[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(glDeleteProgramを使用)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。

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  
 {  
+public:  
+ explicit Program(GLuint programId);  
+ ~Program();  
+  
+public:  
+　　GLuint id; // プログラムID.  
 };  
  
 } // namespace Shader  
  
 #endif // SHADER\_H\_INCLUDED

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・オブジェクトとして使うのは悪い考えです。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); **/\* !!!ERROR!!! \*/**

デストラクタ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IDをもたせます。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関数の引数として設定するときに使います。

## 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。元の関数名からglを抜いて「BindVertexArray(ばいんど・ばーてっくす・あれい)」という名前にすればいいでしょう。VAOは変更されるかもしれないので、引数として受け取るのがいいでしょう。

「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の先頭付近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>  
  
 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。

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];  
+};  
+  
+/\*\*  
+\* ライトをまとめた構造体.  
+\*/  
+struct LightList  
+{  
+ AmbientLight ambient;  
+ DirectionalLight directional;  
+ PointLight point;  
+};  
+  
 /\*\*  
 \* 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。

/\*\*  
 \* シェーダー・プログラム.  
 \*/  
 class Program  
 {  
 public:  
 explicit Program(GLuint programId);  
 ~Program();  
+  
+ void Use();  
+ void BindVertexArray(GLuint);  
+ 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);  
  
 public:  
　　 GLuint id; // プログラムID.  
 };  
  
 } // namespace Shader  
  
 #endif // SHADER\_H\_INCLUDED

## 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。

class Program  
 {  
 public:  
 explicit Program(GLuint programId);  
 ~Program();  
  
 　 void Use();  
 　 void BindVertexArray(GLuint);  
 　 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.  
+  
+ // uniform変数の位置.  
+ GLint locMatMVP;  
+ GLint locAmbLightCol;  
+ GLint locDirLightDir;  
+ GLint locDirLightCol;  
+ GLint locPointLightPos;  
+ GLint locPointLightCol;  
+  
+ glm::mat4 matVP;  
+ LightList lights;  
　};  
  
　} // namespace Shader

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

メンバ関数を定義する前に、頂点データ型をファイルに分離しておきます。  
理由は、Drawメンバ関数は描画のためにMesh構造体のメンバ変数を参照しなければならないからです。つまり、Mesh構造体の内容が分からないと困るのですが、この構造体はMain.cppに定義されているのでShader.cppからは見えないのです。そこで、ヘッダファイルとして分離して、インクルードしようというわけですね。

それでは、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。  
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.  
+};  
+  
+/\*\*  
+\* 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

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

切り取ったままだとビルドエラーを起こしてしまうので、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>

## コンストラクタの定義

それでは、メンバ関数を実装してきます。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");  
+  
+ const GLint texColorLoc = glGetUniformLocation(id, "texColor");  
+ if (texColorLoc >= 0) {  
+ glUseProgram(id);  
+ glUniform1i(texColorLoc, 0);  
+ glUseProgram(0);  
+ }  
+}  
+  
 } // namespace Shader

## 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

デストラクタは、program変数が0でなければglDeleteProgramを呼ぶだけです。

## 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.  
+\*  
+\* プログラム・オブジェクトを使い終わったらglUseProgram(0)を実行して解除すること.  
+\*/  
+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。

## BindVertexArray関数の定義

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VAOを設定する.  
+\*  
+\* @param vao 設定するVAOのID.  
+\*/  
+void Program::BindVertexArray(GLuint vao)  
+{  
+ glBindVertexArray(vao);  
+}  
+  
} // namespace Shader

## BindTexture関数の定義

BindVertexArray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::BindVertexArray(GLuint vao)  
 {  
 glBindVertexArray(vao);  
 }  
  
+/\*\*  
+\* 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。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。そのため、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.  
+\*  
+\* この関数を使う前に、Use()を実行しておくこと.  
+\*/  
+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);  
+ }  
+}  
+  
} // namespace Shader

## 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 (locPointLightCol >= 0) {  
 glUniform3fv(locPointLightCol, 8, &lights.point.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

