruje fala sinusoidalna na rysunku 21.2.

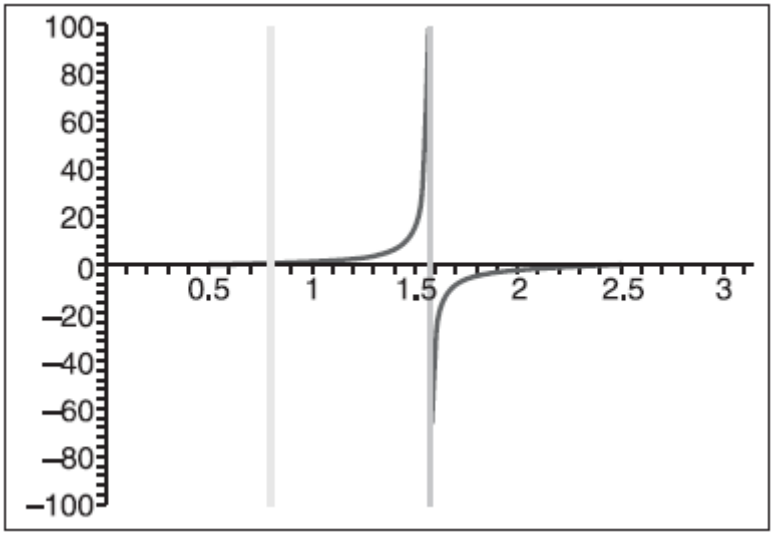


Teraz, gdy zobaczysz, jak wiele możesz osiągnąć dzięki tym funkcjom, zobaczmy, co możesz zrobić z innymi dość popularnymi transcendentalnymi funkcjami.

Aproksymacja styczna

Funkcje sinus i cosinus są całkiem ładne, ponieważ można je przybliżać za pomocą szeregu potęgowego. Funkcje te nie miały żadnego wyjątkowego ani ekstremalnego przypadku, ale to już koniec. Funkcja styczna ma nieskończoną liczbę osobliwości, ale podobnie jak funkcje sinusowe i cosinusowe, rozważamy tylko jeden okres. Ta funkcja ma osobliwości w π / 2. Rozważmy zakres [0 ...π] Dla funkcji tan. Możesz łatwo zmienić to na [-π? / 2,π / 2] lub inny podobny zakres, jeśli chcesz. Wiesz,

że strona pozytywna i negatywna są lustrzanymi odbiciami, więc możesz zwrócić uwagę na zakres [0 ...π / 2]. Dużym problemem w przybliżeniu tej funkcji jest osobliwość, która istnieje w? / 2. Ta funkcja jest asymptotyczna i to jest prawdziwy problem, który musisz naprawić. Ogólnie rzecz biorąc, dowolna asymptotyczna krzywa będzie się powoli zbiegała, co daje bardzo złożone funkcje dla rozsądnego przybliżenia. Klucz do przyspieszenia funkcji stycznej pochodzi z bardzo prostej tożsamości, zilustrowanej na rysunku 21.3.



Oznacza to, że potrzebujesz tylko przybliżenia dla zakresu [0 ..π/ 4], ponieważ możesz przekształcić dowolny inny kąt w tym zakresie z tożsamością. Podział jest naprawdę brzydki, ale w tym przypadku nas uratuje. Nie ma już osobliwości i żadnych asymptot, więc możesz teraz przeprowadzić analizę numeryczną, aby obliczyć najlepszy współczynnik dla funkcji stycznej. Zauważ, że ten kod może przyjmować tylko kąty od 0 ..π. W związku z tym zakłada się, że można dopasować kąt w zakresie wewnątrz kodu, co jest szybsze niż wykorzystanie pozostałej części modułu? (w liczbach zmiennoprzecinkowych). Ponownie, jeśli potrzebujesz innego zakresu, na przykład [-π / 2 ..π / 2], powinieneś być w stanie to osiągnąć dzięki podstawowym obserwacjom geometrii funkcji tan.

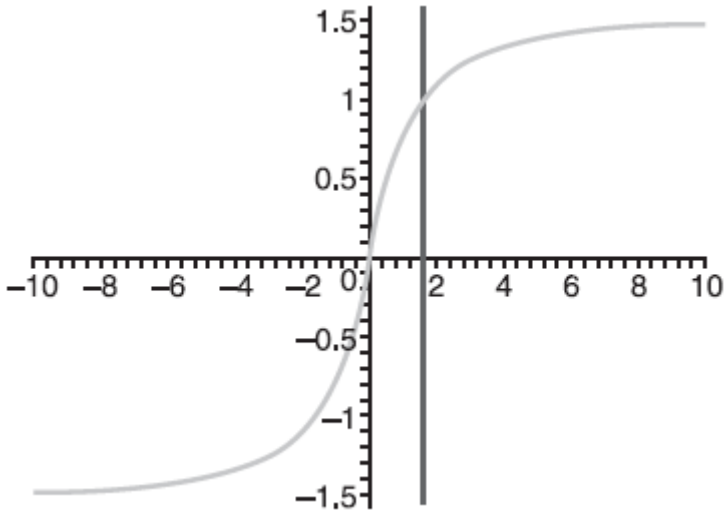
```
float tan (float X)

{

float Wynik, Kwadrat;

// Obrót o 90 °

jeśli (X <= PI / 2) {

Wynik = X0CoEff;

// Zwrot 45 °

if (X> PI / 4) {

Kwadrat = X = PI / 2 - X;

Wynik + = Kwadrat * XiCoEff;

Kwadrat * = X; // Dwie ostatnie linie powtarzają się dla każdego coeffa.
```

```
...
return 1 / Result;
} else {
Kwadrat = X;
Wynik + = Kwadrat * XiCoEff;
Kwadrat * = X; // Dwie ostatnie linie są powtarzane dla każdej współpracy.
...
return Wynik;
}
} else {
Wynik = -X0CoEff;
// Zwrot 45 °
jeśli (X <PI / 2 + PI / 4) {
Kwadrat = PI - X - PI / 2;
X = -Square;
Kwadrat = X;
Wynik + = Kwadrat * XiCoEff;
Kwadrat * = X; // Dwie ostatnie linie są powtarzane dla każdej współpracy.
...
return 1 / Result;
} else {
Kwadrat = X - PI;
X = -Square;
Kwadrat = X;
Wynik + = Kwadrat * XiCoEff;
Kwadrat * = X; // Dwie ostatnie linie są powtarzane dla każdej współpracy.
...
return Wynik;
}
}
```

