[OpenGL講座第03回]

インデックスバッファのある世界線

# インデックスバッファ

## 不必要に頂点データを増やさない方法

プリミティブには四角形がありません。そのため四角形を表示するには、GL\_TRIANGLESを使う場合は2つの三角形、つまり頂点データが合計6つ必要です。現在のOpenGLでは、点・線角・三角形のいずれかしか描画できないため、図形としては角がひとつ増えただけなのに、頂点データは3つも追加しなければならないのです。  
また、GL\_TRIANGLE\_STRIPやGL\_TRIANGLE\_FANを使えば4つで済みますが、これらは「連続した三角形を描く」ためのものです。そのため、四角形のように途中が途切れている図形の場合、図形の数と同じ回数のglDrawArrays関数を実行しなければなりません。

**[補足]** 実はGL\_TRIANGLE\_STRIP, GL\_TRIANGLE\_FANでも、一度のglDrawArrays関数の実行で離れた位置にある四角形を描く方法はあります。しかし、描画されない頂点データを定義するなどの工夫が必要なこと、そして、そのような工夫が必要なデータを取り扱うことが難しいことなどの理由で、2018年現在ではめったに使われません。

頂点データが増えると、それだけメモリを必要となります。また、頂点データの数だけシェーダが実行されるため、処理時間もかかります。増えた角数のデータを追加するだけで描画する方法はないのでしょうか。  
それが実はあるんです。

それがインデックスバッファです。インデックスバッファを使えば、頂点データを直接プリミティブとして扱うのではなく、インデックスによって間接的に扱うことが可能になります。もちろん、インデックスバッファもメモリを使います。しかし、格納されるのはインデックスだけなので、せいぜいint型変数ひとつ、容量にして4バイトです。頂点データは座標だけでも3つのfloat型変数を使うので12バイトとなります。これはインデックスデータの3倍の容量です。頂点データに色や法線など他の情報を加えていけば、その差はさらに広がります。

## インデックスデータ

インデックスデータの使い方を確認するために、頂点データ配列に4つの頂点を追加しましょう。  
配列の先頭に、次のプログラムを追加してください。

const Vertex vertices[] = {  
**+** { {-0.5f, -0.3f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
**+** { { 0.3f, -0.3f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
**+** { { 0.3f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
**+** { {-0.5f, 0.5f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
**+**  
 { {-0.5f, -0.3f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.3f, -0.3f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.3f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
 .  
 .  
 .  
 };

追加した頂点データは、左下を始点として反時計回りに4つの頂点を定義しています。

次に、インデックスデータを追加しましょう。頂点データ配列の定義の下に、次のプログラムを追加してください。

{ {-0.5f, -0.3f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.3f, -0.3f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.3f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
 .  
 .  
 .  
 };  
**+**  
**+**/// インデックスデータ.  
**+**const GLushort indices[] = {  
**+** 0, 1, 2, 2, 3, 0,  
**+**};

インデックスデータは、「対応する頂点データ配列の先頭から何番目の頂点データか」を示す数値の集まりです。頂点データがそうであるように、インデックスデータもプリミティブの指定と組み合わせなければ、実際の形状は確定しません。今回作成したindices(いんでぃしーず、indexの複数形)配列変数の場合、GL\_TRIANGLESを使って描画する前提で、「0, 1, 2」と[2, 3, 0」という2つの三角形を定義しています。

なお、インデックスデータの型はGLubyte(じーえる・ゆー・ばいと)、GLushort(じーえる・ゆー・しょーと)、GLuint(じーえる・ゆー・いんと)、GLuint64(じーえる・ゆー・いんと・ろくじゅうよん)のいずれかを使うことができます。ビット数の大きい型のほうが、より大きい頂点データ配列を扱うことができますが、インデックスデータの容量が増えます。今回はそれほど大きな頂点データ配列を作るつもりはありませんので、16bitのGLushortで十分でしょう。

## Index Buffer Object(IBO)の作成

頂点データを格納するときにVertex Buffer Object(VBO)というオブジェクトを作りましたね。  
同様に、インデックスデータを格納するには「Index Buffer Object(いんでっくす・ばっふぁ・おぶじぇくと、IBO)」というオブジェクトを使います。

それでは、IBOを作成するプログラムを作りましょう。  
CreateVBO関数定義の下に、次のコードを追加してください。

glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  
 glBufferData(GL\_ARRAY\_BUFFER, size, data, GL\_STATIC\_DRAW);  
 glBindBuffer(GL\_ARRAY\_BUFFER, 0);  
 return vbo;  
}  
  
**+**/\*\*  
**+**\* Index Buffer Objectを作成する.  
**+**\*  
**+**\* @param size インデックスデータのサイズ.  
**+**\* @param data インデックスデータへのポインタ.  
**+**\*  
**+**\* @return 作成したIBO.  
**+**\*/  
**+**GLuint CreateIBO(GLsizeiptr size, const GLvoid\* data)  
**+**{  
**+** GLuint ibo = 0;  
**+** glGenBuffers(1, &ibo);  
**+** glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ibo);  
**+** glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, size, data, GL\_STATIC\_DRAW);  
**+** glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);  
**+** return ibo;  
**+**}  
**+** /\*\*  
\* Vertex Array Objectを作成する.  
\*  
\* @param vbo VAOに関連付けられるVBO.

