[OpenGL 3D 2018 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。  
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, -0.43f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.5f, -0.43f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.0f, 0.43f, 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, -0.43f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.5f, -0.43f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.0f, 0.43f, 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,  
**+**};  
**+** // 頂点シェーダー.  
 static const char\* vsCode =

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、次のプログラム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.

CreateIBO(くりえいと・あい・びー・おー)関数の内容は、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 = Shader::Build(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。

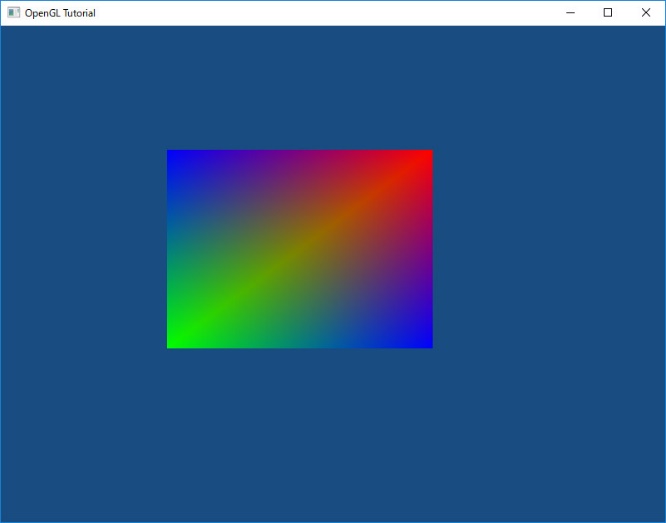
## インデックスデータを使った描画

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。

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();

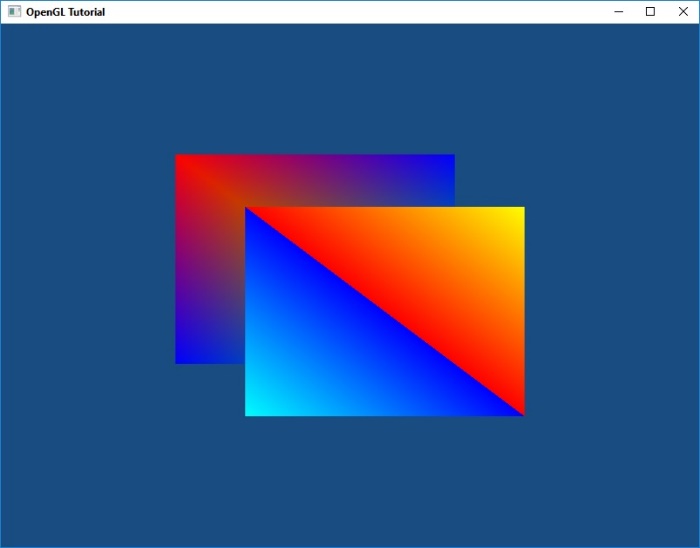
さて、glDrawElements関数の最初の引数には、glDrawArrays関数と同様にプリミティブの種類を指定します。  
2つめの引数は描画するインデックスの数です。  
3つめの引数は、インデックスの型で、GL\_UNSIGNED\_BYTE, GL\_UNSIGNED\_SHORT, GL\_UNSIGNED\_INTのいずれかを指定します。indices変数はGLushort型なので、GL\_UNSIGNED\_SHORTを指定しています。  
4つめの引数には、インデックスデータの描画開始オフセットを、インデックスデータの先頭からのバイト数(バイトオフセットといいます)で指定し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

ここまで実装したら、**ビルドして実行してください。**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, -0.43f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 { { 0.5f, -0.43f, 0.5f}, {0.0f, 1.0f, 0.0f, 1.0f} },  
 { { 0.0f, 0.43f, 0.5f}, {1.0f, 0.0f, 0.0f, 1.0f} },  
 .  
 .  
 .  
 };  
  
