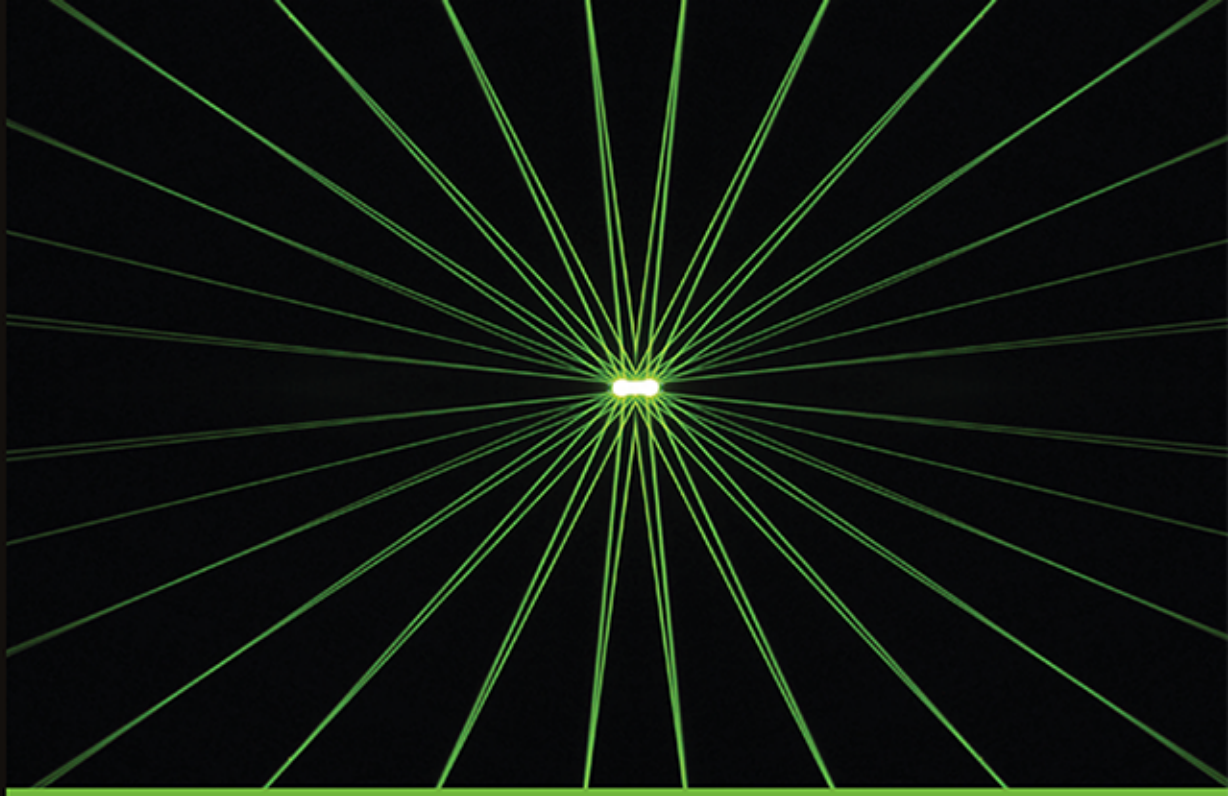


Technologia i rozwiązania

OpenGL Receptury dla programisty

Poznaj możliwości OpenGL!

Muhammad Mobeen Movania

Helion

[PACKT] PUBLISHING

Tytuł oryginału: OpenGL Development Cookbook

Tłumaczenie: Zbigniew Waśko

ISBN: 978-83-283-0018-7

First published in the English language under the title 'OpenGL Development Cookbook'

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/openrp.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/openrp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 7

Techniki renderingu wolumetrycznego bazujące na GPU

W tym rozdziale:

- Implementacja renderingu wolumetrycznego z cięciem tekstury 3D na płaty
- Implementacja renderingu wolumetrycznego z jednoprzebiegowym rzucaniem promieni
- Pseudoizopowierzchniowy rendering w jednoprzebiegowym rzucaniu promieni
- Rendering wolumetryczny z użyciem splattingu
- Implementacja funkcji przejścia dla klasyfikacji objętościowej
- Implementacja wydzielania wielokątnej izopowierzchni metodą maszerujących sześcianów
- Wolumetryczne oświetlenie oparte na technice cięcia połówkowokątowego

Wstęp

Techniki renderowania objętościowego znajdują wiele zastosowań w biomedycynie i inżynierii. W biomedycynie są używane do wizualizacji wyników tomografii komputerowej i rezonansu magnetycznego. W inżynierii służą do wizualizacji pośrednich etapów symulacji FEM, przepływów i analiz strukturalnych. Wraz z pojawieniem się procesorów graficznych wszystkie modele i metody wizualizacyjne zostały przeprojektowane pod kątem pełniejszego wykorzystania

mocy obliczeniowych tych procesorów. W tym rozdziale zaprezentuję kilka algorytmów wizualizacji wolumetrycznej, które można w taki właśnie sposób zrealizować za pomocą funkcji z biblioteki OpenGL w wersji 3.3 lub nowszej. W szczególności będą to trzy najbardziej rozpowszechnione metody polegające na cięciu trójwymiarowej tekstury, jednoprzebiegowym rzucaniu promieni z komponowaniem alfa i renderowaniu izopowierzchni.

Po zapoznaniu się z tymi podstawowymi metodami przyjrzymy się technice klasyfikacji objętościowej z odpowiednią funkcją przejścia. Do wydobywania klasyfikowanych obszarów, takich jak ścianki komórek, często stosuje się metody generowania izopowierzchni. Jedną z nich jest metoda maszerującego czworościanu (*marching tetrahedra*)[1]. Rendering wolumetryczny to także rozmaite techniki oświetlenia objętościowego. Jedną z popularnych technik jest tu cięcie połówkowokątowe i właśnie to spróbujemy zaimplementować.

Implementacja renderingu wolumetrycznego z cięciem tekstury 3D na płaty

Rendering wolumetryczny stanowi specyficzną odmianę algorytmów renderujących, które pozwalają na obrazowanie obiektów i zjawisk o strukturze przestrzennej, takich jak na przykład dym. Algorytmów takich jest wiele, ale nasz przegląd zaczniemy od metody najprostszej, znanej jako cięcie trójwymiarowej tekstury na płaty. Polega ona na aproksymowaniu funkcji opisującej przestrzenny rozkład gęstości przez rozcinanie zbioru danych na płaty w kierunku od przodu ku tyłowi lub od tyłu ku przodowi, a następnie sklejaniu tych płatów przez wspomagane sprzętowo mieszanie. Jako że wszystko to może być realizowane przez sprzęt rasteryzujący, szybkość działania tej metody jest bardzo duża.

Pseudokod cięcia trójwymiarowej tekstury na płaty prostopadłe do kierunku patrzenia przedstawia się następująco:

1. Wyznacz wektor kierunkowy bieżącego widoku.
2. Oblicz minimalną i maksymalną odległość wierzchołków jednostkowego sześcianu, mnożąc skalarnie każdy z tych wierzchołków przez wektor kierunkowy widoku.
3. Wyznacz wszystkie wartości parametru λ określającego możliwe przecięcia krawędzi jednostkowego sześcianu przez płaszczyznę prostopadłą do kierunku widoku, począwszy od wierzchołka najbliższego aż do najdalszego. Wykorzystaj do tego odległości minimalną i maksymalną z punktu 1.

[1] Autor posługuje się tutaj nazwą Marching Tetrahedra (maszerujące czworościany), ale tak naprawdę prezentuje algorytm o nazwie Marching Cubes (maszerujące sześciany) — *przyp. tłum.*

4. Posługując się parametrem λ (z punktu 3.), przesuwaj się zgodnie z kierunkiem widoku i znajdź punkty przecięcia. Powinno ich być od 3 do 6.
5. Zapisz położenia tych punktów we właściwej kolejności i wygeneruj na ich podstawie trójkąty jako zastępczą geometrię.
6. Do obiektu bufora wprowadź nowe wierzchołki.

Przygotowania

Pełny kod dla tej receptury znajdziesz w folderze *Rozdział7/CięcieTekstury3D*.

Jak to zrobić?

Zacznij od następujących czynności:

1. Wczytaj dane objętościowe z zewnętrznego pliku i umieść je w OpenGL-owej teksturze. Włącz też sprzętowe generowanie mipmap. Zazwyczaj dane objętościowe są zbiorem skanów wykonanych metodą rezonansu magnetycznego lub tomografii komputerowej. Każdy taki skan jest dwuwymiarowym płatem. Ułożone na stosie w kierunku osi `Z` tworzą trójwymiarową teksturę, którą można również traktować jak tablicę tekstur dwuwymiarowych. Zapisane w niej wartości określają gęstość prześwietlanej materii, np. gęstości z zakresu od 0 do 20 są typowe dla powietrza. Jeśli są to liczby 8-bitowe bez znaku, możemy je zapisać w tablicy typu `GLubyte`. Dane 16-bitowe bez znaku zapiszemy w tablicy typu `GLushort`. W przypadku tekstur 3D oprócz parametrów `S` i `T` mamy jeszcze parametr `R`, który określa bieżący płat tekstury.

```
std::ifstream infile(volume_file.c_str(), std::ios_base::binary);
if(infile.good()) {
   GLubyte* pData = new GLubyte[XDIM*YDIM*ZDIM];
   infile.read(reinterpret_cast<char*>(pData),
   ↪XDIM*YDIM*ZDIM*sizeof(GLubyte));
   infile.close();
   glGenTextures(1, &textureID);
   glBindTexture(GL_TEXTURE_3D, textureID);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
   ↪GL_LINEAR_MIPMAP_LINEAR);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_BASE_LEVEL, 0);
   glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAX_LEVEL, 4);
   glTexImage3D(GL_TEXTURE_3D,0,GL_RED,XDIM,YDIM,ZDIM,0,GL_RED,GL_
   ↪UNSIGNED_BYTE,pData);
```

```
        glGenerateMipmap(GL_TEXTURE_3D);
        return true;
    } else {
        return false;
    }
```

Parametry filtrowania dla tekstur 3D są podobne do tych, z jakimi mieliśmy do czynienia do tej pory. Mipmapy to zestaw odpowiednio przeskalowanych wersji tej samej tekstury, które stosuje się w zależności od wymaganego **poziomu szczegółowości** (*LOD — level of detail*). Gdy teksturowany obiekt znajduje się daleko od widza (kamery), wybierana jest wersja o odpowiednio małych wymiarach, dzięki czemu wzrasta szybkość działania aplikacji. Wartość `GL_TEXTURE_MAX_LEVEL` określa liczbę takich poziomów szczegółowości, a tym samym liczbę mipmap wygenerowanych z danej tekstury. Poziom podstawowy, czyli numer mipmapy stosowanej przy najmniejszej odległości obiektu od kamery, określa parametr `GL_TEXTURE_BASE_LEVEL`.

Funkcja `glGenerateMipMap` generuje pochodne tablice teksturowe przez redukujące filtrowanie poprzedniego poziomu. Przykładowo załóżmy, że mamy mieć trzy poziomy mipmap, a na poziomie 0 ma być tekstura 3D o wymiarach 256×256×256. Dla poziomu 1. trzeba więc wygenerować teksturę o wymiarach o połowę mniejszych, czyli 128×128×128. Dla poziomu 2. trzeba znów o połowę zmniejszyć wymiary tekstury z poziomu 1., czyli do wartości 64×64×64. Na poziomie 3. będzie tekstura zredukowana do wymiarów 32×32×32.

2. Przygotuj obiekty tablicy i bufora wierzchołków, w których zapiszesz geometrię zastępczych płatów. Upewnij się, że przeznaczenie obiektu bufora jest określone jako `GL_DYNAMIC_DRAW`. Pamięć GPU niezbędną do przechowania maksymalnej liczby płatów alokuje funkcja `glBufferData`. Tablica `vTextureSlices` jest zdefiniowana globalnie i w niej zapisane są wszystkie wierzchołki wyznaczone w procesie cięcia tekstury na płaty. Wartość zerowa wskaźnika do danych oznacza, że dane te będą wprowadzane do bufora dopiero podczas działania aplikacji.

```
const int MAX_SLICES = 512;
glm::vec3 vTextureSlices[MAX_SLICES*12];

glGenVertexArrays(1, &volumeVAO);
glGenBuffers(1, &volumeVBO);
glBindVertexArray(volumeVAO);
glBindBuffer (GL_ARRAY_BUFFER, volumeVBO);
glBufferData (GL_ARRAY_BUFFER, sizeof(vTextureSlices), 0,
↪GL_DYNAMIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,0,0);
glBindVertexArray(0);
```

3. Zaimplementuj cięcie badanego obszaru przestrzeni przez wyznaczanie przecięć jednostkowego sześcianu płatami prostopadłymi do kierunku patrzenia. W naszej aplikacji zadanie to wykonuje funkcja `SliceVolume`. Stosujemy sześcian jednostkowy,

ponieważ nasze dane zajmują obszar o takich samych wymiarach względem wszystkich trzech osi (256×256×256). Gdyby te wymiary nie były jednakowe, należałoby odpowiednio przeskalować sześcian jednostkowy.

```
//wyznacz odległości max i min
glm::vec3 vecStart[12];
glm::vec3 vecDir[12];
float lambda[12];
float lambda_inc[12];
float denom = 0;
float plane_dist = min_dist;
float plane_dist_inc = (max_dist-min_dist)/float(num_slices);

//wyznacz vecStart i vecDir
glm::vec3 intersection[6];
float dL[12];

for(int i=num_slices-1;i>=0;i--) {
   for(int e = 0; e < 12; e++)
      {
      dL[e] = lambda[e] + i*lambda_inc[e];
      }
   if ((dL[0] >= 0.0) && (dL[0] < 1.0)) {
      intersection[0] = vecStart[0] + dL[0]*vecDir[0];
   }
   //podobnie dla wszystkich punktów przecięcia
   int indices[]={0,1,2, 0,2,3, 0,3,4, 0,4,5};
   for(int i=0;i<12;i++)
   vTextureSlices[count++]=intersection[indices[i]];
}
//uaktualnij obiekt bufora
glBindBuffer(GL_ARRAY_BUFFER, volumeVBO);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vTextureSlices),
↪&(vTextureSlices[0].x));
```

4. W funkcji renderującej ustaw mieszanie nakładkowe, zwiąż obiekt tablicy wierzchołków, uruchom shader i wywołaj funkcję `glDrawArrays`.

