Sphere tracing: integracja z klasycznymi metodami symulacji i renderingu

Michał Jarząbek

Uniwersytet im. Adama Mickiewicza

Streszczenie

Sphere tracing to metoda renderowania pól odległości używana ostatnio głównie w produkcjach demoscenowych. Modelowanie przy użyciu pól odległości ma właściwości wizualne, które warto wykorzystać przy tworzeniu gier komputerowych. Nie jest jednak oczywiste w jaki sposób połączyć sphere tracing oraz pola odległości z klasycznymi podejściami stosowanymi w grach. Referat zaprezentuje podstawy omawianych metod oraz aplikację demonstrującą zintegrowane: reprezentację sceny przy użyciu pól odległości, symulację fizyczną w scenie prostych obiektów (sfer) oraz rendering przy jednoczesnym użyciu sphere tracingu i rasteryzacji.

1. Wstęp

Twórcy produkcji demoscenowych przeznaczonych na komputery PC zawsze stali w pierwszym szeregu autorów, testerów i użytkowników

najnowszych metod renderowania grafiki w czasie rzeczywistym. Intra i dema, w przeciwieństwie do gier komputerowych, nie są interaktywne, nie wykonują obliczeń związanych ze sztuczną inteligencją ani grą sieciową, a działać muszą przede wszystkim na high-endowych konfiguracjach sprzętowych używanych podczas konkursów – mogą zatem przeznaczyć znacznie więcej mocy obliczeniowej na proces generowania obrazu. Ponadto ze względu na swoją prostą konstrukcję mogą korzystać z metod, którym brakuje elastyczności niezbędnej przy tworzeniu dużych i zróżnicowanych aplikacji – na przykład, jeżeli opis całej renderowanej sceny można zawrzeć w jednym programie wykonywanym na GPU (tzw. shaderze), to tym lepiej.

Techniki oparte na metodzie śledzenia promieni (ang. ray tracing) zawsze cieszyły się powodzeniem zwolenników atrakcyjnych wizualizacji ze względu na dużo większe możliwości uzyskiwania efektów tzw. globalnego oświetlenia. Omawiana metoda sphere tracingu opiera się na procesie maszerowania wzdłuż promienia (ang. ray marching) i służy do wizualizacji scen opisanych przez powierzchnie niejawne (ang. implicit surface) przy użyciu funkcji odległości (ang. distance function). W rozdziałach 2. i 3. opisano powierzchnie niejawne i funkcje odległości. W rozdziale 4. krótko omówiono podstawy związane z metodami bazującymi na śledzeniu promieni. Rozdział 5. jest poświęcony technice sphere tracingu.

Powierzchnie niejawne mają specyficzne wizualne właściwości (np. łatwa zmiana topologii obiektu), które można byłoby wykorzystać jako podstawę do stworzenia gry komputerowej. Gra taka, przy ograniczeniach wydajnościowych dzisiejszego sprzętu, siłą rzeczy byłaby raczej niewielkiej skali, w zamian jednak otrzymalibyśmy narzędzie do stworzenia specyficznego klimatu lub rozgrywki. Sphere tracing w wydaniu democenowym mógłby jednak okazać się metodą niewystarczająco elastyczną lub po prostu zbyt kosztowną obliczeniowo aby oprzeć na nim cały proces wizualizacji gry. W rozdziale 6. zaprezentowany zostanie schemat integracji renderowania sceny przy użyciu sphere tracingu i renderowania obiektów w scenie za pomocą rasteryzacji. Dodatkowo w rozdziale 7. przedstawiona zostanie metoda symulacji fizycznej prostych obiektów w scenie opisanej przez funkcje odległości.

2. Powierzchnie niejawne

W przeciwieństwie do jawnych modeli opisu powierzchni w modelowaniu 3D (np. siatek trójkątów, powierzchni parametrycznych), powierzchnie niejawne zdefiniowane są przez zbiór punktów spełniających pewne równanie

$$F(x, y, z) = 0 \tag{2.1}$$

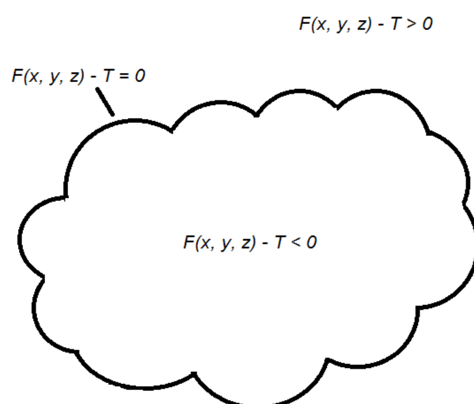
gdzie *F* to funkcja ciągła $R^3 \to R$.

Przykładowo równanie kuli $x^2 + y^2 + z^2 - r^2 = 0$ opisuje nieskończoną liczbę punktów w odległości *r* od środka układu. Punkty te leżą na wspólnej powierzchni (powierzchni kuli). Jeśli weźmiemy jakikolwiek inny punkt przestrzeni i podstawimy go do równania, otrzymamy jakąś wartość niezerową. Wartość ta może posłużyć do określenia czy punkt znajduje się wewnątrz czy na zewnątrz powierzchni.