## Draw関数の定義

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)  
+{  
+ // モデル行列を計算する.  
+ const glm::mat4 matScale = glm::scale(glm::mat4(1), scale);  
+ const glm::mat4 matRotateX = glm::rotate(glm::mat4(1), rotate.x, glm::vec3(1,0,0));  
+ const glm::mat4 matRotateXY = glm::rotate(matRotateX, rotate.y, glm::vec3(0,1,0));  
+ const glm::mat4 matRotateXYZ =  
+ glm::rotate(matRotateXY, rotate.z, glm::vec3(0,0,-1));  
+ const glm::mat4 matTranslate = glm::translate(glm::mat4(1), translate);  
+ const glm::mat4 matModel = matTranslate \* matRotateXYZ \* 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]);  
+  
+ // モデル座標系における指向性ライトの方向を計算し、GPUメモリに転送する.  
+ if (locDirLightDir >= 0) {  
+ const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateXYZ));  
+ 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.  
+ if (locPointLightPos >= 0) {  
+ const glm::mat4 matInvModel = glm::inverse(matModel);  
+ 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);  
+ }  
+  
+ // 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

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 progLightingId =  
 Shader::BuildFromFile("Res/FragmentLighting.vert", "Res/FragmentLighting.frag");  
 if (!vbo || !ibo || !vao || !shaderProgram || !fragmentLightingId) {  
 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。

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");  
-if (locDirLightDir < 0 || locDirLightCol < 0 || locAmbLightCol < 0 ||  
- locPointLightPos < 0 || locPointLightCol < 0) {  
- std::cerr << "ERROR: uniform変数の位置を取得できません.\n";  
- return 1;  
- }  
+ Shader::Program progLighting(progLightingId);  
  
 // テクスチャを作成する.  
 const int tw = 8; // 画像の幅.  
 const int th = 8; // 画像の高さ.

次に、ライトデータとしてLightList構造体を使うように変更します。  
ポイント・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.  
-const glm::vec3 ambLightCol = 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.  
-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;  
+Shader::LightList lights;  
+lights.ambient.color = glm::vec3(0.05f, 0.1f, 0.1f);  
+lights.directional.direction = glm::normalize(glm::vec3(5, -2, -2));  
+lights.directional.color = glm::vec3(1, 1, 1);  
+lights.point.position[0] = glm::vec3(5, 4, 0);  
+lights.point.color[0] = glm::vec3(1.0f, 0.8f, 0.4f) \* 100.0f;  
  
 // メインループ.  
 while (!window.ShouldClose()) {

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);  
 const glm::mat4x4 matView =  
 glm::lookAt(viewPos, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));

次に、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);  
  
-// 光源を設定する.  
-const glm::vec3 ambLightCol = glm::vec3(0.05f, 0.1f, 0.1f);  
-const glm::vec3 dirLightDir = glm::normalize(glm::vec3(5, -2, -2));  
-const glm::vec3 dirLightCol = glm::vec3(1.0f, 1.0f, 1.0f);  
-glUniform3fv(locAmbLightCol, 1, &ambLightCol.x);  
-glUniform3fv(locDirLightDir, 1, &dirLightDir.x);  
-glUniform3fv(locDirLightCol, 1, &dirLightCol.x);  
-glUniform3fv(locPointLightCol, 8, &pointLightCol[0].x);  
+progLighting.SetLightList(lights);  
-  
-glBindVertexArray(vao);  
+progLighting.BindVertexArray(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) {

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 matModelR =  
- glm::rotate(glm::mat4(1), theta \* 5, glm::vec3(0, 1, 0));  
- const glm::mat4x4 matModelT = glm::translate(glm::mat4(1), glm::vec3(x, 0, z));  
- const glm::mat4x4 matMVP = matProj \* matView \* matModelT \* matModelR;  
- const glm::vec3 dirLightDirOnModel =  
- glm::inverse(glm::mat3x3(matModelR)) \* dirLightDir;  
- 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(locDirLightDir, 1, &dirLightDirOnModel.x);  
- glUniform3fv(locPointLightPos, 8, &pointLightPosOnModel[0].x);  
- glUniformMatrix4fv(locMatMVP, 1, GL\_FALSE, &matMVP[0][0]);  
- 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));  
 }

