[OpenGL講義 第08回]

あなたのクラスは何ですか？

# プログラム・オブジェクトのクラス化

## クラスの名前を決める

テクスチャにMVP行列にライトと、順調にモデルを描画するために設定すべき変数が増えてきました。  
しかし、モデルの描画に必要なプログラムは、異なるモデルであっても大きな違いはありません。こういうときは、プログラムを関数にまとめてしまうのが定石ですが、今回はそれをさらに進めて、C++のクラスとしてまとめていこうと思います。

クラスを作るとき最初に考えるのは、「クラスの目的」です。つまり、「これから作るクラスに何をさせたいのか。」ということです。今回の場合、「モデルを表示するためのさまざまなプログラムを簡単に扱えるようにしたい 。」ということになるでしょうが、「モデルを手軽に表示したい。」というのが目的になります。  
次に、それらのプログラムが扱う対象を調べて、それらを扱うのにふさわしいクラス名を考えます。  
例えば、モデルを表示するためには以下の作業が必要です。

- プログラム・オブジェクトの設定  
- VAOの設定  
-テクスチャの設定  
-ライトの設定  
- MVP行列の設定  
-モデルの描画

これらは結局のところ、特定のプログラム・オブジェクトの管理下にあるGPUメモリに向けて、データを送り込む操作にほかなりません。ですから、クラス名はProgramObject…としたいところですが、OpenGLでは猫も杓子も「オブジェクト」なので「Object」の部分がなくても意味は通じるはずです。ということで、クラス名は「Program」としましょう。  
とはいえ、「Program」という名前は名詞として一般的すぎます。そこで名前空間と組み合わせることを考えます。現在、プログラム・オブジェクトを作成する関数はShader名前空間に定義しています。ですからProgramクラスもShader名前空間に定義するのが良さそうです。  
つまり、名前空間名を加えたクラス名は「Shader::Program」になるわけです。これなら、何を表現するためのクラスなのかが十分に伝わりそうですね。

早速クラスを作りましょう。Shader.hに次のプログラムを追加してください。

namespace Shader {  
  
 GLuint Build(const GLchar\* vsCode, const GLchar\* fsCode);  
 GLuint BuildFromFile(const char\* vsPath, const char\* fsPath);  
  
+/\*\*  
+\* シェーダー・プログラム.  
+\*/  
+class Program  
+{  
+};  
+  
 } // namespace Shader  
  
 #endif // SHADER\_H\_INCLUDED

## コンストラクタとデストラクタの宣言

次に、Programクラスに必要なメンバ関数を考えます。まずコンストラクタとデストラクタを決めましょう。コンストラクタやデストラクタをどのように定義するかは、プログラム全体の設計方針に関わります。今回は、コンストラクタでプログラム・オブジェクトのIDを受け取り、デストラクタで受け取ったプログラム・オブジェクトを破棄するという形にしてみます。それでは、Programクラスに次のプログラムを追加してください。

/\*\*  
 \* シェーダー・プログラム.  
 \*/  
 class Program  
 {  
+public:  
+ explicit Program(GLuint programId);  
+ ~Program();  
+  
+private:  
+　　GLuint id; // プログラムID.  
 };

コンストラクタにはexplicit(えくすぷりしっと)というキーワードを指定しています。1個の引数だけを受け取るコンストラクタは「変換コンストラクタ」と呼ばれ、暗黙の型変換の対象とみなされます。コンストラクタにexplicitキーワードを使うと、暗黙の型変換の対象外であると宣言したことになります。  
暗黙の型変換の何が問題なのか、次の例を考えてみましょう。

class Program  
{  
public:  
 Program(GLuint programId);  
};  
  
void DoSomethingToTex(GLuint texId);  
void DoSomethingToProg(const Program& programId);  
…  
Program prog;  
GLuint texId;  
…  
DoSomethingToTex(texId);  
DoSomethingToProg(texId); **/\* !? \*/**

上記のプログラム例では、DoSomethingToProgという関数にtexIdを設定してしまっています(コピペしたときに書き換え忘れたのでしょうか)。しかし、このプログラムのビルドは成功します。

種明かしはこうです。DoSomethingToProg関数の引数はconst Program&ですから、そのままではtexIdを設定できません。しかし、ProgramにはGLuintを受け付ける変換コンストラクタが定義されています。そこで、コンピューターはこのコンストラクタを使ってtexIdをProgram型に変換するプログラムを挿入し(これが暗黙の型変換です)、その結果、上記のプログラムはめでたくビルドに成功する、というわけです。

当然ながら、テクスチャIDをプログラム・オブジェクトのIDとして使うと、実行時に不可思議なエラーを起こしてプログラムが止まってしまいます。ですから、こういったことが起こらないようにしたいですよね。そこでexplicitキーワードの出番です。次のようにexplicitを指定すると、texIdを指定しても、暗黙の型変換が起こらないのでめでたくエラーになるというわけです。

class Program  
{  
public:  
 explicit Program(GLuint programId); **/\* explicitキーワードを追加 \*/**  
};  
  
void DoSomethingToTex(GLuint texId);  
void DoSomethingToProg(const Program& programId);  
…  
Program prog;  
GLuint texId;  
…  
DoSomethingToTex(texId);  
DoSomethingToProg(texId); **/\* ここでビルドに失敗する \*/**

コンストラクタに対してデストラクタでは特に変わったことはしていません。メンバ変数には、プログラム・オブジェクトのIDを持たせます。この変数にはコンストラクタの引数をコピーしておき、必要に応じてOpenGL関数の引数として使います。

## メンバ関数の宣言

残りは、モデルの描画に必要な機能は先に列挙済みですので、あとはメンバ関数名、引数、戻り値を決めるだけです。必要なのは以下の6つでした。

- プログラム・オブジェクトの設定  
- VAOの設定  
-テクスチャの設定  
-ライトの設定  
- MVP行列の設定  
-モデルの描画

「プログラム・オブジェクトの設定」は、glUseProgram関数のことです。これは「Use(ゆーず)」という名前でいいでしょう。プログラムIDをメンバ変数として保持しているので、引数は不要です。

「VAOの設定」は、glBindVertexArray関数のことです。VAOはVertex Puller(VP)に設定されるパラメータで、プログラム・オブジェクトからは独立しています。事実、使用するプログラム・オブジェクトが変わっても、使用するVAOは変化しません。つまり、「VAOの設定」はプログラム・オブジェクトの管轄外なのです。ですから、この機能はメンバ関数にはしません。通常の関数のままにしておきましょう。

「テクスチャの設定」はglActiveTextureとglBindTexture関数のことです。関数名は「BindTexture(ばいんど・てくすちゃ)」でいいと思います。2つの関数を合わせて3つの引数が必要ですが、当面は2Dテクスチャしか使わない予定なので、2つの引数を設定できれば十分でしょう。

「ライトの設定」は、glUniform3fv関数を用いてライトのデータをGPUメモリに転送するプログラムのことです。設定するライトデータは多岐にわたるため、構造体にまとめておくほうがよさそうです。この構造体は「LightList(らいと・りすと)」という名前にして、メンバ関数名は「SetLightList(せっと・らいと・りすと)」としましょう。引数はLightListのポインタか参照になるでしょう。

「MVP行列の設定」は、glUniformMatrix4fv関数を用いてMVP行列をGPUメモリに転送することです。とはいえ、モデル行列は描画のたびに変化します。ですから、この処理は「モデルの描画」としてまとめてしまうほうがよさそうです。ただ、ビュー行列とプロジェクション行列は変化しないので、この2つは一度設定すれば十分なはずです。そこで、この2つを乗じた行列を引数として受け取るメンバ関数を作ります。関数名は「SetViewProjectionMatrix(せっと・びゅー・ぷろじぇくしょん・まとりくす)」としましょう。

「モデルの描画」は、glDrawElementsBaseVertex関数の呼び出しのことです。関数名はずばり「Draw(どろー)」でいいと思います(シンプルにできるならそのほうがよいのです)。Mesh構造体のメンバ変数を設定するだけですから、Mesh構造体のポインタか参照を引数にしましょう。モデル行列も引数として渡したいのですが、ライトのための逆行列を作る関係で、モデル行列そのものを引数にするわけにはいきません。そこで、モデル行列のもとになる移動量・拡大縮小率・回転角の3つのパラメーターを引数にします。

さて、メンバ関数を宣言する前に、いくつかやっておくことがあります。まず、行列やベクトルを扱うことになるので、GLMライブラリから適切なヘッダファイルをインクルードしなければなりません。  
Shader.hを開き、次のようにインクルード文を追加してください。

/\*\*  
 \* @file Shader.h  
 \*/  
 #ifndef SHADER\_H\_INCLUDED  
 #define SHADER\_H\_INCLUDED  
 #include <GL/glew.h>  
+#include <glm/vec3.hpp>  
+#include <glm/mat4x4.hpp>  
  
 namespace Shader {

Mesh構造体は先行宣言だけ追加します。  
mat4x4.hppのインクルード文の下に、次のプログラムを追加してください。

/\*\*  
 \* @file Shader.h  
 \*/  
 #ifndef SHADER\_H\_INCLUDED  
 #define SHADER\_H\_INCLUDED  
 #include <GL/glew.h>  
 #include <glm/vec3.hpp>  
 #include <glm/mat4x4.hpp>  
  