 const GLushort 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。

プログラムが書けたら、**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。

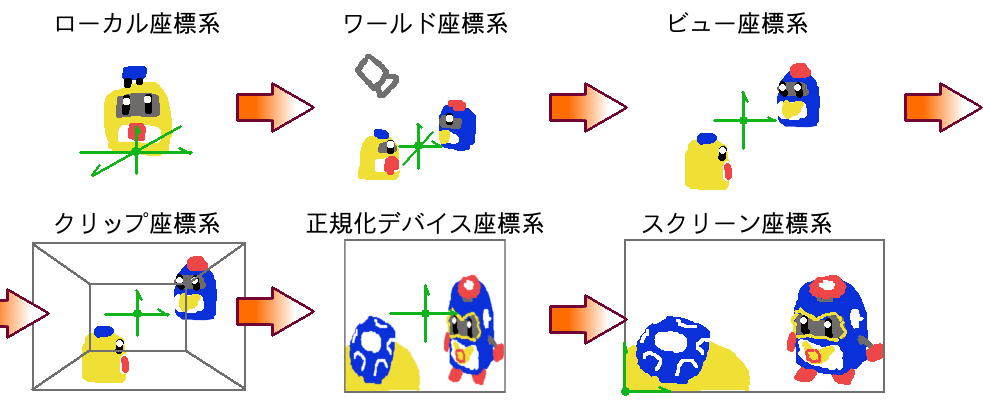
しかし、時にはポリゴン境界をはっきりとさせたい場合があります。そんなときは、ふたつめの四角形のように、ポリゴン毎に別々の頂点データを割り当て、頂点の共有はあきらめることになります。この場合、インデックスデータのぶんだけ余分なメモリが必要になってしまいます。今回追加したような小さなデータではデメリットのほうが目立ちますが、一般的にモデルの頂点数は数百から数万に及びます。そのようなモデルでは、共有される頂点数のほうが多くなるので、インデックスデータを使うほうが有利であることは変わりません。

# 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といった数値を使ってきました。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「ワールド座標系上の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「ワールド座標系」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。

「座標系」は「座標を定めるためのルール」です。このルールには「原点の位置」、「距離の単位」「軸の数や方向」などが含まれます。コンピューター・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。  
以下の図は、コンピューター・グラフィックスで使われる一般的な座標系とその関係を表しています。



**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。一般的に、頂点シェーダの入力はこの座標系になります。  
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とされることが多いです。  
ローカル座標系からワールド座標系への変換は「モデル変換」または「ワールド変換」と呼ばれます。

**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。  
ワールド座標系からビュー座標系への変換は「ビュー変換」と呼ばれます。

**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)から出力される座標は、この座標系て定義されてい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)。  
ビュー座標系からクリップ座標系への変換は「プロジェクション変換」と呼ばれます。

**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(実際に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。  
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。  
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。そのため、特に名前はありません。

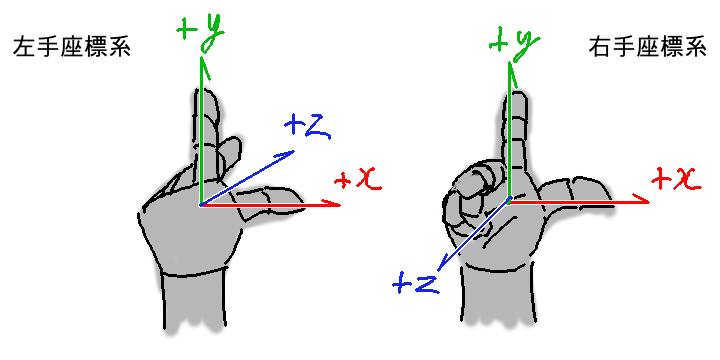
**スクリーン座標系**

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。OpenGLでは左下が原点です。  
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。そのため、特に名前はありません。

さて、頂点データを定義している空間を表す座標系は「ワールド座標系」と説明しました。しかし、実際には「ローカル座標系であり、ワールド座標系であり、ビュー座標系であり、クリップ座標系でもあり、またNDC座標系でもある」というのが正しい説明になります。こんな説明になる理由は、まだ座標変換のプログラムを書いていないので、頂点シェーダに入力する座標系は、出力される座標系と同じになっていなければ困るからです。ローカル座標系からクリップ座標系までが全て同じルールであるならば、座標変換がなくても問題ないわけですね。  
そして、前回書いてもらった頂点シェーダーでは、w要素には常に1を設定しています。1で除算しても座標は変化しないので、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。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

## OpenGL Mathematics(GLM)の導入

ここまでワールド座標系だけで暮らしてきましたが、いよいよ新たな座標系を導入していこうと思います。複数の座標系を扱うとき、ある座標系で定義された座標が、別の座標系ではどのような座標として見えるかを計算する必要が生じます。この計算のことを「座標変換」といいます。コンピューター・グラフィックスの世界では、座標変換は4行4列の行列と座標の積として表わされます。  
行列と座標、あるいは行列と行列の演算はかなり面倒です。しかも、これから頻繁に使っていくことになります。こういった計算は、ライブラリに任せるのがいいでしょう。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てください。