IBOを作成する手順は、VBOのときとほとんど同じです。作成するとき、GL\_ARRAY\_BUFFERではなく、GL\_ELEMENT\_ARRAY\_BUFFERを使っている点に気をつけてください。GL\_ELEMENT\_ARRAY\_BUFFERは、インデックスデータを操作するためのマクロ定数です。

## Iindex Buffer ObjectをVertex Array Objectに関連付ける

VBOと同様に、IBOもVAOに設定することができます。そこで、CreateVAO関数の引数にIBOを追加することにします。CreateVAO関数を、次のように修正してください。

/\*\*  
 \* Vertex Array Objectを作成する.  
 \*  
 \* @param vbo VAOに関連付けられるVBO.  
**+**\* @param ibo VAOに関連付けられるIBO.   
 \*  
 \* @return 作成したVAO.  
 \*/  
**-**GLuint CreateVAO(GLuint vbo)  
**+**GLuint CreateVAO(GLuint vbo, GLuint ibo)  
 {  
 GLuint vao = 0;  
 glGenVertexArrays(1, &vao);  
 glBindVertexArray(vao);  
 glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  
**+** glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ibo);  
 glEnableVertexAttribArray(0);  
 glVertexAttribPointer(0, sizeof(Vertex::position) / sizeof(float),  
 GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (const　GLvoid\*)(offsetof(Vertex, position)));  
 glEnableVertexAttribArray(1);  
 glVertexAttribPointer(1, sizeof(Vertex::color) / sizeof(float),  
 GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (const　GLvoid\*)(offsetof(Vertex, color)));  
 glBindVertexArray(0);  
 glDeleteBuffers(1, &vbo);  
**+** glDeleteBuffers(1, &ibo);  
 return vao;  
 }

それでは、ひとつずつ変更点を見ていきましょう。ひとつめの変更は、引数の説明の追加です。ふたつめの変更は、引数にIBOを追加することです。みっつめは、glBindBuffer関数を使ってVAOにIBOを関連付けるコードの追加です。最後は、glDeleteBuffers関数でIBOに削除マークを付けています。

続いて、CreateVAO関数の呼び出しを修正しましょう。main関数に移動し、CreateVAO関数を呼び出している部分を、次のように修正してください。

const GLuint vbo = CreateVBO(sizeof(vertices), vertices);  
**-**const GLuint vao = CreateVAO(vbo);  
**+**const GLuint ibo = CreateIBO(sizeof(indices), indices);  
**+**const GLuint vao = CreateVAO(vbo, ibo);  
 const GLuint shaderProgram = CreateShaderProgram(vsCode, fsCode);  
**-**if (!vbo || !vao || !shaderProgram) {  
**+**if (!vbo || !ibo || !vao || !shaderProgram) {  
　 　return 1;  
 }

まずCreateIBO関数呼び出しを追加しています。この関数の引数には、インデックスデータのサイズとポインタを渡しています。次に、CreateVAO関数の第2引数に、作成したIBOを渡しています。最後に、if文の中にiboが作成できているかのチェックを追加しています。

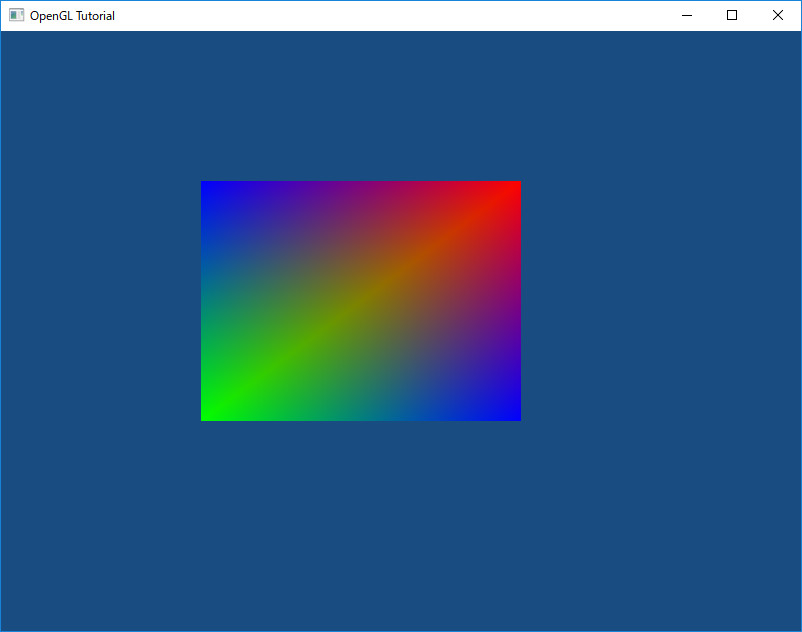
## インデックスデータを使った描画

最後に、描画関数を変更します。というのも、頂点データによる描画と、インデックスデータによる描画では、使う関数が異なるからです。頂点データの場合はglDrawArrays関数を使いますが、インデックスデータでは、glDrawElements関数が使われます。そういうことなので、メインループのコードを次のように変更してください。  
今回は新しい編集記号として「-」を使っています。「-」はその行を削除することを示します。また、削除する行であることが分かりやすいように、行全体を赤字にしてあります。

glUseProgram(shaderProgram);  
 glBindVertexArray(vao);  
**-**glDrawArrays(GL\_TRIANGLES, 0, sizeof(vertices)/sizeof(vertices[0]));  
**+**glDrawElements(GL\_TRIANGLES,  
**+** sizeof(indices)/sizeof(indices[0]), GL\_UNSIGNED\_SHORT, (const GLvoid\*)0);  
 window.SwapBuffers();

