[OpenGL講義 第07回]

フラグメントは光源の夢を見るか

# ポイント・ライト

## ポイント・ライト変数の追加

指向性ライトは、太陽や月のように、非常に遠方にある光源を表現することには長けています。  
しかし現実世界には、たき火やろうそくや電灯など、さまざまな光源があふれています。指向性ライトでは、このような近距離にある光源を表現することが難しいです。

そこで、新しく「ポイント・ライト」というものを導入しようと思います。ポイント・ライトは、日本語では「点光源(てんこうげん)」と呼ばれ、名前のとおり「空間の1点から全方位に光を放つライト」です。  
VertexLighting.vertに、次のプログラムを追加してください。

// 指向性ライト.  
 struct DirectionalLight  
 {  
 vec3 direction;  
 vec3 color;  
 };  
  
+// ポイント・ライト.  
+struct PointLight  
+{  
+ vec3 position[8];  
+ vec3 color[8];  
+};  
+  
 uniform AmbientLight ambientLight;  
 uniform DirectionalLight directionalLight;  
+uniform PointLight pointLight;  
 uniform mat4x4 matMVP;

ポイント・ライトは最大8個まで配置できるように配列にしてみました。  
GLSLではC言語と同様に配列を定義することができます。ただし、構造体と配列の組み合わせには注意が必要です。というのもGLSLには、構造体の配列をuniform変数として使う場合、配列の要素ごとにuniform変数の位置を取得しなければならない、という制限があるからです。これはプログラムが面倒なうえ、データをひとつひとつ転送しなければならないために処理速度も遅くなります。対処法は、上記のプログラムのように「配列を構造体に入れる」ことです。

**[補足]** ライトの数の制限をなくすために考えられたのが、ディファード・ライティングやディファード・シェーディングといった技術です。これらの方法の登場により、理論上は無限個のライトを扱うことが可能になりました。

## 明るさを計算するプログラムの追加

データを定義したら、ポイント・ライトの明るさを計算します。  
VertexLighting.vertのmain関数に、次のプログラムを追加してください。

/\*\*  
 \* 頂点シェーダー.  
 \*/  
 void main()  
 {  
- **// ランバート反射による拡散反射光の計算.**  
+ // 指向性ライトの明るさを計算.  
 float cosTheta = clamp(dot(vNormal, -directionalLight.direction), 0.0, 1.0);  
- **outColor.rgb = (vColor.rgb \* directionalLight.color \* cosTheta);**  
+ vec3 lightColor = directionalLight.color \* cosTheta;  
+  
+ // ポイント・ライトの明るさを計算.  
+ for (int i = 0; i < 8; ++i) {  
+ if (dot(pointLight.color[i], pointLight.color[i]) != 0) {  
+ vec3 lightVector = pointLight.position[i] - vPosition;  
+ float cosTheta = clamp(dot(vNormal, normalize(lightVector)), 0.0, 1.0);  
+ float intensity = 1 / dot(lightVector, lightVector);  
+ lightColor += pointLight.color[i] \* cosTheta \* intensity;  
+ }  
+ }  
+  
+ outColor.rgb = vColor.rgb \* lightColor;  
  
 **// 環境光の計算.**  
 outColor.rgb += ambientLight.color;  
 outColor.a = vColor.a;  
  
 outTexCoord = vTexCoord;  
 gl\_Position = matMVP \* vec4(vPosition, 1.0);  
}

ライトの数が増えたので、lightColor(らいと・からー)という変数を用意して、ライトの明るさを計算するたびに、この変数に加えていくことにします。全てのライトから受ける光の量を計算したあとで、頂点カラーと掛け合わせます。

ポイント・ライトは8個あるのでfor文で処理します。  
最初にあるのは「ライトの色が真っ黒(=すべての成分が0)なら無視する」というif文です。これは内積を計算するdot(どっと)関数を使えば簡単に調べられます。内積とは「2つのベクトルA,Bの各要素を乗じて合計する」ことです。つまり計算結果が0ならば「すべての成分が0」だということが分かるわけです。  
なお、C/C++言語では、数値型からbool型への変換は自動的に行われますが、GLSLにそのような機能はありません。そのため、上記のプログラムのように明示的な比較が必要です。

色が0でなければ、明るさを計算します。指向性ライトと同様に、ポイント・ライトの明るさも、光の向きと法線のなす角によって変化します。違いは、光の向きを計算で求めるところです。向きを求めるには、ライトの位置から頂点の位置を引いて「ライト-頂点」ベクトルを求め、そのベクトルを単位ベクトルに正規化(せいきか)します。正規化を行うにはnormalize(のーまらいず)関数を使います。normalize関数は、引数に設定されたベクトルを単位ベクトルに変換した値を返します。