Równanie 2.1 możemy zgeneralizować:

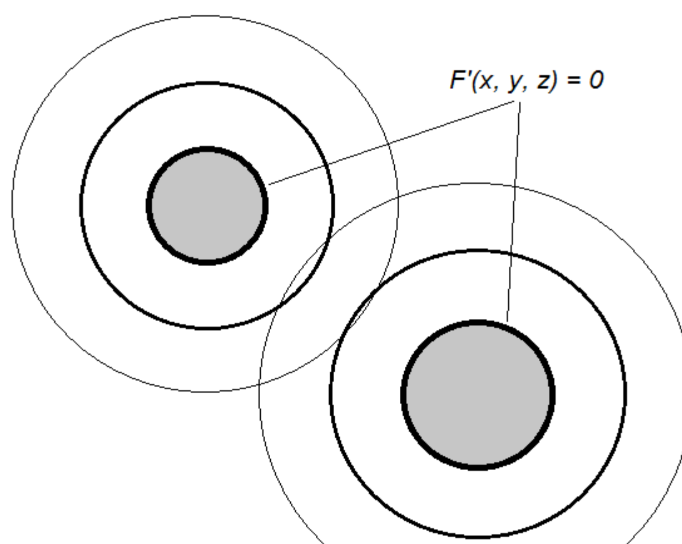
$$F(x, y, z) - T = 0 \tag{2.2}$$

Teraz, zmieniając stałą *T* otrzymamy różne powierzchnie, które będą określone przez zbiór tych punktów *(x, y, z)*, dla których równanie 2.2 przyjmuje wartość zero. Każdą taką powierzchnię nazywamy izopowierzchnią.



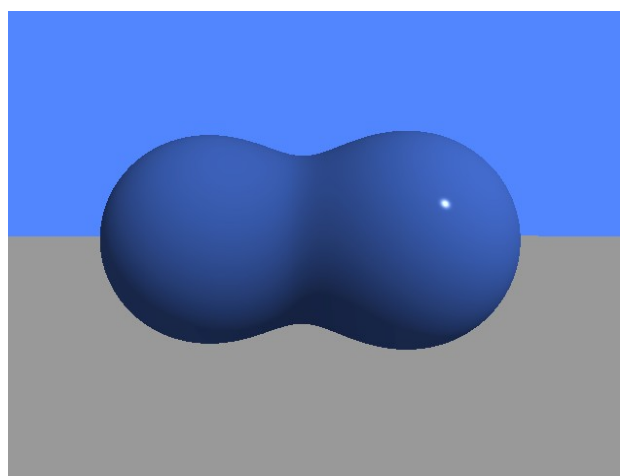
Rys.2.1. Wartość funkcji określa położenie punktu względem powierzchni

Niekiedy warto na funkcję *F* patrzeć jak na funkcję generującą swego rodzaju pole wpływu. Zdefiniujmy *F' = G + H*, gdzie *G* i *H* to funkcje opisujące kulę w punktach odpowiednio *P1* i *P2*. Jeśli umieścimy punkty *P1* i *P2* w dużej odległości od siebie (tzn. znacznie większej niż suma promieni kul) i przyjmiemy *T* = 0, to równanie *F' = 0* opisze powierzchnię składającą się z dwóch rozłącznych elementów, z których każdy będzie niemal dokładnie kulą – niemal, ponieważ nawet w punktach odległych od środka każdej kuli, jej wpływ będzie niezerowy i finalna powierzchnia zostanie nieznacznie zdeformowana.



Rys.2.2. Powierzchnia niejawna powstała z sumy dwóch funkcji

Jeśli jednak zbliżymy dwie kule do pewnej granicznej odległości, sumowanie wpływów spowoduje płynne połączenie dwóch elementów powierzchni w jeden (tzw. blob).



Rys.2.3. Powierzchnia niejawna powstała z sumy dwóch funkcji. Po zbliżeniu dwóch źródeł wpływu do siebie powstała jedna powierzchnia o charakterystycznym wyglądzie – blob

Jak widać, podstawowe manipulacje równaniem mogą spowodować zmianę spójności (ang. connectivity) powierzchni niejawnej.

2.1 Wektor normalny

Kluczowym narzędziem wykorzystywanym w grafice komputerowej jest wektor normalny do powierzchni. Dla powierzchni niejawnej można go zdefiniować przy użyciu gradientu funkcji. Gradient funkcji niejawnej to wektor jej pochodnych cząstkowych:

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right)$$

Jest on prostopadły do izopowierzchni i wskazuje w kierunku najszybszego wzrostu funkcji F. Wskazuje zatem w tym samym kierunku co wektor normalny do powierzchni. W takim razie, jednostkowy wektor normalny to:

$$\vec{N} = \frac{\nabla F}{|\nabla F|}$$

Więcej informacji o powierzchniach niejawnych w [1].

3. Funkcje odległości

Wartością funkcji odległości postaci $F(x, y, z)$ jest odległość punktu (x, y, z) od najbliższego punktu na izopowierzchni. Interesować nas będą przede wszystkim tzw. funkcje odległości ze znakiem (ang. signed distance function), będących podzbiorem funkcji niejawnych.

Funkcja odległości jest zdefiniowana jako:

$$d(\mathbf{x}) = min(|\mathbf{x} - \mathbf{xi}|) \quad \text{dla wszystkich } \mathbf{xi} \text{ w } S \tag{3.1}$$

gdzie $\mathbf{x} = (x, y, z)$, S to zbiór punktów izopowierzchni. Dla punktów na izopowierzchni wartość funkcji odległości wynosi zero.