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glBindVertexArray(volumeVAO);
shader.Use();
glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE, glm::value_ptr(MVP));
glDrawArrays(GL_TRIANGLES, 0, sizeof(vTextureSlices)/
↪sizeof(vTextureSlices[0]));
shader.UnUse();
glDisable(GL_BLEND);
```

Jak to działa?

Metoda cięcia tekstury 3D na płaty aproksymuje całkę renderingu wolumetrycznego przez mieszanie alfa poteksturowanych płatów. Pierwszy krok to wczytanie danych wolumetrycznych i umieszczenie ich w teksturze 3D. Potem następuje cięcie obszaru zajmowanego przez te dane na tymczasowe płaty ustawione prostopadle do kierunku patrzenia. W procesie tym wyznaczane są punkty przecięcia tymi płatami jednostkowego sześcianu. Zadanie to wykonuje funkcja `SliceVolume`. Działa ona tylko wtedy, gdy zmienia się kierunek patrzenia.

Najpierw wyznaczamy wektor kierunku patrzenia (`viewDir`), którego współrzędne stanowią trzecią kolumnę macierzy modelu i widoku. Pierwsza kolumna tej macierzy to wektor zwrócony w prawo, a kolumna druga to wektor zwrócony w górę. Zobaczmy teraz, jak dokładnie działa funkcja `SliceVolume`. Zaczyna od wyznaczenia maksymalnej i minimalnej odległości do wierzchołków sześcianu jednostkowego w kierunku patrzenia. W tym celu mnoży skalarnie położenie każdego z tych wierzchołków przez wektor kierunku patrzenia.

```
float max_dist = glm::dot(viewDir, vertexList[0]);
float min_dist = max_dist;
int max_index = 0;
int count = 0;
for(int i=1;i<8;i++) {
   float dist = glm::dot(viewDir, vertexList[i]);
   if(dist > max_dist) {
      max_dist = dist;
      max_index = i;
   }
   if(dist<min_dist)
      min_dist = dist;
}
int max_dim = FindAbsMax(viewDir);
min_dist -= EPSILON;
max_dist += EPSILON;
```

Są tylko trzy unikatowe ścieżki wiodące od wierzchołka najbliższego do najdalszego. Każdą z nich dla wszystkich wierzchołków umieszczamy w tablicy krawędzi zdefiniowanej w sposób następujący:

```
int edgeList[8][12]={{0,1,5,6, 4,8,11,9, 3,7,2,10 }, //v0 z przodu
   {0,4,3,11, 1,2,6,7, 5,9,8,10 }, // v1 z przodu
   {1,5,0,8, 2,3,7,4, 6,10,9,11}, // v2 z przodu
   { 7,11,10,8, 2,6,1,9, 3,0,4,5 }, // v3 z przodu
   { 8,5,9,1, 11,10,7,6, 4,3,0,2 }, // v4 z przodu
   { 9,6,10,2, 8,11,4,7, 5,0,1,3 }, // v5 z przodu
   { 9,8,5,4, 6,1,2,0, 10,7,11,3}, // v6 z przodu
   { 10,9,6,5, 7,2,3,1, 11,4,8,0 } // v7 z przodu
```

Następnie wyznaczane są odległości do punktów przecięcia płatów z każdą z 12 krawędzi sześcianu:

```
glm::vec3 vecStart[12];
glm::vec3 vecDir[12];
float lambda[12];
float lambda_inc[12];
float denom = 0;
float plane_dist = min_dist;
float plane_dist_inc = (max_dist-min_dist)/float(num_slices);
for(int i=0;i<12;i++) {
   vecStart[i]=vertexList[edges[edgeList[max_index][i]][0]];
   vecDir[i]=vertexList[edges[edgeList[max_index][i]][1]]-vecStart[i];
   denom = glm::dot(vecDir[i], viewDir);
   if (1.0 + denom != 1.0) {
      lambda_inc[i] = plane_dist_inc/denom;
      lambda[i]=(plane_dist-glm::dot(vecStart[i],viewDir))/denom;
   } else {
      lambda[i] = -1.0;
      lambda_inc[i] = 0.0;
   }
}
```

Na koniec przeprowadzana jest interpolacja punktów przecięć z krawędziami sześcianu, idąc od tyłu ku przodowi w kierunku wyznaczonym przez wektor widoku. Po wygenerowaniu płatów nowe dane są umieszczane w obiekcie bufora wierzchołków.

```
for(int i=num_slices-1;i>=0;i--) {
   for(int e = 0; e < 12; e++) {
      dL[e] = lambda[e] + i*lambda_inc[e];
   }
   if ((dL[0] >= 0.0) && (dL[0] < 1.0)) {
      intersection[0] = vecStart[0] + dL[0]*vecDir[0];
   } else if ((dL[1] >= 0.0) && (dL[1] < 1.0)) {
      intersection[0] = vecStart[1] + dL[1]*vecDir[1];
   } else if ((dL[3] >= 0.0) && (dL[3] < 1.0)) {
      intersection[0] = vecStart[3] + dL[3]*vecDir[3];
   } else continue;

   if ((dL[2] >= 0.0) && (dL[2] < 1.0)){
      intersection[1] = vecStart[2] + dL[2]*vecDir[2];
   } else if ((dL[0] >= 0.0) && (dL[0] < 1.0)){
      intersection[1] = vecStart[0] + dL[0]*vecDir[0];
   } else if ((dL[1] >= 0.0) && (dL[1] < 1.0)){
      intersection[1] = vecStart[1] + dL[1]*vecDir[1];
   } else {
      intersection[1] = vecStart[3] + dL[3]*vecDir[3];
   }
   //podobnie dla pozostałych krawędzi, aż do intersection[5]
   int indices[]={0,1,2, 0,2,3, 0,3,4, 0,4,5};
```

```
        for(int i=0;i<12;i++)
            vTextureSlices[count++]=intersection[indices[i]];
    }
    glBindBuffer(GL_ARRAY_BUFFER, volumeVBO);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vTextureSlices),
    ↪&(vTextureSlices[0].x));
```

W funkcji renderującej uruchamiamy odpowiedni program shaderowy. Shader wierzchołków wyznacza położenia wierzchołków w przestrzeni przycięcia, mnożąc ich położenia w przestrzeni obiektu (`vPosition`) przez połączoną macierz modelu, widoku i rzutowania (`MVP`). Oblicza też współrzędne tekstury 3D (`vUV`) dla danych wolumetrycznych. Stosujemy sześcian jednostkowy, a zatem najmniejsze współrzędne wierzchołka będą wynosiły (`-0.5`, `-0.5`, `-0.5`), a największe — (`0.5`, `0.5`, `0.5`). Żeby można było ich użyć jako współrzędnych tekstury 3D, trzeba je przesunąć do przedziału od (`0`, `0`, `0`) do (`1`, `1`, `1`), a więc trzeba je zwiększyć o (`0.5`, `0.5`, `0.5`).

```
smooth out vec3 vUV;
void main() {
    gl_Position = MVP*vec4(vVertex.xyz,1);
    vUV = vVertex + vec3(0.5);
}
```

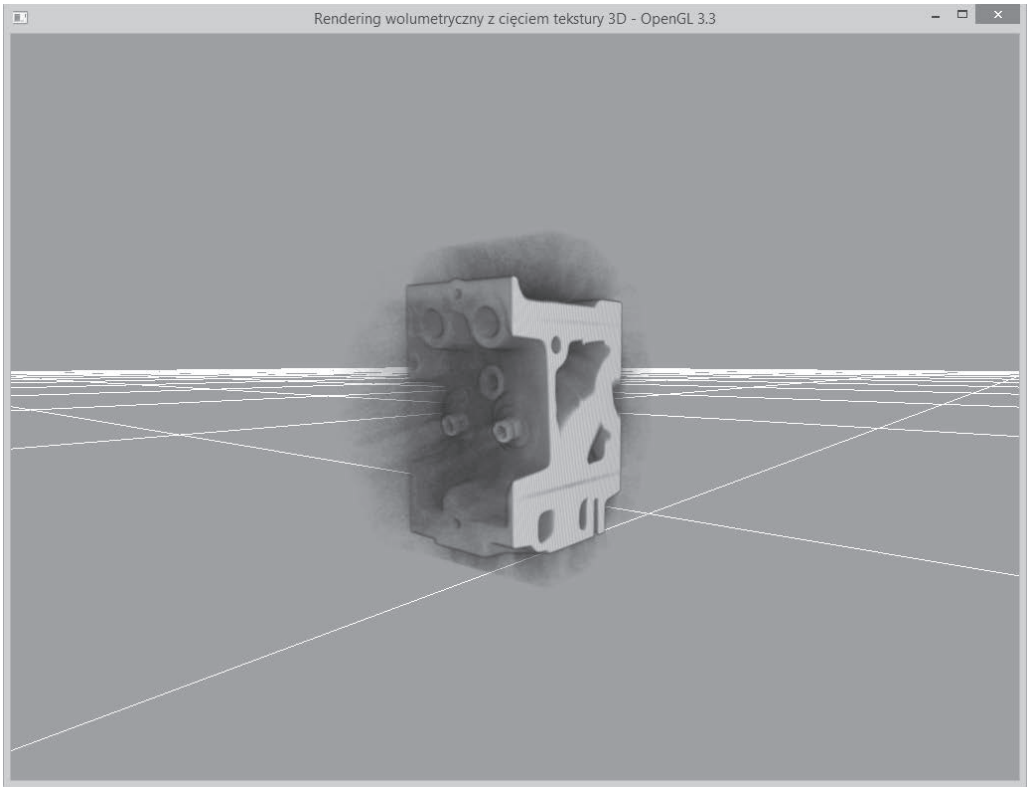
Shader fragmentów używa tych współrzędnych do próbkowania danych wolumetrycznych (dostępnych teraz poprzez nowy typ samplera dla tekstur trójwymiarowych `sampler3D`) w celu określenia koloru fragmentu na podstawie odczytanej gęstości. Podczas tworzenia tekstury 3D określiliśmy jej wewnętrzny format jako `GL_RED` (trzeci parametr funkcji `glTexImage3D`), a zatem teraz możemy pobierać gęstość z czerwonego kanału samplera tekstury. Aby uzyskać odcień szarości, ustawiamy taką samą wartość w pozostałych kanałach koloru.