Zanim przejdę do benchmarków, powinienem powiedzieć kilka słów o kodzie. Wydaje się, że został skopiowany i wklejony cztery razy i słusznie. Jeśli weźmiesz wspólne obliczenia serii poza kodem, musisz zająć się odwrotnością dla połowy przypadków. Możesz dodać kolejne elementy if lub goto z dwoma głównymi rozszerzeniami wielomianowymi, ale możesz wziąć duży hit wydajnościowy z oddziału. Rozszerzanie funkcji, tak jak było tutaj wykonane, powoduje wzrost prędkości o 20%, co jest dość znaczące. Seria Taylora dla tej konkretnej funkcji wypadła bardzo źle.. Do analizy prędkości nie jest jasne, czy uzyskałeś jakąkolwiek prędkość, czy nie. Aby osiągnąć ten cel, potrzebujesz znacznie więcej współczynników i jeden brzydki podział na połowę przypadków. Ale w rzeczywistości, patrząc na średnie cykle zegara Intela dla funkcji tan wyraźnie widać, że ta funkcja cierpi bardziej niż poprzednie dwie. Porównując funkcję tan z aproksymacją seregu, wyniki były mniej więcej tak dobre, jak osiągnięte dzięki sin i cos. Dla 14 bitów dokładności przybliżenie jest dwa razy szybsze niż funkcja C / C ++. W najlepszym przypadku uzyskujesz potężny czterokrotny wzrost prędkości, a dokładniejszy przypadek daje około 70% implementacji MS C / C ++. Ponownie wykonano całkiem dobrą robotę polegającą na optymalizacji tej funkcji. To jest miłe, ale podnieśmy tu żółtą flagę, ponieważ dzieje się dużo. Dokładniej, funkcja odwrotna zaszkodzi precyzji wykraczającej poza teoretyczną kwotę. W związku z tym możliwe jest przybliżenie 14 bitów i kończy się 9 punktami precyzji w praktyce. Zostałeś ostrzeżony. To nie jest prawdziwy problem z cos / sin.

Przybliżenie ArcTangent

Oto kolejny potwór, który jest całkiem przydatny w rozwiązywaniu problemów geometrii. Zazwyczaj ArcTan nie jest czymś, z czego chcesz korzystać, ponieważ jest wolniejszy niż wszystkie inne funkcje, które widziałeś do tej pory. Wiele razy można obejść obliczanie tej funkcji, patrząc na swój problem w inny sposób. Możesz użyć stoków i czterech ćwiartek, aby usunąć użycie tej funkcji. Tak więc morał jest prosty: jeśli potrafisz się bez niego obejść, to bez niego, ponieważ aproksymacja nie będzie tak szybka, jak na przykład korzystanie z nachyleń. Czasem jednak nie da się tego uniknąć, więc przeanalizujmy to. Funkcja ArcTan, przedstawiona na rysunku 21.4,



jest kolejną krzywą asymptotyczną. To przybliża takie krzywe do nietrywialnego zadania i zasadniczo trzeba spojrzeć na krzywą w inny sposób, aby uzyskać dobre przybliżenie. Bezpośrednio od pałki zauważ wyraźną symetrię w X. Dlatego powinieneś brać pod uwagę jedynie dodatnią część krzywej. Jeszcze raz możesz odwołać się do bardzo miłej tożsamości:

$$\arctan(x) = \frac{\pi |x|}{2x} - \arctan\left(\frac{1}{x}\right)$$

Okay, więc to nie jest takie miłe, ale wystarczy dla naszych celów. Co więcej, jeśli bierzesz tylko pod uwagę pozytywne wartości, otrzymujesz ładniejszą formę:

$$\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right)$$

Jest to interesujące, ponieważ teraz możesz ograniczyć swoje przybliżenie do zakresu [0..1] i skutecznie pozbyć się asymptotycznej strony równania:

```
float atan (float X)

{

float Wynik, Kwadrat;

// Obrót o 90 °

jeśli (X <= 1) {

Wynik = X0CoEff;

Kwadrat = X;

} else {

Wynik = PI / 2 - X0CoEff;

X = 1 / X;

Kwadrat = -X;

}

Wynik + = Kwadrat * XiCoEff;

Kwadrat * = X; // Dwie ostatnie linie to

powtarzane dla każdego coeff.

...

return Wynik;

}
```

Ponownie, szereg Taylora nie występuje na wykresie. Ponownie, patrząc na głębokość współczynnika, funkcja ta nie jest łatwym przybliżeniem. Szybko, możesz spodziewać się dobrej bitwy, ale Intel twierdzi, że jego funkcja ArcTan waha się od 150-300 cykli. Ma dość szeroki zakres, ale w niektórych przypadkach wydaje się, że potrafi pokonać cosinus i sinus. Ponieważ jednak pokonałeś już prędkość cosinusa i funkcji sinusoidalnych FPU z przybliżeniem, nie powinno dziwić, że ta funkcja działa również bardzo dobrze. W porównaniu z 15 bitami dokładności, funkcja wynosi około 2,5 razy szybciej niż

wbudowana funkcja. Przy najmniejszej precyzji jest nieco mniej niż czterokrotnie szybsza, a dzięki 22 bitom dokładności wciąż jest o 60% szybsza. Co najlepsze, nie tracisz precyzji, jak przy funkcji tan, ponieważ podział odbywa się przed zbliżeniem, a nie po.

Aproksymacja ArcSin / ArcCos

Ta funkcja terroryzuje programistów nawet bardziej niż te wcześniej widziane. ArcSin i ArcCos są jedynie wzajemnymi odbiciami. Są one bardzo powolne na komputerach, głównie dlatego, że nie ma funkcji sprzętowej ArcSin. Zazwyczaj rozwiązaniem jest obliczenie ArcTan, a następnie wykorzystanie tożsamości trygonometrii w celu konwersji z ArcTan na ArcSin i ArcCos. Aby jeszcze bardziej przybliżyć dyskusję, jedną z nich jest konwersja z ArcSin za pomocą ArcTan