Funkcja odległości ze znakiem jest zdefiniowana bardzo podobnie, z tą różnicą, że wartości dla punktów wewnątrz powierzchni są ujemne. Widać zatem, że funkcje odległości są podzbiorem funkcji niejawnych (dodatnie na zewnątrz, zerowe na, ujemne wewnątrz powierzchni). O wartościach funkcji niejawnych w poprzednim rozdziale nie było powiedziane wiele, poza tym że są ciągłe i jaki jest ich znak. Teraz mówimy o funkcjach, na które nałożono dodatkowe ograniczenie - muszą mieć wartości dokładnie równe odległości punktu od powierzchni. Wyraża się to przez równanie

$$|\nabla d| = 1$$

Jest to intuicyjne w tym sensie, że w punkcie dwa razy dalszym od powierzchni, wartość d jest dwa razy większa.

3.1 Uzyskiwanie funkcji odległości

W rozdziale 2. widzieliśmy funkcję niejawną $F = x^2 + y^2 + z^2 - r^2$ powstałą z równania kuli, która posłużyła nam do zamodelowania powierzchni kuli. Funkcje ta nie spełnia jednak naszych wymagań do funkcji odległości. Na przykład, dla $r = 1$, jej wartość w punkcie $(2, 0, 0)$ wynosi 3, podczas gdy odległość tego punktu od powierzchni kuli o promieniu $r = 1$ w rzeczywistości wynosi 1.

W przypadku kuli sprawa jest prosta. Jej funkcja odległości to po prostu $d(\mathbf{x}) = ||\mathbf{x}|| - r$, gdzie $\mathbf{x} = (x, y, z)$, $||(x, y, z)||$ oznacza normę euklidesową wektora. Łatwo jest uzyskać funkcję odległości również dla innych prostych obiektów geometrycznych jak prostopadłościany, cylindry czy torusy [2].

Co jednak zrobić, gdy powierzchnia jest na tyle skomplikowana, że nie jesteśmy w stanie w prosty sposób określić funkcji odległości? Jeżeli znamy opisującą ją dowolną funkcję niejawną F, która na interesującym nas przedziale jest ciągła w sensie Lipschitza ze stałą λ, to funkcja F/λ jest ograniczeniem dolnym na funkcję odległości opisywanej powierzchni. W rozdziale o sphere tracingu zobaczymy, dlaczego zarówno funkcje odległości jak i ich ograniczenia dolne są dla nas interesujące.

Stała Lipschitza funkcji w pewnym przedziale opisuje, nieformalnie mówiąc, największy możliwy wzrost wartości funkcji w tym przedziale. Im szybciej funkcja rośnie w pewnych miejscach przedziału (a co za tym idzie, im bardziej „odległa" jest w swym zachowaniu od funkcji odległości, której szybkość wzrostu jest dokładnie określona), tym większa jej stała Lipschitza i przez tym większą wartość należy ją podzielić, aby otrzymać funkcję, która rośnie nie szybciej niż funkcja odległości (czyli ograniczenie funkcji odległości). Dokładne omówienie pojęcia ciągłości lipschitzowskiej wykracza poza ramy tego referatu (zob. [2]).

3.2 Konstruktywna geometria brył

Konstruktywna geometria brył (ang, constructive solid geometry, CSG) to metoda tworzenia powierzchni poprzez łączenie innych powierzchni działaniami boolowskimi: sumą, przecięciem i różnicą. Działanie te są dobrze zdefiniowane dla funkcji odległości. Weźmy dwie powierzchnie A, B i opisujące je funkcje odległości FA, FB.

Odległość do sumy powierzchni A i B to minimum wartości ich funkcji: $min(FA(x, y, z), FB(x, y, z))$.

Przecięcie powierzchni definiujemy przy pomocy operacji maksimum: $max(FA(x, y, z), FB(x, y, z))$. Nie jest to jednak dokładna funkcja odległości, a jedynie jej ograniczenie dolne (zob. [2]).

Podobnie, różnica definiowana jest jako *max(FA(x, y, z), -FB(x, y, z))*. Tak jak w przypadku przecięcia, wynik to jedynie dolne ograniczenie na funkcję odległości.

Jeżeli *FA* lub *FB* są dolnymi ograniczeniami na funkcje odległości (zamiast dokładnymi funkcjami), to funkcje będące wynikami powyższych operacji CSG również pozostaną dolnymi ograniczeniami na odpowiednie skonstruowane powierzchnie.

3.3 Transformacje powierzchni niejawnych

Podane wcześniej przykłady funkcji niejawnych i funkcji odległości modelowały powierzchnie w środku układu współrzędnych, co na ogół nie jest satysfakcjonujące – chcemy mieć możliwość dowolnego ustawienia i zorientowania obiektu w przestrzeni. Pożądane przekształcenie powierzchni można by oczywiście zawrzeć w definicji funkcji, byłoby to jednak niewygodne w sytuacji, gdy stosowane do powierzchni transformacje zmieniają się dynamicznie.