+struct Mesh;  
+  
 namespace Shader {

次に、ライトの構造体を作らなけばなりません。  
BuildFromFile関数定義の下に、次のプログラムを追加してください。FragmentLighting.fragにあるライト用構造体とほぼ同じなので、コピーするといいでしょう。

GLuint Build(const GLchar\* vsCode, const GLchar\* fsCode);  
 GLuint BuildFromFile(const char\* vsPath, const char\* fsPath);  
  
+/\*\*  
+\* 環境光.  
+\*/  
+struct AmbientLight  
+{  
+ glm::vec3 color;  
+};  
+  
+/\*\*  
+\* 指向性ライト.  
+\*/  
+struct DirectionalLight  
+{  
+ glm::vec3 direction;  
+ glm::vec3 color;  
+};  
+  
+/\*\*  
+\* ポイントライト.  
+\*/  
+struct PointLight  
+{  
+ glm::vec3 position[8];  
+ glm::vec3 color[8];  
+};  
+  
+/\*\*  
+\* スポットライト.  
+\*/  
+struct SpotLight  
+{  
+ glm::vec4 dirAndCutOff[4];  
+ glm::vec4 posAndInnerCutOff[4];  
+ glm::vec3 color[4];  
+};  
+  
+/\*\*  
+\* ライトをまとめた構造体.  
+\*/  
+struct LightList  
+{  
+ AmbientLight ambient;  
+ DirectionalLight directional;  
+ PointLight point;  
+ PointLight spot;  
+};  
+  
 /\*\*  
 \* シェーダー・プログラム.  
 \*/  
 class Program

続いて、メンバ関数を宣言しましょう。Programクラスに、次のプログラムを追加してください。

class Program  
 {  
 public:  
 explicit Program(GLuint programId);  
 ~Program();  
+  
+ void Use();  
+ void BindTexture(GLuint, GLuint);  
+ void SetLightList(const LightList&);  
+ void SetViewProjectionMatrix(const glm::mat4&);  
+ void Draw(const Mesh& mesh,  
+ const glm::vec3& translate, const glm::vec3& rotate, const glm::vec3& scale);  
  
 private:  
　　 GLuint id; // プログラムID.  
 };

## メンバ変数の定義

次は、どんなメンバ変数が必要かを考えていきます。ライトやMVP行列をGPUメモリに転送するには、転送先uniform変数の位置を知って置かなければなりません。ですから、位置を格納する変数が必要です。  
また、SetLightListメンバ関数とSetViewProjectionMatrixメンバ関数の引数は、Drawメンバ関数が実行されるまで保持できなければなりません。そのため、ライトデータとビュー・プロジェクション行列もメンバ変数に必要でしょう。それでは、Programクラスに次のプログラムを追加してください。

　 void SetLightList(const LightList&);  
 　 void SetViewProjectionMatrix(const glm::mat4&);  
 void Draw(const Mesh& mesh,  
 const glm::vec3& translate, const glm::vec3& rotate, const glm::vec3& scale);  
  
　private:  
　 GLuint id; // プログラムID.  
+  
+ // uniform変数の位置.  
+ GLint locMatMVP;  
+ GLint locAmbLightCol;  
+ GLint locDirLightDir;  
+ GLint locDirLightCol;  
+ GLint locPointLightPos;  
+ GLint locPointLightCol;  
+ GLint locSpotLightDir;  
+ GLint locSpotLightPos;  
+ GLint locSpotLightCol;  
+  
+ glm::mat4 matVP; // ビュー・プロジェクション行列.  
+ LightList lights;  
　};

これでProgramクラスの定義は完了です。

## 頂点データ型をファイルに分ける

Mesh構造体は宣言しかしていないので、名前しか分かりません。しかし、メッシュを描画するためには、構造体にどのようなメンバ変数があるかが分かっていなければなりません。さらに、メンバ変数の型であるVector3やColorやVertexといった構造体の知識も必要です。ただし、現在これらはMain.cppで定義されているので、Shader.cppからは見えません。そこで、必要な構造体をファイルに分離してインクルードできるようにしましょう。

Srcフォルダに「Geometry.h(じおめとり・どっと・えいち)」というヘッダーファイルを追加してください(Geometry=幾何学、幾何学的形状)。そして、追加したGeometry.hを開き、次のプログラムを追加してください。

+/\*\*  
+\* @file Geometry.h  
+\*/  
+#ifndef GEOMETRY\_H\_INCLUDED  
+#define GEOMETRY\_H\_INCLUDED  
+#include <GL/glew.h>  
+  
+#endif // GEOMETRY\_H\_INCLUDED

次に、Main.cppからVector2, Vector3, Color, Vertex構造体を切り取って、

/\*\*  
 \* @file Main.cpp  
 \*/  
 #include "GLFWEW.h"  
 #include "Shader.h"  
 #include "Texture.h"  
 #include <glm/gtc/matrix\_transform.hpp>  
 #include <iostream>  
 #include <vector>  
  
-/// 2Dベクトル型.  
-struct Vector2  
-{  
- float x, y;  
-};  
-  
-/// 3Dベクトル型.  
-struct Vector3  
-{  
- float x, y, z;  
-};  
-  
-/// RGBAカラー型.  
-struct Color  
-{  
- float r, g, b, a;  
-};  
-  
-/// 頂点データ型.  
-struct Vertex  
-{  
- Vector3 position; ///< 座標  
- Color color; ///< 色  
- Vector2 texCoord; ///< テクスチャ座標.  
- Vector3 normal; ///< 法線.  
-};  
-  
 /// 頂点データ.  
 const Vertex vertices[] = {

Geometry.hに貼り付けてください。

#ifndef GEOMETRY\_H\_INCLUDED  
 #define GEOMETRY\_H\_INCLUDED  
 #include <GL\glew.h>  
   
+/// 2Dベクトル型.  
+struct Vector2  
+{  
+ float x, y;  
+};  
+  
+/// 3Dベクトル型.  
+struct Vector3  
+{  
+ float x, y, z;  
+};  
+  
+/// RGBAカラー型.  
+struct Color  
+{  
+ float r, g, b, a;  
+};  
+  
+/// 頂点データ型.  
+struct Vertex  
+{  
+ Vector3 position; ///< 座標  
+ Color color; ///< 色  
+ Vector2 texCoord; ///< テクスチャ座標.  
+ Vector3 normal; ///< 法線.  
+};  
+  
#endif // GEOMETRY\_H\_INCLUDED

同様にMesh構造体を切り取り、

0, 1, 2, 2, 3, 0,  
 4, 5, 6, 7, 8, 9,  
};  
  
-/\*\*  
-\* ポリゴン表示単位.  
-\*/  
-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とみなされる頂点配列内の位置.  
-};  
-  
/\*\*  
\* メッシュ配列.  
\*/  
const Mesh meshList[] = {

Geometry.hに貼り付けてください。

/// 頂点データ型.  
 struct Vertex  
 {  
 Vector3 position; ///< 座標  
 Color color; ///< 色  
 Vector2 texCoord; ///< テクスチャ座標.  
 Vector3 normal; ///< 法線.  
 };  
  
+/\*\*  
+\* ポリゴン表示単位.  
+\*/  
+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とみなされる頂点配列内の位置.  
+};  
+  
#endif // GEOMETRY\_H\_INCLUDED

さて、構造体を奪われたMain.cppは、このままではビルドできません。そこで、作成したばかりのGeometry.hをインクルードします。Main.cppに次のプログラムを追加してください。

　/\*\*  
　\* @file Main.cpp  
　\*/  
　#include "GLFWEW.h"  
　#include "Shader.h"  
　#include "Texture.h"  
+#include "Geometry.h"  
　#include <glm/gtc/matrix\_transform.hpp>  
　#include <iostream>

そして、当初の目的どおりShader.cppにもインクルードしましょう。さらに今回は行列の計算を行うので、matrix\_transform.hppもインクルードしておきます。Shader.cppに次のプログラムを追加してください。

/\*\*  
 \* @file Shader.cpp  
 \*/  
 #include "Shader.h"  
+#include "Geometry.h"  
+#include <glm/gtc/matrix\_transform.hpp>  
 #include <vector>  
 #include <iostream>  
 #include <fstream>

これでメンバ関数を定義する準備ができました。

## コンストラクタの定義

それでは、メンバ関数を定義していきましょう。まずはコンストラクタからです。Shader.cppを開き、BuildFromFile関数定義の下に、次のプログラムを追加してください。

GLuint BuildFromFile(const char\* vsPath, const char\* fsPath)  
 {  
 const std::vector<GLchar> vsCode = ReadFile(vsPath);  
 const std::vector<GLchar> fsCode = ReadFile(fsPath);  
 return Build(vsCode.data(), fsCode.data());  
 }  
  
+/\*\*  
+\* コンストラクタ.  
+\*  
+\* @param programId プログラム・オブジェクトのID.  
+\*/  
+Program::Program(GLuint programId) : id(programId)  
+{  
+ locMatMVP = glGetUniformLocation(id, "matMVP");  
+ locAmbLightCol = glGetUniformLocation(id, "ambientLight.color");  
+ locDirLightDir = glGetUniformLocation(id, "directionalLight.direction");  
+ locDirLightCol = glGetUniformLocation(id, "directionalLight.color");  
+ locPointLightPos = glGetUniformLocation(id, "pointLight.position");  
+ locPointLightCol = glGetUniformLocation(id, "pointLight.color");  
+ locSpotLightDir = glGetUniformLocation(id, "spotLight.dirAndCutOff");  
+ locSpotLightPos = glGetUniformLocation(id, "spotLight.posAndInnerCutOff");  
+ locSpotLightCol = glGetUniformLocation(id, "spotLight.color");  
+  
+ const GLint texColorLoc = glGetUniformLocation(id, "texColor");  
+ if (texColorLoc >= 0) {  
+ glUseProgram(id);  
+ glUniform1i(texColorLoc, 0);  
+ glUseProgram(0);  
+ }  
+}  
+  
 } // namespace Shader