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

Możesz być w stanie pozbyć się podziału za pomocą kilku operacji, ale na pewno nie będziesz w stanie dotknąć tego pierwiastka kwadratowego, kwadratowego mnożenia i ArcTan. Jeśli Intel zdecyduje się nie dodawać funkcji ArcSin, może istnieć ku temu dobry powód, poza przestrzenią i pieniędzmi. Funkcja ArcSin przedstawiona na rysunku 21.5 jest jedną z najgorszych funkcji do przybliżenia. Jest to bardzo nienaturalne, a może bardziej poprawne, niealgebraiczne. Na początku jest to jednak trochę zaskakujące, ponieważ obejmuje tylko od 1 do 1, ale jego kształt po prostu nie pasuje do wzoru serii mocy. Pierwszą rzeczą, którą powinieneś zauważyć, jest symetria, dlatego powinieneś rozważyć tylko górną część krzywej. Jeśli próbowałbyś zbliżyć krzywą od 0 do 1, dokładność byłaby bardzo zła, do punktu, w którym 10 współczynników dawałoby około 8 bitów precyzji. Najwyraźniej nie przydaje się to nam. Ponownie, wyciągnij miłe tożsamości znane ArcSin. W szczególności interesujący jest

$$\arcsin(x) = \frac{\pi}{2} - \arcsin\left(\sqrt{1-x^2}\right)$$

To wciąż jest dość brzydka tożsamość, ale to prawie tak dobre, jak to tylko możliwe. Przynajmniej pozbyłeś się podziału na poprzednią tożsamość. Daje nam to rozpiętość, która jest łatwiejsza do przybliżenia niż cała krzywa. Jedną z przyczyn, dla których ta tożsamość nie powinna Cię niepokoić, jest to, że musisz jej użyć tylko wtedy, gdy x jest większe niż 21/2, co stanowi mniej niż jedną trzecią całej krzywej. Jeśli chodzi o ArcCos, można to łatwo wyrazić jako

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x)$$

Prostszym sposobem na to jest po prostu dostosowanie pierwszego współczynnika tabeli i odwrócić wszystkie znaki. W ten sposób nie ma dodatkowej złożoności do równania. Kod całego zakresu [-1..1] i jest następujący:

```
float ArcSin (float X)

{

float Wynik, Kwadrat;

if (X> = 0) {
```

```
jeśli (X <0,70710678118f) {

Wynik = X0CoEff;

Kwadrat = X;

} else {

Wynik = PI / 2 - X0CoEff;

X = sqrtf (1 - X * X);

Kwadrat = -X;

}

} else {

if (X> = -0.70710678118f) {

Wynik = -X0CoEff;

Kwadrat = X;

X = -X;

} else {

Wynik = -PI / 2 + X0CoEff;

Kwadrat = X = sqrtf (1 - X * X);

}

}

Wynik + = Kwadrat * XiCoEff;

Kwadrat * = X; // Dwie ostatnie linie to

powtarzane dla każdego coeff.

...

return Wynik;

}
```

Nie powinno dziwić, że możesz zrobić tak samo dobrze jak ArcTan z tą funkcją. Dla 15 bitów precyzji jest około 2,5 razy szybszy niż wbudowana funkcja. W najlepszym przypadku jest czterokrotnie szybszy niż funkcja dostarczona przez system, a także Najgorszy przypadek jest około 54% szybszy niż pierwotna prędkość. Zauważ, że przy tych liczbach użyta funkcja pierwiastka kwadratowego jest taka, jaką zapewnia biblioteka. Możesz przyspieszyć kod jeszcze bardziej, jeśli przybliżysz pierwiastek kwadratowy.

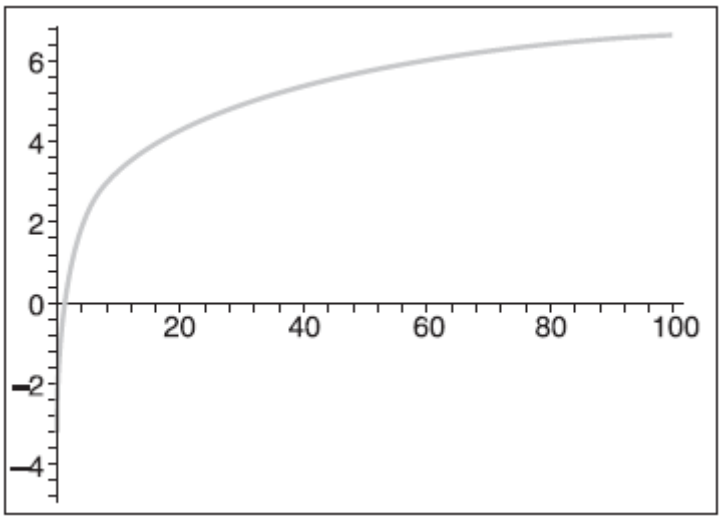
Przybliżenie Logarytmu

Prawdopodobnie nie jest to bardzo przydatna funkcja w grach w czasie rzeczywistym. Ale wiem aż za dobrze, że kiedy to powiem, ktoś inny zrobi krok naprzód i da kontrprzykład. Zauważ, że fizyka i niektóre skomplikowane symulacje powietrza, wody lub innych podobnych zjawisk mogą wymagać

takich funkcji, więc jest to dobry pomysł aby zobaczyć, co możesz z nimi zrobić. Na początek wiesz, że możesz konwertować z dowolnej podstawy logarytmu na dowolną inną z tej prostej tożsamości:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

Ostatni człon jest stały, więc możesz go łatwo obliczyć, jeśli chcesz wykonać funkcję z ustaloną podstawą. Komputery PC są znacznie lepsze w wykonywaniu obliczeń w bazie 2, więc najbardziej logiczną rzeczą do zrobienia jest utworzenie funkcji do obliczania logarytmu bazowego 2, zilustrowanego na rysunku 21.6.



Jeśli pamiętasz sposób, w jaki są zapisywane punkty zmiennoprzecinkowe, musisz znać liczbę całkowitą logarytmu 2 dla x. Przypomnijmy, że zmiennoprzecinkowe są zapisane jako $(1 + 2^{-mantisa}) * 2^{exponent-127}$. Część wykładniczą można uzyskać za pomocą maski bitowej i przesunięcia. To, co da ci trudniejszy czas, to mantysa logarytmu. Z innej tożsamości dziennika, wiesz o tym

$$\log(xy) = \log(x) + \log(y)$$

Ponieważ można obliczyć logarytm z wykładniczego wyrazu, wystarczy znaleźć logarytm pierwszego terminu, a następnie podsumować je. Ponownie, szybkie spojrzenie na wykres logarytmiczny pokazuje, że masz kolejną asymptotę, której musisz się pozbyć. Kolejną rzeczą, której musisz się pozbyć, aby uzyskać dobre przybliżenie, jest nieskończoność po prawej stronie. Ale jeśli przyjrzeć się wyrażeniu dla wartości zmiennoprzecinkowej, nie stanowi to problemu, ponieważ gwarantujemy, że liczba w zakresie [1..2]. Ostatnią rzeczą, którą musisz zrobić, to przybliżenie logarytmu w tym zakresie. Kod:

```
float log2(float X)

{

float Result, Square;

Result = (float)((*(unsigned long *)&X) >> 23) - 127 + x0CoEff;

*(unsigned long *)&X = (*(unsigned long *)&X & 0x007FFFFF) | 0x3F800000;
```

```
Square = X;

Result += Square * XiCoEff;

Square *= X; // The 2 last lines are repeated for every coeff.

...

return Result;

}
```