ろうそくや電球の明かりが遠くまでは届かないのと同様に、ポイント・ライトではライトと頂点の距離が遠いほど明るさが弱くなります。明るさは「距離の2乗に反比例する」ことが知られている(「逆２乗の法則)といいます)ので、上記のプログラムでもこの式を使っています。距離の2乗を求めるにはdot(どっと)関数を使います。dotは「2つのベクトルA,Bの各要素を乗じて合計する」ことで内積を計算する関数なのですが、AとBが同じベクトルだった場合、この計算はベクトルの長さの2乗を計算することと同じです。距離による明るさの変化量はintensity(いんてんしてぃ、強度)という変数に代入します。

角度による明るさ(cosTheta)と距離による明るさ(intensity)が求まれば、あとはその2つを掛け合わせてlightColor(らいと・からー)変数に積算していくだけです。

全てのライトからの明るさを合計できたら、それを物体の色に掛けたものが最終的な色になります。

## ポイント・ライト変数の位置を取得する

次に、ポイント・ライト変数の位置を取得します。  
Main.cppを開き、指向性ライトの位置を取得しているプログラムの上に、次のプログラムを追加してください。

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) {  
+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;  
 }

## ライトデータを作成する

次に、転送するライトを準備します。  
ライトを設定しているプログラムに、次のプログラムを追加してください。

//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;  
+  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);  
 glEnable(GL\_CULL\_FACE);

ポイント・ライトは8個まで設定できますが、とりあえず1個だけ値を設定してみました。

続いて、指向性ライトのデータを転送している部分に、次のプログラムを追加してください。

// 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);  
  
 glBindVertexArray(vao);  
  
 glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, texId);

ポイント・ライトは8個あるので、転送するデータの数も8個になっている点に注意してください。また、指向性ライトと同様に、ポイント・ライトの位置はモデル座標系に変換してから転送しなければならないため、ここでは色だけを転送しています。

## ポイント・ライトの座標をモデル座標系に変換する

それでは、ポイント・ライトの位置をGPUメモリに転送していきましょう。  
木を植えるプログラムに、次のプログラムを追加してください。

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;  
 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ローカル座標系に変換して転送する.  
 const glm::vec3 dirLightDirOnModel =  
 glm::inverse(glm::mat3(matModelR)) \* dirLightDir;  
 glUniform3fv(locDirLightDir, 1, &dirLightDirOnModel.x);  
  
+ // ポイント・ライトの位置をローカル座標系に変換して転送する.  
+ 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);  
+  
 glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
 }

基本的な方法は、指向性ライトの向きの計算と同じです。ただし、今回の計算対象は「向き」ではなく「位置」なので、モデルの回転に加えて移動も逆行列の対象としています。

これでポイント・ライトを追加することができました。**ビルドして実行してください。**

**[課題01]** 地面モデルの表示プログラムに、ポイント・ライトの位置をローカル座標系に変換して転送するプログラムを追加してください。

# フラグメント・ライティング

## 頂点シェーダによるライティングの問題点

ここまでは、頂点シェーダをつかってライトを実装してきました。  
しかし、頂点シェーダによるライティングには、いくつか問題があります。例えば、平らな板の中央付近にライトを置いたとします。現実世界では、中央付近が明るく、外側に向かって暗くなるはずです。しかし、頂点シェーダでは、プリミティブの端にある頂点でしかライトを感じることができません。そのため、板は一様に暗い色に見えてしまいます。ライトはすぐそばにあるのに！

## 頂点シェーダの追加

この問題を解決するためには、ライティングの舞台を頂点シェーダからフラグメントシェーダに移す必要があります。フラグメント単位でライティングを施せば、プリミティブのどの部分であっても、より正確な明るさを計算できるようになるからです。

フラグメント・ライティングには、頂点シェーダーとフラグメントシェーダーの両方が必要です。まずは、Resフォルダに「FragmentLighting.vert(ふらぐめんと・らいてぃんぐ・ばーと)」という名前のファイルを追加してください。  
今回もSimpleシェーダーをベースに改造していきます。追加したFragmentLighting.vertを開き、Simple.vertの内容をコピーして貼り付けてください。

+/\*\*  
+\* @file Simple.vert  
+\*/  
+#version 410  
+  
+layout(location=0) in vec3 vPosition;  
+layout(location=1) in vec4 vColor;  
+layout(location=2) in vec2 vTexCoord;  
+  
+layout(location=0) out vec4 outColor;  
+layout(location=1) out vec2 outTexCoord;  
+  
+uniform mat4x4 matMVP;  
+  
+/\*\*  
+\* 頂点シェーダー.  
+\*/  
+void main()  
+{  
+ outColor = vColor;  
+ outTexCoord = vTexCoord;  
+ gl\_Position = matMVP \* vec4(vPosition, 1.0);  
+}

