Volume Renderingの実装方法について、説明する。

# 3Dグリッドデータ

まず、表示するための3Dグリッドデータを格納するテクスチャと、それにアクセスするためのFBOを作成する必要がある。

追記：FBOは、出力先としてテクスチャを使用したい場合に、必要になるだけ。入力としてテクスチャを使用するだけなら、FBOは不要だ！つまり、与えられた3Dデータを、Volume renderingで表示するだけなら、FBOは不要なのだ！

## 3Dテクスチャの作成

3Dグリッドのデータを表示したいわけだから、まず、そのデータを格納した3Dテクスチャを作成する必要があるよね。

3Dテクスチャ作成の流れは、まず、テクスチャIDを生成する。

glGenTextures(1, &texture);

そして、パラメータを設定する。

glBindTexture(GL\_TEXTURE\_3D, texture);

glTexParameteri(GL\_TEXTURE\_3D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_3D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_3D, GL\_TEXTURE\_WRAP\_R, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_3D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_3D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

最後に3DグリッドのデータをGPUへ転送する。コードは以下の通り。

glTexImage3D(GL\_TEXTURE\_3D, 0, GL\_R16F, width, height, depth, 0, GL\_RED, GL\_FLOAT, data);

なお、Dataには、float型のスカラーの3Dグリッドのデータが格納されているものとする。もし、例えば3次元のベクトルなら、以下のようにすれば良い。

glTexImage3D(GL\_TEXTURE\_3D, 0, GL\_RGB16F, width, height, depth, 0, GL\_RGB, GL\_FLOAT, data);

## FBOの作成

作成した3Dテクスチャへアクセスするには、実際にはFBOを介して行う。そのため、対応するFBOを作成し、3Dテクスチャを、そのFBOに関連付けるという作業が必要になる。

まず、FBOを作成する。

glGenFramebuffers(1, &fbo);

次に、このFBOに、上で作成したtextureを関連付ける。

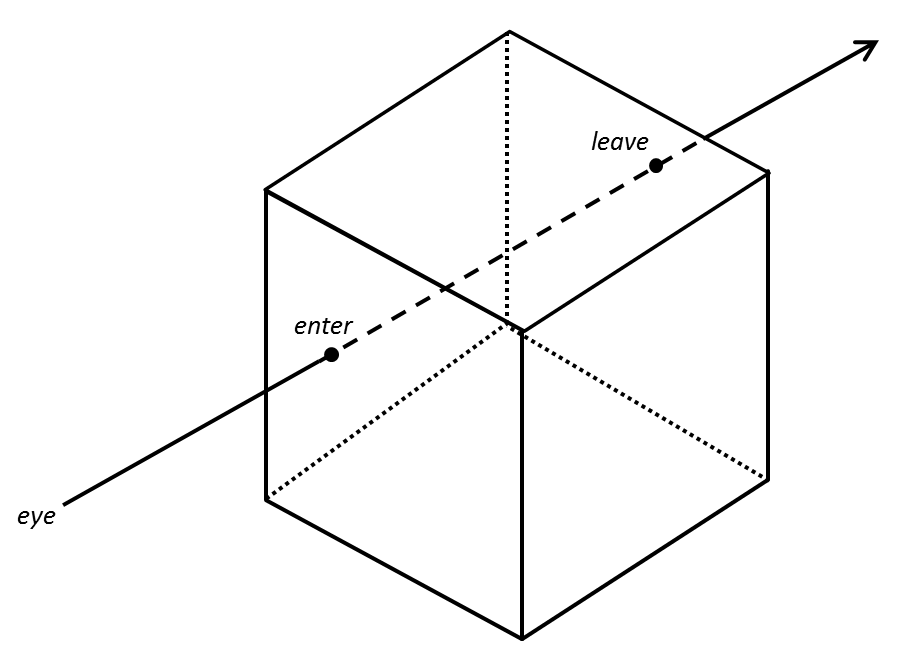
glBindFramebuffer(GL\_FRAMEBUFFER, fbo);

glFramebufferTexture(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, texture, 0);

# Volumetric DataのBounding Box

## Bounding Box

通常、volumetricデータは、のグリッドデータだ。つまり、そのサイズのBounding Boxの中にのみ、データがある。Ray castingでは、このBounding Boxに対してrayをcastする。



Ray castingのプログラムでは、上図の「enter」と「leave」の情報を使って、ray castする。従って、この「enter」と「leave」の情報を事前に計算する必要がある。