```
smooth in vec3 vUV;
uniform sampler3D volume;
void main(void) {
    vFragColor = texture(volume, vUV).rrrr;
}
```

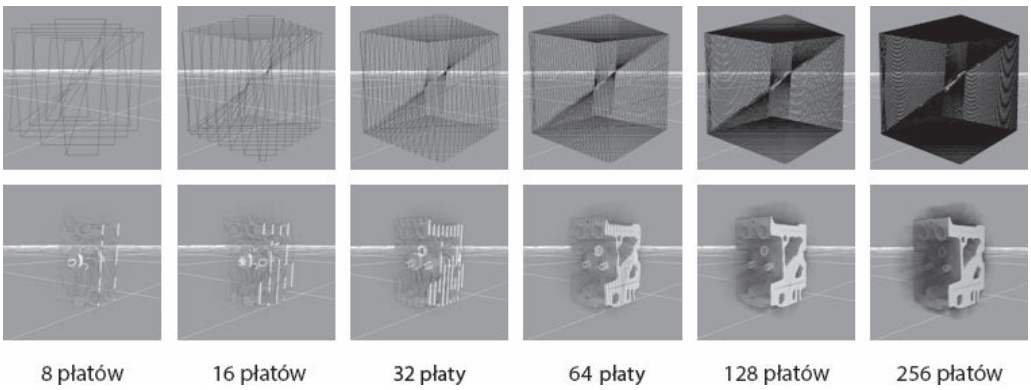
We wcześniejszych wersjach OpenGL zapisalibyśmy gęstości wolumetryczne w specjalnie do tego przeznaczonym formacie `GL_INTENSITY`. Niestety został on usunięty z rdzennego profilu biblioteki w wersji 3.3 i musimy posługiwać się formatami `GL_RED`, `GL_GREEN`, `GL_BLUE` lub `GL_ALPHA`.

I jeszcze jedno?

Przykładowa aplikacja zbudowana na podstawie powyższej receptury wizualizuje dane wolumetryczne fragmentu silnika. Za pomocą klawiszy + (plus) i – (minus) można zmieniać liczbę płatów.



Zobaczmy teraz, jak powstaje taki obraz, wyświetlając coraz większą liczbę płatów tekstury 3D, począwszy od **8** aż po pełne **256**. Rezultaty widać na poniższym rysunku. W górnym rzędzie pokazane są widoki konturowe płatów, a u dołu — rezultaty ich mieszania.



Jak łatwo zauważyć, zwiększanie liczby płatów poprawia wygląd renderowanego obrazu. Jednak po przekroczeniu wartości 256 nie widać już znaczącej poprawy, a powyżej 350 zaczyna być zauważalne spowolnienie działania aplikacji. Przyczyną jest konieczność przesyłania do GPU coraz większych ilości geometrii.

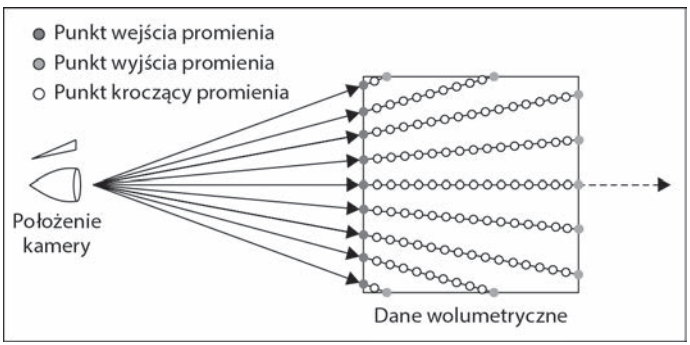
Zwróć uwagę na czarną chmurę otaczającą obiekt wolumetryczny. Jej obecność jest wynikiem błędów, jakie wystąpiły w trakcie rejestrowania danych (np. szum aparatury rejestrującej lub zanieczyszczenie powietrza wokół skanowanego obiektu). Artefakty tego typu można usunąć bądź to przez zastosowanie odpowiedniej funkcji przejścia, bądź przez wyeliminowanie ich w shaderze fragmentów, co zrobimy później w recepturze „Wolumetryczne oświetlenie oparte na technice cięcia połówkowokątowego".

Dowiedz się więcej

- *Real-Time Volume Graphics*, A K Peters/CRC Press, rozdział 3. „GPU-based Volume Rendering", punkt 3.5.2 „Viewport-Aligned Slices", s. 73 – 79.

Implementacja renderingu wolumetrycznego z jednoprzebiegowym rzucaniem promieni

W tej recepturze pokażę, jak można zaimplementować na GPU rendering wolumetryczny z jednoprzebiegowym rzucaniem promieni. Ogólnie rzucanie promieni może być wieloprzebiegowe lub jednoprzebiegowe. Podejścia te różnią się sposobem ustalania kierunków kroczących promieni. Podejście jednoprzebiegowe korzysta z jednego shadera fragmentów, którego zasadę działania najlepiej wyjaśni poniższy rysunek.



Najpierw wyznaczamy kierunek promienia wysyłanego z kamery. W tym celu odejmujemy położenie kamery od położenia wierzchołka wolumetrycznego. Początkowym położeniem kroczącego

promienia (punktem wejścia) jest położenie wierzchołka. Następnie przesuwamy promień wzdłuż wyznaczonego kierunku o ustalony krok. W każdym punkcie pobieramy próbkę danych wolumetrycznych. Wędrówkę promienia kończymy, gdy ten wyjdzie poza obszar danych lub kumulowany kolor fragmentu stanie się kompletnie nieprzezroczysty.

Próbki zebrane przez wędrujący promień łączymy ze sobą według określonego przepisu. Jeśli stosujemy uśrednianie, to po prostu sumujemy wszystkie próbki i wynik dzielimy przez ich liczbę. Jeśli zaś stosujemy mieszanie alfa w kolejności od przodu do tyłu, to mnożymy pobraną próbkę przez składową alfa zakumulowanego koloru i wynik odejmujemy od pobranej próbki. To daje nam składową alfa z poprzednich kroków. Wartość tę mnożymy przez pobraną próbkę i wynik dodajemy do zakumulowanego koloru, a na koniec dodajemy ją do składowej alfa zakumulowanego koloru. Ostatecznie jako kolor fragmentu wyprowadzamy kolor zakumulowany.

Przygotowania

Pełny kod dla tej receptury znajdziesz w folderze *Rozdział7/RzucaniePromieni*.

Jak to zrobić?

Aby zaimplementować na GPU jednoprzebiegowe rzucanie promieni, wykonaj następujące czynności:

1. Podobnie jak w poprzedniej recepturze wczytaj dane wolumetryczne do trójwymiarowej tekstury OpenGL-owej. Dodatkowe objaśnienia tego fragmentu implementacji znajdziesz w definicji funkcji `LoadVolume` podanej w pliku *Rozdział7/RzucaniePromieni/main.cpp*.
2. Przygotuj obiekty tablicy i bufora wierzchołków potrzebne do wyrenderowania jednostkowego sześcianu. Zrób to w sposób następujący:

```
glGenVertexArrays(1, &cubeVAOID);
glGenBuffers(1, &cubeVBOID);
glGenBuffers(1, &cubeIndicesID);
glm::vec3 vertices[8]={  glm::vec3(-0.5f,-0.5f,-0.5f),
↪glm::vec3( 0.5f,-0.5f,-0.5f),glm::vec3( 0.5f, 0.5f,-0.5f),
↪glm::vec3(-0.5f, 0.5f,-0.5f),glm::vec3(-0.5f,-0.5f, 0.5f),
↪glm::vec3( 0.5f,-0.5f, 0.5f),glm::vec3( 0.5f, 0.5f, 0.5f),
↪glm::vec3(-0.5f, 0.5f, 0.5f)};

GLushort cubeIndices[36]={0,5,4,5,0,1,3,7,6,3,6,2,7,4,6,6,4,5,2,1,3,3,1,
↪0,3,0,7,7,0,4,6,5,2,2,5,1};

glBindVertexArray(cubeVAOID);
glBindBuffer (GL_ARRAY_BUFFER, cubeVBOID);
glBufferData (GL_ARRAY_BUFFER, sizeof(vertices), &(vertices[0].x),
↪GL_STATIC_DRAW);
```

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,0,0);

glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, cubeIndicesID);
glBufferData (GL_ELEMENT_ARRAY_BUFFER, sizeof(cubeIndices),
&cubeIndices[0], GL_STATIC_DRAW);
glBindVertexArray(0);
```

3. W funkcji renderującej uaktywnij program shaderowy rzucania promieni (*Rozdział7/RzucaniePromieni/shadery/raycaster.*[*vert*, *frag*]) i wyrenderuj sześcian jednostkowy.

```
glEnable(GL_BLEND);
glBindVertexArray(cubeVAOID);
shader.Use();
glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE, glm::value_ptr(MVP));
glUniform3fv(shader("camPos"), 1, &(camPos.x));
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
shader.UnUse();
glDisable(GL_BLEND);
```

4. Z shadera wierzchołków wyprowadź, poza położeniem wierzchołka w przestrzeni przycięcia, współrzędne trójwymiarowej tekstury potrzebne do jej próbkowania w shaderze fragmentów. Aby uzyskać te współrzędne, po prostu przesuń współrzędne wierzchołka z przestrzeni obiektu o wektor (`0.5`, `0.5`, `0.5`).

```
smooth out vec3 vUV;
void main()
{
   gl_Position = MVP*vec4(vVertex.xyz,1);
   vUV = vVertex + vec3(0.5);
}
```

5. W shaderze fragmentów utwórz pętlę przesuwającą promień wzdłuż kierunku wyznaczonego na podstawie położenia kamery i początkowego wierzchołka wolumetrycznego. Zatrzymaj wykonywanie pętli, gdy promień wyjdzie poza obszar danych wolumetrycznych lub zakumulowany kolor stanie się całkowicie nieprzezroczysty.

```
vec3 dataPos = vUV;
vec3 geomDir = normalize((vUV-vec3(0.5)) - camPos);
vec3 dirStep = geomDir * step_size;
bool stop = false;
for (int i = 0; i < MAX_SAMPLES; i++) {
   // przesuń promień o jeden krok
   dataPos = dataPos + dirStep;
   // warunek zakończenia
   stop=dot(sign(dataPos-texMin),sign(texMax-dataPos)) < 3.0;
   if (stop)
      break;
```

6. Skomponuj pobraną próbkę z istniejącym już kolorem i zwróć wypadkową wartość jako kolor fragmentu.

```
    float sample = texture(volume, dataPos).r;

    float prev_alpha = sample - (sample * vFragColor.a);
    vFragColor.rgb = prev_alpha * vec3(sample) + vFragColor.rgb;
    vFragColor.a += prev_alpha;
    //wcześniejsze zakończenie pętli
    if( vFragColor.a>0.99)
       break;
}
```

Jak to działa?

Receptura składa się dwóch zasadniczych części. W pierwszej generujemy i renderujemy geometrię sześcianu, w którym ma działać shader fragmentów. Moglibyśmy użyć pełnoekranowego czworokąta, tak jak to robiliśmy przy implementowaniu wieloprzebiegowego śledzenia promieni, ale dla renderingu wolumetrycznego korzystniejsze jest zastosowanie jednostkowego sześcianu. Część druga odbywa się w shaderach.

W shaderze wierzchołków (*Rozdział7/RzucaniePromieni/shadery/raycaster.vert*) wyznaczane są współrzędne trójwymiarowej tekstury na podstawie położeń wierzchołków sześcianu jednostkowego. Ponieważ sześcian jest położony w środku układu współrzędnych, dodajemy do każdego wierzchołka wektor `vec(0.5)`, aby uzyskać współrzędne tekstury w zakresie od `0` do `1`.

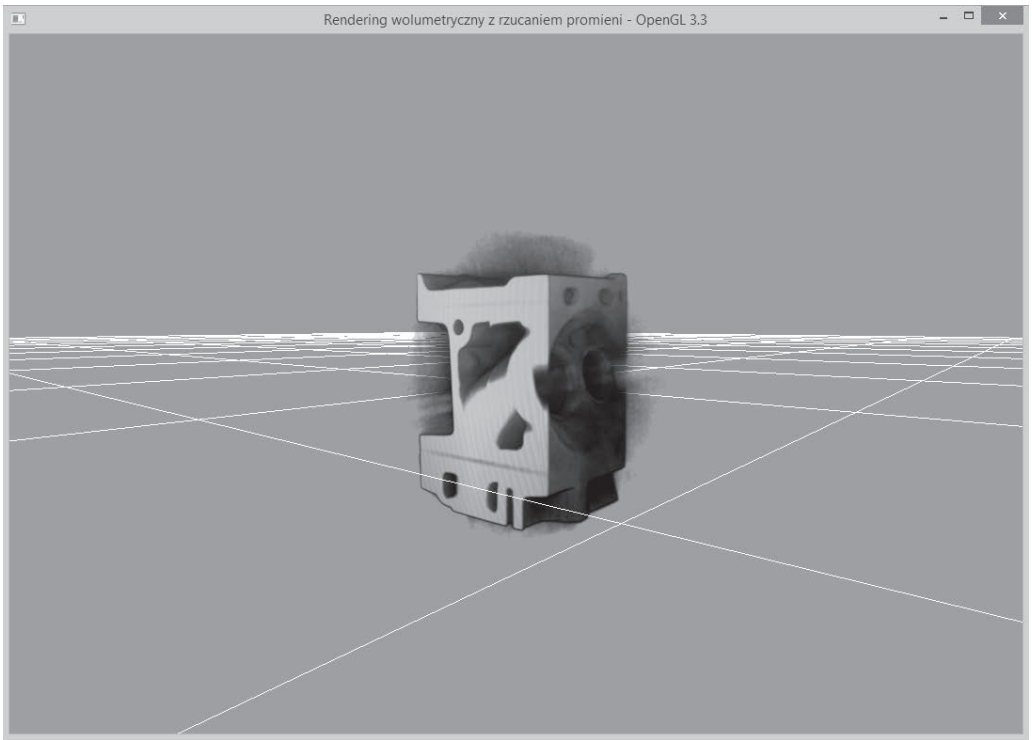
```
#version 330 core
layout(location = 0) in vec3 vVertex;
uniform mat4 MVP;
smooth out vec3 vUV;
void main() {
   gl_Position = MVP*vec4(vVertex.xyz,1);
   vUV = vVertex + vec3(0.5);
}
```

Potem shader fragmentów na podstawie współrzędnych trójwymiarowej tekstury i współrzędnych kamery wyznacza kierunki kroczących promieni. Pętla (pokazana w punkcie 5.) przesuwa promień wzdłuż ustalonego kierunku, pobiera próbki danych wolumetrycznych i zgodnie z wybranym schematem komponuje z nich wypadkowy kolor fragmentu. Proces ten trwa, dopóki promień nie opuści obszaru wolumetrycznego lub składowa alfa zakumulowanego koloru nie osiągnie pełnej swojej wartości.

Stałe `texMin` i `texMax` mają wartości, odpowiednio, `vec3(-1,-1,-1)` i `vec3(1,1,1)`. Aby określić, czy promień opuścił obszar danych, używamy funkcji `sign`, która zwraca `-1`, jeśli jej argument ma wartość mniejszą od zera, `0`, jeśli jest on równy zero, i `1`, jeśli jest większy od zera. Zatem dla położeń skrajnych wywołania tej funkcji w postaci `sign(dataPos-texMin)` i `sign(texMax-dataPos)` dadzą `vec3(1,1,1)`. Jeśli wymnożymy skalarnie dwa takie wektory, otrzymamy wartość `3`. A zatem, jeśli promień będzie w obszarze danych, iloczyn skalarny da wartość mniejszą niż `3`. Jeśli wyjdzie więcej, będzie to oznaczało, że promień jest już poza obszarem danych.

I jeszcze jedno…

Przykładowa aplikacja renderuje dane wolumetryczne fragmentu silnika, wykorzystując metodę jednoprzebiegowego rzucania promieni. Położenie kamery można zmieniać przez przeciąganie myszą z wciśniętym lewym przyciskiem, a przeciąganie z wciśniętym przyciskiem środkowym powoduje przybliżanie lub oddalanie widoku.