Najpierw kilka uwag na temat kodu źródłowego. Pierwszy wiersz funkcji pobiera część całkowitą logarytmu z liczby zmiennoprzecinkowej (pamiętaj, że musisz usunąć 127 stronniczości). Następująca linia usuwa wykładnik i maski w 127 (wykładnik równy 0), dzięki czemu możesz kontynuować z przybliżeniem, biorąc pod uwagę liczbę z przedziału [1..2]. Nic zbyt skomplikowanego się nie dzieje. Jeśli chodzi o prędkość, najbardziej precyzyjny przypadek daje około 70% prędkości oryginalnej funkcji biblioteki MS. 16-bitowa skrzynka jest nieco mniej niż dwa razy szybsza, a najszybszy możliwy przypadek jest trzy razy szybszy niż wersja oryginalna. Należy zauważyć, że porównania te zostały wykonane przy użyciu logarytmu naturalnego, a funkcja log2 została pomnożona przez stałą przy użyciu wyżej wymienionej tożsamości.

Aproksymacja potęgi

Funkcja potęgowa nie jest pośrednio funkcją transcendentalną. Na przykład liczbę 28 można obliczyć, mnożąc 2 przez siebie osiem razy. Zanim zajmiemy się typem transcendentalnym, najpierw zoptymalizujmy typy algebraiczne, z którymi łatwiej się uporać. Załóżmy, że chcesz obliczyć moc xy, gdzie x jest liczbą rzeczywistą, ale y jest dodatnią liczbą całkowitą (liczbą naturalną). Najbardziej oczywistą rzeczą do wypróbowania jest pomnożenie x przez y. Ale możesz zrobić o wiele lepiej niż to, ponieważ wielokrotnie oblicza wiele tych samych informacji. Jest to podstawowa tożsamość dla wykładników

$$x^{a+b} = x^a x^b$$

Jeśli spojrzysz na binarną reprezentację swojej naturalnej liczby y, zauważ, że jest to w istocie garść dodatków. Gdy bit jest włączony, możesz dodać 2i, a jeśli nie, nie musisz nic robić. Zasadniczo masz to:

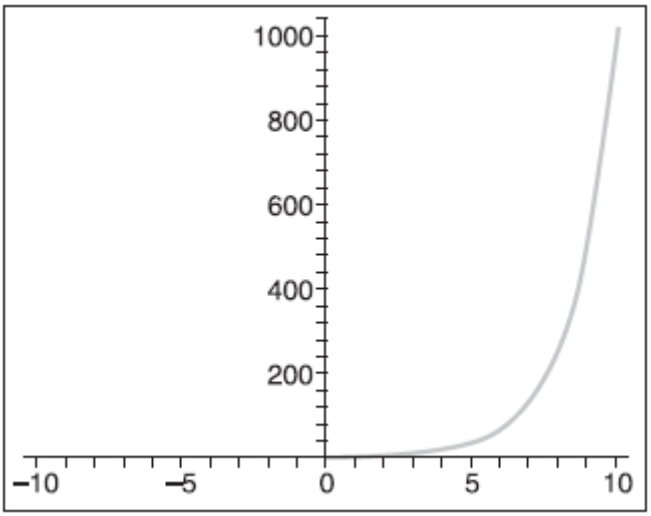
$$x^{1+2+4+8+\cdots+2^i} = xx^2x^4x^8\cdots x^{2^i}$$

Prawa strona jest o wiele bardziej interesująca, ponieważ oznacza to, że musisz obliczyć x2 [^ i]. Reszta to tylko kilka mnożeń tych terminów. Algorytm jest prosty. Dla mocy xy, gdzie y jest naturalną, pętla do momentu, gdy wykładnik y wynosi 0. Ale tymczasem, jeśli prawy bit jest włączony (innymi słowy, jeśli liczba jest nieparzysta), możesz pomnożyć twoją bieżącą moc x przez wynik. Bez względu na wartość, musisz mnożyć x przez siebie na każdy obrót, obliczając tym samym wzrost mocy. Kod źródłowy:

```
float IntPow (float X, unsigned long n)

{

wynik zmiennoprzecinkowy (1,0f);

while (n) {
```

```
jeśli (n & 1)

Wynik * = Kwadrat;

X * = X;

n >> = 1;

}

return Wynik;

}
```

Algorytm ten zależy od liczby bitów ustawionych w liczbie całkowitej, więc jeśli jest wiele bitów , otrzymasz większą wydajność, niż gdy kilka z nich jest ustawionych. Z pierwszym zestawem 16 bitów, algorytm jest około trzy razy szybszy niż funkcja pow dostarczona z kompilatorem MS. Z zestawem 8 bitów, to jest mniej więcej pięć razy szybciej. Tak czy owak, ta funkcja jest znacznie lepsza niż to, co zapewnia kompilator. Ma jednak jeden poważny problem: po prostu nie może obliczyć wykładników ułamkowych ani ujemnych. W wielu równaniach masz stałą moc, więc to nie jest problem. Możesz rozwinąć poprzedni algorytm, aby uzyskać skuteczne rozwiązanie tego problemu. Oczywiście nie jest to w stanie zaspokoić wszystkich twoich potrzeb, więc musisz poszukać czegoś innego. Ta funkcja jest wyjątkowo brzydka, ponieważ niemożliwe jest przybliżenie tej krzywej, biorąc pod uwagę, że masz dwie zmienne. Możesz to zrobić, ale przybliżenie byłoby tak straszne, że nawet nie chciałbyś z niego skorzystać. Zamiast tego można użyć tożsamości do konwersji na Xy. Co więcej, tożsamość do wyboru to Tak, to oznacza inną aproksymację, która nie jest największą funkcją, na jaką można liczyć, ale ta funkcja jest kosztowna obliczeniowo. Już wiesz, jak obliczyć logarytm o wartości 2, więc jedyną rzeczą, której aktualnie brakuje, jest funkcja potęgowa z bazą 2, zilustrowana na rysunku 21.7.