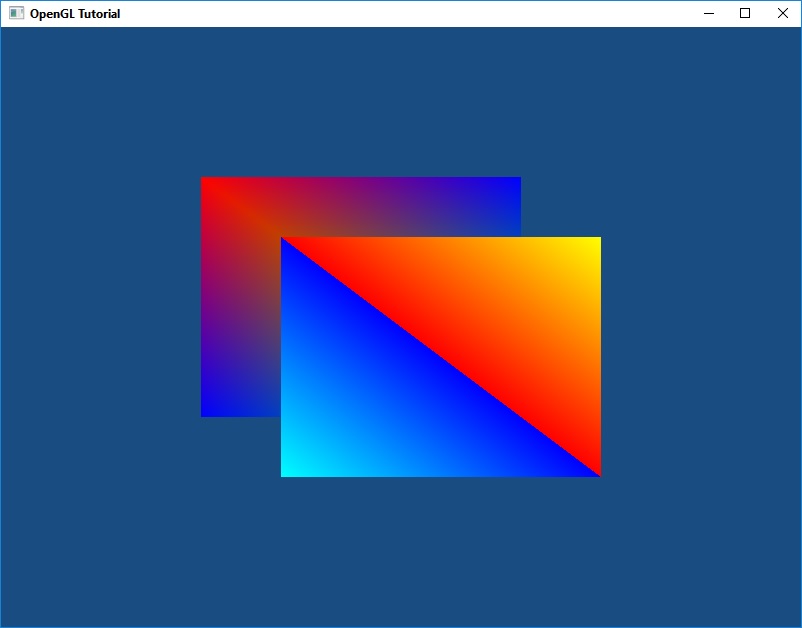
さて、glDrawElements関数の最初の引数には、glDrawArrays関数と同様にプリミティブの種類を指定します。  
2つめの引数は描画するインデックスの数です。  
3つめの引数は、インデックスの型で、GL\_UNSIGNED\_BYTE, GL\_UNSIGNED\_SHORT, GL\_UNSIGNED\_INTのいずれかを指定します。indices変数はGLuint型なので、GL\_UNSIGNED\_INTを指定しています。  
4つめの引数には、インデックスデータの描画開始オフセットを、インデックスデータの先頭からのバイト数で指定します。オフセットは整数ですが、OpenGLの歴史的な事情で、引数の型は「const GLvoid\*」になっています。そこで、キャストして型を変換しています。

ここまで実装したら、ビルドして実行してみてください。次のような画像が表示されたでしょうか。



## 頂点を共有しないことについて

表示したい図形によっては、ポリゴン間で頂点を共有したくないことがあります。  
例えば、四角形を追加して、以下の画像のように表示したいとします。



追加した四角形は、対角線にそって色が混ざらずに分かれています。このような結果を得るには、ポリゴンごとに異なる色を設定してあげなければなりません。言葉で説明されても分かりにくいと思いますので、ちょっとやってみましょう。頂点データとインデックスデータを、次のように変更してください。

const Vertex vertices[] = {  
 { {-0.5f, -0.3f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.3f, -0.3f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.3f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
 { {-0.5f, 0.5f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
  
**+** { {-0.3f, 0.3f, 0.1f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
**+** { {-0.3f, -0.5f, 0.1f}, {0.0f, 1.0f, 1.0f, 1.0f} },  
**+** { { 0.5f, -0.5f, 0.1f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
**+** { { 0.5f, -0.5f, 0.1f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
**+** { { 0.5f, 0.3f, 0.1f}, {1.0f, 1.0f, 0.0f, 1.0f} },  
**+** { {-0.3f, 0.3f, 0.1f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
**+**  
 { {-0.5f, -0.3f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.3f, -0.3f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.3f, 0.5f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
 .  
 .  
 .  
 };  
  
 const GLuint indices[] = {  
 0, 1, 2, 2, 3, 0,  
**+** 4, 5, 6, 7, 8, 9,  
 };

四角形をひとつ追加しただけなのですが、頂点データは4個ではなく6個追加されています。また、追加されたインデックスデータには重複する番号がありません。  
書けたら、ビルドして実行してください。先に示した画像のように表示されたら成功です。

OpenGLにおいて、ポリゴン内の各フラグメントの色は、各頂点の色を、頂点からの距離の比率によって混合したものになります。そのため、最初に作った四角形のように複数のポリゴンで頂点を共有している場合、境界線はあまりはっきりとは見えません。これは、基本的には望ましい動作です。というのも、現実の物体には、はっきりとした境界のない部分のほうが多いからです。