Dowiedz się więcej

Zapoznaj się z następującymi publikacjami:

- *Real-Time Volume Graphics*, A K Peters/CRC Press, rozdział 7. „GPU-based Ray Casting", s. 163 – 184.
- *Single-Pass Raycasting* w serwisie The Little Grasshopper, *http://prideout.net/blog/?p=64*.

Pseudoizopowierzchniowy rendering w jednoprzebiegowym rzucaniu promieni

Teraz zaimplementujemy renderowanie pseudoizopowierzchni w jednoprzebiegowym rzucaniu promieni. Większość kodu będzie taka sama jak w poprzedniej recepturze, a jedyna różnica będzie polegała na zastosowaniu innego schematu komponowania próbek w shaderze fragmentów rzucającym promienie. Będzie on próbował znaleźć określoną izopowierzchnię i jeśli ją znajdzie, wyznaczy dla niej normalną w punkcie próbkowania, a następnie przeprowadzi obliczenia oświetleniowe dla tego punktu.

Rozwiązanie to zapisane w pseudokodzie wygląda następująco:

```
Wyznacz kierunek patrzenia kamery i początkowe położenie promienia
Określ długość promienia
Dla każdej próbki na drodze promienia
   Pobierz pierwszą próbkę danych (sample1) z bieżącego położenia promienia
   Pobierz drugą próbkę (sample2) z następnego położenia promienia
   Jeśli (sample1-isoValue) < 0 i (sample2-isoValue) > 0
      Uściślij położenie punktu przecięcia, stosując metodę bisekcji
      Wyznacz gradient w punkcie przecięcia
      Zastosuj cieniowanie Phonga w punkcie przecięcia
      Przypisz fragmentowi bieżący kolor
      Przerwij
   Koniec warunku
Koniec pętli
```

Przygotowania

Pełny kod dla tej receptury znajdziesz w folderze *Rozdział7/Izoppowierzchnia*. Samodzielne tworzenie możesz rozpocząć od skopiowania kodu poprzedniej receptury — z jednoprzebiegowym rzucaniem promieni.

Jak to zrobić?

Zacznij od następujących prostych czynności:

1. Podobnie jak w poprzedniej recepturze wczytaj dane wolumetryczne do trójwymiarowej tekstury OpenGL-owej. Dodatkowe objaśnienia znajdziesz w definicji funkcji `LoadVolume` podanej w pliku *Rozdział7/Izopowierzchnia/main.cpp*.
2. Przygotuj obiekty tablicy i bufora wierzchołków potrzebne do wyrenderowania jednostkowego sześcianu — tak jak poprzednio.
3. W funkcji renderującej uaktywnij program shaderowy rzucania promieni (*Rozdział7/Izopowierzchnia/shadery/raycaster.*[*vert*, *frag*]) i wyrenderuj sześcian jednostkowy.

```
glEnable(GL_BLEND);
glBindVertexArray(cubeVAOID);
shader.Use();
glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE, glm::value_ptr(MVP));
glUniform3fv(shader("camPos"), 1, &(camPos.x));
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
shader.UnUse();
glDisable(GL_BLEND);
```

4. Z shadera wierzchołków wyprowadź, poza położeniem wierzchołka w przestrzeni przycięcia, współrzędne trójwymiarowej tekstury potrzebne do jej próbkowania w shaderze fragmentów. Aby uzyskać te współrzędne, po prostu przesuń obiektowe współrzędne wierzchołka w sposób następujący:

```
smooth out vec3 vUV;
void main()
{
   gl_Position = MVP*vec4(vVertex.xyz,1);
   vUV = vVertex + vec3(0.5);
}
```

5. W shaderze fragmentów utwórz pętlę przesuwającą promień wzdłuż kierunku wyznaczonego na podstawie położenia kamery i początkowego wierzchołka wolumetrycznego. Zatrzymaj wykonywanie pętli, gdy promień wyjdzie poza obszar danych lub zakumulowany kolor stanie się całkowicie nieprzezroczysty.

```
vec3 dataPos = vUV;
vec3 geomDir = normalize((vUV-vec3(0.5)) - camPos);
vec3 dirStep = geomDir * step_size;
bool stop = false;
for (int i = 0; i < MAX_SAMPLES; i++) {
   // przesuń promień o jeden krok
   dataPos = dataPos + dirStep;
   // warunek zakończenia
   stop=dot(sign(dataPos-texMin),sign(texMax-dataPos)) < 3.0;
   if (stop)
      break;
```

6. W celu wyznaczenia izopowierzchni bierz po dwie próbki i sprawdzaj, czy przy przejściu od jednej do drugiej promień przeciął izopowierzchnię. Gdy coś takiego stwierdzisz, ustal dokładne miejsce przecięcia, stosując metodę bisekcji. Na koniec zastosuj na izopowierzchni cieniowanie Phonga, przyjmując, że źródło światła znajduje się tam, gdzie kamera.

```
float sample=texture(volume, dataPos).r;
float sample2=texture(volume, dataPos+dirStep).r;
if( (sample -isoValue) < 0 && (sample2-isoValue) >= 0.0)
{
   vec3 xN = dataPos;
   vec3 xF = dataPos+dirStep;
   vec3 tc = Bisection(xN, xF, isoValue);
```

```
            vec3 N = GetGradient(tc);
            vec3 V = -geomDir;
            vec3 L = V;
            vFragColor = PhongLighting(L,N,V,250, vec3(0.5));
            break;
         }
      }
```

Funkcję `Bisection` zdefiniuj następująco:

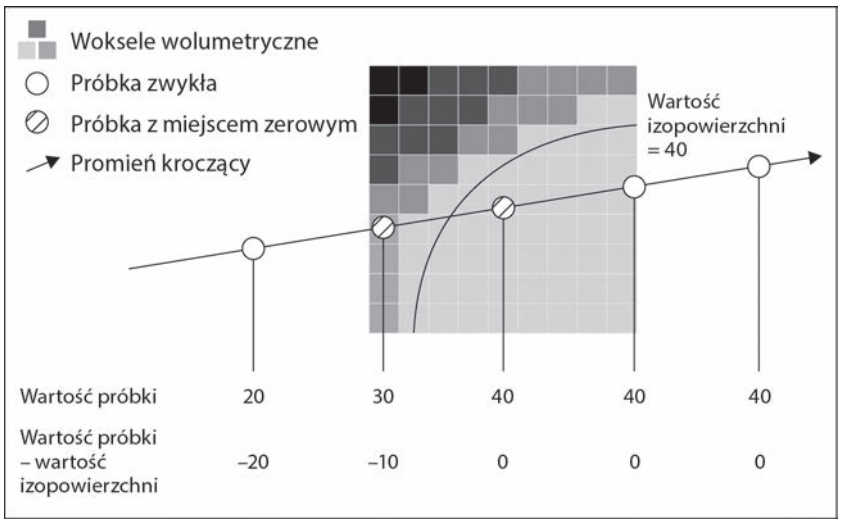
```
vec3 Bisection(vec3 left, vec3 right , float iso) {
   for(int i=0;i<4;i++) {
      vec3 midpoint = (right + left) * 0.5;
      float cM = texture(volume, midpoint).x ;
      if(cM < iso)
         left = midpoint;
      else
         right = midpoint;
   }
   return vec3(right + left) * 0.5;
}
```

Funkcja ta bierze dwie próbki, między którymi leży zadana wartość, i stara się wyznaczyć jej dokładne położenie. W tym celu uruchamia pętlę, w której cyklicznie wyznacza punkt środkowy między próbkami i porównuje jego wartość wolumetryczną z zadaną wartością izopowierzchni. Jeśli wartość w punkcie środkowym jest mniejsza od wartości izopowierzchni, punkt środkowy zastępuje lewą próbkę. Gdy jest inaczej, zastępuje próbkę prawą. W ten sposób szybko zawęża się obszar poszukiwań. Po wykonaniu określonej liczby takich operacji zwracane jest położenie punktu środkowego. Funkcja `Gradient` wyznacza gradient wartości wolumetrycznych metodą skończonych różnic centralnych.

```
vec3 GetGradient(vec3 uvw)
{
   vec3 s1, s2;
   //wyznaczanie centralnego ilorazu różnicowego
   s1.x = texture(volume, uvw-vec3(DELTA,0.0,0.0)).x ;
   s2.x = texture(volume, uvw+vec3(DELTA,0.0,0.0)).x ;

   s1.y = texture(volume, uvw-vec3(0.0,DELTA,0.0)).x ;
   s2.y = texture(volume, uvw+vec3(0.0,DELTA,0.0)).x ;

   s1.z = texture(volume, uvw-vec3(0.0,0.0,DELTA)).x ;
   s2.z = texture(volume, uvw+vec3(0.0,0.0,DELTA)).x ;

   return normalize((s1-s2)/2.0);
}
```

Jak to działa?

Większość kodu jest podobna do tego, który napisaliśmy w recepturze z jednoprzebiegowym rzucaniem promieni. Różnica pojawia się dopiero w pętli realizującej ruch promienia poprzez obszar wolumetryczny. Tym razem nie stosujemy żadnego komponowania koloru, lecz wyznaczamy miejsca zerowe funkcji opisującej izopowierzchnię przez sprawdzanie próbek z dwóch kolejnych kroków. Dobrze ilustruje to poniższy rysunek. Jeśli między badanymi próbkami jest miejsce zerowe, precyzujemy jego położenie, stosując metodę bisekcji.



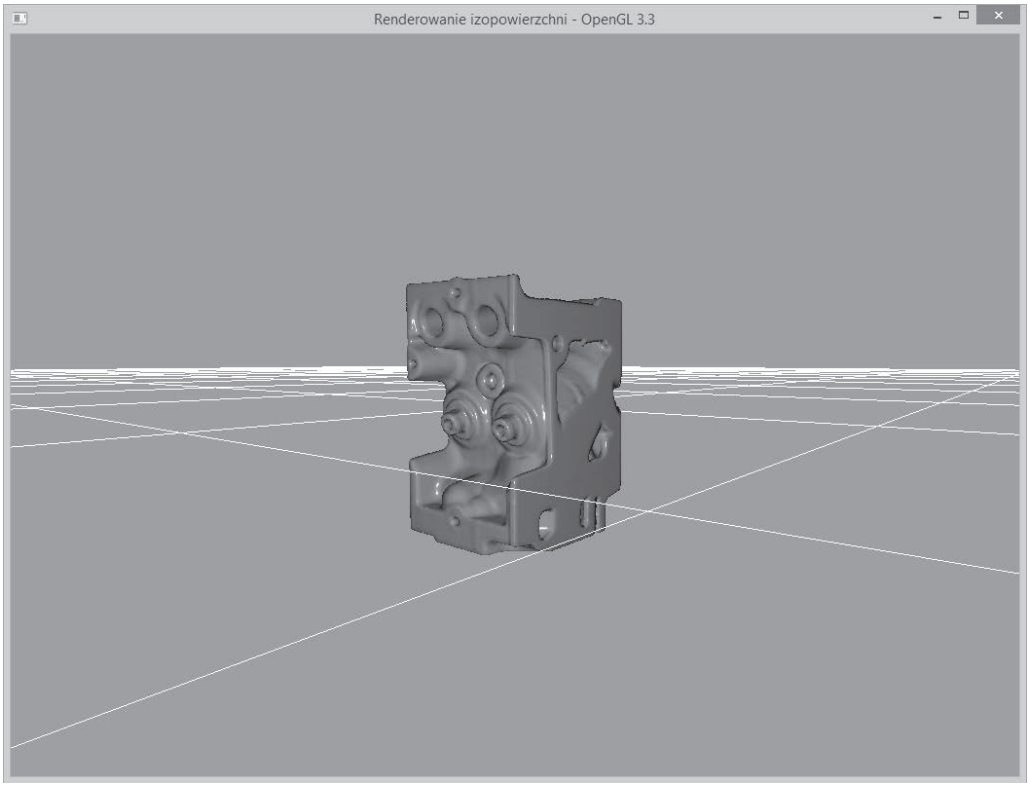
Następnie renderujemy izopowierzchnię, stosując model oświetleniowy Phonga, i opuszczamy pętlę maszerującego promienia. W ten sposób wyrenderujemy izopowierzchnię położoną najbliżej kamery. Gdybyśmy chcieli wyrenderować wszystkie izopowierzchnie o zadanej wartości, musielibyśmy usunąć instrukcję przerywającą wykonywanie pętli.

I jeszcze jedno…

Przykładowa aplikacja stanowiąca implementację powyższej receptury renderuje dane wolumetryczne zeskanowanego fragmentu silnika. Po uruchomieniu wyświetla obraz pokazany na rysunku na następnej stronie.

Dowiedz się więcej

- *Advanced Illumination Techniques for GPU-based Volume Rendering*, notatki z konferencji SIGGRAPH 2008, dostępne pod adresem *http://www.voreen.org/files/sa08-coursenotes_1.pdf*.



Rendering wolumetryczny z użyciem splattingu

Tym razem zaimplementujemy technikę zwaną splattingiem. Jej algorytm sprowadza się do zamiany wokseli na placki (*splats*) przez splatanie ich z jądrem filtra gaussowskiego. Gaussowskie jądro wygładzające usuwa wyższe częstotliwości i wygładza krawędzie, przez co wyrenderowany obraz wygląda na lepiej dopracowany.

Przygotowania

Gotowy kod receptury znajduje się w folderze *Chapter7/Splatting*.

Jak to zrobić?