**[課題01]** 地面など、描画にprogLightingを使用しているプログラムを、木の描画プログラムと同様にprogLighting.Drawを使うように変更してください。

## 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。**問題なく実行できていれば成功です。   
なお、課題01を完了させていないとビルドに失敗する可能性があります。

# 経過時間を計る

## 移動速度を決めるためには

ところで、前回ライトを移動させましたが、その移動速度は適当に決めています。移動速度を適切に決めるには、メインループがひとまわりするのにかかった時間を知らなければなりません。そこで、経過時間を計測する機能を追加しましょう。

GLFWライブラリは、ライブラリが初期化されてからどれだけの時間が経過したかを計測しています。  
これを利用して、経過時間を調べようと思います。適当なタイミングでその時間を記録し、しばらく経ってから再び時間を記録します。すると、2つの記録の差が経過時間になりますね。

それでは、プログラムを書いていきましょう。GLFWEW.hを開き、次のプログラムを追加してください。

class Window  
 {  
 public:  
 static Window& Instance();  
 bool Init(int w, int h, const char\* title);  
 bool ShouldClose() const;  
 void SwapBuffers() const;  
  
+ void InitTimer();  
+ void UpdateTimer();  
+ double DeltaTime() const;  
  
 bool IsKeyPressed(int key) const;  
  
 private:  
 Window();  
 ~Window();  
 Window(const Window&) = delete;  
 Window& operator=(const Window&) = delete;  
  
 bool isGLFWInitialized = false;  
 bool isInitialized = false;  
 GLFWwindow\* window = nullptr;  
+ double previousTime = 0;  
+ double deltaTime = 0;  
 };

経過時間を計測するために、3つのメンバ関数と2つのメンバ変数を追加しています。  
InitTimer(いにっと・たいまー)関数はGLFWの計測時間を初期化します。  
UpdateTimer(あっぷでーと・たいまー)関数は、経過時間を計測します。  
DeltaTime(でるた・たいむ)関数は、UpdateTimer関数で計測した経過時間を取得します。  
previousTime(ぷればいおす・たいむ)は前回の時間を記録するための変数です。  
deltaTimeは経過時間を記録するための変数です。

続いて、メンバ関数を定義していきます。  
GLFWEW.cppを開き、Window::IsKeyPressed関数の定義の下に、次のプログラムを追加してください。

bool Window::IsKeyPressed(int key) const  
 {  
 return glfwGetKey(window, key) == GLFW\_PRESS;  
 }  
  
+/\*\*  
+\* タイマーを初期化する.  
+\*/  
+void Window::InitTimer()  
+{  
+ glfwSetTime(0.0);  
+ previousTime = 0.0;  
+ deltaTime = 0.0;  
+}  
+  
 } // namespace GLFWEW

InitTimer関数は、GLFWライブラリの計測時間を初期化し、経過時間の計算を始める準備を行います。  
計測時間を初期化するにはglfwSetTime(じーえる・えふ・だぶりゅー・せっと・たいむ)関数を使います。この関数は、引数で指定された値を計測時間に設定します。そこで、0を指定して初期化しています。

次に、UpdateTimer関数を定義します。InitTimer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 Window::InitTimer()  
　{  
 　 glfwSetTime(0.0);  
 　 previousTime = 0.0;  
 　 deltaTime = 0.0;  
　}  
  
+/\*\*  
+\* タイマーを更新する.  
+\*/  
+void Window::UpdateTimer()  
+{  
+ // 経過時間を計測.  
+ const double currentTime = glfwGetTime();  
+ deltaTime = currentTime - previousTime;  
+ previousTime = currentTime;  
+  
+ // 経過時間が長くなりすぎないように調整.  
+ const float upperLimit = 0.25f; // 経過時間として許容される上限.  
+ if (deltaTime > upperLimit) {  
+ deltaTime = 1.0f / 60.0f;  
+ }  
+}  
+  
 } // namespace GLFWEW