コンストラクタでは、glGetUniformLocation関数を使い、全てのuniform変数の位置を取得してメンバ変数に代入します。それから、glUniform1i関数を使ってtexColorサンプラーを0番目のテクスチャ・イメージ・ユニットに割り当てます。

## デストラクタの定義

次はデストラクタです。Programコンストラクタ定義の下に、次のプログラムを追加してください。

if (texColorLoc >= 0) {  
 glUseProgram(id);  
 glUniform1i(texColorLoc, 0);  
 glUseProgram(0);  
 }  
 }  
  
+/\*\*  
+\* デストラクタ.  
+\*  
+\* プログラム・オブジェクトを削除する.  
+\*/  
+Program::~Program()  
+{  
+ if (id) {  
+ glDeleteProgram(id);  
+ }  
+}  
+  
} // namespace Shader

デストラクタではglDeleteProgram関数を使ってプログラム・オブジェクトを破棄します。デストラクタはProgramクラスの変数が削除されるときに自動的に呼び出されます。ですから、このクラスを使う限りglDeleteProgramを手書きする必要はありません。

## Use関数の定義

次はUse関数を定義します。デストラクタ定義の下に、次のプログラムを追加してください。

Program::~Program() {  
 if (id) {  
 glDeleteProgram(id);  
 }  
 }  
  
+/\*\*  
+\* プログラム・オブジェクトをグラフィックス・パイプラインに割り当てる.  
+\*/  
+void Program::Use()  
+{  
+ if (id) {  
+ glUseProgram(id);  
+ }  
+}  
+  
} // namespace Shader

この関数はglUseProgram関数を呼び出すだけです。

## BindTexture関数の定義

続いて、Use関数定義の下に、次のプログラムを追加してください。

void Program::Use()  
 {  
 if (id) {  
 glUseProgram(id);  
 }  
 }  
  
+/\*\*  
+\* 描画に使用するテクスチャを設定する.  
+\*  
+\* @param unitNo 設定するテクスチャ・イメージ・ユニットの番号(0～).  
+\* @param texId 設定するテクスチャのID.  
+\*/  
+void Program::BindTexture(GLuint unitNo, GLuint texId)  
+{  
+ glActiveTexture(GL\_TEXTURE0 + unitNo);  
+ glBindTexture(GL\_TEXTURE\_2D, texId);  
+}  
+  
} // namespace Shader

GL\_TEXTURE0などの定数はテクスチャ・イメージ・ユニットを示します。テクスチャ・イメージ・ユニット用の定数名の数値と定数自身の値には相関があり、GL\_TEXTURE0から連続した番号が割り当てられることが保証されています。おかげで、上記コードのように数値を加算して設定することが可能なのです。

## SetLightList関数の定義

BindTexture関数定義の下に、次のプログラムを追加してください。

void Program::BindTexture(GLuint unitNo, GLuint texId)  
 {  
 glActiveTexture(GL\_TEXTURE0 + unitNo);  
 glBindTexture(GL\_TEXTURE\_2D, texId);  
 }  
  
+/\*\*  
+\* 描画に使われるライトを設定する.  
+\*  
+\* @param lights 設定するライト.  
+\*/  
+void Program::SetLightList(const LightList& lights)  
+{  
+ this->lights = lights;  
+  
+ // ライトの色情報をGPUメモリに転送する.  
+ if (locAmbLightCol >= 0) {  
+ glUniform3fv(locAmbLightCol, 1, &lights.ambient.color.x);  
+ }  
+ if (locDirLightCol >= 0) {  
+ glUniform3fv(locDirLightCol, 1, &lights.directional.color.x);  
+ }  
+ if (locPointLightCol >= 0) {  
+ glUniform3fv(locPointLightCol, 8, &lights.point.color[0].x);  
+ }  
+ if (locSpotLightCol >= 0) {  
+ glUniform3fv(locSpotLightCol, 4, &lights.spot.color[0].x);  
+ }  
+}  
+  
} // namespace Shader

最初に、引数で渡された値をライト変数に代入します。そして、位置を示す変数が0以上なら、対応するuniform変数にライトデータを転送します。

## SetViewProjectionMatrix関数の定義

SetLightList関数定義の下に、次のプログラムを追加してください。

if (locSpotLightCol >= 0) {  
 glUniform3fv(locSpotLightCol, 4, &lights.spot.color[0].x);  
 }  
 }  
  
+/\*\*  
+\* 描画に使われるビュー・プロジェクション行列を設定する.  
+\*  
+\* @param matVP 設定するビュー・プロジェクション行列.  
+\*/  
+void Program::SetViewProjectionMatrix(const glm::mat4& matVP)  
+{  
+ this->matVP = matVP;  
+}  
+  
} // namespace Shader

この関数は引数の値をmatVPメンバ変数に代入するだけです。このプログラムの興味深い点は、引数の名前とメンバ変数の名前が同じになっていることでしょう。引数(またはローカル変数)に、メンバ変数と同じ名前を付けた場合、その名前は引数(またはローカル変数)を指すようになります。同名のメンバ変数を扱いたいときは、「this->」をつけてメンバ変数であることを明示しなければなりません。

## Draw関数の定義

最後は描画を行う関数です。この関数は少し長いので、雛形から始めましょう。SetViewProjectionMatrix関数定義の下に、次のプログラムを追加してください。

void Program::SetViewProjectionMatrix(const glm::mat4& matVP)  
 {  
 this->matVP = matVP;  
 }  
  
+/\*\*  
+\* メッシュを描画する.  
+\*  
+\* @param mesh 描画するメッシュ.  
+\* @param translate平行移動量.  
+\* @param rotate 回転角度(ラジアン).  
+\* @param scale 拡大縮小率(1=等倍, 0.5=1/2倍, 2.0=2倍).  
+\*  
+\* この関数を使う前に、Use()を実行しておくこと.  
+\*/  
+void Program::Draw(const Mesh& mesh,  
+ const glm::vec3& translate, const glm::vec3& rotate, const glm::vec3& scale)  
+{  
+}  
+  
 } // namespace Shader

それでは、Draw関数に次のプログラムを追加してください。

void Program::Draw(const Mesh& mesh,  
 const glm::vec3& translate, const glm::vec3& rotate, const glm::vec3& scale)  
 {  
+ // モデル行列を計算する.  
+ const glm::mat4 matScale = glm::scale(glm::mat4(1), scale);  
+ const glm::mat4 matRotateZ = glm::rotate(glm::mat4(1), rotate.z,glm::vec3(0, 0, -1));  
+ const glm::mat4 matRotateXZ = glm::rotate(matRotateZ, rotate.x, glm::vec3(1, 0, 0));  
+ const glm::mat4 matRotateYXZ = glm::rotate(matRotateXZ, rotate.y,glm::vec3(0, 1, 0));  
+ const glm::mat4 matTranslate = glm::translate(glm::mat4(1), translate);  
+ const glm::mat4 matModel = matTranslate \* matRotateYXZ \* matScale;  
 }  
  
 } // namespace Shader

最初はモデル行列の作成です。まず拡大縮小行列(matScale)、回転行列(matRotateYXZ)、平行移動行列(matTranslate)の3つを計算し、それらをかけ合わせてモデル行列(matModel)を計算します。  
回転行列の順序が「Y軸回転→X軸回転→Z軸回転」になっているのは、経験的に回転後の姿勢を想像しやすいとされているからです。また、Z軸の向きが(0,0,-1)なのは、OpenGLでは奥側がZ軸のマイナス方向だからですね。

続いてMVP行列を計算します。モデル行列を計算するプログラムの下に、次のプログラムを追加してください。

const glm::mat4 matRotateYXZ = glm::rotate(matRotateXZ, rotate.y,glm::vec3(0, 0, 1));  
 const glm::mat4 matTranslate = glm::translate(glm::mat4(1), translate);  
 const glm::mat4 matModel = matTranslate \* matRotateYXZ \* matScale;  
+  
+ // モデル・ビュー・プロジェクション行列を計算し、GPUメモリに転送する.  
+ const glm::mat4 matMVP = matVP \* matModel;  
+ glUniformMatrix4fv(locMatMVP, 1, GL\_FALSE, &matMVP[0][0]);  
 }  
  