Zacznij od następujących prostych czynności:

1. Wczytaj dane wolumetryczne i umieść je w tablicy.

```
std::ifstream infile(filename.c_str(), std::ios_base::binary);
if(infile.good()) {
   pVolume = new GLubyte[XDIM*YDIM*ZDIM];
   infile.read(reinterpret_cast<char*>(pVolume),
XDIM*YDIM*ZDIM*sizeof(GLubyte));
   infile.close();
   return true;
} else {
   return false;
}
```

2. Utwórz trzy pętle, które będą przebiegały przez całą objętość danych wolumetrycznych, woksel po wokselu.

```
vertices.clear();
int dx = XDIM/X_SAMPLING_DIST;
int dy = YDIM/Y_SAMPLING_DIST;
int dz = ZDIM/Z_SAMPLING_DIST;
scale = glm::vec3(dx,dy,dz);
for(int z=0;z<ZDIM;z+=dz) {
   for(int y=0;y<YDIM;y+=dy) {
      for(int x=0;x<XDIM;x+=dx) {
         SampleVoxel(x,y,z);
      }
   }
}
```

   Funkcja `SampleVoxel` jest zdefiniowana w klasie `VolumeSplatter` następująco:

```
void VolumeSplatter::SampleVoxel(const int x, const int y, const int z) {
   GLubyte data = SampleVolume(x, y, z);
   if(data>isoValue) {
      Vertex v;
      v.pos.x = x;
      v.pos.y = y;
      v.pos.z = z;
      v.normal = GetNormal(x, y, z);
      v.pos *= invDim;
      vertices.push_back(v);
   }
}
```

3. W każdym kroku pobierz próbkę wartości wolumetrycznych z bieżącego woksela. Jeśli pobrana wartość jest większa niż wartość określająca izopowierzchnię, zapisz położenie woksela i jego normalną w tablicy wierzchołków.

```
GLubyte data = SampleVolume(x, y, z);
if(data>isoValue) {
   Vertex v;
   v.pos.x = x;
   v.pos.y = y;
   v.pos.z = z;
   v.normal = GetNormal(x, y, z);
   v.pos *= invDim;
   vertices.push_back(v);
}
```

Funkcja `SampleVolume` bierze współrzędne wskazanego punktu i zwraca najbliższą wartość wolumetryczną. Jest ona zdefiniowana w klasie `VolumeSplatter` w sposób następujący:

```
GLubyte VolumeSplatter::SampleVolume(const int x, const int y, const int
↪z) {
   int index = (x+(y*XDIM)) + z*(XDIM*YDIM);
   if(index<0)
      index = 0;
   if(index >= XDIM*YDIM*ZDIM)
      index = (XDIM*YDIM*ZDIM)-1;
   return pVolume[index];
}
```

4. Po pobraniu próbek przekaż wygenerowane wierzchołki do **obiektu tablicy wierzchołków** (**VAO**) zawierającego **obiekt bufora wierzchołków** (**VBO**).

```
glGenVertexArrays(1, &volumeSplatterVAO);
glGenBuffers(1, &volumeSplatterVBO);
glBindVertexArray(volumeSplatterVAO);
glBindBuffer (GL_ARRAY_BUFFER, volumeSplatterVBO);
glBufferData (GL_ARRAY_BUFFER, splatter-
>GetTotalVertices()*sizeof(Vertex), splatter->GetVertexPointer(),
↪GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,sizeof(Vertex), 0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,sizeof(Vertex), (const
↪GLvoid*) offsetof(Vertex, normal));
```

5. Przygotuj dwa FBO dla renderingu pozaekranowego. Pierwszego z nich (`filterFBOID`) użyj do wygładzania gaussowskiego.

```
glGenFramebuffers(1,&filterFBOID);
glBindFramebuffer(GL_FRAMEBUFFER,filterFBOID);
glGenTextures(2, blurTexID);
for(int i=0;i<2;i++) {
   glActiveTexture(GL_TEXTURE1+i);
   glBindTexture(GL_TEXTURE_2D, blurTexID[i]);
   //ustaw parametry tekstury
```

```
   glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA32F,IMAGE_WIDTH,IMAGE_HEIGHT,0,
   ↪GL_RGBA,GL_FLOAT,NULL);
   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0+
   ↪i,GL_TEXTURE_2D,blurTexID[i],0);
}
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE) {
   cout<<"Ustawienie filtrujacego FBO powiodlo sie."<<endl;
} else {
   cout<<"Problem z ustawieniem filtrujacego FBO."<<endl;
}
```

6. Drugiego FBO (`fboID`) użyj do wyrenderowania sceny, która będzie wygładzana po pierwszym przebiegu. Dodaj do tego FBO obiekt bufora renderingu, aby umożliwić testowanie głębi.

```
glGenFramebuffers(1,&fboID);
glGenRenderbuffers(1, &rboID);
glGenTextures(1, &texID);
glBindFramebuffer(GL_FRAMEBUFFER,fboID);
glBindRenderbuffer(GL_RENDERBUFFER, rboID);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texID);
//ustaw parametry tekstury
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, IMAGE_WIDTH, IMAGE_HEIGHT, 0,
↪GL_RGBA, GL_FLOAT, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
↪GL_TEXTURE_2D, texID, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
↪GL_RENDERBUFFER, rboID);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT32, IMAGE_WIDTH,
↪IMAGE_HEIGHT);
status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE) {
   cout<<" Ustawienie pozaekranowego FBO powiodlo sie."<<endl;
} else {
   cout<<" Problem z ustawieniem pozaekranowego FBO."<<endl;
}
```

7. W funkcji renderującej najpierw wyrenderuj do tekstury punktowe placki. Użyj do tego celu drugiego FBO (`fboID`).

```
glBindFramebuffer(GL_FRAMEBUFFER,fboID);
glViewport(0,0, IMAGE_WIDTH, IMAGE_HEIGHT);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glm::mat4 T = glm::translate(glm::mat4(1), glm::vec3(-0.5,-0.5,-0.5));
glBindVertexArray(volumeSplatterVAO);
shader.Use();
   glUniformMatrix4fv(shader("MV"), 1, GL_FALSE, glm::value_ptr(MV*T));
```

```
        glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
        ↪glm::value_ptr(glm::inverseTranspose(glm::mat3(MV*T))));
        glUniformMatrix4fv(shader("P"), 1, GL_FALSE, glm::value_ptr(P));
        glDrawArrays(GL_POINTS, 0, splatter->GetTotalVertices());
    shader.UnUse();
```

Budowa shadera wierzchołków realizującego splatting (*Rozdział7/Splatting/shadery/splatShader.vert*) jest podana poniżej. Jest tu wyznaczana normalna w przestrzeni oka. Rozmiar placka jest obliczany na podstawie rozmiarów obszaru wolumetrycznego i próbkowanego woksela. Po uwzględnieniu położenia placka względem kamery jego rozmiar jest zapisywany przez shader w zmiennej `gl_PointSize`.

```
#version 330 core
layout(location = 0) in vec3 vVertex;
layout(location = 1) in vec3 vNormal;
uniform mat4 MV;
uniform mat3 N;
uniform mat4 P;
smooth out vec3 outNormal;
uniform float splatSize;
void main() {
    vec4 eyeSpaceVertex = MV*vec4(vVertex,1);
    gl_PointSize = 2*splatSize/-eyeSpaceVertex.z;
    gl_Position = P * eyeSpaceVertex;
    outNormal = N*vNormal;
}
```

Shader fragmentów biorący udział w realizacji splattingu (*Rozdział7/Splatting/shadery/splatShader.frag*) ma budowę następującą:

```
#version 330 core
layout(location = 0) out vec4 vFragColor;
smooth in vec3 outNormal;
const vec3 L = vec3(0,0,1);
const vec3 V = L;
const vec4 diffuse_color = vec4(0.75,0.5,0.5,1);
const vec4 specular_color = vec4(1);
void main() {
    vec3 N;
    N = normalize(outNormal);
    vec2 P = gl_PointCoord*2.0 - vec2(1.0);
    float mag = dot(P.xy,P.xy);
    if (mag > 1)
    discard;

    float diffuse = max(0, dot(N,L));
    vec3 halfVec = normalize(L+V);
    float specular=pow(max(0, dot(halfVec,N)),400);
    vFragColor = (specular*specular_color) + (diffuse*diffuse_color);
}
```

8. Następnie ustaw filtrujący FBO i rysując pełnoekranowy czworokąt zastosuj gaussowskie wygładzanie najpierw w pionie, a potem w poziomie — tak jak w recepturze z wariancyjnym mapowaniem cieni z rozdziału 4.

```
glBindVertexArray(quadVAOID);
glBindFramebuffer(GL_FRAMEBUFFER, filterFBOID);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
gaussianV_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
glDrawBuffer(GL_COLOR_ATTACHMENT1);
gaussianH_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
```

9. Odwiąż filtrujący FBO, przywróć domyślny bufor rysowania i wyrenderuj przefiltrowany rezultat na ekranie.

```
glBindFramebuffer(GL_FRAMEBUFFER,0);
glDrawBuffer(GL_BACK_LEFT);
glViewport(0,0,WIDTH, HEIGHT);
quadShader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
quadShader.UnUse();
glBindVertexArray(0);
```

Jak to działa?

Algorytm splattingu polega na renderowaniu wokseli jako gaussowskich plam i rzutowaniu ich na ekran. Aby to zrealizować, najpierw wybieramy z obszaru danych wolumetrycznych odpowiednie woksele. W tym celu przeglądamy cały obszar w poszukiwaniu wokseli o zadanej izowartości. Gdy napotykamy właściwy, zapisujemy jego położenie i normalną w tablicy wierzchołków. Dla własnej wygody wszystkie potrzebne do tego funkcje umieszczamy w klasie `VolumeSplatter`.

Po utworzeniu nowej instancji klasy `VolumeSplatter` (o nazwie `splatter`) ustalamy wymiary obszaru z danymi wolumetrycznymi i wczytujemy te dane. Następnie określamy wartość wyznaczającą izopowierzchnię i liczbę próbkowanych wokseli. Na koniec wywołujemy funkcję `Volume➥Splatter::SplatVolume`, która dokonuje przeglądu całego obszaru wolumetrycznego woksel po wokselu.

```
splatter = new VolumeSplatter();
splatter->SetVolumeDimensions(256,256,256);
splatter->LoadVolume(volume_file);
splatter->SetIsosurfaceValue(40);
splatter->SetNumSamplingVoxels(64,64,64);
std::cout<<"Generuje punktowe placki ...";
splatter->SplatVolume();
std::cout<<"Gotowe."<<std::endl;
```

Obiekt `splatter` umieszcza wygenerowane wierzchołki i ich normalne w tablicy wierzchołków i w wiązanym z nią obiekcie bufora wierzchołków. W funkcji renderującej najpierw rysujemy w jednym przebiegu cały zbiór placków do pozaekranowego celu, a rezultat poddajemy filtrowaniu przez dwa gaussowskie filtry splotowe. Na koniec wyświetlamy przefiltrowany obraz na pełnoekranowym czworokącie.

Shader wierzchołków (*Rozdział7/Splatting/shadery/splatShader.vert*) oblicza rozmiary punktów wyświetlanych na ekranie w zależności od głębokości, na jakiej leży dany placek. Żeby to było możliwe do wykonania w shaderze wierzchołków, musi być włączony stan `GL_VERTEX_PROGRAM_POINT_SIZE`, więc wywołujemy funkcję `glEnable(GL_VERTEX_PROGRAM_POINT_SIZE)`. Shader ten wyznacza również normalne dla placków w przestrzeni oka.

```
vec4 eyeSpaceVertex = MV*vec4(vVertex,1);
gl_PointSize = 2*splatSize/-eyeSpaceVertex.z;
gl_Position = P * eyeSpaceVertex;
outNormal = N*vNormal;
```

Aby nadać plackom okrągłe kształty na ekranie, shader fragmentów (*Rozdział7/Splatting/shadery/splatShader.frag*) odrzuca wszystkie fragmenty leżące poza okręgiem o promieniu równym promieniowi wyświetlanego placka.

```
vec3 N;
N = normalize(outNormal);
vec2 P = gl_PointCoord*2.0 - vec2(1.0);
float mag = dot(P.xy,P.xy);
if (mag > 1) discard;
```

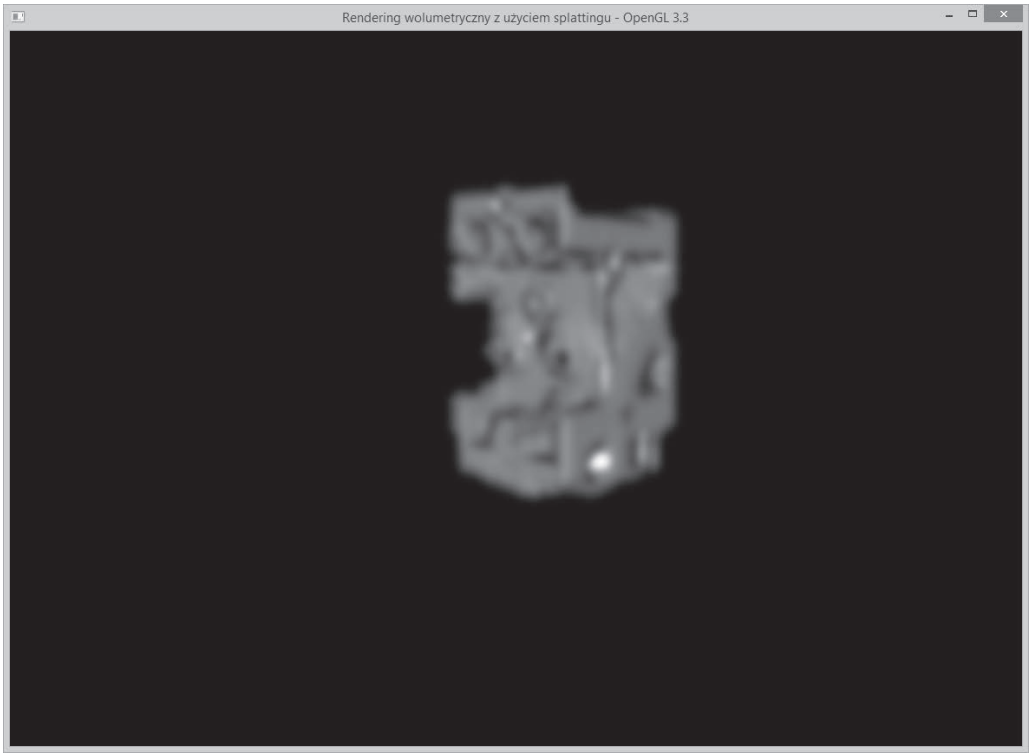
Potem shader wyznacza składowe rozproszenia i odblasku, aby po uwzględnieniu jeszcze normalnej wyświetlanego placka podać na wyjście ostateczny kolor bieżącego fragmentu.

```
float diffuse = max(0, dot(N,L));
vec3 halfVec = normalize(L+V);
float specular = pow(max(0, dot(halfVec,N)),400);
vFragColor = (specular*specular_color) + (diffuse*diffuse_color);
```

I jeszcze jedno…

Przykładowa aplikacja, podobnie jak poprzednie, renderuje dane wolumetryczne zeskanowanego fragmentu silnika. Jak widać na poniższym rysunku, obraz uzyskany metodą splattingu jest nieco rozmyty, a jest to skutek działania wygładzających filtrów gaussowskich.