経過時間の計測方法は、この節のはじめに説明したとおりです。GLFWライブラリが計測している時間を取得するにはglfwGetTime(じーえる・えふ・だぶりゅー・げっと・たいむ)関数を使います。この関数の結果と前回の計測時間との差が、「経過時間」になるわけですね。  
経過時間をdeltaTime変数に格納したら、previousTime変数を今回取得した計測時間に更新します。  
こうすることで、次にUpdateTimer関数が実行された時、直前のUpdateTimer関数実行からの経過時間を計ることができるわけです。

ところで、経過時間を取得したあとで、その値が0.25を越えていたら0.1にするプログラムがありますが、これは何なのでしょう？　これが必要な理由はglfwGetTime関数の仕様にあります。実はこの関数、現実世界の時間を計測しているのです。それの何がまずいかというと、デバッグのためにブレーク・ポイントなどでプログラムを一時停止している間も時間が進んでしまうことです。デバッグを終えてアプリケーションを再開したとき、デバッグに費やした時間がそのまま経過時間になるわけですね。こうなると、非常にデバッグしづらい状況が容易に作り出せてしまいます。(再開した途端、デバッグ前にちょうど命中しそうな弾が敵をすりぬけてしまったり、あるいは自分の眼の前に突然敵の弾が現れたり、などという状況を想像してみてください)。そこで、ある程度の時間で制限をかけているというわけです。経過時間の上限を示す変数名はupperLimit (あっぱー・りみっと=上限)としました。  
上限を超えた場合、経過時間を1/60秒にします。

**[補足]** 1/60秒というのは、現在市販されているほとんどのモニターの画像更新速度です。多くのゲームでは、この時間を基準にゲームの世界を動かしています。たとえプログラムがこの時間より高速に画像を更新できたとしても、モニターの表示が追いつかないため正しく表示されません。

定義を続けましょう。次はDeltaTime関数です。  
UpdateTimer関数の定義の下に、次のプログラムを追加してください。

// 経過時間が長くなりすぎないように調整.  
 const float upperLimit = 0.25f; // 経過時間として許容される上限.  
 if (deltaTime > upperLimit) {  
 deltaTime = 0.1f;  
 }  
 }  
  
+/\*\*  
+\* 経過時間を取得する.  
+\*  
+\* @return 直前の2回のUpdateTimer()呼び出しの間に経過した時間.  
+\*/  
+double Window::DeltaTime() const  
+{  
+ return deltaTime;  
+}  
+  
 } // namespace GLFWEW

この関数はただdeltaTimeを返すだけの関数です。  
これで経過時間の計測プログラムは完成です。

## 経過時間を使って正しい速度を計算する

それでは、経過時間を測って、ライトの移動速度を適切に決めましょう。  
Main.cppを開き、メインループに次のプログラムを追加してください。

// メインループ.  
+window.InitTimer();  
 while (!window.ShouldClose()) {  
+ window.UpdateTimer();  
+ const float deltaTime = (float)window.DeltaTime();  
  
 glEnable(GL\_DEPTH\_TEST);  
 glEnable(GL\_CULL\_FACE);

メインループの直前で経過時間の計測を初期化し、ループの先頭で経過時間を更新、取得しています。  
この構造では、最初の経過時間が非常に短くなります。しかし、一瞬のことなので問題にはならないでしょう。

次に、ライトを移動するプログラムを、次のように変更してください。

// ポイント・ライトの位置が分かるように適当なモデルを表示.  
 {  
 // 0番のポイント・ライトを移動する.  
- const float speed = 0.05f;  
+ const float speed = 10.0f \* deltaTime;  
 if (window.IsKeyPressed(GLFW\_KEY\_A)) {  
 pointLightPos[0].x -= speed;  
 } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
 pointLightPos[0].x += speed;  
 }

ワールド座標の単位は「1.0f=1m」なので、ライトの移動速度は10m/sということになります。  
プログラムが書けたら、**ビルドして実行してください。**  
見た目には変化はありませんが、異なるGPUで実行した場合でも、同じ速度で移動できるようになっています。