アイデアはいたってシンプルだ。上図のbounding boxを普通に描画してやると、フラグメントシェーダでは、画面上の各ピクセルについて、対応するポリゴン上の点の座標が得られる。例えば、上図の例だと、画面上のあるピクセルに対して、前面のポリゴンの「enter」についてフラグメントシェーダが起動し、また、背面のポリゴンの「leave」についてフラグメントシェーダが起動する。つまり、各ピクセルについて２回フラグメントシェーダが起動する。なので、フラグメントシェーダの中で、２つの点「enter」と「leave」の座標を記録しておけば良い。

## 2Dテクスチャ

というわけで、この２種類の点「enter」と「leave」を記録するために、２つのテクスチャを用意する。まず、

glGenTextures(1, &texture);

次に、テクスチャのパラメータを設定する。

glBindTexture(GL\_TEXTURE\_2D, texture);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

そして、テクスチャ用にメモリを確保する。

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB16F, width, height, 0, GL\_RGB, GL\_HALF\_FLOAT, 0);

## FBOの作成

例によって、テクスチャにアクセスするためには、FBOを介す必要がある。というわけで、FBOを作成する。

glGenFramebuffers(1, &fbo);

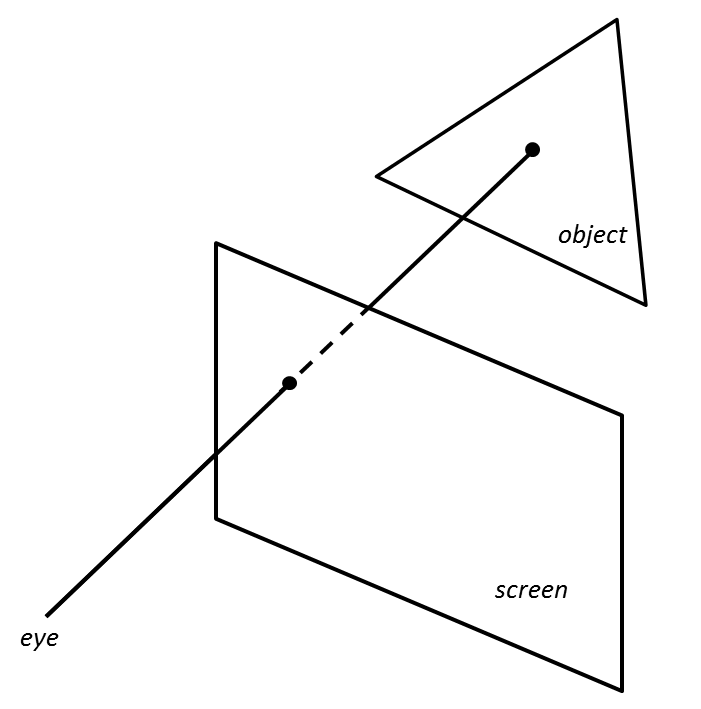
そして、先ほど作成した2Dテクスチャを、このFBOに関連付ける。ただし、２つのテクスチャを関連付けたいので、１つ目を「GL\_COLOR\_ATTACHMENT0」、２つ目を「GL\_COLOR\_ATTACHMENT1」として関連付ける。こうすることで、後でフラグメントシェーダから２つのテクスチャにアクセスできるようになる。

glBindFramebuffer(GL\_FRAMEBUFFER, fbo);

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT*i*, GL\_TEXTURE\_2D, texture, 0);

## Enter、leaveの座標の計算

画面上の各ピクセルに対応する、各ポリゴン内部の点に対して、フラグメントシェーダが起動される。つまり、フラグメントシェーダからは、それに対応するポリゴン内部の点と、対応する画面上のピクセルを取得できるわけだ。



しかし、１つのピクセルに対して、アウトプットは通常１つのRGBカラーだ。つまり、例えば、点「leave」に対してフラグメントシェーダが起動し、点「leave」の座標をRGBカラーとして出力した後、点「enter」に対してフラグメントシェーダが起動し、点「enter」をRGBカラーとして出力したら、点「enter」の方がカメラに近いので、上書きし、「leave」の座標は消えてしまう。一体どうやって、２つの点「enter」と「leave」の座標を共に記録すれば良いのか？

ここで、先ほど用意した２つの2Dテクスチャをアウトプット先として使用することになる。そのために、まず、アウトプット先をスクリーンではなく、この２つの2Dテクスチャにしなければならない。そこで、対応するfboをフレームバッファとしてバインドする。

glBindFramebuffer(GL\_FRAMEBUFFER, fbo);