Zaprezentowana receptura umożliwia poznanie metody splattingu, ale zastosowane przez nas rozwiązanie polegające na sprawdzaniu wszystkich wokseli nie jest zbyt wyszukane i w przypadku większego zbioru danych wolumetrycznych należałoby użyć jakiejś struktury, np. drzewa ósemkowego, która pozwoliłaby szybciej zlokalizować woksele o odpowiednich wartościach.



Dowiedz się więcej

Zapoznaj się z następującymi projektami:

- Projekt Qsplat, *http://graphics.stanford.edu/software/qsplat/*.
- Prace nad rozwojem splattingu w ETH Zurych, *http://graphics.ethz.ch/research/past_projects/surfels/surfacesplatting/*.

Implementacja funkcji przejścia dla klasyfikacji objętościowej

W tej recepturze pokażę, jak można zaimplementować klasyfikację danych wolumetrycznych w połączeniu z prezentowaną wcześniej metodą cięcia trójwymiarowej tekstury na płaty. Klasyfikacja będzie polegała na przypisywaniu określonym wartościom wolumetrycznym kolorów pobieranych z wygenerowanej w tym celu jednowymiarowej tekstury. Przydzielanie właściwych kolorów będzie wykonywał specjalny shader fragmentów. Rezultatem jego działania będzie więc kolor fragmentu uzależniony od wartości wolumetrycznej reprezentowanej przez ten fragment.

Cała reszta receptury wygląda tak samo jak w przypadku cięcia tekstury 3D. Oczywiście klasyfikację danych wolumetrycznych można stosować w połączeniu z dowolnym algorytmem renderowania.

Przygotowania

Gotowy kod dla tej receptury znajdziesz w folderze *Rozdział7/CięcieTekstury3DKlasyfikacja*.

Jak to zrobić?

Zacznij od następujących prostych czynności:

1. Wczytaj dane wolumetryczne i ustaw cięcie tekstury tak samo jak w recepturze „Implementacja renderingu wolumetrycznego z cięciem tekstury 3D na płaty”.
2. Dla funkcji przejścia przygotuj zestaw kolorów. Zakoduj tylko niektóre, a wszystkie pośrednie niech zostaną wygenerowane na zasadzie interpolacji w trakcie działania aplikacji. Szczegóły takiego rozwiązania znajdziesz w pliku *Rozdział7/CięcieTekstury3DKlasyfikacja/main.cpp*.

```
float pData[256][4];
int indices[9];
for(int i=0;i<9;i++) {
   int index = i*28;
   pData[index][0] = jet_values[i].x;
   pData[index][1] = jet_values[i].y;
   pData[index][2] = jet_values[i].z;
   pData[index][3] = jet_values[i].w;
   indices[i] = index;
}
for(int j=0;j<9-1;j++)
{
   float dDataR = (pData[indices[j+1]][0] - pData[indices[j]][0]);
   float dDataG = (pData[indices[j+1]][1] - pData[indices[j]][1]);
   float dDataB = (pData[indices[j+1]][2] - pData[indices[j]][2]);
   float dDataA = (pData[indices[j+1]][3] - pData[indices[j]][3]);
   int dIndex = indices[j+1]-indices[j];
   float dDataIncR = dDataR/float(dIndex);
   float dDataIncG = dDataG/float(dIndex);
   float dDataIncB = dDataB/float(dIndex);
   float dDataIncA = dDataA/float(dIndex);
   for(int i=indices[j]+1;i<indices[j+1];i++)
   {
      pData[i][0] = (pData[i-1][0] + dDataIncR);
      pData[i][1] = (pData[i-1][1] + dDataIncG);
      pData[i][2] = (pData[i-1][2] + dDataIncB);
```

```
            pData[i][3] = (pData[i-1][3] + dDataIncA);
        }
    }
```

3. Dla kolorów wygenerowanych w punkcie 1. utwórz jednowymiarową teksturę i zwiąż ją z jednostką teksturującą nr 1 (`GL_TEXTURE1`).

```
glGenTextures(1, &tfTexID);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_1D, tfTexID);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage1D(GL_TEXTURE_1D,0,GL_RGBA,256,0,GL_RGBA,GL_FLOAT,pData);
```

4. W shaderze fragmentów dodaj nowy sampler dla tej dodatkowej tekstury. Jako że są teraz dwie tekstury, zwiąż jedną z jednostką teksturującą 0 (`GL_TEXTURE0`), a drugą — z jednostką 1 (`GL_TEXTURE1`).

```
shader.LoadFromFile(GL_VERTEX_SHADER, "shaders/textureSlicer.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/textureSlicer.frag");
shader.CreateAndLinkProgram();
shader.Use();
   shader.AddAttribute("vVertex");
   shader.AddUniform("MVP");
   shader.AddUniform("volume");
   shader.AddUniform("lut");
   glUniform1i(shader("volume"),0);
   glUniform1i(shader("lut"),1);
shader.UnUse();
```

5. Na koniec pobierz wartość wolumetryczną z tekstury trójwymiarowej i odszukaj odpowiadający jej kolor w teksturze jednowymiarowej. Kolor ten skieruj do wyjścia jako kolor bieżącego fragmentu. Dokładniejszy opis tej operacji znajdziesz w pliku *Rozdział7/CięcieTekstury3DKlasyfikacja/shadery/textureSlicer.frag*.

```
uniform sampler3D volume;
uniform sampler1D lut;
void main(void) {
   vFragColor = texture(lut, texture(volume, vUV).r);
}
```

Jak to działa?

Recepturę można podzielić na dwie części: przygotowanie tekstury dla funkcji przejścia i pobieranie z niej kolorów w shaderze fragmentów. Obie są dość łatwe do zrozumienia. Pierwszą rozpoczynamy od utworzenia niewielkiej tablicy (o nazwie `jet_values`) z kilkoma podstawowymi kolorami. Definiujemy ją globalnie w sposób następujący:

```
const glm::vec4 jet_values[9]={glm::vec4(0,0,0.5,0),
   glm::vec4(0,0,1,0.1),
   glm::vec4(0,0.5,1,0.3),
   glm::vec4(0,1,1,0.5),
   glm::vec4(0.5,1,0.5,0.75),
   glm::vec4(1,1,0,0.8),
   glm::vec4(1,0.5,0,0.6),
   glm::vec4(1,0,0,0.5),
   glm::vec4(0.5,0,0,0.0)};
```

W momencie tworzenia tekstury powiększamy liczbę kolorów do 256, stosując zwykłą interpolację. Po prostu najpierw wyznaczamy różnicę między wartościami sąsiednich kolorów podstawowych i dzielimy ją przez odległość między tymi kolorami. Uzyskany w ten sposób przyrost dodajemy do bieżącego koloru i otrzymujemy interpolowaną wartość koloru pośredniego. Proces ten powtarzamy aż do zapełnienia całej tekstury, którą następnie przekazujemy do shadera fragmentów za pomocą dodatkowego samplera. W shaderze wartość pobrana z przetwarzanej próbki danych wolumetrycznych służy jako indeks wskazujący właściwy kolor w teksturze funkcji przejścia. Kolor ten jest ostatecznie przypisywany bieżącemu fragmentowi.

I jeszcze jedno…

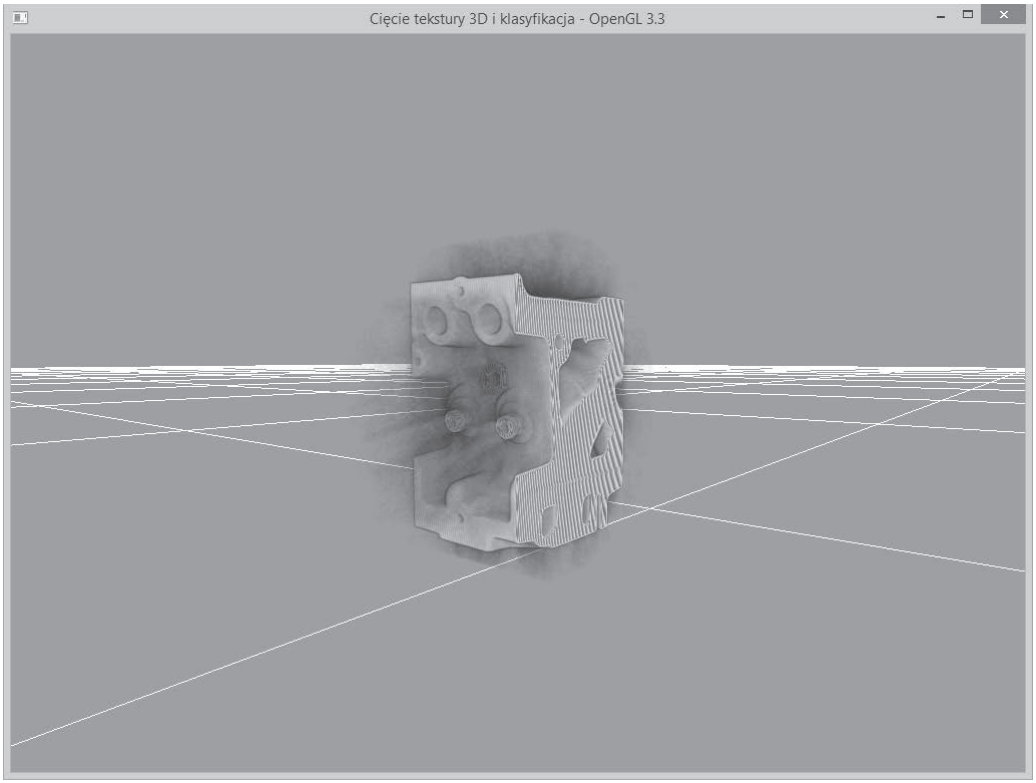
Aplikacja ilustrująca powyższą recepturę renderuje fragment silnika w sposób analogiczny do tego, jaki zastosowaliśmy w recepturze z cięciem tekstury 3D, ale teraz wprowadzenie funkcji przejścia spowodowało pokolorowanie wygenerowanego obrazu. Rezultat działania tej aplikacji jest pokazany na rysunku na następnej stronie.

Dowiedz się więcej

- *Real-Time Volume Graphics*, A K Peters/CRC Press, rozdział 4. „Transfer Functions” i rozdział 10. „Transfer Functions Reloaded”.

Implementacja wydzielania wielokątnej izopowierzchni metodą maszerujących sześcianów

W recepturze zatytułowanej „Pseudoizopowierzchniowy rendering w jednoprzebiegowym rzucaniu promieni” mieliśmy już do czynienia z izopowierzchnią, ale tamta nie była zbudowana z trójkątnych ścianek i gdybyśmy chcieli jednoznacznie wskazać jakiś konkretny jej obszar, byłoby to raczej niemożliwe. Coś takiego można osiągnąć przez wydzielenie izopowierzchni



metodą **maszerujących sześcianów** (*MC — marching cubes*)[2]. Metoda ta polega na przeczesywaniu całego zbioru danych wolumetrycznych i wstawianiu określonych wielokątów w miejscach spełniających kryterium przecięcia. W rezultacie powstaje wielokątna siatka obrazująca kształt zadanej izopowierzchni.

Przygotowania

Pełny kod przykładowej aplikacji znajdziesz w folderze *Rozdział7/MaszerującySześciany*. Cały algorytm MT zawiera się tam w klasie o nazwie `TetrahedraMarcher`.

[2] Autor posługuje się tutaj nazwą Marching Tetrahedra (maszerujące czworościany), ale tak naprawdę prezentuje algorytm o nazwie Marching Cubes (maszerujące sześciany) — *przyp. tłum.*

Jak to zrobić?