 } // namespace Shader

VP行列はSetViewProjectionMatrix関数で受け取りますので、ここではmatVPにmatModelをかけあわせるだけです。

次に、モデル座標系におけるライトの方向を計算します。MVP行列を計算するプログラムの下に、次のプログラムを追加してください。

// モデル・ビュー・プロジェクション行列を計算し、GPUメモリに転送する.  
 const glm::mat4 matMVP = matVP \* matModel;  
 glUniformMatrix4fv(locMatMVP, 1, GL\_FALSE, &matMVP[0][0]);  
+  
+ // 指向性ライトの向きをモデル座標系に変換してGPUメモリに転送する.  
+ const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateYXZ));  
+ if (locDirLightDir >= 0) {  
+ const glm::vec3 dirLightDirOnModel = matInvRotate \* lights.directional.direction;  
+ glUniform3fv(locDirLightDir, 1, &dirLightDirOnModel.x);  
+ }  
+ // モデル座標系におけるポイントライトの座標を計算し、GPUメモリに転送する.  
+ const glm::mat4 matInvModel = glm::inverse(matModel);  
+ if (locPointLightPos >= 0) {  
+ glm::vec3 pointLightPosOnModel[8];  
+ for (int i = 0; i < 8; ++i) {  
+ pointLightPosOnModel[i] = matInvModel \* glm::vec4(lights.point.position[i], 1);  
+ }  
+ glUniform3fv(locPointLightPos, 8, &pointLightPosOnModel[0].x);  
+ }  
+ // モデル座標系におけるスポットライトの座標を計算し、GPUメモリに転送する.  
+ if (locSpotLightDir >= 0 && locSpotLightPos) {  
+ glm::vec4 spotLightDirOnModel[4];  
+ glm::vec4 spotLightPosOnModel[4];  
+ for (int i = 0; i < 4; ++i) {  
+ const glm::vec3 invDir = matInvRotate \* glm::vec3(lights.spot.dirAndCutOff[i]);  
+ spotLightDirOnModel[i] = glm::vec4(invDir, lights.spot.dirAndCutOff[i].w);  
+ const glm::vec3 pos = lights.spot.posAndInnerCutOff[i];  
+ spotLightPosOnModel[i] = matInvModel \* glm::vec4(pos, 1);  
+ spotLightPosOnModel[i].w = lights.spot.posAndInnerCutOff[i]w;  
+ }  
+ glUniform4fv(locSpotLightDir, 4, &spotLightDirOnModel[0].x);  
+ glUniform4fv(locSpotLightPos, 4, &spotLightPosOnModel[0].x);  
+ }  
 }  
  
 } // namespace Shader

このプログラムは、Main.cppにあるライトの計算をコピーし、変数名などを変更したものです。

最後に描画関数を呼び出します。ライトデータをGPUメモリに転送するプログラムの下に、次のプログラムを追加してください。

glUniform4fv(locSpotLightDir, 4, &spotLightDirOnModel[0].x);  
 glUniform4fv(locSpotLightPos, 4, &spotLightPosOnModel[0].x);  
 }  
  
+ // メッシュを描画する.  
+ glDrawElementsBaseVertex(  
+ mesh.mode, mesh.count, GL\_UNSIGNED\_SHORT, mesh.indices, mesh.baseVertex);  
+}  
  
 } // namespace Shader

これでメンバ関数の定義は完了です。

## Shader::Programクラスを使う

それでは、作成したクラスを使ってみましょう。  
Main.cppを開き、フラグメント・ライティング・シェーダーを作成するプログラムを、次のように変更してください。

const GLuint vbo = CreateVBO(sizeof(vertices), vertices);  
 const GLuint ibo = CreateIBO(sizeof(indices), indices);  
 const GLuint vao = CreateVAO(vbo, ibo);  
 const GLuint shaderProgram =  
 Shader::BuildFromFile("Res/Simple.vert", "Res/Simple.frag");  
-const GLuint progLighting =  
+const GLuint pidLighting =  
 Shader::BuildFromFile("Res/FragmentLighting.vert", "Res/FragmentLighting.frag");  
-if (!vbo || !ibo || !vao || !shaderProgram || !progLighting) {  
+if (!vbo || !ibo || !vao || !shaderProgram || !pidLighting) {  
 return 1;  
 }

そして、uniform変数の位置を取得するプログラムを次のように変更してください。

const GLint texColorLoc = glGetUniformLocation(shaderProgram, "texColor");  
 if (texColorLoc >= 0) {  
 glUniform1i(texColorLoc, 0);  
 }  
  
-const GLint locMatMVP = glGetUniformLocation(progLighting, "matMVP");  
-const GLint locAmbLightCol =  
- glGetUniformLocation(progLighting, "ambientLight.color");  
-const GLint locDirLightDir =  
- glGetUniformLocation(progLighting, "directionalLight.direction");  
-const GLint locDirLightCol =  
- glGetUniformLocation(progLighting, "directionalLight.color");  
-const GLint locPointLightPos =  
- glGetUniformLocation(progLighting, "pointLight.position");  
-const GLint locPointLightCol =  
- glGetUniformLocation(progLighting, "pointLight.color");  
-const GLint locSpotLightDir =  
- glGetUniformLocation(progLighting, "spotLight.dirAndCutOff");  
-const GLint locSpotLightPos =  
- glGetUniformLocation(progLighting, "spotLight.posAndInnerCutOff");  
-const GLint locSpotLightCol =  
- glGetUniformLocation(progLighting, "spotLight.color");  
-if (locDirLightDir < 0 || locDirLightCol < 0 || locAmbLightCol < 0 ||  
- locPointLightPos < 0 || locPointLightCol < 0 ||  
- locSpotLightDir < 0 || locSpotLightPos < 0 || locSpotLightCol < 0) {  
- std::cerr << "ERROR: uniform変数の位置を取得できません.\n";  
- return 1;  
-}  
-{  
- glUseProgram(progLighting);  
- const GLint locTexCol = glGetUniformLocation(progLighting, "texColor");  
- if (locTexCol >= 0) {  
- glUniform1i(locTexCol, 0);  
- }  
-}  
-  
-glUseProgram(0);  
+Shader::Program progLighting(pidLighting);  
  
 // テクスチャを作成する.  
 const int imageWidth = 8; // 画像の幅.  
 const int imageHeight = 8; // 画像の高さ.

それから、glUseProgram関数をUse関数で置き換えます。glUseProgram関数の呼び出し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(-20, 30, -30, 1);  
  
-glUseProgram(progLighting);  
+progLighting.Use();  
  
 // 座標変換行列を作成する.  
 const glm::mat4x4 matProj =  
 glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 500.0f);

## ライトの変数をLightList構造体で置き換える

次に、ライトの変数にLightList構造体を使うように変更します。  
ライトの設定をしているプログラムを、次のように変更してください。

GLuint texId = Texture::CreateImage2D(tw, th, imageData, GL\_RGBA, GL\_UNSIGNED\_BYTE);  
 GLuint texHouse = Texture::LoadImage2D("Res/House.tga");  
  
 // ライトの設定.  
+Shader::LightList lights;  
-const glm::vec3 ambLightCol = glm::vec3(0.05f, 0.1f, 0.1f); // 環境光の色.  
+lights.ambient.color = glm::vec3(0.05f, 0.1f, 0.1f); // 環境光の色.  
-const glm::vec3 dirLightDir = glm::normalize(glm::vec3(5,-2,-2)); // 指向性ライトの方向.  
-const glm::vec3 dirLightCol = glm::vec3(1.0f, 1.0f, 1.0f); // 指向性ライトの色.  
+lights.directional.direction = glm::normalize(glm::vec3(5, -2, -2)); // 指向性ライトの方向.  
+lights.directional.color = glm::vec3(1.0f, 1.0f, 1.0f); // 指向性ライトの色.  
-glm::vec3 pointLightPos[8] = {};  
-glm::vec3 pointLightCol[8] = {};  
-pointLightPos[0] = glm::vec3(5, 4, 0);  
-pointLightCol[0] = glm::vec3(1.0f, 0.8f, 0.4f) \* 100.0f;  
+lights.point.position[0] = glm::vec3(5, 4, 0);  
+lights.point.color[0] = glm::vec3(1.0f, 0.8f, 0.4f) \* 100.0f;  
-glm::vec3 spotLightDirAndCutOff[4] = {};  
-glm::vec4 spotLightPosAndInnerCutOff[4] = {};  
-glm::vec3 spotLightCol[4] = {};  
 const glm::vec3 tmpSpotLightDir = glm::normalize(glm::vec3(-1,-2,-2));  
-spotLightDirAndCutOff[0] = glm::vec4(tmpSpotLightDir, std::cos(glm::radians(20.0f));  
-spotLightPosAndInnerCutOff[0] = glm::vec4(-6, 6, 8, std::cos(glm::radians(15.0f));  
-spotLightCol[0] = glm::vec3(0.4f, 0.8f, 1.0f) \* 200.0f;  
+lights.spot.dirAndCutOff[0] = glm::vec4(tmpSpotLightDir, std::cos(glm::radians(20.0f));  
+lights.spot.posAndInnerCutOff[0] = glm::vec4(-6, 6, 8, std::cos(glm::radians(15.0f));  
+lights.spot.color[0] = glm::vec3(0.4f, 0.8f, 1.0f) \* 200.0f;  
  
 // メインループ.  
 while (!window.ShouldClose()) {

ライトの変数名を変更したので、ライトを移動させるプログラムも新しい名前に対応させます。  
ライトを動かすプログラムを、次のように変更してください。

// 0番のポイント・ライトを移動する.  
 const float speed = 0.05f;  
 if (window.IsKeyPressed(GLFW\_KEY\_A)) {  
- pointLightPos[0].x -= speed;  
+ lights.point.position[0].x -= speed;  
 } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
- pointLightPos[0].x += speed;  
+ lights.point.position[0].x += speed;  
 }  
 if (window.IsKeyPressed(GLFW\_KEY\_LEFT\_SHIFT)) {  
 if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
- pointLightPos[0].y += speed;  
+ lights.point.position[0].y += speed;  
 } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
- pointLightPos[0].y -= speed;  
+ lights.point.position[0].y -= speed;  
 }  
 } else {  
 if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
- pointLightPos[0].z -= speed;  
+ lights.point.position[0].z -= speed;  
 } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
- pointLightPos[0].z += speed;  
+ lights.point.position[0].z += speed;  
 }  
 }