Oczywiście możesz użyć dowolnej bazy, ale 2 jest wybrana po prostu dlatego, że komputer działa bardziej efektywnie w ten sposób. Wykonując analizę krzywej, znowu może się okazać, że jest to brzydka krzywa, ponieważ ma asymptotę i inną nieskończoną wartość dla liczb dodatnich. Musisz ponownie zmniejszyć przybliżony segment, aby mieć dobre przybliżenie. Jeśli do tej pory stosowałeś się do tego schematu, w tym przypadku zwykle przychodzi tożsamość i oszczędza cały dzień. Oczywiście ta sekcja nie jest inna. Możesz przywołać tożsamość:

$$2^{a+b} = 2^a 2^b$$

W rzeczywistości jest to ta sama tożsamość, co w przypadku x = 2. Ta tożsamość pozwala nam na przybliżenie zakresu od [0..1]. W szczególności tożsamość, której będziesz używał, jest

$$2^{[n]+n-[n]} = 2^{[n]} 2^{n-[n]}$$

(n - [n]) jest ułamkową resztą n i [n], całkowitą wartością n. Funkcję 2[n] można obliczyć w poprzednim przykładzie, ale pracujesz z komputerami, więc jest łatwiejszy sposób obejścia tego. Aby obliczyć tę wartość, można pobrać część całkowitą n i po prostu wstawić tę wartość, plus błąd, w bity wykładnicze punktu zmiennoprzecinkowego. W drugim semestrze ponownie będziesz potrzebować aproksymacji numerycznej do szeregu potęgowego. Łatwiej to powiedzieć niż zrobić. Oto kod:

```
float Exp2(float X)
{
float Result, Square, IntPow;
if (X < 0) {
const unsigned long IntVal = *(unsigned long *)&X & 0x7FFFFFFF;
const unsigned long Int = (IntVal >> 23) - 127;
if ((long)Int > 0) {
*(unsigned long *)&IntPow = ((((IntVal & 0x007FFFFF) |

0x00800000) >> (23 - Int)) + 127 + 1) << 23;
*(unsigned long *)&X = (((IntVal << Int) & 0x007FFFFF)
| 0x3F800000);
X = 2.0f - X;
} else {
IntPow = 2.0f;
X++;
}
Result = X0CoEff + Square * X1CoEff;
Square *= X; // The 2 last lines are repeated for every coeff.
Result += Square * XiCoEff;
...
return Result / IntPow;
} else {
const unsigned long IntVal = *(unsigned long *)&X;
const unsigned long Int = (IntVal >> 23) - 127;
if ((long)Int > 0) {
*(unsigned long *)&IntPow = ((((IntVal & 0x007FFFFF) |
0x00800000) >> (23 - Int)) + 127) << 23;
*(unsigned long *)&X = (((IntVal << Int) & 0x007FFFFF)
| 0x3F800000);
X—;
} else
IntPow = 1.0f;
Square = X;
Result = X0CoEff + Square * X1CoEff;
Square *= X; // The 2 last lines are repeated for every coeff.
Result += Square * XiCoEff;
...
```

```
return Result * IntPow;
}
}
```

Teraz, gdy pył ustał, zbadajmy kod nieco bliżej. Są dwa przypadki, z którymi trzeba się uporać: wykładniki ujemne i pozytywne. Pierwszą można obliczyć bez wartości ujemnej, a następnie obliczyć jej odwrotność, która z definicji jest wykładnikiem ujemnym. Jedyny haczyk z wartościami ujemnymi polega na tym, że kiedy zaokrąglasz swoją liczbę całkowitą, musisz upewnić się, że jest ona wypełniona, ponieważ wymaga ona przybliżonego terminu z zakresu od 0 do 1. Na przykład, jeśli masz -4,25, to chcesz zrobić (-5 + 0,75), a nie (- 4 -0,25). Teraz możesz zauważyć, że kod nie ma żadnych obliczeń zmiennoprzecinkowych w złożeniu, takich jak fscale i frndint. Przyczyna tego jest prosta. Te brzydkie funkcje po prostu nie mogą być sparowane na Pentium 3 bez zrywania zasobów (Pentium 4 nie ma drugiego dekodera). Jeśli zaimplementujesz całkowitą wersję tych funkcji, co zostało tutaj zrobione, będziesz czerpać korzyści z szybkości z powodu instrukcji całkowitych. Różnica jest dość zauważalna. Byłoby znacznie łatwiej, gdybyś użył funkcji zmiennoprzecinkowych, ale niestety, liczby całkowite są szybsze. Wykładnik liczby reprezentuje liczbę bitów mantysy opisujących całkowitą część liczby. W gruncie rzeczy to właśnie zawiera zmienna Int. Obliczanie IntPow, które jest określeniem dla największej potęgi 2, jest tylko kwestią izolowania bitów całkowitych opisanych przez mantysę (pamiętaj, aby dodać niejawny 0x00800000) i przesuwać je w pozycji wykładnika (z uprzedzeniem). Tak właśnie działa obliczanie IntPow. Następna linia oblicza resztę dla X. W tym przypadku możesz przesunąć bity, które opisują całkowitą część mantysy, zachowując tylko bity opisujące nieintegrujące bity. Następnie musisz pozbyć się domyślnego 1. A następnie algorytm jest dobry do przejścia, z przybliżeniem i ostatecznym mnożeniem (lub dzieleniem, jeśli masz do czynienia z przypadkiem negatywnym). Może to wyglądać jak wiele obliczeń dla prostej funkcji złożenia, która może zrobić to samo, ale pamiętaj, że te funkcje wykonują po 30 i 60 cykli, według Intel, a parowanie tych już bardzo szybkich instrukcji (jeśli to możliwe) oznacza, że pokonasz powolne instrukcje FPU. Jest to coś, o czym należy pamiętać przy wyborze dowolnej technologii programowania (SIMD, FPU, CPU). Byłoby to bardzo trudne, gdyby nie wyszło z tego nic dobrego, ale wprowadzono tu również pewne dobre ulepszenia. Funkcja została porównana z funkcją zasilania dostarczoną przez kompilator, a także z duo log2 i 2x, które zostały zbudowane do tej pory. Gdy precyzja jest ustawiona dla obu funkcji na około 16 bitów, działa około trzy razy szybciej niż funkcja pow. . Najlepszy przypadek jest nieco mniejszy niż trzy razy szybciej, a najlepsza precyzja jest ponad dwa razy szybsza. Ale znowu, fakt, że używasz zestawu funkcji całkowitych do wykonania zadania fskale i frndint (które , co wykorzystuje kompilator) zdecydowanie pomaga.