GLMライブラリの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 "Shader.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。

## シェーダーの変更

座標変換はシェーダーで行うので、シェーダー・プログラムを変更する必要があります。  
頂点シェーダーを次のように書き換えてください。

static const char\* vsCode =  
 "#version 410\n"  
 "layout(location=0) in vec3 vPosition; \n"  
 "layout(location=1) in vec4 vColor; \n"  
 "layout(location=0) out vec4 outColor; \n"  
**+** "uniform mat4x4 matMVP; \n"  
 "void main() { \n"  
 " outColor = vColor; \n"  
**-** " gl\_Position = vec4(vPosition, 1.0); \n"  
**+** " gl\_Position = matMVP \* vec4(vPosition, 1.0); \n"  
 "}";

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(ゆにふぉーむ)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(まっと・ふぉー・ばい・ふぉー)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(まっと・ふぉー)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。行列とベクトルの積を求めるには、乗算が16回、加算が12回も必要となります。積を計算するたびにそんなに多くの計算を書くのは面倒すぎるので、GLSLでは「\*」記号でつなげば(つまり、普通の掛け算と同じ書き方をすれば)、それらの計算を書いたのと同じ結果が得られるようになっています。簡単に書けるだけであって、実際に行われる計算回数は全く変わらないことに注意してください。

## 座標変換行列をシェーダに転送する

シェーダーの準備ができたら、次はプログラムで座標変換行列を作成し、uniform変数に転送していきます。アプリケーションからシェーダーにデータを送るには、転送先のuniform変数の位置情報が必要となりますので、まずはそれを取得するプログラムを書きましょう。  
メインループの手前に、次のプログラムを追加してください。

if (!vbo || !ibo || !vao || !shaderProgram) {  
　 　return 1;  
 }  
  
**+**// uniform変数の位置を取得する. **+**const GLint matMVPLoc = glGetUniformLocation(shaderProgram, "matMVP");  
**+**if (matMVPLoc < 0) {  
**+** std::cerr << "ERROR: uniform変数’matMVP’が見つかりません.\n";  
**+** 　return 1;  
**+**}  
**+** // メインループ. while (!window.ShouldClose()) {

uniform変数の位置を取得するにはglGetUniformLocation(じーえる・げっと・ゆにふぉーむ・ろけーしょん)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、その位置情報を返してくれます。指定された名前のuniform変数が見つかった場合、その位置を指す0以上の値が返されます。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、上記のプログラムでは、結果が負数だった場合にエラーメッセージを表示してプログラムを終了させるようにしています。

続いて、座標変換行列を作成して、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、以下のプログラム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);  
  
 glUseProgram(shaderProgram);  
**+**  
**+**// 座標変換行列を作成してシェーダーに転送する. **+**const glm::mat4x4 matProj =  
**+** glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 500.0f);  
**+**const glm::mat4x4 matView =  
**+** glm::lookAt(glm::vec3(2, 3, 3), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
**+**const glm::mat4x4 matModel = glm::mat4x4(1);  
**+**const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
**+**glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
**+**  
 glBindVertexArray(vao);

このプログラムでは、いくつかの行列を計算しています。glm::mat4x4(じーえるえむ・まっと・ふぉー・ばい・ふぉー)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)。

glm::perspective(じーえるえむ・ぱーすぺくてぃぶ)関数は、透視投影を行う座標変換行列(プロジェクション変換行列)を作成します。「透視投影」は、近くにあるほど大きく、遠くにあるほど小さくなるような変換で、みなさんが現実の世界を見るのと同じように3D空間を見るための変換方法です。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比率です。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、奥行き方向の描画範囲を設定します。今回は、Z座標が0.1～500の範囲にあるプリミティブだけが描画されるようにしました。

glm::lookAt(じーえるえむ・るっく・あっと)関数は、ワールド座標系からビュー座標系への変換行列(ビュー変換行列)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。位置と方向だけでは、この「傾ける」という動作を表現できませんよね。だから、この引数が必要なのです。  
そういうわけで、上記のプログラムは、「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ことになります。

matModel(まっと・もでる)変数は、ローカル座標系からワールド座標系への変換行列(モデル変換行列)です。現在のところ、ローカル座標系とワールド座標系は同じルールになっています。そこで、glm::mat4x4(1)という式で、掛けても何も変化しない(ルールを変えない)変換行列を作っています。掛けても何も変化しない行列のことを「単位行列」といいます(実数の1.0のようなものです)「。単位行列」は次のような、左上から右下の斜めの部分だけ1で、他はすべて0になっている行列です。