さらに、ビュー・プロジェクション行列の設定、光源の設定、VAOのバインド、テクスチャの設定を置き換えます。光源の設定プログラムを、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));  
+progLighting.SetViewProjectionMatrix(matProj \* matView);  
  
-// ライトをGPUメモリに転送する.  
-glUniform3fv(locAmbLightCol, 1, &ambLightCol.x);  
-glUniform3fv(locDirLightDir, 1, &dirLightDir.x);  
-glUniform3fv(locDirLightCol, 1, &dirLightCol.x);  
-glUniform3fv(locPointLightCol, 8, &pointLightCol[0].x);  
-glUniform3fv(locSpotLightCol, 4, &spotLightCol[0].x);  
+progLighting.SetLightList(lights);  
  
 glBindVertexArray(vao);  
-glActiveTexture(GL\_TEXTURE0);  
-glBindTexture(GL\_TEXTURE\_2D, texId);  
+  
+progLighting.BindTexture(0, texId);  
  
 const float treeCount = 10; // 木を植える本数.  
 const float radius = 8; // 半径.  
 for (float i = 0; i < treeCount; ++i) {

## 木の描画にShader::Programを使うように変更する

続いて、木を描画するプログラムを次のように変更してください。

const float treeCount = 10; // 木を植える本数.  
 const float radius = 8; // 半径.  
 for (float i = 0; i < treeCount; ++i) {  
 const float theta = 3.14f \* 2 / treeCount \* i;  
 const float x = std::cos(theta) \* radius;  
 const float z = std::sin(theta) \* radius;  
- const glm::mat4x4 matModelT = glm::translate(glm::mat4(1), glm::vec3(x, 0, z));  
- const glm::mat4x4 matModelR =  
- glm::rotate(glm::mat4(1), theta \* 5, glm::vec3(0, 1, 0));  
- const glm::mat4x4 matMVP = matProj \* matView \* matModelT \* matModelR;  
- glUniformMatrix4fv(locMatMVP, 1, GL\_FALSE, &matMVP[0][0]);  
-  
- // 指向性ライトの向きをモデル座標系に変換してGPUメモリに転送する.  
- const glm::vec3 dirLightDirOnModel =  
- glm::inverse(glm::mat3x3(matModelR)) \* dirLightDir;  
- glUniform3fv(locDirLightDir, 1, &dirLightDirOnModel.x);  
-  
- // ポイント・ライトの位置をモデル座標系に変換してGPUメモリに転送する.  
- const glm::mat4 matInvModel = glm::inverse(matModelT \* matModelR);  
- glm::vec3 pointLightPosOnModel[8];  
- for (int i = 0; i < 8; ++i) {  
- pointLightPosOnModel[i] = matInvModel \* glm::vec4(pointLightPos[i], 1);  
- }  
- glUniform3fv(locPointLightPos, 8, &pointLightPosOnModel[0].x);  
-  
- // スポット・ライトの方向と位置をモデル座標系に変換してGPUメモリに転送する.  
- glm::vec4 spotLightDirOnModel[4];  
- glm::vec4 spotLightPosOnModel[4];  
- for (int i = 0; i < 4; ++i) {  
- const glm::vec3 invDir = matInvRotate \* glm::vec3(spotLightDirAndCutOff[i]);  
- spotLightDirOnModel[i] = glm::vec4(invDir, spotLightDirAndCutOff[i].w);  
- const glm::vec3 pos = glm::vec3(spotLightPosAndInnerCutOff[i]);  
- spotLightPosOnModel[i] = matInvModel \* glm::vec4(pos, 1);  
- spotLightPosOnModel[i].w = spotLightPosAndInnerCutOff[i].w;  
- }  
- glUniform4fv(locSpotLightDir, 4, &spotLightDirOnModel[0].x);  
- glUniform4fv(locSpotLightPos, 4, &spotLightPosOnModel[0].x);  
-  
- glDrawElementsBaseVertex(meshList[0].mode,  
- meshList[0].count, GL\_UNSIGNED\_SHORT,meshList[0].indices,meshList[0].baseVertex);  
+ progLighting.Draw(meshList[0],  
+ glm::vec3(x, 0, z), glm::vec3(0, theta \* 5, 0), glm::vec3(1));  
 }

Shader::Program型を使う場合、MVP行列を計算してGPUメモリに転送する必要はありません。また、ライトの位置や向きを計算してGPUメモリに転送するプログラムも必要ありません。なぜなら、それらの作業は全てDrawメンバ関数の中で行われるからです。逆に言うと、それ以外の部分はやってくれませんので、残しておかなければなりません。上記のプログラムの場合は、木を植える座標(x, z)の計算と、植える角度(theta)の計算は消していません。

**[課題01]** 地面など、木以外でprogLightingを使用しているプログラムについて、Drawメンバ関数を使って描画するように変更してください。木の描画プログラムに行った変更を参考にしてください。

## プログラム・オブジェクト削除プログラムを削除

最後に、メインループのあとでプログラム・オブジェクトを削除しているプログラムを削除します。  
メインループの下にある、オブジェクトを削除するプログラムを、次のように変更してください。

window.SwapBuffers();  
 }  
  
 glDeleteTextures(1, &texHouse);  
 glDeleteTextures(1, &texId);  
-glDeleteProgram(progLighting);  
 glDeleteProgram(shaderProgram);  
 glDeleteVertexArrays(1, &vao);  
  
 return 0;

これで、プログラム・オブジェクトをクラス化することができました。  
**ビルドして実行してください。**クラス化する前と同じように表示されていれば成功です。   
なお、課題01を完了させていないとビルドに失敗します。課題を終えてからビルドしてください。

# モデルデータを追加しやすくする

## MeshListクラスを定義する

メッシュリストにモデルを追加するには、モデルが使用する頂点データとインデックスデータの位置を正確に設定しなければなりません。当然のことではありますが、正確な位置を設定するのは意外と面倒です。それに、あるモデルの頂点やインデックスの数が変化したら、それ以降のデータを使っているメッシュは全てデータを書き直さなければなりません。こういったことを手作業で行うのは、面倒なうえに間違えやすいものです。そこで、モデルデータをもう少し扱いやすくしていきましょう。

まず、メッシュリストをクラスにします。クラスの名前は分かりやすいことが大事ですから、素直にMeshList(めっしゅ・りすと)としましょう。また、これ以上Main.cppを煩雑にしないため、このクラスは別のファイルに定義することにします。  
というわけで、ソリューションエクスプローラーの「ヘッダーファイル」フィルタを右クリックし、「追加→新しい項目」を選択して、Srcフォルダに「MeshList.h」という名前のファイルを追加してください。そして、追加したMeshList.hを開き、次のプログラムを追加してください。

+/\*\*  
+\* @file MeshList.h  
+\*/  
+#ifndef MESHLIST\_H\_INCLUDED  
+#define MESHLIST\_H\_INCLUDED  
+#include <GL/glew.h>  
+#include "Geometry.h”  
+#include <vector>  
+  
+/\*\*  
+\* メッシュ管理クラス.  
+\*/  
+class MeshList  
+{  
+public:  
+ MeshList();  
+ ~MeshList();  
+  
+ bool Allocate();  
+ void Free();  
+ void Add(const Vertex\*, const Vertex\*, const GLushort\*, const GLushort\*);  
+  
+ void BindVertexArray();  
+ const Mesh& Get(size\_t index) const;  
+  
+private:  
+ GLuint vao = 0;  
+ std::vector<Mesh> meshes;  
+  
+ std::vector<Vertex> tmpVertices;  
+ std::vector<GLushort> tmpIndices;  
+};  
+  
+#endif // MESHLIST\_H\_INCLUDED

最初のコメントのあとにインクルード・ガードがあり、次にヘッダーファイルをインクルードしています。インクルードしているファイルはOpenGLの機能を使うためのglew.hと、Vertex構造体やMesh構造体を使うためのGeometry.h、それからstd::vectorクラスを使うためのvectorヘッダです。

MeshListクラスのメンバ関数は、コンストラクタ、デストラクタ、Allocate(あろけーと=割り当てる)、Free(ふりー=解放する)、Add(あど=追加する)、BindVertexArray(ばいんど・ばーてっくす・あれい)、Get(げっと)の7つです。コンストラクタとデストラクタは、クラスの生成と破棄を行います。Allocateメンバ関数は、モデル・データからMeshのリストを作成します。Freeメンバ関数は、Allocateメンバ関数が作成したリストを破棄します。Addメンバ関数はモデルデータを追加します。BindVertexArrayメンバ関数はグラフィックス・パイプラインにVAOを割り当てます。Getメンバ関数は、指定された番号のMeshデータを取得します。

モデルデータを管理するためにはVBO, IBO, VAOが必要ですが、VBOとIBOは、VAOを作成するときだけIDが分かっていればいいので、VAOだけをメンバ変数にしています。そして、Mesh構造体の配列を管理するのがvector<Mesh>型の変数meshes(めっしず)です。また、tmpVertices(てんぷ・ばーてぃしーず)とtmpIndices(てんぷ・いんでぃしーず)は、VAOを作成するとき、一時的にモデル・データを貯めておくために使います(tmpはtemporary(=一時的な)の省略形です)。