手始めに、先頭のファイル名を修正しましょう。  
最初のコメントを次のように変更してください。

/\*\*  
-\* @file Simple.vert  
+\* @file FragmentLighting.vert  
 \*/  
 #version 410

次に、入力変数と出力変数を追加します。入力および出力変数を定義している部分を、次のように変更してください。

/\*\*  
 \* @file FragmentLighting.vert  
 \*/  
 #version 410  
  
　layout(location=0) in vec3 vPosition;  
　layout(location=1) in vec4 vColor;  
　layout(location=2) in vec2 vTexCoord;  
+layout(location=0) in vec3 vNormal;  
  
+layout(location=0) out vec3 outPosition;  
-layout(location=0) out vec4 outColor;  
-layout(location=1) out vec2 outTexCoord;  
+layout(location=1) out vec4 outColor;  
+layout(location=2) out vec2 outTexCoord;  
+layout(location=3) out vec3 outNormal;  
  
 uniform mat4x4 matMVP;

最後に、入力変数を出力変数に代入します。main関数に次のプログラムを追加してください。

/\*\*  
 \* 頂点シェーダー.  
 \*/  
 void main()  
 {  
+ outPosition = vPosition;  
 outColor = vColor;  
 outTexCoord = vTexCoord;  
+ outNormal = vNormal;  
 gl\_Position = matMVP \* vec4(vPosition, 1.0);  
 }

頂点シェーダーの変更は以上です。

今回作成するフラグメント・ライティングでは、頂点シェーダーでするべきことはあまり多くありません。頂点座標をクリップ座標系へと変換するほかは、VPから入力された頂点データをフラグメントシェーダーへと送り出すだけです。

## フラグメントシェーダの追加

続いてフラグメントシェーダを追加します。Resフォルダに「FragmentLighting.frag(ふらぐめんと・らいてぃんぐ・ふらぐ)」という名前のファイルを追加してください。  
FragmentLighting.fragを開き、Simple.fragの内容をコピーして貼り付けてください。

+/\*\*  
+\* @file Simple.frag  
+\*/  
+#version 410  
+  
+layout(location=0) in vec4 inColor;  
+layout(location=1) in vec2 inTexCoord;  
+  
+out vec4 fragColor;  
+  
+uniform sampler2D texColor;  
+  
+/\*\*  
+\* フラグメントシェーダー.  
+\*/  
+void main()  
+{  
+ fragColor = inColor \* texture(texColor, inTexCoord);  
+}

まず先頭のファイル名を修正しましょう。  
最初のコメントを次のように変更してください。

/\*\*  
-\* @file Simple.frag  
+\* @file FragmentLighting.frag  
 \*/  
 #version 410

次に入力変数を追加します。入力変数を定義している部分を、次のように変更してください。

/\*\*  
 \* @file FragmentLighting.frag  
 \*/  
 #version 410  
   
+layout(location=0) in vec3 inPosition;  
-layout(location=0) in vec4 inColor;  
-layout(location=1) in vec2 inTexCoord;  
+layout(location=1) in vec4 inColor;  
+layout(location=2) in vec2 inTexCoord;  
+layout(location=3) in vec3 inNormal;  
   
 out vec4 fragColor;

続いて、VertexLighting.vertからライトの構造体とuniform変数をコピーして貼り付けてください。

layout(location=0) in vec3 inPosition;  
 layout(location=1) in vec4 inColor;  
 layout(location=2) in vec2 inTexCoord;  
 layout(location=3) in vec3 inNormal;  
   
 out vec4 fragColor;  
  
+// 環境光.  
+struct AmbientLight  
+{  
+ vec3 color;  
+};  
+  
+// 指向性ライト.  
+struct DirectionalLight  
+{  
+ vec3 direction;  
+ vec3 color;  
+};  
+  
+// ポイント・ライト.  
+struct PointLight  
+{  
+ vec3 position[8];  
+ vec3 color[8];  
+};  
+  
+uniform AmbientLight ambientLight;  
+uniform DirectionalLight directionalLight;  
+uniform PointLight pointLight;  
+  
 uniform sampler2D texColor;  
   
 /\*\*  
 \* フラグメントシェーダー.  
 \*/

変数をコピーしたら、次は計算プログラムもコピーします。VertexLighting.vertにあるライトの明るさを計算するプログラム.をコピーして、次のようにFragmentLighting.fragのmain関数に貼り付けてください。