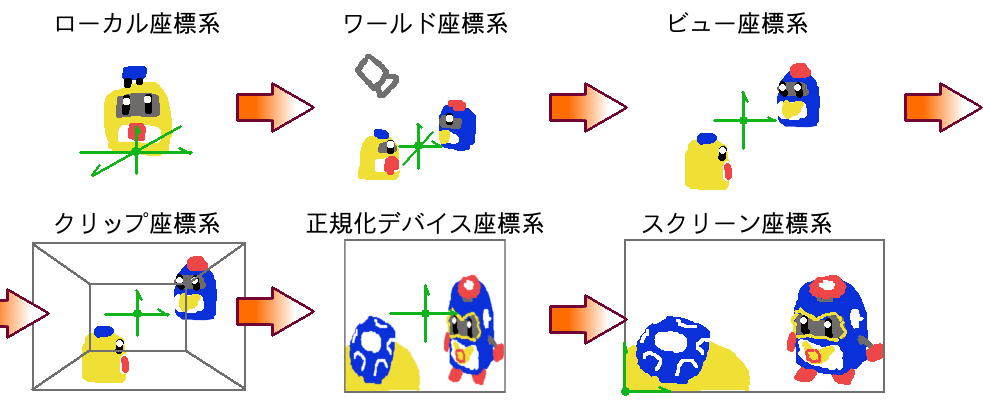
しかし、時にはポリゴン境界をはっきりとさせたい場合があります。そんなときは、ふたつめの四角形のように、ポリゴン毎に別々の頂点データを割り当て、頂点の共有はあきらめることになります。この場合、インデックスデータのぶんだけ余分なメモリが必要になってしまいます。今回追加したような小さなデータではデメリットのほうが目立ちますが、一般的にモデルの頂点数は数百から数万に及びます。そのようなモデルでは、共有される頂点数のほうが多くなるので、インデックスデータを使うほうが有利であることは変わりません。

# 座標系

## さまざまな座標系

これまで、頂点データの座標には0.5とか-0.3といった数値が使われていました。例えば最初の頂点の座標は(-0.5, 0.5, 0.5)です。前回説明したように、これはこの頂点が「クリップ座標系上のX座標-0.5、Y座標0.5、Z座標0.5の位置にある」ことを示しているわけですが、この「クリップ座標系」とは何者なのでしょう？  
それを知るには、まずOpenGLで使われる座標系について理解しなければなりません。

コンピュータグラフィックスでは、用途に応じてさまざまな座標系を使い分けます。OpenGLで使われる座標系を下図に示します。また、座標系はそれぞれ異なる原点と軸を持っています。ある座標系で定義された座標について、異なる座標系で同じ位置を示す座標を得るには、座標変換を行う必要があります。



**ローカル座標系**

ポリゴンモデルを定義する座標系です。モデル座標系、オブジェクト座標系とも呼ばれます。  
原点は、モデルの中心か足元とされることが多いです。

**ワールド座標系**

ポリゴンモデル同士の位置関係を定義する座標系です。  
原点はシーンの中心か、上からみたときにシーンの左上となる位置とされることが多いです。

**ビュー座標系**

視点を基準とする座標系です。カメラ座標系、視点(Eye)座標系とも呼ばれます。  
原点は視点の座標になります。

**クリップ座標系**

クリッピングという、画面に映らない部分を除去する処理のための座標系です。頂点シェーダ(テッセレータやジオメトリシェーダも使う場合、そのうちで最後に実行されるシェーダ)の出力はこの座標系でなければなりません。クリップ座標系はXYZ各軸が±Wの大きさを持つ空間です(検索すると±1としている文章を見かけると思いますが、概念的には正しい説明です。それでも、ほとんどのグラフィックスハードウェアの実装では±Wとして扱われます)。±Wを超えた位置にあるポリゴンはグラフィックスパイプラインから除去されます。境界にまたがるポリゴンは外側になる部分だけが除去されます。ローカル座標系、ワールド座標系、ビュー座標系からクリップ座標系への変換は、シェーダで行わなければなりません。  
この座標系の空間は、座標変換の方法によって立方体か角錐台(ピラミッドの上端を切り落としたような形)になります。  
原点は空間の中心です(通常は視点座標と一致します)。

**正規化デバイス座標系**

クリップ座標系のX,Y,Z要素をW要素で除算した座標系です。この変換はクリッピングの直後に行われ、PAへの入力になります。英語の頭文字(Normalized Device Coordinates)から、NDCとも呼ばれます。ところで、この座標系とクリップ座標系はWで除算している以外に違いはありません。実際、グラフィックスハードウェアが必要とするのは正規化デバイス座標系のほうです。それならば、クリッピングも正規化デバイス座標系でやれば良さそうに思えますが(実際、そうなっているハードウェアもあります)、わざわざ分けているのは、コンピュータにとって除算が時間のかかる処理だからです。クリッピングによって表示されなくなる頂点を除算するのは時間の無駄でしかありません。そこで、除算の前にクリッピングできるように定義されたのが、クリップ座標系というわけです。OpenGLにおいて、正規化デバイス座標系は±1の大きさを持つ空間です。クリップ座標系から正規化デバイス座標系への変換は、グラフィックスハードウェアが行います。  
この座標系の空間はXYZ各軸が±1の範囲を持つ立方体です。  
原点は空間の中心になります。