Typowo, powierzchnie niejawne przekształcamy poprzez zastosowanie transformacji odwrotnej do dziedziny funkcji. Jeśli T jest transformacją przestrzeni, zaś F definiuje powierzchnię niejawną, wtedy przekształcona powierzchnia jest zdefiniowana jako powierzchnia niejawna takiej funkcji:

$$F(T^{-1}(x, y, z)) = 0 \qquad (3.2)$$

Niestety, jeżeli funkcja F jest funkcją odległości, to niektóre transformacje mogą spowodować, że funkcja po lewej stronie równania 3.2 nie będzie już prawidłową funkcją odległości. Aby nową funkcję przekształcić z powrotem w funkcję odległości (lub przynajmniej jej dolne ograniczenie – zależnie od zastosowanej transformacji), znów należy posłużyć się pojęciem ciągłości lipschitzowskiej – należy określić stałą Lipschitza λ stosowanej transformacji na interesującym przedziale i podzielić lewą stronę równania 3.2 przez λ.

Translacja i obrót są przykładami izometrii – transformacji zachowujących odległości. W przypadku ich stosowania, korekta wartości zwracanej przez *F* nie jest konieczna.

Przykładem przekształcenia, które nie zachowuje odległości jest skalowanie jednorodne. Jeżeli współczynnik skalowania to s, transformacja odwrotna to skalowanie ze współczynnikiem *1/s*. Stała Lipschitza takiego przekształcenia również wynosi *1/s*. Zatem odległość *d'* od przeskalowanej powierzchni, której funkcja odległości to *d*, wynosi:

$$d'(x, y, z) = d(S^{-1}(x, y, z)) / (1/s) = sd(S^{-1}(x, y, z)) \qquad (2.3)$$

3.4 Deformacje

Oprócz liniowych przekształceń pokazanych w poprzednim paragrafie, do dziedziny funkcji możemy zastosować przekształcenia nieliniowe – deformacje przestrzeni, np. skręcenie (twist), wygięcie (bend).

Zdeformować możemy również wynik funkcji – zwróconą odległość. Przykładem takiego zniekształcenia byłoby 'zmieszanie' wyników dwóch funkcji odległości, aby z dwóch obiektów utworzyć blob. Zauważmy, że zwyczajna suma funkcji wpływu pokazana w rozdziale 2. dla ogólnych powierzchni niejawnych, nie będzie prawidłową funkcją odległości ani jej ograniczeniem dolnym. Zamiast tego, [2] podaje wzory na dolne ograniczenia funkcji odległości pewnej odmiany blobów.

Zarówno w przypadku deformacji dziedziny funkcji, jak i jej wartości, należy szczególnie uważnie zmodyfikować otrzymaną funkcję tak, aby pozostała ona funkcją odległości lub jej ograniczeniem dolnym, poprzez odpowiednie przeskalowanie jej wyniku.

4. Podstawy generowania obrazu

4.1 Metoda śledzenia promieni

Proces generowania obrazu w metodzie śledzenia promieni jest, w uproszczeniu, następujący:

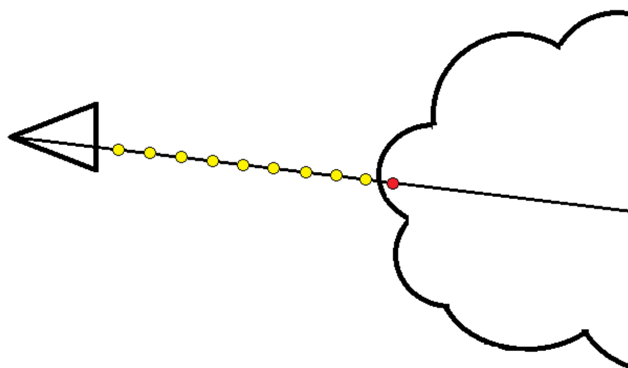
Dla każdego piksela finalnego obrazu:
1. wygeneruj promień o początku w punkcie patrzenia i przechodzący przez punkt odpowiadający w przestrzeni rzutowania danemu pikselowi
2. znajdź punkt przecięcia promienia z najbliższą powierzchnią
3. znajdź kolor powierzchni w punkcie przecięcia

Ostatni punkt został tu bardzo okrojony, jednak jego szczegóły nie są dla nas teraz interesujące. Od opisów powierzchni używanych w klasycznym ray tracingu wymaga się możliwości analitycznego znalezienia punktu przecięcia powierzchni z danym promieniem. Funkcje niejawne w ogólności nie pozwalają nam na to.

4.2 Metoda maszerowania wzdłuż promienia

W metodzie maszerowania wzdłuż promienia punkt 2. algorytmu z poprzedniego paragrafu wygląda (znów w uproszczeniu) następująco:

2. przesuń się wzdłuż promienia o odległość równą KROK i sprawdź, czy nowy punkt znajduje się wewnątrz powierzchni. Jeśli nie, ponów. Jeśli tak, zakończ – punkt przecięcia został znaleziony.



Rys.4.1. Ilustracja procesu ray marchingu

W tej metodzie nie oczekuje się już od reprezentacji powierzchni, aby umożliwiała analityczne znalezienie przecięcia z promieniem. Wymaga się tylko, aby można było łatwo sprawdzić, czy punkt znajduje się wewnątrz, czy na zewnątrz powierzchni. Reprezentacją spełniającą to wymaganie są funkcje niejawne.