/\*\*  
 \* フラグメントシェーダー.  
 \*/  
 void main()  
 {  
+ // 指向性ライトの明るさを計算する.  
+ float cosTheta = clamp(dot(inNormal, -directionalLight.direction), 0.0, 1.0);  
+ vec3 lightColor = directionalLight.color \* cosTheta;  
+  
+ // ポイントライトの明るさを計算する.  
+ for (int i = 0; i < 8; ++i) {  
+ if (dot(pointLight.color[i], pointLight.color[i]) != 0) {  
+ vec3 lightVector = (pointLight.position[i] - inPosition);  
+ float power = 1.0 / dot(lightVector, lightVector);  
+ float cosTheta = clamp(dot(inNormal, normalize(lightVector)), 0.0, 1.0);  
+ lightColor += pointLight.color[i] \* cosTheta \* power;  
+ }  
+ }  
+  
 fragColor = inColor \* texture(texColor, inTexCoord);  
 }

最後に、計算したライトの明るさを反映します。  
main関数の末尾にあるfragColorに値を代入している部分に、次のプログラムを追加してください。

lightColor += pointLight.color[i] \* cosTheta \* power;  
 }  
　 }  
  
　 fragColor = inColor \* texture(texColor, inTexCoord);  
+ fragColor.rgb \*= lightColor;  
+ fragColor.rgb += ambientLight.color;  
 }

これでフラグメントシェーダーの修正は完了です。

頂点ライティングとフラグメント・ライティングは、どの部分について計算を行うかが違うだけで、計算式は同じものです。ですから、コピー&貼り付けをするだけでほぼ完成するんですね。

今回は、最大4つのライトを使用できるようにしてみました。ライトの数は自由に変更できますが、あまり大きくするとフラグメントシェーダのパフォーマンスが低下します。

次は、頂点シェーダに書いていたライティング処理を移植し、複数のライトに対応させます。フラグメントシェーダのmain関数を次のように変更してください。

void main()  
 {  
 fragColor = inColor \* texture(colorSampler, inTexCoord);  
**+** vec3 lightColor = lightData.ambientColor.rgb;  
**+** for (int i = 0; i < maxLightCount; ++i) {  
**+** vec3 lightVector = lightData.light[i].position.xyz - inWorldPosition;  
**+** float lightPower = 1.0 / dot(lightVector, lightVector);  
**+** float cosTheta = clamp(dot(vec3(0, 0, 1), normalize(lightVector)), 0, 1);  
**+** lightColor += lightData.light[i].color.rgb \* cosTheta \* lightPower;  
**+** }  
**+** fragColor.rgb \*= lightColor;  
}

複数のライトに対応させるためにfor文を使っています。使い方はC++のfor文と同じです。なお、GLSLでは範囲for文は使えません。これでフラグメントシェーダの変更は完了です。

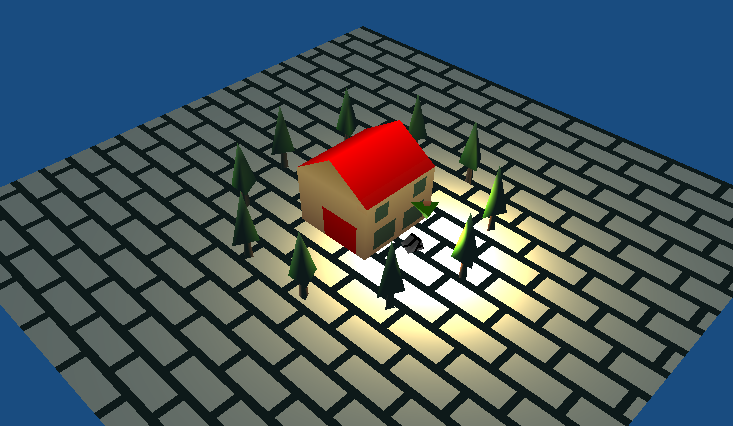
## フラグメント・ライティング・シェーダーを読み込む

頂点ライティング・シェーダーの代わりに、フラグメント・ライティング・シェーダーを読み込みます。  
頂点ライティング・シェーダーを読み込むプログラムを、次のように変更してください。

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

これで、ライティングがフラグメントシェーダで実行されるようになるはずです。

プログラムが書けたら、**ビルドして実行してください。**  
プリミティブの中間部分が、ポイント・ライトの光で照らされているように見えたら成功です。



# 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

モデルを描画するために設定すべき変数が増えてきたため、メインループが見づらくなってきていますね。  
しかし、モデルごとに必要な処理はほとんどおなじプログラムになっています。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。