Przejście do czwartego biegu za pomocą SIMD

Jakby dotychczasowa prędkość nie była wystarczająca, możesz przesuwać kopertę jeszcze dalej za pomocą instrukcji SIMD. Jednym z problemów z instrukcjami SIMD jest brak funkcji transcendentalnej. Jeśli potrzebujesz kalkulacji z udziałem cosinus / sinus lub jakakolwiek inna funkcja transcendentalna, musisz obliczyć wartość za pomocą wartości zmiennoprzecinkowych, po czym następuje konwersja do rejestru SIMD. Nigdy więcej. Funkcje, które wymyśliliśmy, mogą być stosowane bardzo dobrze, jeśli potrzebujesz dużo obliczeń funkcji transcendentalnych, co jeszcze bardziej zwiększa ograniczenie prędkości. Niefortunne jest to, że wszystkie te operacje wymagają tożsamości, które zmieniają funkcję podlegającą ocenie, w zależności od wartości wejściowej. SIMD nie jest zbyt dobrze przystosowany do wykonywania obliczeń na pojedynczej wartości, więc na ogół te przypadki będą musiały być załatwione zanim użyjemy SIMD. SIMD staje się przydatny podczas obliczania wielomianu. W przypadku niektórych funkcji, takich jak cosinus / sinus, istnieją sposoby obejścia wszystkich tych weryfikacji przed użyciem SIMD, usuwając w ten sposób wszystkie gałęzie w kodzie. Skoro już widziałeś cosinus,

spójrzmy na funkcję sinusa za pomocą skrętu instrukcji SSE. Przypomnijmy, że ideą kodu cosinus było sprawdzenie pod kątem symetrii, gdzie jedna była tylko lustrem, a druga zmieniła znak. Metoda zmiany znaków w całym kodzie polega na odwróceniu znaku pierwszego współczynnika i zmianie znaku zmiennej, która mnoży X w każdym ruchu. Teraz oczywiście chciałbyś całkowicie pozbyć się skoków, a możesz. Sztuczka pochodzi z instrukcji min / max i innego "bitu" zabawy. Krzywa sinusoidalna ma dwa półodbicia przy π2 i 3π2, a także można uznać środek krzywej (π) Za odbicie, ale z odwróceniem znaku. Aby uzyskać znak, możesz obliczyć maskę bitową za pomocą instrukcji SSS cmpss, której możesz użyć na końcu, aby odwrócić znak. Jedynym krokiem, jaki pozostajesz, jest obliczenie odbicia w taki sposób, aby wybrany numer był tym z zakresu [0 ...π / 2]. Można to łatwo osiągnąć za pomocą instrukcji min / max.

Kod wersji funkcji zatokowej serii Taylora następuje:

```
__declspec (wyrównanie (16)) const zmienny statyczny Full [4] =

{2 * PI, 2 * PI, 2 * PI, 2 * PI};

__declspec (align (16)) const statyczny float Half [4] =

{PI, PI, PI, PI};

__declspec (wyrównanie (16)) const static unsigned long Mask [4] =

{0x80000000, 0x80000000, 0x80000000, 0x80000000};

__declspec (align (16)) const statyczny float CoEffi [4] = {...}

// ^^ Powtórz tę linię dla każdego współczynnika wielomianu

voidSin (float X [4])

{

__jako M {

movaps xmm0, [X]

movaps xmm1, [Full]

subps xmm1, xmm0 // xmm1 = 2PI - X

movaps xmm7, xmm1

cmpps x mm7, x mm0, 1

andps xmm7, [maska] // xmm7 = <maska znaku>

minps xmm0, xmm1 // xmm0 = X [0..PI]

movaps xmm1, [połowa]

subps xmm1, xmm0 // xmm1 = PI - X

minps xmm1, xmm0

movaps xmm2, xmm1 // xmm1 = X [0..PI / 2]

mulps xmm2, xmm2 // xmm2 = X ^ 2
```

```
movaps xmm0, xmm1 // xmm0 = X

mulps xmm1, xmm2 // xmm1 = X ^ 3

movaps xmm3, [CoEffi]

mulps xmm3, xmm1

addps x mm0, xmm3 // xmm0 + = X ^ 3/6

... // Powtórz ostatni blok wszystkie coeff.

xorps x mm0, xmm7 // Zmień odpowiednio znak

movaps [X], xmm0

}

}
```

Wbrew prawdzie ten kod działa bardzo dobrze. W porównaniu z funkcją grzechu dostarczoną przez kompilator jest nieco mniej niż 1,5 razy szybszy dla 20 bitów precyzji. To około 75% oryginalnej prędkości dla pełnej precyzji i około 60% dla najgorszej precyzji. Jest to niesamowite, ponieważ dzięki wersji SIMD można przybliżać cztery liczby na raz i wykonać czterokrotnie więcej pracy. Odtąd w rzeczywistości jesteś (6, 5,3, 6,6) razy szybszy, w tej kolejności, niż funkcja kompilatora. Porównawczo, jeśli weźmiesz średnią liczbę cykli Intela dla tej funkcji, będzie ona wynosić 160-180. Dzięki tym ulepszeniom, środkowa skrzynka pozwala nam to osiągnąć w ~ 27-30 cyklach średnia, która jest raportowaną prędkością mnożenia 2-całkowitych. Ponieważ robisz stałe obliczenia, twój kod nie ma zasięgu. Stąd bierze się około 28 cykli na grzech. Konwertowanie innych funkcji na SIMD zapewne przyniesie również doskonałe wyniki, ale ogólnie wyniki nie powinny być tak dobre. W tym konkretnym przypadku udało się całkowicie pozbyć się wszystkich gałęzi, co nie jest możliwe do osiągnięcia dla większości poprzednich funkcji.