## モデルデータを移動する

まずは定義を描くためのファイルを追加しましょう。ソリューションエクスプローラーの「ソースファイル」フィルタを右クリックして「追加→新しい項目」を選択し、Srcフォルダに「MeshList.cpp」という名前のファイルを追加してください。追加したMeshList.cppを開き、次のプログラムを追加してください。

+/\*\*  
+\* @file MeshList.cpp  
+\*/  
+#include "MeshList.h"  
+#include <iostream>  
+

頂点データとインデックスデータをMain.cppからMeshList.cppに移動します。  
まずは移動先となる変数を定義しましょう。  
MeshList.cppに次のプログラムを追加してください。

/\*\*  
 \* @file MeshList.cpp  
 \*/  
 #include "MeshList.h"  
 #include <iostream>  
  
+const Vertex vTree[] = {  
+};  
+  
+const GLushort iTree[] = {  
+};  
+  
+const Vertex vHouse[] = {  
+};  
+  
+const GLushort iHouse[] = {  
+};  
+  
+const Vertex vRock[] = {  
+};  
+  
+const GLushort iRock[] = {  
+};  
+  
+const Vertex vGround[] = {  
+};  
+  
+const GLushort iGround[] = {  
+};

手始めに木のモデルデータを移動します。Main.cppの木の頂点データを切り取り、

/// 頂点データ.  
 const Vertex vertices[] = {  
 // 木  
- { { 0.00f, 5.0f, 0.00f}, {0.5f, 0.8f, 0.3f, 1.0f}, { 0.0f, 1.0f },  
- { 0.00f, 1.00f, 0.00f } },  
- { { 0.00f, 1.5f,-1.10f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.0f, 0.0f },  
- { 0.00f,-0.49f,-0.87f } },  
- { {-0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.0f, 0.0f },  
- {-0.76f,-0.49f, 0.43f } },  
- { { 0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.5f, 0.0f },  
- { 0.76f,-0.49f, 0.43f } },  
- { { 0.00f, 4.0f, 0.00f}, {0.2f, 0.1f, 0.0f, 1.0f}, { 1.0f, 1.0f },  
- { 0.00f, 1.00f, 0.00f } },  
- { { 0.00f, 0.0f,-0.37f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 0.0f, 0.0f },  
- { 0.00f,-0.49f,-0.87f } },  
- { {-0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 0.5f, 0.0f },  
- {-0.76f,-0.49f, 0.43f } },  
- { { 0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f },  
- { 0.76f,-0.49f, 0.43f } },  
  
 // 家

MeshList.cppのvTree配列変数に貼り付けてください。

namespace Model {  
  
 const Vertex vTree[] = {  
+ { { 0.00f, 5.0f, 0.00f}, {0.5f, 0.8f, 0.3f, 1.0f}, { 0.0f, 1.0f },  
+ { 0.00f, 1.00f, 0.00f } },  
+ { { 0.00f, 1.5f,-1.10f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.0f, 0.0f },  
+ { 0.00f,-0.49f,-0.87f } },  
+ { {-0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.0f, 0.0f },  
+ {-0.76f,-0.49f, 0.43f } },  
+ { { 0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.5f, 0.0f },  
+ { 0.76f,-0.49f, 0.43f } },  
+ { { 0.00f, 4.0f, 0.00f}, {0.2f, 0.1f, 0.0f, 1.0f}, { 1.0f, 1.0f },  
+ { 0.00f, 1.00f, 0.00f } },  
+ { { 0.00f, 0.0f,-0.37f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 0.0f, 0.0f },  
+ { 0.00f,-0.49f,-0.87f } },  
+ { {-0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 0.5f, 0.0f },  
+ {-0.76f,-0.49f, 0.43f } },  
+ { { 0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f },  
+ { 0.76f,-0.49f, 0.43f } },  
 };  
  
 const GLushort iTree[] = {  
 };

同様に、Main.cppにある木のインデックスデータを切り取り、

/// インデックスデータ.  
 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,

MeshList.cppのiTree配列変数に貼り付けてください。

{ { 0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f },  
 { 0.76f,-0.49f, 0.43f } },  
 };  
  
 const GLushort iTree[] = {  
+ 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3,  
+ 4, 5, 6, 4, 6, 7, 4, 7, 5,  
 };  
  
 const Vertex vHouse[] = {  
 };

次は家のモデルを移動します。Main.cppの家の頂点データを切り取り、

/// 頂点データ.  
 const Vertex vertices[] = {  
 // 木  
  
 // 家  
- { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 1.000f, 0.00f } },  
- { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 1.000f, 0.69f } },  
- { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.875f, 1.00f } },  
- { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.750f, 0.69f } },  
- { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.750f, 0.00f } },  
-  
- { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.500f, 0.00f } },  
- { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.500f, 0.69f } },  
- { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.375f, 1.00f } },  
- { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.250f, 0.69f } },  
- { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.250f, 0.00f } },  
-  
- { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.000f, 0.00f } },  
- { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.000f, 0.69f } },  
- { { 0.0f, 6.0f, 3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.750f, 1.00f } },  
- { { 0.0f, 6.0f,-3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.500f, 1.00f } },  
- { { 0.0f, 6.0f,-3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.250f, 1.00f } },  
- { { 0.0f, 6.0f, 3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.000f, 1.00f } },  
  
 // 岩

MeshList.cppのvHouse配列変数に貼り付けてください。

　const GLushort iTree[] = {  
 　 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 2, 3,  
　 4, 5, 6, 4, 6, 7, 4, 7, 5,  
 };  
  
　const Vertex vHouse[] = {  
+ { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 1.000f, 0.00f } },  
+ { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 1.000f, 0.69f } },  
+ { { 0.0f, 6.0f, 3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.875f, 1.00f } },  
+ { {-3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.750f, 0.69f } },  
+ { {-2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.750f, 0.00f } },  
+  
+ { {-2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.500f, 0.00f } },  
+ { {-3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.500f, 0.69f } },  
+ { { 0.0f, 6.0f,-3.0f}, {0.5f, 0.4f, 0.2f, 1.0f}, { 0.375f, 1.00f } },  
+ { { 3.0f, 4.0f,-3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.250f, 0.69f } },  
+ { { 2.8f, 0.0f,-3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.250f, 0.00f } },  
+  
+ { { 2.8f, 0.0f, 3.0f}, {0.4f, 0.3f, 0.2f, 1.0f}, { 0.000f, 0.00f } },  
+ { { 3.0f, 4.0f, 3.0f}, {0.6f, 0.5f, 0.3f, 1.0f}, { 0.000f, 0.69f } },  
+ { { 0.0f, 6.0f, 3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.750f, 1.00f } },  
+ { { 0.0f, 6.0f,-3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.500f, 1.00f } },  
+ { { 0.0f, 6.0f,-3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.250f, 1.00f } },  
+ { { 0.0f, 6.0f, 3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.000f, 1.00f } },  
 };  
  
　const GLushort iHouse[] = {  
 };

次に、Main.cppから家のインデックスデータを切り取り、

/// インデックスデータ.  
 const GLushort indices[] = {  
 // 木  
  
 // 家  
- 0, 1, 3, 3, 4, 0, 1, 2, 3,  
- 5, 6, 8, 8, 9, 5, 6, 7, 8,  
- 9, 8, 11, 11, 10, 9,  
- 4, 3, 6, 6, 5, 4,  
- 3, 12, 13, 13, 6, 3,  
- 8, 14, 15, 15, 11, 8,  
  
 // 岩

MeshList.cppのiHouse配列変数に貼り付けてください。

{ { 0.0f, 6.0f,-3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.250f, 1.00f } },  
 { { 0.0f, 6.0f, 3.0f}, {1.0f, 1.0f, 1.0f, 1.0f}, { 0.000f, 1.00f } },  
 };  
  
　const GLushort iHouse[] = {  
+ 0, 1, 3, 3, 4, 0, 1, 2, 3,  
+ 5, 6, 8, 8, 9, 5, 6, 7, 8,  
+ 9, 8, 11, 11, 10, 9,  
+ 4, 3, 6, 6, 5, 4,  
+ 3, 12, 13, 13, 6, 3,  
+ 8, 14, 15, 15, 11, 8,  
 };  
  
　const Vertex vRock[] = {  
 };

**[課題02]** 木や家のモデルと同じように、岩と地面のモデルをMain.cppからMeshList.cppへ移動してください。  
そのほか、自分で作成したモデルがある場合は、それらのための配列変数を追加して移動してください。全ての頂点データとインデックスデータの移動が済んだら、Main.cppにあるvertices変数とindices変数を削除してください。

## VBO, IBO, VAO作成関数を移動する

MeshListクラスのメンバ関数を定義する前に、VBO, IBO, VAOを作成する関数をMain.cppから移動します。  
Main.cppのCreateVBO, CreateIBO, CreateVAOの3つの関数を切り取り(長いので一部省略して表記していますが、関数全体を切り取ってください)、

const Mesh meshList[] = {  
 { GL\_TRIANGLES, 21, (const GLvoid\*)0, 0 },  
 { GL\_TRIANGLES, 42, (const GLvoid\*)(21 \* sizeof(indices[0])), 8 },  
 { GL\_TRIANGLES, 30, (const GLvoid\*)(63 \* sizeof(indices[0])), 24 },  
 { GL\_TRIANGLES, 96, (const GLvoid\*)(93 \* sizeof(indices[0])), 32 },  
};  
  