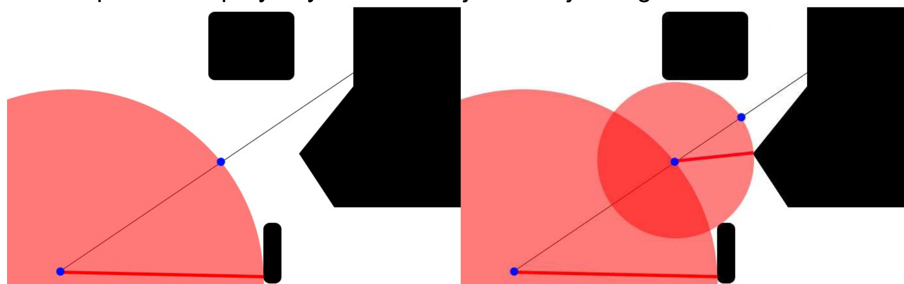
Metoda maszerowania wzdłuż promienia ma swoje charakterystyczne wady. Przy implementacji należy znaleźć odpowiedni kompromis między szybkością działania i dokładnością – im mniejsza wartość KROK, tym większy koszt obliczeniowy; im większa wartość KROK, tym większa szansa na to, że pomiędzy dwoma kolejnymi krokami powierzchnia o niewielkiej grubości wzdłuż promienia zostanie pominięta, dając błędny rezultat.

W następnym rozdziale zobaczymy, że zapewniając pewne dodatkowe informacje o powierzchni, możemy znacząco zwiększyć szybkość i dokładność liczenia punktu przecięcia promienia z powierzchnią.

5. Sphere tracing

Metoda maszerowania wzdłuż promienia wykonuje wiele kroków, których można by uniknąć, wiedząc jak dużo można posunąć się wzdłuż promienia bez „wskoczenia” w powierzchnię. Taką informację zapewniają nam funkcje odległości. Wystarczy sprawdzić wartość funkcji odległości w punkcie, w którym aktualnie znajduje się proces maszerowania, aby wiedzieć jak daleko można przesunąć się w następnym kroku. Zaznaczmy, że funkcja odległości nie mówi nam w jakiej odległości promień, wzdłuż którego maszerujemy, przetnie się z powierzchnią – funkcja odległości nie

zawiera informacji kierunkowej, my zaś podążamy wzdłuż promienia w określonym kierunku. Zamiast tego, otrzymujemy informację w jakiej odległości od sprawdzanego punktu znajduje się powierzchnia w jakimkolwiek kierunku. Nie wiemy gdzie ten najbliższy punkt powierzchni się znajduje, wiemy jednak jak daleko na pewno możemy się przesunąć, nie przecinając powierzchni. Rys. 5.1 ilustruje dwa kolejne kroki maszerowania wzdłuż promienia przy użyciu informacji z funkcji odległości.



Rys.5.1. Dwa kolejne kroki sphere tracingu (dzięki uprzejmości Inigo Quilez [3])

Wartość *d* uzyskana z funkcji odległości informuje zatem o kuli o środku w sprawdzanym punkcie i promieniu *d*, w której nie znajduje się żaden punkt powierzchni. Możemy w dowolnym kierunku zrobić krok aż do brzegu kuli bez obawy, że promień przetnie lub ominie powierzchnię. Stąd nazwa *sphere tracing*. Przecięcie z obiektem zostanie znalezione gdy, przy założeniu pewnego epsilonu dokładności, wartość funkcji odległości w którymś kroku będzie mniejsza od tego epsilonu.

Gdy znamy metodę działania sphere tracingu, staje się jasne, dlaczego ograniczenia dolne na funkcje odległości również są dla nas atrakcyjne – ponieważ w każdym kroku zwracają wartość nie większą niż wartość prawdziwej funkcji odległości. Punkt przecięcia zostanie znaleziony co prawda wolniej, jednak wynik nadal będzie poprawny. Gdy większą część sceny skonstruujemy z prawdziwych funkcji odległości, a resztę z ograniczeń dolnych, przyrost wydajności w stosunku do raymarchingu ze stałym krokiem i tak będzie znaczny.

Wektor normalny obliczamy na podstawie wzoru z paragrafu 2.1, przy użyciu metody numerycznej centralnych różnic skończonych.

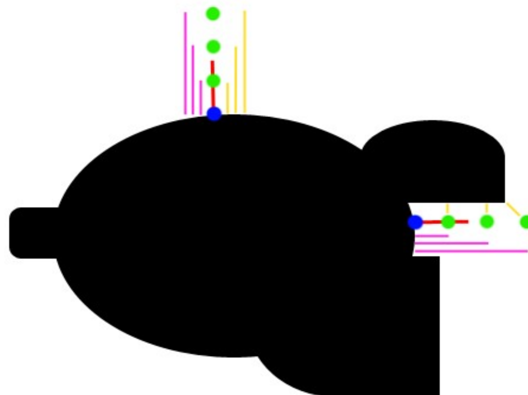
W rozdziale 3. kilkukrotnie wspomniano o transformacjach i deformacjach, które powodują, że funkcje odległości przestają nimi być. Największy problem jest oczywiście wtedy, gdy maksymalna wielkość gradientu nowej funkcji jest większa od 1 (gdy jest mniejsza, funkcja staje się po prostu ograniczeniem dolnym, które – jak zostało powiedziane – również są korzystne). Powiedziane zostało również, że należy taką otrzymaną funkcję odpowiednio przeskalować przez odwrotność jej maksymalnego wzrostu. Nie zawsze jednak chcemy lub potrafimy określić jak duża powinna