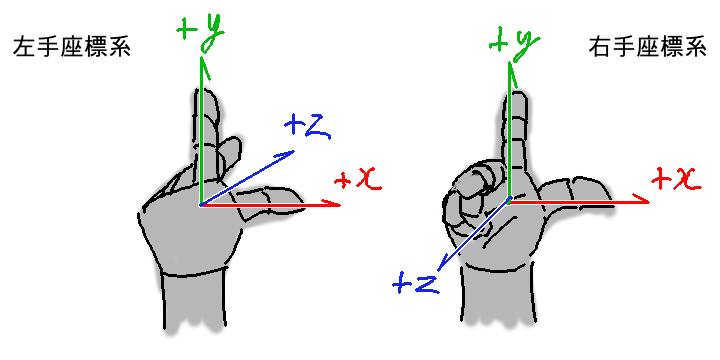
**スクリーン座標系**

フレームバッファ上の座標系です。ウィンドウ座標系とも呼ばれます。正規化デバイス座標系からスクリーン座標系への変換は、グラフィックスハードウェアが行います。  
OpenGLでは左下が原点です。

さて、頂点データを定義している空間を表す座標系はどれなのかというと、これは「クリップ座標系」になります。というのも、まだ座標変換を実装していないので、頂点シェーダの入力と出力の座標系を同じにする必要があるからです。また、いま使っている頂点シェーダではw要素には常に1を指定しています。1で除算しても値は変化しないので、クリップ座標系の座標と正規化デバイス座標系の座標は常に等しくなります。

## 右手座標系と左手座標系

ここまでは、目的に応じた座標系を説明してきました。それとは別に、各軸の向きに注目した座標系の分類があります。それが、「右手座標系」と「左手座標系」です。これは、親指から順にX,Y,Zの軸を割り当てていった時、ある座標系がどちらの手で表現できるかを示しています。



OpenGLでは伝統的に右手座標系が使われます。ところが、クリップ座標系から先は左手座標系が使われるんです。このため、シェーダでこの座標系の変換も行う必要がありますが、幸いなことに、右手座標系と左手座標系の変換はZ座標の符号を逆にするだけです。

**[補足]  
Q:** 最初から左手座標系を使うわけにはいかないの？  
**A:** 基本的にはなんの問題もありません。実際、そのほうが座標変換は分かりやすくなるでしょう。しかしながら、歴史的な事情により既存のOpenGL用プログラムやライブラリは、ほとんどが右手座標系を採用しています。そういった既存の資産を捨て去り、全てを自分で作り上げるつもりなら、どちらでも好きな座標系を使うことができます。そうでなければ、右手座標系を選ぶほうが無難でしょう。

# 座標変換

## GLMの導入

座標変換には4行4列の行列を使います。  
行列の計算は面倒かつ頻繁に登場するので、ライブラリを使うべきでしょう。今回はGLMというライブラリを使うことにします。

メニューから「プロジェクト→NuGetパッケージの管理」を選択し、ソリューションのNuGetパッケージ管理ウィンドウが開いてください。ウィンドウが開いたら「参照」を選択して、検索用テキストボックスに「glm」と入力します。  
いくつか候補が表示されると思いますが、その中から「glm」というパッケージを選び、インストールしてください。インストールが完了したら、NuGetパッケージ管理ウィンドウを閉じます。

インストールが終わったら、Main.cppにヘッダファイルをインクルードします。インクルード文を次のように変更してください。

#include "GLFWEW.h”  
**+**#include <glm/gtc/matrix\_transform.hpp>  
 #include <iostream>  
 #include <vector>

glmライブラリは用途別にヘッダファイルが分けられています。今回インクルードしているmatrix\_transform.hppには座標変換を行う関数が定義されています。

## シェーダの変更

座標変換はシェーダで行うので、シェーダコードを変更する必要があります。  
頂点シェーダを次のように書き換えてください。

static const char\* vsCode =  
 "#version 400\n"  
 "layout(location=0) in vec3 vPosition;"  
 "layout(location=1) in vec4 vColor;"  
 "layout(location=0) out vec4 outColor;"  
**+** "uniform mat4x4 matMVP;"   
 "void main() {"  
 " outColor = vColor;"  
**!** " gl\_Position = matMVP \* vec4(vPosition, 1.0);"   
 "}";

いつものように、行頭に記号が書かれているのが追加や変更のあった行です。  
さて、ひとつめの変更は、matMVPというグローバル変数が追加されていることです(MVPという名前はModel、View、Projectionの頭文字が由来です)。この変数は「uniform mat4x4」として定義されています。uniformというのは、アプリケーションから渡されるパラメータであることを示す修飾子です。そしてmat4x4は4行4列の行列のことです。行列には他にもmat4x3やmat3x3、mat2x2といったものがあります。また、正方行列(行と列の要素数が等しい行列)の場合、mat4やmat3のように、列数を省略した表記も可能です。

ふたつ目の変更は、gl\_Positionの計算です。といっても、ここでは頂点座標にmatMVPを掛けているだけです。

## 座標変換行列をシェーダに転送する

シェーダの準備ができたら、次はプログラムで座標変換行列を作成し、uniform変数に転送します。  
メインループ内のglUseProgram関数呼び出しの下に、次のコードを追加してください。

glUseProgram(shaderProgram);  
  
**+**// 座標変換行列を作成してシェーダに転送する. **+**const GLint matMVPLoc = glGetUniformLocation(shaderProgram, "matMVP");  
**+**if (matMVPLoc >= 0) {  
**+** const glm::mat4x4 matProj =  
**+** glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);  
**+** const glm::mat4x4 matView =  
**+** glm::lookAt(glm::vec3(2, 3, 3), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
**+** const glm::mat4x4 matMVP = matProj \* matView;  
**+** glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
**+**}  
  
 glBindVertexArray(vao);