-/\*\*  
-\* Vertex Buffer Objectを作成する.  
- ... 省略 ...  
-\*/  
-GLuint CreateVBO(GLsizeiptr size, const GLvoid\* data)  
-{  
- ... 省略 ...  
- return vbo;  
-}  
-  
-/\*\*  
-\* Index Buffer Objectを作成する.  
- ... 省略 ...  
-\*/  
-GLuint CreateIBO(GLsizeiptr size, const GLvoid\* data)  
-{  
- ... 省略 ...  
- return ibo;  
-}  
-  
-/\*\*  
-\* Vertex Array Objectを作成する.  
- ... 省略 ...  
-\*/  
-GLuint CreateVAO(GLuint vbo, GLuint ibo)  
-{  
- ... 省略 ...  
- return vao;  
-}  
-  
 /// エントリーポイント.  
 int main()  
 {

MeshList.cppに貼り付けてください。

const Vertex vGround[] = {  
 };  
  
 const GLushort iGround[] = {  
 };  
  
+/\*\*  
+\* Vertex Buffer Objectを作成する.  
+ ... 省略 ...  
+\*/  
+GLuint CreateVBO(GLsizeiptr size, const GLvoid\* data)  
+{  
+ ... 省略 ...  
+ return vbo;  
+}  
+  
+/\*\*  
+\* Index Buffer Objectを作成する.  
+ ... 省略 ...  
+\*/  
+GLuint CreateIBO(GLsizeiptr size, const GLvoid\* data)  
+{  
+ ... 省略 ...  
+ return ibo;  
+}  
+  
+/\*\*  
+\* Vertex Array Objectを作成する.  
+ ... 省略 ...  
+\*/  
+GLuint CreateVAO(GLuint vbo, GLuint ibo)  
+{  
+ ... 省略 ...  
+ return vao;  
+}  
+

関数を切り取ったことでMain.cppは一時的にビルドできない状態になります。ですが、エラー箇所は最終的にMeshListクラスで置き換えていきますので、今は気にしないでください。

## コンストラクタを定義する

それでは、メンバ関数を定義していきましょう。最初はコンストラクタです。  
CreateVAO関数の定義の下に、次のプログラムを追加してください。

glDeleteBuffers(1, &vbo);  
 glDeleteBuffers(1, &ibo);  
 return vao;  
 }  
  
+/\*\*  
+\* コンストラクタ.  
+\*/  
+MeshList::MeshList()  
+{  
+}  
+

コンストラクタは特に何もしません。

## デストラクタを定義する

続いてデストラクタを定義します。  
コンストラクタの定義の下に、次のプログラムを追加してください。

MeshList::MeshList()  
 {  
 }  
  
+/\*\*  
+\* デストラクタ.  
+\*/  
+MeshList::~MeshList()  
+{  
+ Free();  
+}  
+

デストラクタはFreeメンバ関数を呼び出すだけです。必要な後始末はすべてFreeメンバ関数で行います。

## Addメンバ関数を定義する

順番が前後しますが、Allocateメンバ関数で使うので先に定義しておきます。  
デストラクタ定義の下に、次のプログラムを追加してください。

MeshList::~MeshList()  
 {  
 Free();  
 }  
  
+/\*\*  
+\* Meshを追加する.  
+\*  
+\* @param vBegin 追加する頂点データ配列の先頭.  
+\* @param vEnd 追加する頂点データ配列の終端.  
+\* @param iBegin 追加するインデックスデータ配列の先頭.  
+\* @param iEnd 追加するインデックスデータ配列の終端.  
+\*/  
+void MeshList::Add(const Vertex\* vBegin, const Vertex\* vEnd,  
+ const GLushort\* iBegin, const GLushort\* iEnd)  
+{  
+ // メッシュを追加.  
+ Mesh m;  
+ m.mode = GL\_TRIANGLES;  
+ m.count = iEnd - iBegin;  
+ m.indices = (const GLvoid\*)(tmpIndices.size() \* sizeof(GLushort));  
+ m.baseVertex = (GLint)tmpVertices.size();  
+ meshes.push\_back(m);  
+  
+ // モデルのデータをvectorにコピー.  
+ tmpVertices.insert(tmpVertices.end(),vBegin, vEnd);  
+ tmpIndices.insert(tmpIndices.end(),iBegin, iEnd);  
+}

Addメンバ関数はvBegin,vEndで頂点データを、iBegin, iEndでインデックスデータを受け取り、それらをメッシュとして追加します。まず、Mesh型の変数mにモデルのメッシュデータを設定します。プリミティブの種類はGL\_TRIANGLES固定としました。インデックスデータの数は、データの終端(iEnd)から先頭(iBegin)を引くことで得られます。  
インデックスデータはtmpIndicesの末尾に追加されるので、インデックスデータの開始バイト位置は、追加前のtmpIndicesのサイズにGLushort型のバイト数をかけたものになります。同様に、頂点データの開始位置は、頂点データを追加する前のtmpVerticesのサイズになります。変数mを設定したら、push\_back(ぷっしゅ・ばっく)メンバ関数を使ってmeshes変数に追加します。

次に、insert(いんさーと)メンバ関数を使って頂点データとインデックスデータをtmpVerticesとtmpIndicesに追加します。insertメンバ関数は、最初の引数で指定した位置に、続く2つの引数で指定された範囲のデータを追加する関数です。位置の指定ではendメンバ関数を使っています。endメンバ関数は、vectorに格納されたデータの末尾の位置を返します。こうすることで、vectorの末尾にデータを追加することができます。

このように、プログラムでメッシュデータを作ることによって、インデックスデータの個数や位置、頂点データの位置の計算をコンピューターに任せてしまうことができるわけです。

## Allocateメンバ関数を定義する

次は、メッシュのリストを作成するAllocate(あろけーと)関数です。  
デストラクタの定義の下に、次のプログラムを追加してください。

MeshList::~MeshList()  
 {  
 Free();  
 }  
  
+/\*\*  
+\* モデルデータからMeshのリストを作成する.  
+\*  
+\* @retval true 作成成功.  
+\* @retval false 作成失敗.  
+\*/  
+bool MeshList::Allocate()  
+{  
+ return true;  
+}

この関数は長くなるので、少しずつ作っていきます。  
まずVAOを作成します。Allocateメンバ関数に次のプログラムを追加してください。

bool MeshList::Allocate()  
 {  
+ Free();  
+  
+ // 容量を予約.  
+ meshes.reserve(100);  
+ tmpVertices.reserve(10’000);  
+ tmpIndices.reserve(10’000);  
+  
+ // VAOを作成する.  
+ GLuint vbo = CreateVBO(tmpVertices.size() \* sizeof(Vertex), tmpVertices.data());  
+ GLuint ibo = CreateIBO(tmpIndices.size() \* sizeof(GLushort), tmpIndices.data());  
+ vao = CreateVAO(vbo, ibo);  
+  
+ // vectorのメモリを解放する.  
+ std::vector<Vertex>().swap(tmpVertices);   
+ std::vector<GLushort>().swap(tmpIndices);   
+  
+ if (!vbo || !ibo || !vao) {  
+ std::cerr << "ERROR: VAOの作成に失敗.\n";  
+ return false;  
+ }  
 return true;  
 }

最初にFreeメンバ関数を呼び出していますが、これは万が一連続でAllocateメンバ関数が呼び出された場合に、確保済みのバッファ・オブジェクトが解放されるようにするためです。すぐあとでvbo, ibo, vao変数を上書きしているので、古い値がわからなくなってしまう前に解放しておきます。

次のvector::reserve(りざーぶ)関数は、vector型の変数がすぐに確保できる容量を予約します。vector型は最初に適当な大きさのメモリを確保して、そこにデータを詰め込んでいきます。データが確保したメモリに入り切らなくなったとき、より大きい別のメモリを確保しなおして、そちらにデータをコピーし、古いメモリを解放します。vector型がいくらでも大きな配列を作れるのは、このシステムのおかげです。ただし、この方法にはひとつ問題があります。データのコピーにはそれなりの時間がかかるので、プログラムなかなか進まなくなってしまうことがあるのです。そこでreserve関数の出番です。reserve関数は、指定された大きさのメモリを確保します。十分な大きさのメモリを確保しておくことで、データのコピーが起こらないようにできるわけです。今回は、メッシュは100個、頂点とインデックスは1万個まで、データのコピーをしなくてもいいようにしてみました。いきなりこんな数のデータを追加することはないでしょうが、あとからいろいろなデータを追加することを考えて大きめに取っています。  
その次のVBO, IBO, VAOの作成は、Main.cppに書いたプログラムとほぼ同じですね。

さて、VAOを作成してしまえばtmpVerticesとtmpIndicesは用済みです。使わないデータを残しておくのはメモリの無駄使いですから、vectorが確保したメモリを解放します。これには空のvector型変数とswap(すわっぷ)メンバ関数を使います。「std::vector<Vertex>()」の部分は空のvector型の変数を作成しています。変数を作るとき、通常は「std::vector<Vertex> name()」のように変数名を設定しますが、C++では名前をつけない変数を作ることができるのです。このような名無しの変数のことを「一時変数」と呼びます。