być to wartość. Możemy wtedy próbować dobrać ją eksperymentalnie, obserwując poprawność renderowanej powierzchni. Możemy również próbować „zabezpieczyć się” na poziomie globalnym – poprzez zmniejszenie wielkości kroku sphere tracingu – na przykład przesuwając się w każdym kroku nie o całą odległość zwróconą przez funkcję, ale o pewien jej ułamek. Jest to popularne podejście w produkcjach demoscenowych – łatwe i szybkie do zaimplementowania, choć nieoptymalne wydajnościowo (posuwamy się przez całą scenę z mniejszym krokiem, mimo że być może tylko część jej elementów potrzebuje korekcji odległości).

5.1 Właściwości renderowania przy użyciu funkcji odległości

Funkcje odległości dostarczają nam pewnej globalnej informacji o scenie, którą, jak się okazuje, łatwo wykorzystać do symulowania efektów oświetlenia – miękkich cieni oraz tzw. ambient occlusion (AO).

Efekt ambient occlusion w danym punkcie powierzchni jest obliczany w sposób następujący: funkcja odległości jest ewaluowana w kilku punktach wzdłuż wektora normalnego powierzchni. Każda wartość jest porównywana z odległością próbkowanego punktu od wyjściowego punktu na powierzchni. Doskonale ilustruje to rys. 5.2.



Rys.5.2. Obliczanie ambient occlusion (dzięki uprzejmości Inigo Quilez [3])

Widać, że w punkcie, który nie jest „osłonięty” i jego wartość AO powinna być niska, wartości funkcji odległości (żółte) i odległości od samego punktu (fioletowe) są niemal równe. W przypadku punktu „osłoniętego” przez pobliską geometrię, odpowiadające sobie wartości różnią się znacząco.

Otrzymany zestaw różnic należy w jakiś sposób agregować. Nie ma tu nauki, raczej eksperymenty – co będzie lepiej wyglądać. [3] proponuje zsumować różnice z wykładniczo malejącymi wagami.

Efekt miękkich cieni obliczany jest następująco: funkcja odległości znów

jest ewaluowana w kilku punktach, tym razem na drodze od sprawdzanego punktu do pozycji światła. Jeżeli w którymś punkcie wartość jest mniejsza lub równa zeru, wiadomo że punkt jest w cieniu. Możemy jednak osiągnąć więcej. Znajdujemy minimum z otrzymanych wartości odległości i obliczamy z niej współczynnik cienia w taki sposób, aby poczynając od pewnego progu, wartości coraz bliższe zera dawały coraz większą wartość współczynnika cienia. Daje nam to efekt miękkiego półcienia na krawędziach cieni.

Odległości na których operujemy, możemy wykorzystać w jeszcze jeden sposób – aby otrzymać cień, którego granica jest twarda przy krawędzi powierzchni, która cień rzuca, i zmiękcza się wraz z oddalaniem od powierzchni. W tym celu uwzględniamy znów odległość, w jakiej znajduje się każda próbka od punktu, którego cień obliczamy. Ponownie całość należy w jakiś sposób agregować. Znów, [3] proponuje, zamiast szukania minimum wartości funkcji odległości w punktach próbkowania, znaleźć minimum wyrażenia $k * d / m$, gdzie d to wartość funkcji odległości, m to odległość punktu próbkowania od punktu dla którego liczymy cień, k to stała ustalona doświadczalnie. Znalezione minimum to (po być może uprzednim przeskalowaniu) wartość współczynnika cienia.

6. Integracja z rasteryzacją

Pokażemy jak połączyć renderowanie części obiektów na scenie przy użyciu sphere tracingu z renderowniem reszty obiektów przy użyciu rasteryzacji. Całość przetwarzania będzie odbywać się na GPU z wykrzystaniem shaderów wierzchołków i pikseli.

Sphere tracing prostych scen w czasie rzeczywistym na potrzeby produkcji demoscenowych stał się możliwy dzięki implementacji na GPU. Implementacja ta jest oczywista – należy pokryć cały obszar renderowania (viewport) geometrią rysowaną z użyciem pixel shadera, w którym proceduralnie zdefiniowana jest scena przy użyciu funkcji odległości. Znając koordynaty renderowanego piksela na ekranie jesteśmy w stanie skonstruować promień zaczepiony w punkcie widzenia i przechodzący przez punkt w przestrzeni świata odpowiadający pikselowi. Do pokrycia obszaru widzenia geometrią tradycyjnie używa się dwóch trójkątów pokrywających idealnie płaszczyznę rzutowania (tzw. full-screen quad), lub jednego trójkąta (oczywiście większego niż przestrzeń rzutowania) – dzięki drugiemu rozwiązaniu skracamy o drobny procent czas wykonywania całego rysowania, gdyż w rozwiązaniu z dwoma trójkątami pewna liczba pikseli leżąca na linii styku dwóch trójkątów przetwarzana jest dwukrotnie.

