[OpenGL講義 第06回]

神は「光あれ。」と言われた。

すると光があった。  
神はその光を見て「最初はこんなもんだろ」と思い、良しとされた。第1日である。

# シェーダー・プログラムをファイルから読み込む

## 関数宣言の追加

これまでシェーダー・プログラムは、文字列として直接Main.cppに書いていました。しかし、この方法ではシェーダーの内容を書き換えるたびにビルドが必要になってしまいます。  
そこで、シェーダー・プログラムをファイルから読み込むように変更しましょう。  
Shader.hを開き、以下のプログラムを追加してください。

#ifndef SHADER\_H\_INCLUDED  
 #define SHADER\_H\_INCLUDED  
 #include <GL/glew.h>  
  
 namespace Shader {  
  
 GLuint Build(const GLchar\* vsCode, const GLchar\* fsCode);  
+GLuint BuildFromFile(const char\* vsPath, const char\* fsPath);  
  
 }  
  
 #endif

Shader名前空間に、ファイルからプログラム・オブジェクトを作成する関数宣言を追加しています。

## ヘッダファイルのインクルード

続いて、宣言した関数を定義していきます。まず、ファイル読み込み操作ために「fstream」ヘッダと「stdint.h」ヘッダをインクルードしましょう。Shader.cppを開き、次のプログラムを追加してください。

/\*\*  
 \* @file Shader.cpp  
 \*/  
 #include "Shader.h"  
 #include <vector>  
 #include <iostream>  
+#include <fstream>  
+#include <stdint.h>  
  
　namespace Shader {

シェーダー・プログラムを作成する関数は既に定義済みですから、足りないのはファイルを読み込む部分だけです。ですから、BuildFromFile関数でやるべきなのは、ファイルを読み込んで、その内容使ってBuild関数を実行することです。

## ファイル読み込み関数を作成する

プログラム・オブジェクトをビルドするには、頂点シェーダー用とフラグメントシェーダー用で2回のファイルの読み込みが必要になります。同じことを2度書くのは無駄ですから、読み込みだけを行う関数を定義することにしましょう。Build関数の定義の下に、次のプログラムを追加してください。

glDeleteProgram(program);  
 return 0;  
 }  
 return program;  
 }  
  
+/\*\*  
+\* ファイルを読み込む.  
+\*  
+\* @param path 読み込むファイル名.  
+\*  
+\* @return 読み込んだデータ.  
+\*/  
+std::vector<GLchar> ReadFile(const char\* path)  
+{  
+ std::basic\_ifstream<GLchar> ifs;  
+ ifs.open(path, std::ios\_base::binary);  
+ if (!ifs.is\_open()) {  
+ std::cerr << "ERROR: " << path << " を開けません.\n";  
+ return {};  
+ }  
+ ifs.seekg(0, std::ios\_base::end);  
+ const size\_t length = (size\_t)ifs.tellg();  
+ ifs.seekg(0, std::ios\_base::beg);  
+ std::vector<GLchar> buf(length);  
+ ifs.read(buf.data(), length);  
+ buf.push\_back('\0');  
+ return buf;  
+}  
+  
 } // namespace Shader

ファイルを読み込む方法は、基本的にはTGAファイルを読み込んだときと同じです。ただ、シェーダーファイルは決まったヘッダを持たないので、ファイル全体を読み取らなくてはなりません。そのために、ファイルのバイト数を調べる必要があります。

残念なことにC++14までのバージョンには、ファイルのバイト数を直接取得する方法がありません(この機能はC++17で追加されました)。そこで別の方法を使わなければなりません。ところで、ファイルをバイナリ・モードで開いている場合、読み出し位置はファイルの先頭からのバイト数になります。つまり、読み出し位置を終端に設定すれば、その位置はファイルのバイト数と同じ数値になるのです。  
読み出し位置をファイル終端に移動させるにはbasic\_ifstream(べーしっく・あい・えふ・すとりーむ)のメンバ関数seekg(しーく・じー)を使います(gはgetの頭文字です)。この関数は2つの引数を受け取ります。最初の引数は変更する量です。その次の引数は変更の基準となる場所で、次の3種類から選択します。

|  |  |
| --- | --- |
| std::ios\_base::beg | ファイルの先頭 |
| std::ios\_base::cur | 現在の読み出し位置 |
| std::ios\_base::end | ファイルの終端 |