単位行列 =

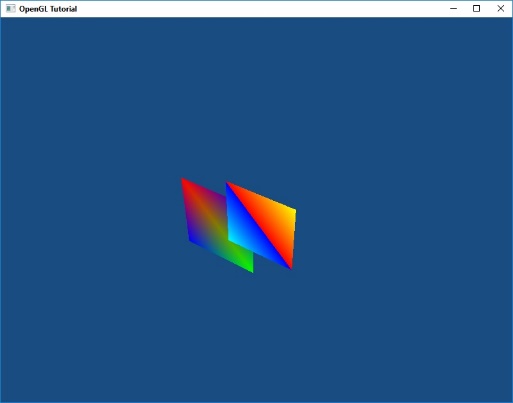
GLMライブラリを使って単位行列を作るには、実数をひとつだけ設定できるコンストラクタを使い、引数に1を設定します。

3つの座標変換行列を手に入れたあとは、それをかけ合わせることで行列を合成しています。個々の行列を座標に掛けるのではなく、行列同士の積を求めているのには理由があります。ひとつには、行列の性質のひとつに「行列の積を座標に掛けた結果は、積の元になった2つの行列を順番に座標に掛けた場合と等しい」というものがあるからです(精度の問題でわずかな誤差はあります)。この性質のおかげで、シェーダーには合成した座標変換行列をひとつ転送するだけで済ませられるのです。ふたつめの理由は、合成した行列なら行列と座標の積の計算は1回で済むので、行列と座標の積を都度計算するより速いからです。

**[補足]** 座標をv、2つの行列をmA, mBとしたとき、次の式が成り立つ、という意味です。  
 **mA × (mB × v) = (mA × mB) × v**また、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、みなさんが慣れ親しんでいる交換法則が成り立たないからです。実数または虚数の数値aとbがあるとき、  
 **a × b = b × a**  
が成り立ちますよね。だから、2×4でも4×2でも、答えは同じ8になるわけです。ですが、行列の場合は、  
 **mA × mB = mB × mA**は成り立たないので、  
 **mA × mB ≠ mB × mA**となります。  
プリミティブが思ったように表示されないときは、行列を掛ける順番が正しいかどうかを確認してみるといいでしょう。

そして、glUniformMatrix4fv(じーえる・ゆにふぉーむ・まとりくす・ふぉー・えふ・ぶい)関数で、シェーダーに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(=シェーダー)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。

それでは、**ビルドして実行してください。**次のような画像が表示されたら成功です。



## 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、次のプログラム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 += 0.01f;  
**+**if (degree >= 360.0f) { degree -= 360.0f; }  
**+**const glm::vec3 viewPos = glm::rotate(  
**+** glm::mat4(1), glm::radians(degree), glm::vec3(0, 1, 0)) \* glm::vec4(2, 3, 3, 1);  
  
 glUseProgram(shaderProgram);  
  
// 座標変換行列を作成してシェーダに転送する.

さらに、matViewを作成するプログラムを次のように書き換えてください。

// 座標変換行列を作成してシェーダに転送する.  
const glm::mat4x4 matProj =  
 glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 500.0f);  
const glm::mat4x4 matView =  
**-** glm::lookAt(glm::vec3(2, 3, 3), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
**+** 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]);

プログラムが書けたら、**ビルドして実行してください。**  
四角形がくるくると回り続けるようになると思います。

さて、ここでふたつの重要なお知らせがあります。  
座標変換行列を導入した結果、**頂点データを定義しているワールド座標のルールと、それ以外の座標系のルールは違うものになりました。**これまで、座標系を気にするようなプログラムを書いてはいなかったので、このことに気づかなかったかもしれません。特に注意してもらいたいのは、一般的なOpenGLの世界において、クリップ座標系は左座標系に属し、ワールド座標系は右手座標系に属していることです。座標変換に使っているGLMライブラリでは、OpenGLの一般的なワールド座標系を作り出します。ですから、作成しているプログラムも一般的なものに変わったわけです。これにより、Z軸の方向が逆になりました。  
加えて、以前はクリップ座標系では正方形として定義されているはずの四角形が、横長に表示されていましたよね。これは、画面サイズが800x600と横長なので、スクリーン座標系への変換の過程で横に引き伸ばされるからです。画面サイズを変えると見た目も変わってしまうようでは困るので、一般的には透視投影変換のタイミングで縦横の比率を考慮した変換行列を作るようにします。当然、GLMライブラリも比率を考慮した行列を作ってくれるようになっていますので、今後は頂点データで定義したとおりの図形を表示できるようになりました。