Taki shader jest bardzo kosztowny obliczeniowo – jest w nim co najmniej jedna pętla, która może mieć wiele iteracji, a w każdej ewaluowana jest, być może bardzo rozbudowana, funkcja odległości. Po znalezieniu punktu przecięcia należy jeszcze znaleźć wektor normalny (3-6 ewaluacji

funkcji odległości, w zależności od wybranej metody różnic skończonych), oraz obliczyć współczynniki cienia i ambient occlusion (kilka kolejnych wywołań funkcji odległości dla każdego z efektów).

Ponieważ funkcja odległości jest bardzo kosztowna, dobrze jest opisywać przy jej użyciu tylko obiekty, których nie można wyrenderować w inny sposób. Ponadto, jeżeli interesuje nas umieszczenie w scenie siatek trójkątów, sphere tracing w ogóle nie nadaje się do ich wizualizacji. Z tych powodów, może być w pewnych sytuacjach wartościowe rysowanie części sceny przy użyciu innej metody – na przykład rasteryzacji.

Rasteryzacja oczywiście świetnie komponuje się z już używanym przez nas GPU. Kwestia integracji obrazu powstałego w procesie sphere tracingu i tego w procesie rasteryzacji nie jest jednak oczywista.

6.1 Zasłanianie

Pierwszym problem, który należy rozwiązać jest kwestia odpowiedniego zasłaniania powierzchni dalszych przez bliższe. W rasteryzacji jest to rozwiązywane przy użyciu bufora głębi, zaś w ray tracingu w sposób automatyczny – algorytm zawsze znajduje przecięcie promienia z najbliższą powierzchnią.

Tu musimy w jakiś sposób zintegrować oba podejścia. Możliwe są dwa rozwiązania. Możemy podczas tworzenia obrazu w sphere tracingu do drugiego rendertargetu zapisać wartość głębi punktu, na który pada promień i użyć go jako bufora głębi przy renderowaniu obiektów w rasteryzacji.

Możemy też zrobić odwrotnie – w trakcie rasteryzacji tworzyć dodatkowy rendertarget z głębią sceny i korzystać z niego przy sphere tracingu, choć już nie jako hardware-owy bufor głębi (nic by to nam nie dało, bo nie rysujemy w tym etapie przecież „prawdziwej” geometrii), tylko poprzez porównywanie w shaderze wartości głębi punktu przecięcia promienia ze sceną z wartością pobraną z przygotowanego bufora.

W przykładowej aplikacji zaimplementowano pierwszy sposób, choć w wersji jeszcze bardziej uproszczonej – sphere tracing nie tworzy klasycznego logarytmicznego bufora głębi, tylko zapisuje do rendertargetu odległość punktu na powierzchni od punktu widzenia. Tak przygotowanych danych nie można już oczywiście użyć jako hardware-owego bufora głębi przy rasteryzacji, jest on więc wykorzystywany podobnie jak opisano w metodzie drugiej – pixel shader pobiera odpowiednią wartość z tekstury i porównuje ją z odległością renderowanego piksela od punktu widzenia, aby określić zasłonięcie.

6.2 Cienie

Aby wizualizacja była bardziej spójna, częściowo integrujemy również rzucanie cienia między obiektami rasteryzowanymi i rysowanymi w sphere tracingu. W tym momencie mamy jedynie miękkie cienie (opisane wcześniej) rzucane w ramach sceny w sphere tracingu. Istnieje tu kilka możliwości rozszerzenia, my wykorzystamy jedną: wygenerujemy mapę cienia (ang. shadow map) sceny opisanej powierzchniami niejawnymi i użyjemy jej przy renderowaniu obiektów w procesie rasteryzacji.

W tym celu stworzona została uproszczona wersja shadera do sphere tracingu – znajduje ona punkt przecięcia promienia ze sceną, ale nie wykonuje obliczeń oświetlenia. Scena jest renderowana z punktu widzenia światła. Do rendertargetu, podobnie jak w przypadku zasłaniania, zapisywana jest odległość punktu od punktu widzenia (w tym przypadku pozycji światła). Dodatkowo, zamiast rzutowania perspektywicznego mamy rzutowanie ortogonalne, gdyż korzystamy ze światła kierunkowego – wpływa to na procedurę generowania promienia.

Gdy mapa cienia zostanie utworzona, reszta procesu jest identyczna jak w klasycznej metodzie map cienia – należy zrzutować rasteryzowany punkt do przestrzeni projekcji światła i pobrać odpowiednią wartość z mapy cienia, aby określić czy rysowana powierzchnia znajduje się w cieniu.

7. Prosta symulacja fizyczna

Funkcje odległości zapewniają informację, którą można wykorzystać w celu detekcji kolizji. Na potrzeby prezentacji zrealizowana została symulacja fizyczna bardzo prostych obiektów – sfer w scenie opisanej funkcjami odległości. Kule będą odbijać się od powierzchni niejawnych. Odbicia nie będą wpływać na powierzchnie niejawne (traktujemy ja jak obiekty nieruchome na potrzeby kolizji).

Zasada działania jest dość oczywista. W poszukiwaniu kolizji kuli ze sceną ewaluowana jest funkcja odległości w punkcie, w którym znajduje się środek kuli. Jeżeli wartość odległości jest mniejsza niż promień kuli, to znaczy, że mamy kolizję.