アプリケーションからシェーダにデータを送るには、転送先のuniform変数の位置を取得する必要があります。それにはglGetUniformLocation関数を使います。この関数は、第1引数で指定されたプログラムオブジェクトの中から、第2引数で指定された名前のuniform変数を検索し、その位置を返します。uniform変数が見つかれば0以上の値が返ってきますが、もし同じ名前のuniform変数が見つからない場合は-1が返ってきます。見つからなかった場合はデータを転送できないので、見つかった場合のみ転送されるようにif文で制御しています。

if文の内側を見ていきましょう。最初に出くわすのは行列の計算です。  
glm::mat4x4は4行4列の行列型です。GLMはGLSLとほぼ同じコードが書けるように設計されているので、型名にも同じ名前が使えます(名前空間名で修飾する必要はありますが)。

glm::perspective関数は、透視投影を行う座標変換行列を作成します。最初の引数にはY軸方向の視野角をラジアンで指定します。ここでは45度としています。2つめの引数は、X軸方向の視野角を決めるためのアスペクト比です。通常は、「描画範囲の幅/描画範囲の高さ」を渡します。3つめと4つめの引数は、視点から角錐台の天井および底までの距離です。これはクリップ座標系のZ軸の座標0と1に対応します。

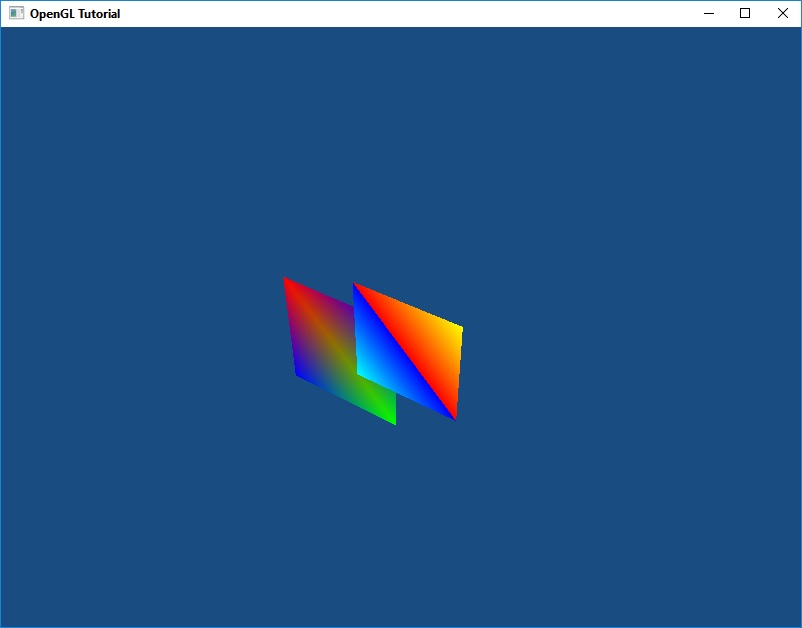
glm::lookAt関数は、ワールド座標系からビュー座標系への変換行列を作成します。最初の引数は、視点の座標です。2つめの引数は注視点の座標です。3つめの引数は視点の上方向のベクトルです。上方向ベクトルは視点の傾きを表します。みなさんの頭が視点だとすると、上方向ベクトルは頭のてっぺんからまっすぐ上にのびています。首を傾けると、上方向ベクトルも傾きます。  
つまり、上記のコードでは、「ワールド座標系の(2, 3, 3)の位置から、(0, 0, 0)を見ていて、上方向が(0, 1, 0)を向いている視点座標系」への変換行列を作っているわけです。

2つの座標変換行列を手に入れたあとは、このふたつをかけ合わせることで2つの座標変換を合成しています。  
座標に行列の積を掛けた結果は、積の元になった2つの行列を順番に掛けた場合と等しくなります(精度の問題でわずかな誤差はありますが)。

NOTE: 行列を掛ける順番はとても重要です。なぜなら、行列の乗算では交換法則が成り立たないからです。  
また、積の右辺と左辺のどちらが先に行われる変換になるかは、行列の定義に依存します。GLSLやGLMの場合、先に行われる変換は右辺です。

そして、glUniformMatrix4fv関数で、uniform変数に対して変換行列を転送します。この関数の最初の引数は、転送先のuniform変数の位置です。2つめの引数は、転送するデータの数です。3つめの引数は、行列を転置するかどうかを指定します(転置というのは行と列の内容を入れ替えることです)。4つめの引数は、転送するデータへのポインタで、ここでは「&vp[0][0]」とすることで、行列の最初の要素へのポインタを渡しています。  
また、glUniformMatrix4fv関数は、現在設定されているプログラムオブジェクトに対してデータを転送します。そのため、glUseProgramよりあとで呼び出さなければなりません。

さて、ここまで実装したコードを実行すると、次のように表示されると思います。



## 視点を動かす

四角形が斜めになったのは分かりますが、この状態では2つの図形の位置関係がさっぱり分かりません。そこで、視点を動かしてみることにします。  
glUseProgram関数呼び出しの前に、次のコードを追加してください。

glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
 **+**// 視点を回転移動させる.  
**+**static float degree = 0.0f;  
**+**degree += 0.1f;  
**+**if (degree >= 360.0f) { degree -= 360.0f; }  
**+**const glm::vec3 viewPos = glm::rotate(  
**+** glm::mat4(), glm::radians(degree), glm::vec3(0, 1, 0)) \* glm::vec4(2, 3, 3, 1);  
  
 glUseProgram(shaderProgram);