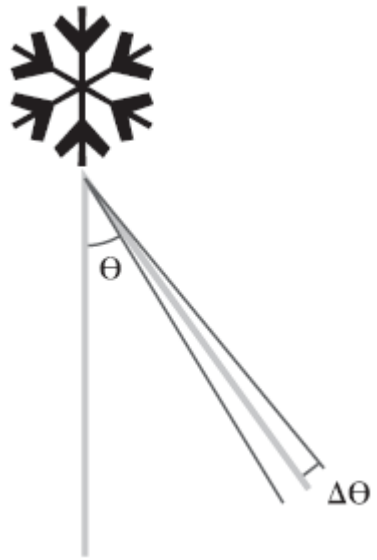
Przybliżenie modelu fizycznego

Podczas projektowania gry zazwyczaj chcesz zmaksymalizować wykorzystanie zasobów. Zasoby obejmują moc procesora, moc GPU, moc FPU, dźwięk IO itd. Jeśli masz dużo wolnych zasobów, możesz dodać kilka fajnych efektów specjalnych, zwiększyć dokładność w różnych obszarach lub po prostu odciążyć trochę wagi z jednego zasobu do drugiego osiągnąć większą liczbę klatek na sekundę. Oczywiście musisz wziąć pod uwagę, że czas jest także zasobem i zazwyczaj nie masz zbyt wiele do stracenia. Ale często możesz dodać fajne efekty specjalne w ciągu jednego lub dwóch dni. Ideą przybliżenia modelu fizycznego jest modelowanie procesu fizycznego za pomocą bardzo szorstkiego równania z powrotem do pieluchy. Jednym z pokazów dołączonych do tej części jest symulacja opadów śniegu połączona z efektem wody, symulującym obiekt wpadający do wody. Oczywiście możesz zajrzeć do podręczników fizyki i spróbować zasymulować matematykę za efektem, ale dlaczego robisz to, gdy efekty są czysto wizualne? Prawdziwym testem tutaj jest "Czy to dobrze wygląda?"

Symulacja opadów śniegu

Wiele gier wdrożyło symulację opadów śniegu, ale prawie wszystkie z nich stosowały najnowocześniejszą technikę, w zależności od mocy komputerów w momencie ich wydania. Najgorsze, jakie widziałem, to cząstki białego kwadratu opadające w linii prostej. Niektórzy bardziej sprytni programiści zastosowali nieco inną technikę w 2D, co daje całkiem przyzwoite wyniki w grze 2D. Pomysł polegał na tym, aby cząsteczka opadła liniowo, ale aby nieznacznie przesunąć cząstkę o małą liczbę

losową. Nie było tak źle, ale wyglądało na to, że cząsteczka trzęsła się histerycznie. Prawdziwy fizyczny model uwzględniałby tarcie powietrza, grawitację, powierzchnię płatka śniegu i wiele innych szczegółów. To spowodowałoby, że symulacja była zbyt powolna. Jeśli kiedykolwiek widziałeś śnieg, wiesz jak opisać ruch spadającego płatka śniegu. Zachowuje się mniej więcej jak spadający kawałek papieru, ale przy znacznie mniejszym obrocie. Kluczem do tego jest ostatnie słowo: rotacja. Jeśli chcesz zasymulować płatek śniegu, zmień jego obrót, zamiast zmieniać jego pozycję. Gdy spada płatek śniegu, tarcie powietrza sprawia, że lekko kołysze się w lewo i prawo. To jest efekt, który chcemy symulować. Losowanie pozycji przesunięcia cząstki nie było złym pomysłem, ale nie było płynne ani stopniowe. Jeśli wybierzesz losowo kąt cząstki, otrzymasz naprawdę dobre przejście od lewej do prawej. To wygląda bardziej naturalnie. Rysunek 21.8 pokazuje kąt padania płatka śniegu, a także jego losowe przesunięcie kąta, które jest obliczane dla każdej klatki.



Nie chcesz, aby twoje płatki śniegu zaczęły latać w górę, więc musisz ograniczyć wartości kąta bez określonego zakresu. Jeśli chcesz mieć ochotę, możesz również użyć kąta delta, który jest proporcjonalny do czasu. Byłoby to bardziej poprawne niż statyczna kątowa delta. Wybrałem jednak użycie prostszej metody w demo. W demo zaimplementowano to, że odległość przebyta przez cząstkę jest proporcjonalna do czasu, więc użytkownik na wolniejszym komputerze nie widzi naprawdę wolnego śniegu spadającego. Widzi tylko mniej drastyczne (X, Z) ruchy w płatkach śniegu, co powinno być dopuszczalne. Teraz musimy wymyślić równanie, które zrobi to wszystko. Jest to całkiem proste, ponieważ ta konwersja jest podobna do konwersji współrzędnych sferycznych na współrzędne kartezjańskie. Jedyna różnica polega na tym, że chcesz, by kąty miały zakres [-a, a]. Możesz łatwo użyć krzywej sinusoidalnej, dla której wartość 0 w ogóle nie przesunie cząstki i dla której dowolna wartość graniczna będzie miała maksymalny ruch dla tego zakresu. Jeśli jest to nieco mylące, wróć i spójrz na krzywą sinusoidalną, aby zobaczyć, jak zachowuje się dla wartości z zakresem [-π/ 2,π / 2], gdzie wartość Y jest przesunięciem, które zastosujesz w X lub Z. Tym, co musisz się zastanowić, jest wartość, którą powinieneś użyć dla Y. Możesz zwiększyć Y o stałą wartość, ale byłoby ładniej, gdyby cząstka nie spadła tak szybko, jeśli porusza się dużo w X lub Z, na przykład . Aby to osiągnąć, możesz po prostu obliczyć cosinus każdego kąta i pomnożyć je razem. Ponownie, jeśli spojrzysz na krzywą cosinus, jej wartość jest maksymalna na 0 i spada, gdy zbliżasz się do twoich granic dla twojego podanego zakresu. W skrócie, równanie to :

$$\langle x, y, z \rangle = Speed \cdot Time \left\langle \sin\left(AngleX, \cos\left(AngleX\right)\cos\left(AngleZ\right), \sin\left(AngleZ\right)\right)\right\rangle$$

Nie ma powodu, aby wybrać tę formułę dla Y, inną niż fakt, że maleje wraz ze wzrostem kątów dla dowolnego kąta AngleZ lub AngleX. Równie ważnym wyborem byłoby obliczenie cosinusa dodawania dla bezwzględnej wartości kątów. Osiągnęłoby to mniej więcej ten sam efekt. Dopóki jest wydajne i wygląda dobrze, powinno uzyskać zielone światło.

Symulacja baniek

Teraz, kiedy już rozmawialiśmy o płatkach śniegu, ten aspekt powinien być dość łatwy do symulacji. Kiedy podążysz pod wodę, możesz chcieć symulować bąbelki docierające na powierzchnię. W symulacji płatka śniegu potrzebowaliśmy funkcji, która płynnie przechodziła od lewej do prawej, ponieważ w ten sposób lekkie przedmioty spadają z gazu. Bańka jest trochę inna. Powietrze ma na przykład znacznie mniejszą gęstość niż woda, a bąbelek nigdy nie jest naprawdę sferyczny, nawet jeśli użyjemy kuli do przybliżenia bańki. Z tego powodu niektóre części bańki mają tendencję do poruszania się szybciej niż niektóre inne obszary pęcherzyka, a to powoduje masowe przesunięcie powietrza z jednego obszaru do drugiego. Jeśli uznasz bańkę za pojedynczy punkt (jej środek), zauważysz, że punkt faktycznie porusza się szybko od lewej do prawej (z powodu tej zmiany masy, którą możesz obserwować). Jest to spowodowane ciśnieniem płynu. Biorąc pod uwagę to, formuła określająca położenie bańki staje się dość łatwa. Możesz po prostu wybrać losowo pozycję bańki, a będziesz skutecznie renderować całkiem przyzwoitą symulację. W wersji demonstracyjnej możesz wejść pod wodę, aby zobaczyć bąbelki. Nie zawsze musisz komplikować symulacje więcej.