Potrzebujemy jeszcze wektora normalnego kolizji. Wystarczyłoby w tym celu sprawdzić wektor normalny powierzchni w punkcie kolizji, nie znamy jednak tego punktu Przybliżony wektor normalny kolizji uzyskujemy, znajdując wektor normalny do powierzchni w punkcie, w którym znajduje się kula. Okazuje się to być wystarczająco dobrym przybliżeniem.

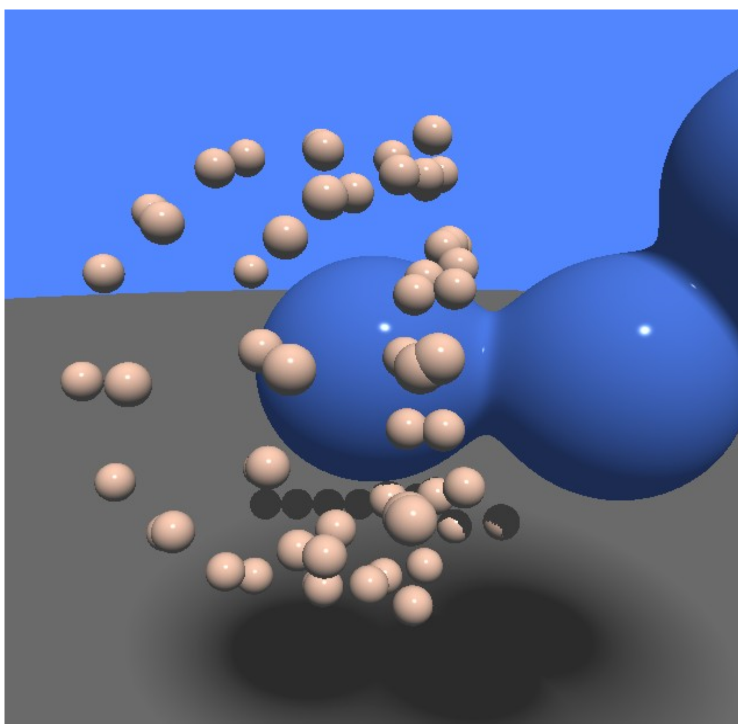
Kolizja jest rozwiązywana w dwóch etapach. Po pierwsze, ponieważ kulka „wpadła" w powierzchnię na pewną głębokość, pozycja kulki jest korygowana, poprzez przesunięcie jej o głębokość penetracji wzdłuż wektora normalnego kolizji. Po drugie, prędkość kulki jest „odbijana" względem

wektora normalnego kolizji. Krok pierwszy jej przydatny, gdyż zmiana prędkości nie zawsze wystarcza by prawidłowo rozwiązać kolizję, zwłaszcza w przypadku dużej penetracji lub tzw. kontaktów spoczynkowych.

Symulacja jest zrealizowana w całości na GPU. W tym celu utworzone zostają dwa zestawy tekstur, które w każdym pikselu będą przechowywać aktualny stan innej kulki. Dwa zestawy są potrzebne, gdyż przetwarzania będziemy dokonywać w pixel shaderze, gdzie niemożliwe jest odczytywanie i zapisywanie jednocześnie tej samej tekstury (bufory read-write z DirectX11 mogłyby tu pomóc). W jednej klatce zatem czytamy z bufora A i zapisujemy do bufora B, w kolejnej zaś czytamy z B i zapisujemy do A.

Na każdy zestaw składają się dwie tekstury w formacie R32G32B32A32. W pierwszej, w kanałach RGB każdego piksela przechowywana jest aktualna pozycja kulki. W drugiej, również w kanałach RGB, jej prędkość. Jeden kanał w każdej z tekstur pozostaje niewykorzystany.

Aby zatem wykonać krok symulacji, należy wywołać piksel shader, który zostanie wykonany dla każdego piksela, pobierze dwie wartości z tekstur z jednego zestawu, obliczy nowe, i zapisze je w tych samych pozycjach w teksturach z drugiego zestawu.



Rys. 7.1. Zrzut ekranu z aplikacji demonstracyjnej

8. Zakończenie

W artykule wprowadzono pojęcia powierzchni niejawnych, funkcji odległości i sphere tracingu. Pokazano też metodę integracji renderowania metodami sphere tracingu i rasteryzacji.

Autor jest zdania, że właściwości wizualne powierzchni niejawnych stanowią ciekawe narzędzie, które przez twórców gier nie zostało jeszcze chyba wykorzystane. Jednak wydajność i elastyczność takiego rozwiązania pozostawia jeszcze wiele do życzenia i jest tu wiele miejsca dla przyszłych innowacji.

Bibliografia

[1] Fedkiw R., Osher S., „Level Set Methods and Dynamic Implicit Surfaces", Wydawnictwo Springer, 2003
[2] Hart C.J., „Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces" 1994
[3] Quilez I., Rendering worlds with two triangles with raytracing on the GPU in 4096 bytes", http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf 2008

Sphere tracing: integration with classical methods of simulation and rendering

Abstract

Sphere tracing is a method of rendering distance fields used recently in demoscene productions. Modelling with distance fields has visual characteristics that computer games could take advantage of. It is not obvious though how to combine sphere tracing and distance fields with classical approaches used in games. The paper will present basics of these methods along with the application demonstrating: scene representation using distance fields, physical simulation of simple objects (spheres) in a scene, and rendering simultaneously with sphere tracing and rasterization.