今回の場合、ファイルの終端に移動したいので、最初の引数には0、2つめの引数にはstd::ios\_base::endを指定しました。

次に、読み出し位置を取得します。これにはtellg(てる・じー)メンバ関数を使います(seekgと同様、gはgetの頭文字です)。tellg関数の戻り値の型はstd::streampos(えすてぃーでぃー・すとりーむ・ぽす、posはポジションの略)という型です。この型は数値型ではないので、size\_t型にキャストしています。

**[補足]** 実はstd::streamposを直接size\_tに変換することはできません。ただし、std::streamposはstd::streamoff(えすてぃーでぃー・すとりーむ・おふ、offはオフセットの略)という数値型には変換できます。つまり、std::streampos → std::streamoff → size\_tという順番で変換されているわけです。

ファイルサイズを取得したら、再びseekgメンバ関数を使って、読み出し位置を先頭に戻します。戻しておかないとファイルを読みだせませんから、これはとても重要です。

ファイルサイズが分かったら、あとは簡単です。サイズと同じ大きさのvector型変数を作り、readメンバ関数でデータを読み込むだけです。ただし、今回読み込んだデータは文字列として扱われます。C/C++の文字列は終端に「\0」が必要ですが、readメンバ関数は自動的に終端を付け足してくれたりはしません。そこで、push\_backメンバ関数を使って、バッファの最後に「\0」を追加しています。

**[補足]** シェーダーファイルはテキストファイルですが、読み込みはバイナリ・モードで行っています。主な理由はバイナリ・モードのほうが高速だからです。というのは、テキスト・モードでは必要に応じて改行コードの変換が行われるので、読み出しが遅くなってしまうのです。また、改行コードの変換が行われると、ファイルサイズとデータのバイト数が違ってしまう場合があります。そのため、上記の方法でvectorの大きさを決めることができません。これらの理由から、単に読み込むだけならバイナリ・モードのほうが効率的です。

## ファイルからシェーダー・プログラムを作成する関数を定義する

完成した読み込み関数と、シェーダー・プログラムを作成する関数をまとめます。  
ReadFile関数定義の下に、次のプログラムを追加してください。

ifs.read(buf.data(), length);  
 buf.push\_back('\0');  
 return buf;  
 }  
+  
+/\*\*  
+\* ファイルからプログラム・オブジェクトを作成する.  
+\*  
+\* @param vsPath 頂点シェーダー・ファイル名.  
+\* @param fsPath フラグメントシェーダー・ファイル名.  
+\*  
+\* @return 作成したプログラム・オブジェクト.  
+\*/  
+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());  
+}  
  
} // namespace Shader

このプログラムがやっていることは、とてもシンプルです。まず頂点シェーダー・ファイルとフラグメントシェーダー・ファイルを読み込みます。そして、Build関数に読み込んだデータを渡し、シェーダー・プログラムを作成します。

## ファイルが読み込めなかった場合に備える

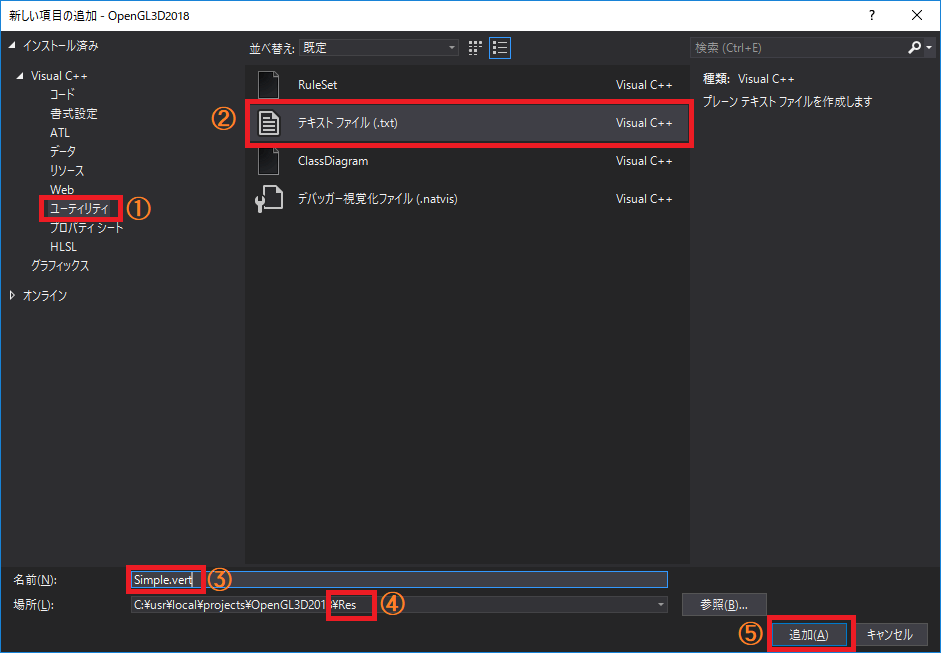
ファイルが読み込めなかった場合、vector型の変数はからっぽです。その状態でdataメンバ関数を呼び出すとnullptrが返されます。ここで問題なのは、Build関数とCompile関数がnullptrに対応していないことです。  
そこで、nullptrが設定されても大丈夫なようにプログラムを修正しましょう。  
Compile関数に次のプログラムを追加してください。