もうひとつ、ワールド座標系の大きさの単位を「**1メートル**」に決めました。プログラムで距離を1.0と書いたとき、それは1メートルを意味することになります。これまでは特に単位は決めておらず謎のままでしたが、これで四角形が縦横0.8mの大きさであることや、視点がどのあたりの位置にあるのかが理解しやすくなると思います。

**[補足]** ワールド座標系における1.0をどれくらいの大きさにするべきかは、表示したい内容によって異なります。  
人間やそれに近い大きさのものを表示するなら1mは分かりやすい単位でしょう。惑星規模の映像を表示したいなら1kmにしてもいいでしょう。また、小さな動物が活躍するようなゲームなら、1cm単位にすることも考えられます。  
ワールド座標の単位はプランナー、アーティスト、プログラマーが相談して決めます。

## 深度バッファを使う

さて、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…？  
いいえ、錯覚ではありません。実際に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、現実なら手前のプリミティブで隠れてしまう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」はピクセルごとの奥行き情報を格納しているメモリ領域です。ラスタライザ(RS)は、深度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。

実は、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、メインループの先頭に次のプログラムを追加してください。

// メインループ.  
 while (!window.ShouldClose()) {  
**+** glEnable(GL\_DEPTH\_TEST);  
**+**  
 glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClaer(GL\_COLOR\_BUFFER\_BIT|GL\_DEPTH\_BUFFER\_BIT);

glEnable(じーえる・いねーぶる)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(じーえる・でぃせーぶる)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(じーえる・でぷす・てすと)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(じーえる・かる・ふぇいす)でしょう。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(かる)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。

プログラムが書けたら**ビルドして実行してください。**  
正しい前後関係で描画されるはずです。

**[課題01]** 頂点データの座標を変更して、2つの四角形が交差するようにしてください。

**[課題02]** 裏面カリングを有効にしてください。

# VBO、IBOを部分的に描画する

## データの追加

現在はインデックスバッファに設定されているすべてのプリミティブを、一度のglDrawElements関数で描画しています。風景のムービーを作りたいならこのままでも構わないのですが、ゲームのようなインタラクティブ性のあるアプリケーションを作る場合は、頂点やインデックスの一部分だけを表示したり、移動したりといったことが出来るべきでしょう。  
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、部分的な描画に使うデータを追加しましょう。緑と茶色の2つの三角錐を立てに重ねて木を表現してみることにします。Main.cppを開き、vertices配列の先頭に、次のデータを追加してください。

const Vertex vertices[] = {  
**+** // 木  
**+** { { 0.00f, 5.0f, 0.00f}, {0.5f, 0.8f, 0.3f, 1.0f} },  
**+** { { 0.00f, 1.5f,-1.10f}, {0.1f, 0.3f, 0.0f, 1.0f} },  
**+** { {-0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f} },  
**+** { { 0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f} },  
**+** { { 0.00f, 4.0f, 0.00f}, {0.2f, 0.1f, 0.0f, 1.0f} },  
**+** { { 0.00f, 0.0f,-0.37f}, {0.5f, 0.3f, 0.2f, 1.0f} },  
**+** { {-0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f} },  
**+** { { 0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 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} },  
 { {-0.5f, 0.5f, 0.5f}, {0.0f, 0.0f, 1.0f, 1.0f} },  
 .  
 .  
 .  
 };

三角錐には4つの頂点が必要です。それが2つなので、合計8個の頂点データを追加しています。  
追加した頂点データはローカル座標系で定義されることに注意してください。今回作成する木のモデルでは、幹の中心が地面に接する点を原点としています。

頂点データの次はインデックスデータです。indices変数の先頭に、次のデータを追加してください。

const GLushort indices[] = {  
**+** // 木  
**+** 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3,  
**+** 4, 5, 6, 4, 6, 7, 4, 7, 5,  
**+** 0, 1, 2, 2, 3, 0,  
 4, 5, 6, 7, 8, 9,  
 };

木を構成する三角形の数は7枚です(葉っぱにあたる緑の三角錐は4枚。幹にあたる茶色の三角錐は、地面当たる部分は見えないので底面なしで3枚)。また、木はGL\_TRIANGLESプリミティブで描画する予定なので、3×7で21個のインデックスを追加しています。