Zacznij od następujących prostych czynności:

1. Wczytaj dane wolumetryczne i umieść je w tablicy.

```
std::ifstream infile(filename.c_str(), std::ios_base::binary);
if(infile.good()) {
   pVolume = new GLubyte[XDIM*YDIM*ZDIM];
   infile.read(reinterpret_cast<char*>(pVolume),
   ↪XDIM*YDIM*ZDIM*sizeof(GLubyte));
   infile.close();
   return true;
} else {
   return false;
}
```

2. Uruchom trzy pętle, aby pobrać próbki z całego obszaru wolumetrycznego, woksel po wokselu.

```
vertices.clear();
int dx = XDIM/X_SAMPLING_DIST;
int dy = YDIM/Y_SAMPLING_DIST;
int dz = ZDIM/Z_SAMPLING_DIST;
glm::vec3 scale = glm::vec3(dx,dy,dz);
for(int z=0;z<ZDIM;z+=dz) {
   for(int y=0;y<YDIM;y+=dy) {
      for(int x=0;x<XDIM;x+=dx) {
         SampleVoxel(x,y,z, scale);
      }
   }
}
```

3. W każdym kroku próbkowania wyznacz wartości wolumetryczne we wszystkich ośmiu narożnikach próbkującego sześcianu.

```
GLubyte cubeCornerValues[8];
for( i = 0; i < 8; i++) {
   cubeCornerValues[i] = SampleVolume(
   x + (int)(a2fVertexOffset[i][0] *scale.x),
   y + (int)(a2fVertexOffset[i][1]*scale.y),
   z + (int)(a2fVertexOffset[i][2]*scale.z));
}
```

4. Wyznacz wartość wskaźnika krawędziowego, aby zidentyfikować przypadek maszerującego sześcianu dla izopowierzchni o zadanej wartości.

```
int flagIndex = 0;
for( i= 0; i<8; i++) {
   if(cubeCornerValues[i]<= isoValue)
      flagIndex |= 1<<i;
}
edgeFlags = aiCubeEdgeFlags[flagIndex];
```

5. Za pomocą tablicy przeglądowej (`a2iEdgeConnection`) znajdź właściwe krawędzie dla danego przypadku, a następnie użyj tablicy przesunięć (`a2fVertexOffset`), aby wyznaczyć wierzchołki krawędzi i normalne. Tablice te są zdefiniowane w pliku nagłówkowym *Tables.h* umieszczonym w folderze *Rozdział7/MaszerującySześciany/*.

```
for(i = 0; i < 12; i++)
{
   if(edgeFlags & (1<<i))
   {
   float offset = GetOffset(cubeCornerValues[a2iEdgeConnection[i][0] ],
     cubeCornerValues[ a2iEdgeConnection[i][1] ]);
   edgeVertices[i].x = x + (a2fVertexOffset[a2iEdgeConnection[i][0] ][0]
     + offset * a2fEdgeDirection[i][0])*scale.x ;
   edgeVertices[i].y = y + (a2fVertexOffset[a2iEdgeConnection[i][0] ][1]
     + offset *    a2fEdgeDirection[i][1])*scale.y ;
   edgeVertices[i].z = z + (a2fVertexOffset[a2iEdgeConnection[i][0] ][2]
     + offset * a2fEdgeDirection[i][2])*scale.z ;
   edgeNormals[i] = GetNormal( (int)edgeVertices[i].x,
   ↪(int)edgeVertices[i].y,
     (int)edgeVertices[i].z );
   }
}
```

6. Na koniec skorzystaj z tablicy przeglądowej połączeń trójkątów, aby połączyć właściwe wierzchołki i normalne dla danego przypadku.

```
for(i = 0; i< 5; i++) {
   if(a2iTriangleConnectionTable[flagIndex][3*i] < 0)
      break;
   for(int j= 0; j< 3; j++) {
      int vertex = a2iTriangleConnectionTable[flagIndex][3*i+j];
      Vertex v;
      v.normal = (edgeNormals[vertex]);
      v.pos = (edgeVertices[vertex])*invDim;
      vertices.push_back(v);
   }
}
```

7. Gdy już maszerujący sześcian przebiegnie cały obszar wolumetryczny, przekaż wygenerowane wierzchołki do obiektu tablicy wierzchołków zawierającej obiekt bufora wierzchołków.

```
glGenVertexArrays(1, &volumeMarcherVAO);
glGenBuffers(1, &volumeMarcherVBO);
glBindVertexArray(volumeMarcherVAO);
glBindBuffer (GL_ARRAY_BUFFER, volumeMarcherVBO);
glBufferData (GL_ARRAY_BUFFER, marcher->
GetTotalVertices()*sizeof(Vertex), marcher-> GetVertexPointer(),
↪GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,sizeof(Vertex),0);
```

```
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,sizeof(Vertex),
↪(const GLvoid*)offsetof(Vertex, normal));
```

8. W celu wyrenderowania powstałej geometrii zwiąż VAO z wygenerowanymi wierzchołkami, uaktywnij shader i wyrenderuj trójkąty. W tej recepturze jako kolor fragmentu wyprowadź normalną wierzchołka.

```
glBindVertexArray(volumeMarcherVAO);
shader.Use();
   glUniformMatrix4fv(shader("MVP"),1,GL_FALSE,
   glm::value_ptr(MVP*T));
   glDrawArrays(GL_TRIANGLES, 0, marcher->GetTotalVertices());
shader.UnUse();
```

Jak to działa?

Dla wygody umieszczamy całą procedurę maszerującego sześcianu w klasie `TetrahedraMarcher`. Zgodnie ze swą nazwą procedura ta przesuwa próbkujący sześcian po całym obszarze wolumetrycznym i wyznacza wartości wolumetryczne we wszystkich narożnikach próbki. W zależności od relacji między tymi ośmioma wartościami a wartością określoną dla izopowierzchni generowany jest specjalny indeks znakujący. Za pomocą tego indeksu wybierany jest z tablicy przeglądowej indeks krawędziowy, a ten z kolei pozwala wybrać jedną z predefiniowanych konfiguracji przecięcia sześcianu przez izopowierzchnię. Następnie z tablicy połączeń krawędziowych wybierane są względne położenia narożników sześcianu próbkującego i na ich podstawie wyznaczane są wierzchołki i normalne wielokąta należącego do izoprzestrzeni. Po zgromadzeniu tych wierzchołków przeprowadzana jest triangulacja.

Przyjrzyjmy się poszczególnym etapom nieco dokładniej. Indeks znakujący jest określany w wyniku porównania wartości wolumetrycznych we wszystkich ośmiu narożnikach sześcianu próbkującego z zadaną wartością izopowierzchni. Indeks ten umożliwia wybranie znaczników krawędzi z tablicy przeglądowej `aiCubeEdgeFlags`.

```
flagIndex = 0;
for( i= 0; i<8; i++) {
   if(cubeCornerValues[i] <= isoValue)
      flagIndex |= 1<<i;
   }
edgeFlags = aiCubeEdgeFlags[flagIndex];
```

Wierzchołki i normalne wielokąta odpowiadające danemu indeksowi są obliczane na podstawie danych pobranych z tablicy połączeń krawędziowych (`a2iEdgeConnection`) i umieszczane w tablicach lokalnych.

```
for(i = 0; i < 12; i++) {
if(edgeFlags & (1<<i)) {
float offset = GetOffset(cubeCornerValues[a2iEdgeConnection[i][0] ],
  cubeCornerValues[a2iEdgeConnection[i][1] ]);
```

```
edgeVertices[i].x = x + (a2fVertexOffset[a2iEdgeConnection[i][0] ][0]
  + offset * a2fEdgeDirection[i][0])*scale.x;
edgeVertices[i].y = y + (a2fVertexOffset[a2iEdgeConnection[i][0] ][1]
  + offset * a2fEdgeDirection[i][1])*scale.y;
edgeVertices[i].z = z + (a2fVertexOffset[a2iEdgeConnection[i][0] ][2]
  + offset * a2fEdgeDirection[i][2])*scale.z;
edgeNormals[i] = GetNormal( (int)edgeVertices[i].x, (int)edgeVertices[i].y,
  (int)edgeVertices[i].z );
```

Na koniec użyta zostaje tablica przeglądowa połączeń trójkątów (`a2iTriangleConnectionTable`) i z niej pobierane są właściwe uporządkowania wierzchołków i normalnych. Atrybuty te trafiają ostatecznie do odpowiednich wektorów.

```
for(i = 0; i< 5; i++) {
   if(a2iTriangleConnectionTable[flagIndex][3*i] < 0)
      break;
   for(int j= 0; j< 3; j++) {
      int vertex = a2iTriangleConnectionTable[flagIndex][3*i+j];
      Vertex v;
      v.normal = (edgeNormals[vertex]);
      v.pos = (edgeVertices[vertex])*invDim;
      vertices.push_back(v);
   }
}
```

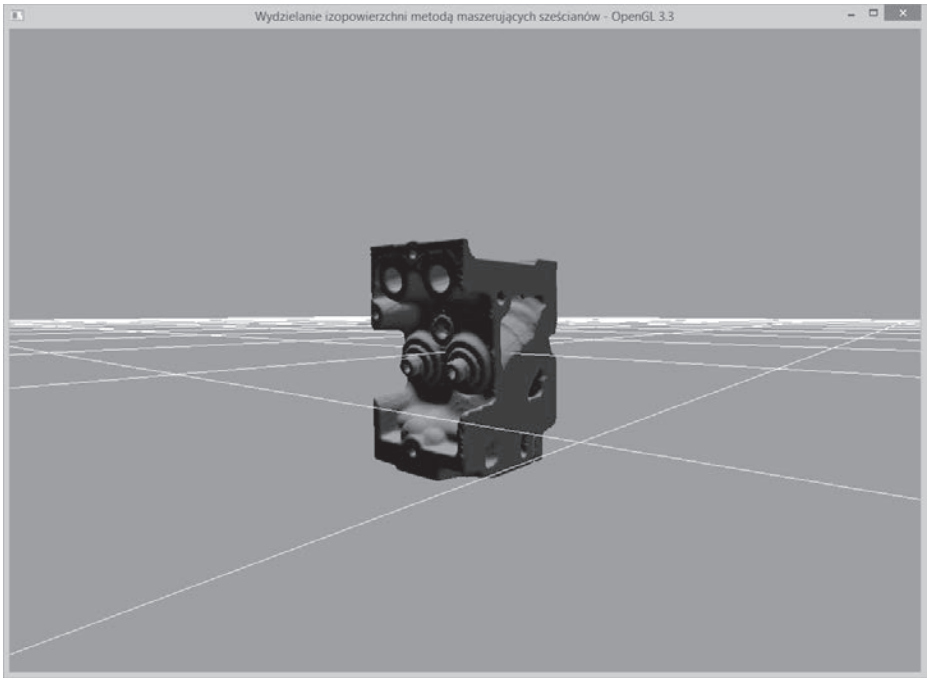
Po zrealizowaniu procedury maszerujących sześcianów umieszczamy wygenerowane wierzchołki i normalne w obiekcie bufora. W funkcji renderującej wiążemy odpowiedni obiekt tablicy wierzchołków, uaktywniamy shader i rysujemy trójkąty. Zastosowany tu shader fragmentów podaje na wyjście, jako kolor bieżącego fragmentu, współrzędne wektora normalnego.

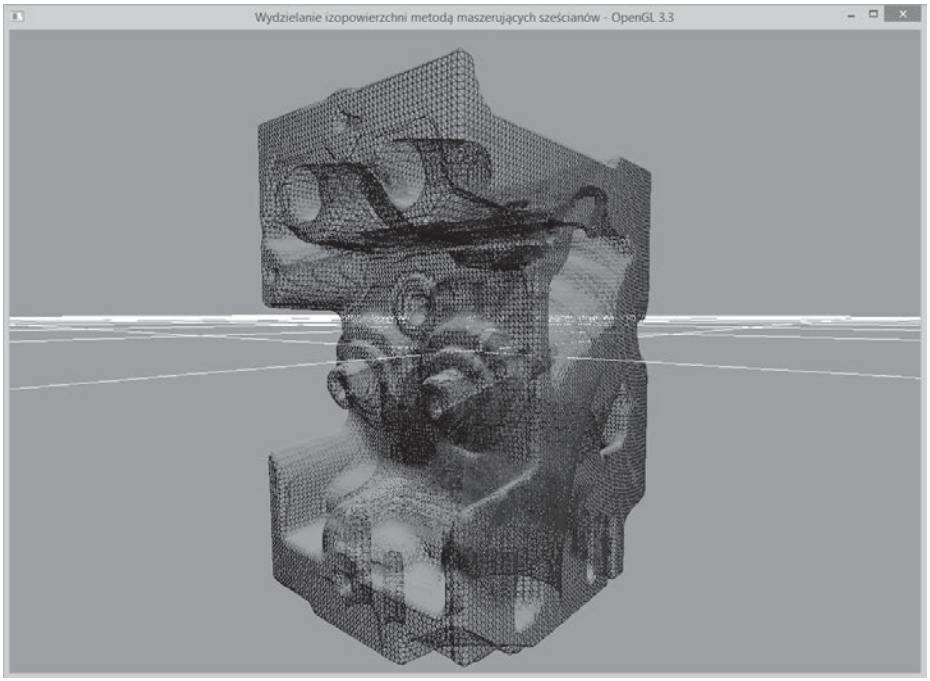
```
#version 330 core
layout(location = 0) out vec4 vFragColor;
smooth in vec3 outNormal;
void main() {
   vFragColor = vec4(outNormal,1);
}
```

I jeszcze jedno…

Podobnie jak w poprzednich recepturach aplikacja przykładowa renderuje dane wolumetryczne fragmentu silnika, co widać na rysunku na następnej stronie. Kolory są tu ustalane na podstawie normalnych izopowierzchni.

Wciśnięcie klawisza *W* spowoduje włączenie renderingu krawędziowego (wireframe). Można wtedy zobaczyć wielokąty izopowierzchni o wartości 40 (patrz na drugi rysunek na następnej stronie).





Powyższa receptura jest oparta na algorytmie maszerujących sześcianów, ale istnieje też metoda **maszerujących czworościanów** (*marching tetrahedra*), która pozwala na dokładniejszą triangulację izopowierzchni.