次の「.swap(tmpVertices);」の部分は、この一時変数のswapメンバ関数の呼び出しです。swapメンバ関数は、引数で設定された変数と自分自身の値を交換します。その結果、一時変数はさっきまでtmpVerticesが持っていたデータを持つようになり、tmpVerticesのほうは空っぽになります。  
そして、swapが終わって「;」に到達すると、一時変数は削除されます(一時変数の寿命は行の終わりまでなんです。儚いですね)。このときswapしたデータも削除されるので、メモリを解放することができるのです。

**[補足]** swapで解放されるのはCPU側のメモリにあるデータだけです。CreateVBOやCreateIBOでGPUメモリに転送したデータは削除されません。GPU側のメモリが解放されるのは、VAOを削除する時です。

データを削除したら、VAOの作成に成功したかどうかを調べ、成功していたらtrue、失敗していたらエラーメッセージを表示してfalseを返します。

## メッシュを追加する

次に、リストにメッシュデータを追加します。  
vectorの容量を予約するプログラムの下に、次のプログラムを追加してください。

// 容量を予約.  
 meshes.reserve(100);  
 tmpVertices.reserve(10’000);  
 tmpIndices.reserve(10’000);  
  
+ // メッシュを追加.  
+ Add(std::begin(vTree), std::end(vTree), std::begin(iTree), std::end(iTree));  
+  
 // VAOを作成する.  
 GLuint vbo = CreateVBO(tmpVertices.size() \* sizeof(Vertex), tmpVertices.data());  
 GLuint ibo = CreateIBO(tmpIndices.size() \* sizeof(GLushort), tmpIndices.data());  
 vao = CreateVAO(vbo, ibo);

メッシュを追加する機能はAddメンバ関数として作成済みなので、追加するデータの範囲を指定してAdd関数を呼び出すだけでメッシュを追加することができます。

木のモデルに続いて家のモデルも追加しましょう。木のメッシュを追加するプログラムの下に、次のプログラムを追加してください。家のモデルを追加するプログラムは、「Tree」を「House」に変えただけです。こういうときは「コピー&貼り付け」を使うといいでしょう。

// メッシュを追加.  
 Add(std::begin(vTree), std::end(vTree), std::begin(iTree), std::end(iTree));  
+ Add(std::begin(vHouse), std::end(vHouse), std::begin(iHouse), std::end(iHouse));  
  
 // VAOを作成する.  
 GLuint vbo = CreateVBO(tmpVertices.size() \* sizeof(Vertex), tmpVertices.data());  
 GLuint ibo = CreateIBO(tmpIndices.size() \* sizeof(GLushort), tmpIndices.data());  
 vao = CreateVAO(vbo, ibo);

これで、木と家のモデルを追加することができました。

**[課題03]** 課題02で追加した岩と地面のメッシュデータを、Addメンバ関数を使ってメッシュリストに追加してください。  
そのほか、自分で作成したモデルがある場合は、それらも同様にメッシュリストに追加してください。

## Freeメンバ関数を定義する

続いて、Allocate関数の対になるFree関数を定義します。  
Allocate関数の定義の下に、次のプログラムを追加してください。

if (!vbo || !ibo || !vao) {  
 std::cerr << "ERROR: VAOの作成に失敗.\n";  
 return false;  
 }  
 return true;  
 }  
  
+/\*\*  
+\* メッシュリストを破棄する.  
+\*/  
+void MeshList::Free()  
+{  
+ glDeleteVertexArrays(1, &vao);  
+ vao = 0;  
+ std::vector<Mesh>().swap(meshes);  
+}  
+

デストラクタでは、glDeleteVertexArray関数を使ってvaoを破棄したあと、メンバ変数を0で初期化します(0は「バッファ・オブジェクトが存在しない」ことを表す番号でしたね)。そしてvector::swap関数でメッシュリストを空にします。

## BindVertexArrayメンバ関数を定義する

次に、VAOをバインドする関数を定義します。  
Free関数の定義の下に、次のプログラムを追加してください。

vao = 0;  
 std::vector<Mesh>().swap(meshes);  
 }  
  
+/\*\*  
+\* 描画に使用するVAOを設定する.  
+\*/  
+void MeshList::BindVertexArray()  
+{  
+ glBindVertexArray(vao);  
+}  
+

この関数はglBindVertexArray関数を呼び出すだけです。

## Getメンバ関数を定義する

最後に、メッシュデータを取得する関数を定義します。  
BindVertexArray関数の定義の下に、次のプログラムを追加してください。

void MeshList::BindVertexArray()  
{  
 glBindVertexArray(vao);  
}  
  
+/\*\*  
+\* メッシュを取得する.  
+\*  
+\* @param index 取得するメッシュの番号.  
+\*  
+\* @return index番目のメッシュ.  
+\*/  
+const Mesh& MeshList::Get(size\_t index) const  
+{  
+ return meshes[index];  
+}

この関数は、meshes変数のindex番目にあるメッシュデータを返します。  
これでメッシュリストをクラスにすることができました。

## MeshListクラスを使う

それでは、MeshListクラスを使うようにMain.cppを変更していきましょう。まずはMeshList.hをインクルードします。Main.cppの先頭付近に、次のプログラムを追加してください。

/\*\*  
 \* @file Main.cpp  
 \*/  
 #include "GLFWEW.h"  
 #include "Shader.h"  
 #include "Texture.h"  
 #include "Geometry.h"  
+#include "MeshList.h"  
 #include <glm/gtc/matrix\_transform.hpp>  
 #include <iostream>

最初に、meshList変数の型をMeshListクラスに変更します。Main.cppを開き、meshList変数定義を次のように変更してください。

// 地面.  
 };  
  
-/\*\*  
-\* メッシュ配列.  
-\*/  
-const Mesh meshList[] = {  
- { GL\_TRIANGLES, 3 \* 7, (const GLvoid\*)0, 0 },  
- { GL\_TRIANGLES, 3 \* 14, (const GLvoid\*)(21 \* sizeof(indices[0])), 8 },  
- { GL\_TRIANGLES, 3 \* 10, (const GLvoid\*)(63 \* sizeof(indices[0])), 24 },  
- { GL\_TRIANGLES, 6 \* 16, (const GLvoid\*)(93 \* sizeof(indices[0])), 32 },  
-};  
+MeshList meshList;  
  
 /// エントリーポイント.  
 int main()

次に、CreateVBOなどの関数を使っている部分がエラーになっているので、ここを置き換えます。  
Main.cppのVAOを作成しているプログラムを、次のように変更してください。

GLFWEW::Window& window = GLFWEW::Window::Instance();  
 if (!window.Init(800, 600, "OpenGL Tutorial")) {  
 return 1;  
 }  
  
-const GLuint vbo = CreateVBO(sizeof(vertices), vertices);  
-const GLuint ibo = CreateIBO(sizeof(indices), indices);  
-const GLuint vao = CreateVAO(vbo, ibo);  
+MeshList meshList;  
+if (!meshList.Allocate()) {  
+ return 1;  
+}  
+  
 const GLuint shaderProgram =  
 Shader::BuildFromFile("Res/Simple.vert", "Res/Simple.frag");  
 const GLuint pidLighting =  
 Shader::BuildFromFile("Res/FragmentLighting.vert", "Res/FragmentLighting.frag");  
-if (!vbo || !ibo || !vao || !shaderProgram || !pidLighting) {  
+if (!shaderProgram || !pidLighting) {  
 return 1;  
 }

続いて、glBindVertexArray関数を、MeshList::BindVertexArray関数で置き換えます。  
メインループにあるglBindVertexArray関数の呼び出しを、次のように変更してください。

// 座標変換行列を作成する.  
 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));  
 progLighting.SetViewProjectionMatrix(matProj \* matView);  
  
-glBindVertexArray(vao);  
+meshList.BindVertexArray();  
  
 progLighting.SetLightList(lights);  
 progLighting.BindTexture(0, texId);

それから、meshList変数の型を変更したために、メッシュデータの取得方法が変わっています。木のモデルを表示するプログラムを次のように変更してください。

for (float i = 0; i < treeCount; ++i) {  
 const float theta = 3.14f \* 2 / treeCount \* i;  
 const float x = std::cos(theta) \* radius;  
 const float z = std::sin(theta) \* radius;  
- progLighting.Draw(meshList[0],  
+ progLighting.Draw(meshList.Get(0),  
 glm::vec3(x, 0, z), glm::vec3(0, theta \* 5, 0), glm::vec3(1));  
 }

**[課題04]** 木のモデルと同様に、家と岩のモデルを表示するプログラムを、MeshList::Get関数を使うように書き換えてください。自分で作成したモデルを表示している場合は、それらも同様に書き換えてください。

最後に、vaoを破棄するプログラムを削除します。次のように、メインループの下にある、glDeleteVertexArrays関数の呼び出しを削除してください。

glDeleteTextures(1, &texRock);  
 glDeleteTextures(1, &texHouse);  
 glDeleteTextures(1, &texId);  
- glDeleteVertexArrays(1, &vao);  
  
 return 0;  
 }

これでメッシュリストをクラスにすることができました。**ビルドして実行してください。**  
変更前と同じように表示できていれば成功です。  
なお、課題02, 03, 04が完了していないとビルドに失敗します。課題を終えてからビルドしてください。

**[課題05]** 家の頂点データに法線を追加し、progLightingを使って描画してください。  
以下に家の法線の数値を示します。