## Mesh(めっしゅ)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、次のプログラムを追加してください。

0, 1, 2, 2, 3, 0,  
 4, 5, 6, 7, 8, 9,  
 };  
  
**+**/\*\*  
**+**\* 描画データ.  
**+**\*/  
**+**struct Mesh  
**+**{  
**+** GLenum mode; ///< プリミティブの種類.  
**+** GLsizei count; ///< 描画するインデックス数.  
**+** const GLvoid\* indices; ///< 描画開始インデックスのバイトオフセット.  
**+** GLint baseVertex; ///< インデックス0番とみなされる頂点配列内の位置. **+**};  
**+**  
 // 頂点シェーダー.  
 static const char\* vsCode =

Mesh(めっしゅ)は「網、網目模様の織物」という意味の単語です。ポリゴンで作られたモデルが網目模様に似ていることから、モデルの形状データを指す単語として使われています。さて、Mesh構造体には、glDrawElements関数の引数に対応するデータを格納します。glDrawElements関数の流儀に従って、オフセットの型は「const 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。  
Mesh構造体の定義の下に、次のプログラムを追加してください。

GLsizei count; ///< 描画するインデックス数.  
 const GLvoid\* indices; ///< 描画開始インデックスのバイトオフセット.  
 GLint baseVertex; ///< インデックス0とみなされる頂点配列内の位置.};  
  
**+**/\*\*  
**+**\* 描画データリスト.  
**+**\*/  
**+**const Mesh meshList[] = {  
**+** { GL\_TRIANGLES, 21, (const GLvoid\*)0, 0 }, // 木  
**+**};  
**+** // 頂点シェーダー.  
 static const char\* vsCode =

meshList(めっしゅ・りすと)は名前のとおり、Mesh構造体の配列です。  
今は、さきほど追加した木のモデルを描画するためのデータがひとつあるだけです。この構造体の最初のメンバーは描画に使用するプリミティブの種類です。先ほど作成した「木」のモデルは、GL\_TRIANGLESで描画するつもりでデータを作ったので、同じプリミティブを設定しています。次のメンバーは、描画するインデックスデータの数です。木のインデックスデータの数は21個でしたね。3つめのメンバーは、木の描画に使う「最初のインデックスデータのバイト単位の位置」です。木のインデックスデータはindices配列の先頭、つまり0バイト目にありますから、0を設定しています。4つめのメンバーは、木の描画に使う「最初の頂点データのインデックス」です。木の頂点データはvertices配列の先頭、つまり0番目にありますから、0を設定しています。

## Mesh構造体を使う

それでは、Mesh構造体を使ってみましょう。  
glDrawElements関数呼び出しを次のように書き換えてください。

glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
  
 glBindBufferArray(vao);  
**-**glDrawElements(GL\_TRIANGLES, sizeof(indices)/sizeof(indices[0]),  
**-** GL\_UNSIGNED\_SHORT, (const GLvoid\*)0);  
**+**glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
**+** GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
  
 window.SawpBuffers();

glDrawElementsBaseVertex(じーえる・どろー・えれめんつ・べーす・ばーてっくす)関数は、glDrawElements関数の兄弟で、追加の引数として「インデックスの0番とみなす頂点データの位置」を設定することができます。  
頂点データを増減させた場合に、このベース・バーテックスの値だけを変更するだけで済み、インデックス全体を書き換える手間をなくせます。

ところで、今回追加した木のモデルは四角形より大きいので、現在の視点座標からでは全体を画面に映すころができません。そこで、視点の位置も変更しましょう。視点を回転させるプログラムを、次のように変更してください。

// 視点を回転移動させる.  
static float degree = 0.0f;  
degree += 0.01f;  
if (degree >= 360.0f) { degree -= 360.0f; }  
const glm::vec3 viewPos = glm::rotate(  
**-** glm::mat4(1), glm::radians(degree), glm::vec3(0, 1, 0)) \* glm::vec4(2, 3, 3, 1);  
**+** glm::mat4(1), glm::radians(degree), glm::vec3(0, 1, 0)) \* glm::vec4(-20, 20,-30, 1);  
  
 glUseProgram(shaderProgram);  
// 座標変換行列を作成してシェーダに転送する.