さらに、matView作成コードを次のように書き換えます。

if (matMVPLoc >= 0) {  
 const glm::mat4x4 matProj =  
 glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);  
const glm::mat4x4 matView =  
**!** glm::lookAt(viewPos, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
 const glm::mat4x4 matMVP = matProj \* matView;  
 glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
}

これで、視点がワールド座標系の原点を中心として、くるくると回り続けるようになるはずです。

## おや、奥行きのようすが…？

視点を動かしてみると、2枚の四角形の前後関係がおかしいような気がします。目の錯覚なんでしょうか…？  
いいえ、錯覚ではありません。実際おかしくなっています。  
OpenGLのプリミティブの描画順は、単純にインデックスデータで渡された順番になります。また、何もしなければピクセルはどんどん上書きされていきます。こういう仕組みなので、視点から見て奥にあるプリミティブがあとから描画されると、手前のプリミティブで隠されるべき部分まで描画されてしまうのです。

これを防ぐには、深度バッファというものを使います。

# 深度バッファを使う

深度バッファは、ピクセルごとの奥行き情報を格納しているフレームバッファです。ラスタライザは、このバッファに書かれた奥行きとフラグメントのZ値を比較し、深度バッファより手前にあると判断されたフラグメントだけを、フラグメントシェーダに送ります。

深度バッファはGLFWが自動的に作成してくれています。わたしたちがしなければならないのは、深度バッファの利用を許可することだけです。  
Main.cppに移動し、メインループの手前に次のコードを追加してください。

if (!vbo || !ibo || !vao || !shaderProgram) {  
 return 1;  
 }  
  
**+**glEnable(GL\_DEPTH\_TEST);  
  
 // メインループ.  
 while (!window.ShouldClose()) {

glEnable関数は、OpenGLが持つさまざまな機能を有効にする関数です。無効にするにはglDisable関数を使います。引数は有効にする機能を示す定数です。GL\_DEPTH\_TESTは深度バッファによる奥行きの比較(深度テスト)を示します。

引数にはさまざまな定数を指定することができますが、深度テスト以外で特に有用なのはGL\_CULL\_FACEです。GL\_CULL\_FACEは「裏面カリング(back face culling)」という機能を示す定数です(cullは「集める、(集めて)取り除く」という意味の英単語です)。ポリゴンで球や直方体を作った場合を考えてください。これらの形状が描画されるとき、裏を向いている面は表を向いている面によって隠されてしまうため、画面に表示されることはありません。表示されないのに描画するのは無駄です。「裏面カリング」が有効になっていると、OpenGLは裏向きのポリゴンを描画しなくなります。

これで、前後関係が正しく描画されるはずです。  
ビルドして実行してみてください。

# VBO、IBOを部分的に描画する

## データの追加

現在、オフスクリーンバッファとバックバッファへの描画には、同じ頂点データとインデックスデータを使っています。  
これは使い勝手がよくありません。なんとか違うデータを使うようにできないでしょうか。  
幸いなことに、glDrawElements関数には描画するインデックス数とインデックスデータの始点オフセットを指定する機能があります。この機能を使えば、インデックスバッファの一部分だけを描画することができそうです。

まずはバックバッファ用の頂点データとインデックスデータを追加しましょう。  
Main.cppを開き、vertices変数の末尾に次のデータ列を追加してください。

const Vertex vertices[] = {  
 { {-0.5f, -0.3f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {0.0f, 0.0f} },  
 { { 0.3f, -0.3f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {1.0f, 0.0f} },  
 { { 0.3f, 0.5f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {1.0f, 1.0f} },  
 { {-0.5f, 0.5f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, {0.0f, 1.0f} },  
  
 { {-0.3f, 0.3f, 0.1f}, {0.0f, 0.0f, 1.0f, 1.0f}, {0.0f, 1.0f} },  
 { {-0.3f, -0.5f, 0.1f}, {0.0f, 1.0f, 1.0f, 1.0f}, {0.0f, 0.0f} },  
 { { 0.5f, -0.5f, 0.1f}, {0.0f, 0.0f, 1.0f, 1.0f}, {1.0f, 0.0f} },  
 { { 0.5f, -0.5f, 0.1f}, {1.0f, 0.0f, 0.0f, 1.0f}, {1.0f, 0.0f} },  
 { { 0.5f, 0.3f, 0.1f}, {1.0f, 1.0f, 0.0f, 1.0f}, {1.0f, 1.0f} },  
 { {-0.3f, 0.3f, 0.1f}, {1.0f, 0.0f, 0.0f, 1.0f}, {0.0f, 1.0f} },  
  
**+** { {-1.0f,-1.0f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 1.0f, 0.0f} },  
**+** { { 1.0f,-1.0f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.0f, 0.0f} },  
**+** { { 1.0f, 1.0f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.0f, 1.0f} },  
**+** { {-1.0f, 1.0f, 0.5f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 1.0f, 1.0f} },  
 };

そして、indices変数の末尾に次のデータ列を追加してください。

const GLuint indices[] = {  
 0, 1, 2, 2, 3, 0,  
 4, 5, 6, 7, 8, 9,  
**+** 10, 11, 12, 12, 13, 10,  
 };