Dowiedz się więcej

Zapoznaj się z następującymi publikacjami:

- Paul Bourke, *Polygonising a scalar field*, *http://paulbourke.net/geometry/polygonise/*.
- *Volume Rendering: Marching Cubes Algorithm*, *http://cns-alumni.bu.edu/~lavanya/Graphics/cs580/p5/web-page/p5.html*.
- *An implementation of Marching Cubes and Marching Tetrahedra Algorithms*, *http://www.siafoo.net/snippet/100*.

Wolumetryczne oświetlenie oparte na technice cięcia połówkowokątowego

W tej recepturze zaimplementujemy oświetlenie wolumetryczne, a jako technikę renderingu zastosujemy cięcie połówkowokątowe. Zamiast ciąć obszar wolumetryczny na plastry prostopadłe do kierunku patrzenia potniemy go ukośnie, co umożliwi nam symulowanie absorpcji światła przez kolejne plastry.

Przygotowania

Pełny kod przykładowej aplikacji znajduje się w folderze *Rozdział7/CięciePołówkowokątowe*. Jak łatwo się domyślić, dużą część kodu zapożyczymy z receptury stanowiącej przykład implementacji renderingu wolumetrycznego z cięciem tekstury 3D na płaty.

Jak to zrobić?

Zacznij od następujących prostych czynności:

1. Ustaw pozaekranowy rendering z użyciem FBO wyposażonego w dwa przyłącza: jedno dla pozaekranowego renderingu bufora światła i jedno dla pozaekranowego renderingu bufora oka.

```
glGenFramebuffers(1, &lightFBOID);
glGenTextures (1, &lightBufferID);
glGenTextures (1, &eyeBufferID);
glActiveTexture(GL_TEXTURE2);
```

```
lightBufferID = CreateTexture(IMAGE_WIDTH, IMAGE_HEIGHT, GL_RGBA16F,
↪GL_RGBA);
eyeBufferID = CreateTexture(IMAGE_WIDTH, IMAGE_HEIGHT, GL_RGBA16F,
↪GL_RGBA);
glBindFramebuffer(GL_FRAMEBUFFER, lightFBOID);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
↪GL_TEXTURE_2D, lightBufferID, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
↪GL_TEXTURE_2D, eyeBufferID, 0);
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE )
   printf("Ustawienie FBO dla swiatla powiodlo sie !!! \n");
else
   printf("Problem z ustawieniem FBO dla swiatla ");
```

Dla wygody umieściłem tworzenie tekstury i ustalanie jej parametrów w jednej funkcji o nazwie `CreateTexture`. Jej definicja wygląda następująco:

```
GLuint CreateTexture(const int w,const int h, GLenum internalFormat,
↪GLenum format) {
   GLuint texid;
   glGenTextures(1, &texid);
   glBindTexture(GL_TEXTURE_2D, texid);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
   glTexImage2D(GL_TEXTURE_2D, 0, internalFormat, w, h, 0, format,
   ↪GL_FLOAT, 0);
   return texid;
}
```

2. Podobnie jak w recepturze ze zwykłym cięciem tekstury 3D wczytaj dane wolumetryczne.

```
std::ifstream infile(volume_file.c_str(),
std::ios_base::binary);
if(infile.good()) {
   GLubyte* pData = new GLubyte[XDIM*YDIM*ZDIM];
   infile.read(reinterpret_cast<char*>(pData),
   ↪XDIM*YDIM*ZDIM*sizeof(GLubyte));
   infile.close();
   glGenTextures(1, &textureID);
   glActiveTexture(GL_TEXTURE0);
   glBindTexture(GL_TEXTURE_3D, textureID);
   // ustaw parametry tekstury
   glTexImage3D(GL_TEXTURE_3D,0,GL_RED,XDIM,YDIM,ZDIM,0,GL_RED,GL_
   ↪UNSIGNED_BYTE,pData);
   GL_CHECK_ERRORS
   glGenerateMipmap(GL_TEXTURE_3D);
```

```
        return true;
    } else {
        return false;
    }
```

3. Podobnie jak w technikach mapowania cieni wyznacz macierz cienia jako iloczyn macierzy modelu i widoku, rzutowania oraz przesunięcia.

```
MV_L=glm::lookAt(lightPosOS,glm::vec3(0,0,0), glm::vec3(0,1,0));
P_L=glm::perspective(45.0f,1.0f,1.0f, 200.0f);
B=glm::scale(glm::translate(glm::mat4(1), glm::vec3(0.5,0.5,0.5)),
↪glm::vec3(0.5,0.5,0.5));
BP = B*P_L;
S = BP*MV_L;
```

4. W kodzie renderingu wyznacz wektor połówkowy względem wektorów kierunkowych widoku i światła.

```
viewVec = -glm::vec3(MV[0][2], MV[1][2], MV[2][2]);
lightVec = glm::normalize(lightPosOS);
bIsViewInverted = glm::dot(viewVec, lightVec)<0;
halfVec = glm::normalize( (bIsViewInverted?-viewVec:viewVec) +
↪lightVec);
```

5. Potnij obszar wolumetryczny tak samo jak w recepturze ze zwykłym cięciem tekstury 3D. Jedyną różnicą niech będzie to, że zamiast ciąć prostopadle do kierunku widoku zastosujesz cięcie w kierunku dwusiecznej kąta między wektorami widoku i światła.

```
float max_dist = glm::dot(halfVec, vertexList[0]);
float min_dist = max_dist;
int max_index = 0;
int count = 0;
for(int i=1;i<8;i++) {
    float dist = glm::dot(halfVec, vertexList[i]);
    if(dist > max_dist) {
        max_dist = dist;
        max_index = i;
    }
    if(dist<min_dist)
    min_dist = dist;
}
//reszta funkcji SliceVolume jak w recepturze ze zwykłym cięciem tekstury 3D
//zmienia się tylko viewVec na halfVec
```

6. Zwiąż FBO, a następnie wyczyść bufor światła białym kolorem `(1,1,1,1)` i bufor oka kolorem czarnym `(0,0,0,0)`.

```
glBindFramebuffer(GL_FRAMEBUFFER, lightFBOID);
glDrawBuffer(attachIDs[0]);
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT );
```

```
glDrawBuffer(attachIDs[1]);
glClearColor(0,0,0,0);
glClear(GL_COLOR_BUFFER_BIT );
```

7. Zwiąż obiekt `volumeVAO` i uruchom pętlę przebiegającą przez wszystkie płaty. W każdym przebiegu najpierw wyrenderuj płat do bufora oka, ale z buforem światła związanym jako teksturą. Następnie wyrenderuj płat do bufora światła.

```
glBindVertexArray(volumeVAO);
for(int i =0;i<num_slices;i++) {
   shaderShadow.Use();
   glUniformMatrix4fv(shaderShadow("MVP"), 1, GL_FALSE,
   glm::value_ptr(MVP));
   glUniformMatrix4fv(shaderShadow("S"), 1, GL_FALSE, glm::value_ptr(S));
   glBindTexture(GL_TEXTURE_2D, lightBuffer);
   DrawSliceFromEyePointOfView(i);

   shader.Use();
   glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
   ↪glm::value_ptr(P_L*MV_L));
   DrawSliceFromLightPointOfView(i);
}
```

8. W funkcji renderującej płat z punktu widzenia oka ustaw odpowiedni zwrot funkcji mieszania w zależności od tego, czy kierunek patrzenia jest zgodny z kierunkiem światła, czy też jest do niego przeciwny.

```
void DrawSliceFromEyePointOfView(const int i) {
   glDrawBuffer(attachIDs[1]);
   glViewport(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
   if(bIsViewInverted) {
      glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
   } else {
      glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
   }
   glDrawArrays(GL_TRIANGLES, 12*i, 12);
}
```

9. W przypadku bufora światła zastosuj zwykłe mieszanie „nakładkowe”.

```
void DrawSliceFromLightPointOfView(const int i) {
   glDrawBuffer(attachIDs[0]);
   glViewport(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glDrawArrays(GL_TRIANGLES, 12*i, 12);
}
```

10. Na koniec odwiąż FBO i przywróć domyślny bufor rysowania. Ustaw okno widokowe na cały ekran i używając shadera, wyrenderuj bufor oka.

```
glBindVertexArray(0);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glDrawBuffer(GL_BACK_LEFT);
```

```
glViewport(0,0,WIDTH, HEIGHT);
glBindTexture(GL_TEXTURE_2D, eyeBufferID);
glBindVertexArray(quadVAOID);
quadShader.Use();
glDrawArrays(GL_TRIANGLES, 0, 6);
quadShader.UnUse();
glBindVertexArray(0);
```

Jak to działa?

Prezentowana technika polega na akumulowaniu pośrednich rezultatów w dwóch odrębnych buforach i cięciu obszaru wolumetrycznego na plastry w kierunku dwusiecznej kąta między wektorami światła i widoku. Gdy scena jest renderowana z punktu widzenia oka, bufor światła służy za teksturę wskazującą, czy bieżący fragment jest w cieniu, czy nie. Sprawdzian ten odbywa się w shaderze fragmentów z użyciem macierzy cienia, tak jak w algorytmie mapowania cieni. Na tym etapie następuje też zmiana równania mieszającego w zależności od wzajemnej relacji między wektorami kierunkowymi widoku i światła. Gdy wektor widoku jest odwrócony w stosunku do wektora światła, mieszanie zachodzi od tyłu ku przodowi i jest właśnie tak ustawiane za pomocą instrukcji `glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE)`. Natomiast gdy oba wektory są zwrócone w tę samą stronę, mieszanie jest ustawiane przez instrukcję `glBlendFunc(GL_ONE, ↪GL_ONE_MINUS_SRC_ALPHA)`. Zauważ, że nie stosujemy tu mieszania nakładkowego, bo już wcześniej kolor został pomnożony przez wartość alfa w shaderze fragmentów (*Rozdział7/CięciePołówkowokątowe/shadery/slicerShadow.frag*), co widać w poniższym fragmencie jego kodu:

```
vec3 lightIntensity = textureProj(shadowTex, vLightUVW.xyw).xyz;
float density = texture(volume, vUV).r;
if(density > 0.1) {
   float alpha = clamp(density, 0.0, 1.0);
   alpha *= color.a;
   vFragColor = vec4(color.xyz*lightIntensity*alpha, alpha);
}
```

Następny etap to renderowanie sceny z punktu widzenia źródła światła. Tym razem stosujemy mieszanie nakładkowe, aby elementy składowe światła kumulowały się tak jak w zwykłych warunkach. Użyty do tego shader fragmentów jest dokładnie taki sam jak ten, którego używaliśmy przy zwykłym cięciu tekstury 3D (patrz *Rozdział7/CięciePołówkowokątowe/shadery/textureSlicer.frag*).

```
vFragColor = texture(volume, vUV).rrrr * color ;
```

I jeszcze jedno…

Aplikacja będąca implementacją powyższej receptury renderuje scenę znaną z innych aplikacji opisanych w tym rozdziale. Położenie źródła światła można zmieniać przez przeciąganie myszą z wciśniętym prawym przyciskiem. Widać wtedy, że cienie są dynamiczne i na bieżąco dosto-

sowują się do nowych warunków oświetleniowych. Za pomocą uniformów shadera wprowadzone zostało pokolorowane zanikanie światła wraz z odległością. To z tego powodu widać niebieskawe zabarwienie finalnego obrazu. Zauważ, że tym razem nie widać czarnej otoczki wokół renderowanego obiektu. Zniknęła, bo w shaderze fragmentów dopuściliśmy do obliczeń tylko te wartości wolumetryczne, które są większe od 0,1. W ten sposób pozbyliśmy się niepożądanych artefaktów i uzyskaliśmy dużo lepszy rezultat.

Dowiedz się więcej

Zapoznaj się z następującymi publikacjami:

- „Volume Rendering Techniques", w *GPU Gems 1*, rozdział 39., dostępny pod adresem *http://http.developer.nvidia.com/GPUGems/gpugems_ch39.html*.
- *Real-Time Volume Graphics*, A K Peters/CRC Press, rozdział 6., „Global Volume Illumination".

Skorowidz

D

E

F

G

H

I

K

L

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ź

OpenGL
Receptury dla programisty

OpenGL dostarcza programistom API zaawansowanych grafik i animacji do renderowania. To oprogramowanie umożliwia tworzenie niezwykle atrakcyjnych gier, prezentacji oraz efektów graficznych. Jeżeli chcesz poznać możliwości OpenGL, jeżeli szukasz odpowiedzi na nurtujące Cię pytania, to trafiłeś na świetną książkę!

Znajdziesz w niej zbiór receptur cenionych przez programistów. Dzięki nim błyskawicznie rozwiążesz typowe problemy oraz zobaczysz, jak podejść do przeróżnych zagadnień związanych z OpenGL. Sięgnij po tę lekturę, a nauczysz się wybierać obiekty na podstawie ich różnych właściwości, mapować środowisko, filtrować obraz oraz tworzyć realne scenerie za pomocą odpowiedniej gry światła i cienia. Ponadto zobaczysz, jak śledzić promienie, ścieżki oraz tworzyć animacje szkieletowe i symulacje fizyczne. Ta książka to kopalnia najlepszych przepisów na wykorzystanie OpenGL!

Dzięki tej książce:

- zaznajomisz się z możliwościami OpenGL
- zaimplementujesz system wykrywania kolizji oraz system cząsteczkowy
- poznasz formaty modeli siatkowych
- nauczysz się mapować środowisko
- wykorzystasz w pełni OpenGL

Najlepsze przepisy na OpenGL!

[PACKT] PUBLISHING

Helion

28924 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-0018-7

cena: 54,90 zł