書けたらビ**ルドして実行してください。**画面中央に、緑と茶色でできた細長い物体(木のつもり)が表示されていれば成功です。

## 頂点データをローカル座標系で定義する

VBOとIBOの一部だけを描画することができるようになりましたが、今のままでは最初に決めた座標に表示することしかできません。なぜなら、頂点データをワールド座標系で定義しているからです。  
好きな位置に、好きな数のメッシュを表示できるようにするには、頂点データをローカル座標系で定義しなおす必要があります。

座標系を変えるなんて大変そうに思えますが、みなさんは既に一度、経験しています。座標変換のプログラムを追加したとき、頂点データを定義する座標系は、クリップ座標系からワールド座標系に変わり、Z軸の向きが逆になりました。つまり、頂点データに手を付けなくても、変換行列を追加するだけで、結果的に望みの座標系で定義されていることにできるのです。

つまり、わたしたちがやらなければならないのは、「ローカル座標系からワールド座標系へと変換するプログラムを追加する」ことです。  
それでは、座標変換プログラムに、次のプログラムを追加してください。

// 座標変換行列を作成してシェーダに転送する.const glm::mat4x4 matProj =  
 glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 500.0f);  
const glm::mat4x4 matView =  
 glm::lookAt(viewPos, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
**-**const glm::mat4x4 matModel = glm::mat4x4(1);  
**+**const glm::mat4x4 matModel = glm::translate(glm::mat4x4(1), glm::vec3(10, 0, 1));  
const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);

glm::translate(じーえるえむ・とらんすれーと)は、平行移動行列を作成する関数です。最初の引数が合成先の行列、2つめの引数が平行移動させる距離になっています。追加したプログラムは「ローカル座標系で定義された木のモデルを、ワールド座標系の(10, 0, 1)の位置に配置する」という意味になります。

プログラムが書けたら、**ビルドして実行してください。**木が先程と違う位置に立っていたら成功です。頂点データは何も変えていないのに、位置を変えられるということが、ローカル座標系を追加した効果です。

## 複数の木を植える

木が1本だけではさびしいので、もっとたくさん植えたいですね。  
好きな場所に好きな数だけ木を植えるには、木を植えるたびにローカル座標系からワールド座標系への変換行列を書き換えるだけです。例として、木を円形に植えてみましょう。  
木を表示するプログラムを、次のように変更してください。

// 座標変換行列を作成してシェーダに転送する.const glm::mat4x4 matProj =  
 glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 500.0f);  
 const glm::mat4x4 matView =  
 glm::lookAt(viewPos, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
**-**const glm::mat4x4 matModel = glm::translate(glm::mat4x4(1), glm::vec3(10, 0, 1));  
**-**const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
**-**glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
**-**  
**-**glBindVertexArray(vao);  
**-**glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
**-** GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
 **+**glBindVertexArray(vao);  
**+  
+**const float treeCount = 10; // 木の本数.  
**+**const float radius = 8; // 木を植える円の半径.  
**+**for (float i = 0; i < treeCount; ++i) {  
**+** const float x = std::cos(3.14f \* 2 / treeCount \* i) \* radius;  
**+** const float z = std::sin(3.14f \* 2 / treeCount \* i) \* radius;  
**+** const glm::mat4x4 matModel = glm::translate(glm::mat4(1), glm::vec3(x, 0, z));  
**+** const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
**+** glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
**+** glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
**+** GL\_UNSIGNED\_SHORT, meshList[0].indices,meshList[0].baseVertex);  
**+**}  
**+**  
 window.SwapBuffers();

このプログラムは、木のモデルをtreeCount(つりー・かうんと)本、半径radius(れでぃあす)メートルの円周上に植樹します。

プログラムが書けたら、**ビルドして実行してください。**10本の木が植えられていたら成功です。

## 家を建てる

木だけではさびしいので、家を建てましょう。  
頂点データ配列に、次のデータを追加してください。

const Vertex vertices[] = {  
 // 木  
 { { 0.00f, 5.0f, 0.00f}, {0.5f, 0.8f, 0.3f, 1.0f} },  
 { { 0.00f, 1.5f,-1.10f}, {0.1f, 0.3f, 0.0f, 1.0f} },  
 { {-0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f} },  
 { { 0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f} },  
 { { 0.00f, 4.0f, 0.00f}, {0.2f, 0.1f, 0.0f, 1.0f} },  
 { { 0.00f, 0.0f,-0.37f}, {0.5f, 0.3f, 0.2f, 1.0f} },  
 { {-0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f} },  
 { { 0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f} },  
  