頂点データやインデックデータを追加している場合、それに応じてインデックスをずらしてください。

追加した頂点データの座標系は正規化デバイス座標系として扱われる点に注意してください。そして、ポリゴンが画面全体を覆うように-1～+1の範囲を設定しています。また、テクスチャ座標の並びが最初の4つの頂点データとは違っています。U座標を反転すると、画像を左右反転して表示することができます。

## RenderningPart構造体

次に、描画するインデックスデータの範囲を構造体として表現します。  
indices変数定義の下に、次のコードを追加してください。

/\*\*  
\* 部分描画データ.  
\*/  
struct RenderingPart  
{  
 GLvoid\* offset; ///< 描画開始インデックスのバイトオフセット.  
 GLsizei size; ///< 描画するインデックス数.  
};

RenderingPartは開始オフセットと描画するインデックス数だけを持つシンプルな構造体です。glDrawElements関数の流儀に従って、オフセットの型は「GLvoid\*」としています。

続いて、この構造体を使って描画範囲を定義しましょう。  
RenderingPart構造体定義の下に、次のコードを追加してください。

/\*\*  
\* RenderingPartを作成する.  
\*  
\* @param offset 描画開始インデックスのオフセット(インデックス単位).  
\* @param size 描画するインデックス数.  
\*  
\* @return 作成した部分描画オブジェクト.  
\*/  
constexpr RenderingPart MakeRenderingPart(GLsizei offset, GLsizei size) {  
 return { reinterpret\_cast<GLvoid\*>(offset \* sizeof(GLuint)), size };  
}  
  
/\*\*  
\* 部分描画データリスト.  
\*/  
static const RenderingPart renderingParts[] = {  
 MakeRenderingPart( 0, 12),  
 MakeRenderingPart( 12, 6),  
};

MakeRenderingPart関数は、RenderingPartオブジェクトを作成する関数です。  
わざわざ関数を作ったのには理由があります。RenderingPartのパラメータはglDrawElementsのインデックス数とオフセット引数に渡されますが、この2つは単位が違います。さらに、オフセットはreinterpret\_castを用いてGLvoid\*に変換する必要があります。これを毎回書き下すのは手間ですし、間違いやすいので、関数にしてしまうことにしました。

NOTE: 関数の先頭にある「constexpr」というのは、そのあとに続く関数が「コンパイル時定数」になりうることを示す識別子です。「costant expression(定数式)」を略して名付けられました。この識別子を持つ関数・クラス・値は、可能な限りコンパイル時に計算され、アプリケーションを実行するときには計算済みの値が使われます。計算をしなくていいので、実行速度の高速化が期待できます。

renderingPartsは名前のとおり、RenderingPartのリストです。  
0番目にこれまでのインデックスデータの範囲、1番めに今回追加したデータの範囲を設定しています。MakeRenderingPart関数を使うことで、オフセットはバイト単位ではなく、インデックス単位で指定できるようになっている点に注目してください(「やってみよう」等でインデックスデータを追加している場合は、ここの数値も調整してくださいね)。

## 部分描画オブジェクトを使う

それでは、部分描画データを使ってみましょう。  
2つあるglDrawElements関数呼び出しを、それぞれ次のように書き換えます。

ひとつめは、単に引数をrenderingParts変数に置き換えるだけです。

glBindBufferArray(vao);  
 glDrawElements(  
**!** GL\_TRIANGLES, renderingParts[0].size, GL\_UNSIGNED\_INT, renderingParts[0].offset);  
  
 glBindFrameBuffer(GL\_FRAMEBUFFER, 0);

ふたつめは、ちょっと手を加えます。

if (colorSamplerLoc >= 0) {  
 glBindTexture(GL\_TEXTURE\_2D, offscreen->GetTexutre());  
 }  
**+**if (matMVPLoc >= 0) {  
**+** glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &glm::mat4()[0][0]);  
**+**}  
glDrawElements(  
**!** GL\_TRIANGLES, renderingParts[1].size, GL\_UNSIGNED\_INT, renderingParts[1].offset);  
  
 window.SwapBuffers();

まずMVP行列として単位行列を渡します。単位行列とは「変換を行わない」行列です。これによって、以後の描画では、頂点データはクリップ座標系として扱われることになります。シェーダに変換行列を渡していなかった頃と同じ状態になるわけです。  
そして、ひとつめと同様にglDrawElements関数の引数をrenderingParts変数で置き換えます。こちらはインデックス1を使っていることに注意してください。

NOTE: このやりかたをさらに進めて、プリミティブの種類やデータ型もRenderingPartに格納してしまうこともできます。うまく設定すれば、最初の256個のインデックスを使うデータはGL\_UNSIGNED\_BYTEで格納し、257～65536個まではGL\_UNSIGNED\_SHORT、さらに大きいインデックスを使うデータはGL\_UNSIGNED\_INTで格納する、などという芸当も可能です。

# やってみよう

* 頂点データの座標を変更して、前後関係を変更してみましょう。
* 頂点データとインデックスデータを追加して、六角形や八角形を表示してみましょう。
* GL\_CULL\_FACEを有効にしてみましょう。

NOTE: 上記の変更を行う前にプロジェクトのバックアップを作成し、次回の講義に入る前に変更を元に戻しておいてください。