さらに、アウトプット先として、関連付けた２つの2Dテクスチャを使用できるようにする。これにより、GL\_COLOR\_ATTACHMENT0は、glFragColor[0]、GL\_COLOR\_ATTACHMENT1は、glFragColor[1]として、アプトプット先としてアクセスできるようになった。

GLenum buffers[2] = { GL\_COLOR\_ATTACHMENT0, GL\_COLOR\_ATTACHMENT1 };

glDrawBuffers(2, buffers);

さらに、ブレンディングのオプション「GL\_ONE」を使う。

glEnable(GL\_BLEND);

glBlendFunc(GL\_ONE, GL\_ONE);

シェーダ側では、以下のようにgl\_FrontFacingフラグを使って「enter」か「leave」かをチェックし、２つのテクスチャに分けて出力している。上のブレンディングオプションのおかげで、glFragColor[i]への出力がORで加算されていくため、「leave」の座標が消えず、両方の座標が共に記録できるわけだ。非常にうまいね！

if (gl\_FrontFacing) {

glFragColor[0] = 0.5 \* (pos + 1.0);

glFragColor[1] = vec3(0);

} else {

glFragColor[0] = vec3(0);

glFragColor[1] = 0.5 \* (pos + 1.0);

}

なお、上で述べたとおり、「leave」に対応するポリゴンは後ろを向いている。通常、後ろ向きのポリゴンに対してはフラグメントシェーダは起動されない。そこで、GULL\_FACEをOFFにして、後ろ向きのポリゴンに対してもフラグメントシェーダが起動すようにしておく必要がある。

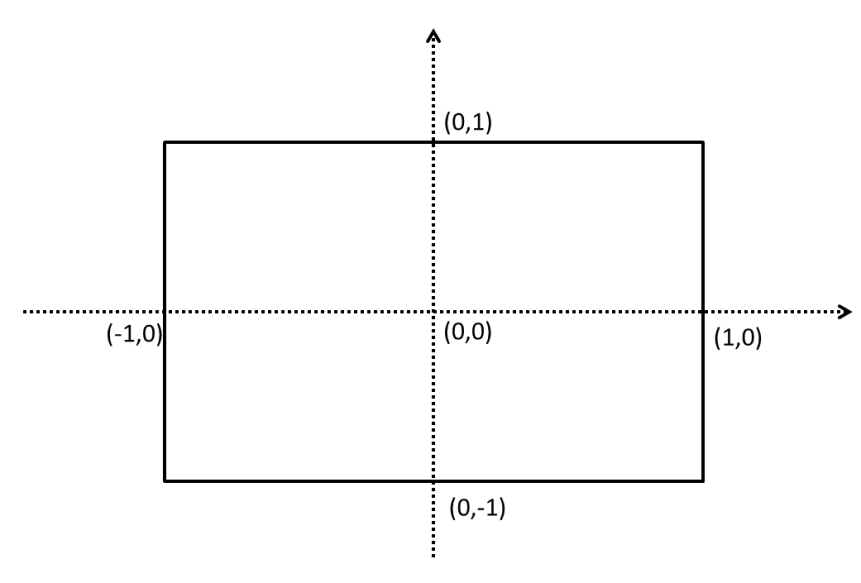
glDisable(GL\_CULL\_FACE);

# Ray Casting

ここまで来たら、後はray castするだけだ。なお、ここでは、通常のレンダリングのようにmodelviewMatrixとprojectionMatrixを使ってオブジェクトのポリゴンの座標を変換したりはしない。ここ、すごく重要！！

画面上の各ピクセルの色を計算するために必要な情報は、volumetric dataを格納した3Dテクスチャと、既に計算した「enter」、「leave」が格納された2Dテクスチャがあれば十分なのだ。だから、座標変換とかはもはや必要ないのだ。

こういう時、画面上の各ピクセルについて、フラグメントシェーダを起動するためには、一番良い方法は、(-1,-1)-(1,1)の2Dの四角形を、ortho射影で描画する方法だ。つまり、画面と同じ大きさの四角形を描画することで、各ピクセルについてフラグメントシェーダが必ず１回ずつ起動される。



## テクスチャのバインド

各テクスチャをバインドし、フラグメントシェーダからアクセスできるようにする。

glActiveTexture(GL\_TEXTURE0); glBindTexture(GL\_TEXTURE\_2D, cubeinter.texture[0]);

glActiveTexture(GL\_TEXTURE1); glBindTexture(GL\_TEXTURE\_2D, cubeinter.texture[1]);

glActiveTexture(GL\_TEXTURE2); glBindTexture(GL\_TEXTURE\_3D, texture);