**+** // 家  
**+** { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**+** { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**+** { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f} },  
**+** { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**+** { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**+**  
**+** { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**+** { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**+** { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f} },  
**+** { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f} },  
**+** { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f} },  
**+**  
**+** { { 3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**+** { { 0.0f, 6.0f, 3.0f}, {0.3f, 0.2f, 0.2f, 1.0f} },  
**+** { {-3.0f, 4.0f, 3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**+** { {-3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**+** { { 0.0f, 6.0f,-3.0f}, {0.3f, 0.2f, 0.2f, 1.0f} },  
**+** { { 3.0f, 4.0f,-3.0f}, {0.2f, 0.1f, 0.1f, 1.0f} },  
**+  
+** // 四角形 { {-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} },

さらに、indices変数の先頭に次のデータを追加してください。

const GLushort indices[] = {  
 // 木  
 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3,  
 4, 5, 6, 4, 6, 7, 4, 7, 5,  
  
**+** // 家  
**+** 0, 1, 3, 3, 4, 0, 1, 2, 3,  
**+** 5, 6, 8, 8, 9, 5, 6, 7, 8,  
**+** 9, 8, 1, 1, 0, 9,  
**+** 4, 3, 6, 6, 5, 4,  
**+** 15, 14, 11, 11, 10, 15,  
**+** 12, 11, 14, 14, 13, 12,  
**+  
+** // 四角形 0, 1, 2, 2, 3, 0,  
 4, 5, 6, 7, 8, 9,  
 };

頂点データとインデックスデータを追加したら、それを使うための描画データを作ります。  
描画データリストに次のプログラムを追加してください。

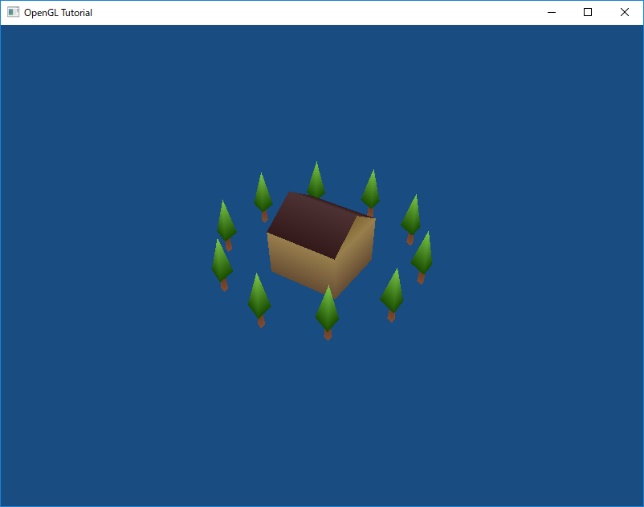
/\*\*  
\* 描画データリスト.  
\*/  
const Mesh meshList[] = {  
 { GL\_TRIANGLES, 21, (const GLvoid\*)0, 0 }, // 木  
**+** { GL\_TRIANGLES, 42, (const GLvoid\*)(21 \* sizeof(GLushort)), 8 }, // 家  
};

家は14枚の三角形でできています。ですから、インデックスデータは3×14=42個になります。そして、家のデータは木の直後に追加しましたから、家のインデックスデータのバイト位置は「木のインデックスデータ数(21個)\*データ一個のバイト数(GLushort型のバイト数)」となるわけです。同様に、家の頂点データのインデックスは、0～7番目が木の頂点なので、その直後の8番目からが家の頂点データになるわけです。

最後に、追加した描画データを使って家を建てます。メインループに次のプログラムを追加してください。

const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
 glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
 glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices,meshList[0].baseVertex);  
 }  
  
**+**{  
**+** const glm::mat4x4 matModel = glm::translate(glm::mat4(1), glm::vec3(0, 0, 0));  
**+** const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
**+** glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
**+** glDrawElementsBaseVertex(meshList[1].mode, meshList[1].count,  
**+** GL\_UNSIGNED\_SHORT, meshList[1].indices,meshList[1].baseVertex);  
**+**}  
**+**  
 window.SwapBuffers();

家を建てる場所はワールド座標の原点としました。  
プログラムが書けたら、**ビルドして実行してください。**  
木に囲まれた中央に家が建てられていれば成功です。



**[課題03]** 家をあと2軒表示してください。位置はどこでも構いませんが、他の家や木に重ならないようにしてください。

**[課題04]** 岩のモデルを作成して、1軒目の家の横に表示してください。  
直方体を作って少し形状をくずす程度のもので十分です。