Symulacja obiektu uderzającego w płyn

Jednym bardzo zgrabnym (ale raczej kosztownym) efektem jest generowanie tętnienia przez obiekt zderzający się z płynem. Problem polega na tym, że woda jest dość złożonym obiektem fizycznym, a jej poprawna symulacja wymaga sporo pracy matematycznej. Oczywiście, nie możesz sobie na to pozwolić w grze w czasie rzeczywistym. Załóżmy, że renderujesz mapę wysokości, a zmarszczki są obliczane przez funkcję, która po prostu dodaje określoną wysokość do twojej mapy wysokości, aby wygenerować falowanie. Kiedy obiekt penetruje płyn, najpierw tworzy otwór, popychając płyn ze wszystkich stron. Gdy ciecz zasysa przedmiot pod nią, z powodu masy otaczającej otwór, w miejscu, w którym obiekt spadł, powstaje wysoka kolumna płynu. W całym tym procesie płyn wokół obszaru docelowego powoli zaczyna generować okrągłe fale rozciągające się tak daleko, jak pozwala na to lepkość płynu. Możesz zobaczyć, dlaczego ten model nie jest bardzo popularny w grach. Ma dość wysoki stopień złożoności i zajmuje trochę czasu na procesorze. Wszystko, co chcesz zrobić, to zasymulować ten efekt. Pierwszą rzeczą, o której powinieneś pomyśleć, jest okrągły aspekt wzoru. Musi mieć tę samą wysokość dla każdego kąta, o ustalonym promieniu. Podaj równania, twoje dane wejściowe (X, Z) faktycznie zamieniają się na następujące wartości:

$$In = \sqrt{x^2 + z^2}$$

To jest miłe, ponieważ nie musisz już myśleć w 3D. Możesz myśleć w 2D, ponieważ

Pomyślnie zwinąłeś swoje dwie wartości wejściowe w jedną. Możesz łatwo symulować małe fale z ujemną krzywą cosinus. Krzywa cosinus jest wybrana, ponieważ zaczyna się od maksimum, stąd dziura. Ponadto wybierasz ujemną wartość krzywej cosinus, ponieważ chcesz mieć dziurę, a nie bryłę. W przeciwnym razie generowałbyś ciągłość krzywej cosinusu w miarę zbliżania się do nieskończoności. Niestety, istnieje problem. Kiedy wskakujesz do basenu, fale są znacznie mniejsze na drugim końcu niż w miejscu, w którym skaczesz. Musisz wziąć pod uwagę fakt, że Im dalej od 0 twojego wejścia, tym mniejsze są fale. Masz do wyboru wiele opcji, ale w przypadku wersji demo wybrałem krzywą wykładniczą. Ponownie, jeśli spojrzysz na wykres krzywej wykładniczej, zauważysz, że krzywa jest dość

wysoka dla dużych wartości i dąży do zera dla mniejszych wartości. Dokładnie tak chcesz zachować krzywą. Wszystko, co musisz zrobić, to wybrać dobry punkt wyjścia na krzywej wykładniczej i skalować wejście tak, aby krzywa zmniejszała się tak szybko, jak chcesz. Równanie, jakie dotychczas mieliśmy, jest podobne do tego:

$$e^{a-In\cdot Scale1}\cos\left(In\cdot Scale2\right)$$

W tym momencie nie zrobiłeś nic o czasie, który upłynął, podczas gdy ta krzywa jest renderowana. Po pierwsze, czas powinien wypchnąć fale z punktu centralnego. Efekt ten można osiągnąć po prostu przez odjęcie skalarnej wartości czasu od krzywej cosinus. Ponownie, jeśli tego nie widzisz, spójrz na krzywą cosinus i przesuń całą krzywą w lewo. Pamiętaj tylko, aby zmniejszyć jego wysokość o tę samą funkcję i powinieneś to zobaczyć. Na koniec musisz wyjaśnić, że fale koncentrują się wokół celu, ale potem się rozpraszają. Jest to trudny efekt, ale można go osiągnąć w dwóch etapach. Pierwszym krokiem jest stworzenie fali, w której dotknięta rozpiętość rośnie na zewnątrz. Drugim krokiem jest zmniejszenie całkowitej wysokości fali. Ten drugi krok nie jest zbyt trudny. Krzywa wykładnicza zmierza w kierunku zera, gdy jej wartość jest ujemna i bardzo mała, więc można po prostu zmniejszyć czynnik czasu w wykładniczej dla drugiego przypadku. Ale co z pierwszym przypadkiem? Jak możesz symulować rozwijającą się falę bez zwiększania złożoności? Jeśli zmniejszysz wartość wejściową i pozwolisz jej rozszerzać się stopniowo, aż osiągnie pełną rozpiętość, możesz łatwo osiągnąć pożądany efekt. Najlepszym sposobem wizualizacji jest przyjęcie krzywej wykładniczej w danym zakresie i początkowo ściskanie krzywej tak, aby pasowało do bardzo małego zakresu. Z czasem pozwól mu rozszerzyć się do normalnego zakresu. To skutkuje ekspansją dotkniętego regionu. Równania dla pierwszego i drugiego kroku są następujące:

Pierwszy Krok

$$a^{1-\frac{In\cdot Scale1}{Time\cdot Scale3}}\cos\left(In\cdot Scale2-Time\cdot Scale4\right)$$

Drugi Krok .

$$e^{a-In\cdot Scale1-Time\cdot Scale3}\cos\left(In\cdot Scale2-Time\cdot Scale4\right)$$

W sumie jest to dość skomplikowane. Masz dwa wejścia, {Czas, W}, więc możesz przybliżyć tę funkcję jako powierzchnię 3D. Zamiast tego zdecydowałem się przybliżyć funkcję w sposób chaotyczny. Spójrz na kod źródłowy. W żadnym wypadku nie jest tak optymalne, jak to tylko możliwe. Jako ćwiczenie, zbadaj go i spróbuj zoptymalizować równania jeszcze bardziej z tym samym pożądanym efektem. To jest możliwe!