GLuint Compile(GLenum type, const GLchar\* string)  
 {  
+ if (!string) {  
+ return 0;  
+ }  
+  
 GLuint shader = glCreateShader(type);  
 glShaderSource(shader, 1, &string, nullptr);  
 glCompileShader(shader);

追加したのは、string引数がnullptrだったら0を返して関数を終了するプログラムです。  
ファイルが開けなければstring引数にはnullptrが設定されるので、Compile関数は「0(作成失敗)」を返します。これを受け取ったBuild関数はリンク・エラーになって「0(作成失敗)」を返すので、最終的にBuildFromFile関数も「0(作成失敗)」になる、という流れです。

## シェーダー・プログラムをファイルにする

ファイルからプログラム・オブジェクトを作成できるようになったので、次はシェーダー・プログラムをファイルに書いていきましょう。元になるシェーダー・プログラムは既に文字列として定義済みですから、あとは新規にファイルを作ってそれをコピーし、調整を加えるだけです。頂点シェーダーは「Simple.vert(しんぷる・ばーと)」、フラグメントシェーダーは「Simple.frag(しんぷる・ふらぐ)」という名前にしましょうか(vertはバーテックスの略、fragはフラグメントの略です)。それと、シェーダー・ファイルはResフォルダに置くことにしましょう。  
それでは、ソリューションエクスプローラーの「リソースファイル」フィルターの上で右クリックし「追加→新しい項目」を選択してください。



「新しい項目の追加」ウィンドウが開くので、左側のリストから「Visual C++→ユーティリティ」を選択し(①)、中央のリストから「テキストファイル」を選んでください(②)。そして、ファイルの名前を「Simple.vert」に書き換えます(③)。  
さらに、「場所」が「プロジェクトフォルダ\Res」となるように書き換えてください(④)。名前と場所を確認して、問題ないようなら「追加」ボタンをクリックすると(⑤)ファイルが追加されます。

ファイルを追加したら、Main.cppにあるvsCode変数を切り取ります(Ctrl+X)。

-/// 頂点シェーダー.  
-static const char\* vsCode =  
-"#version 410 \n"  
-"layout(location=0) in vec3 vPosition; \n"  
-"layout(location=1) in vec4 vColor; \n"  
-"layout(location=2) in vec2 vTexCoord; \n"  
-"layout(location=0) out vec4 outColor; \n"  
-"layout(location=1) out vec2 outTexCoord; \n"  
-"uniform mat4x4 matMVP; \n"  
-"void main() { \n"  
-" outColor = vColor; \n"  
-" outTexCoord = vTexCoord; \n"  
-" gl\_Position = matMVP \* vec4(vPosition, 1.0); \n"  
-"} \n";  
-  
/// フラグメントシェーダー.  
static const char\* fsCode =

そして、Simple.vertを開いて貼り付けてください(Ctrl+V)。

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

貼り付けただけではC++言語のプログラムのままです。GLSLとして使えるように、内容を修正しましょう。  
Simple.vertを次のように変更してください。

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

主な変更点は下記のとおりです。

1. 不要な行の削除
2. 行の前後にある「"」や「\n」の削除
3. 構造を見やすくするために改行とコメント行を追加

続いて、フラグメントシェーダーもファイルにしていきます。Simple.vertを作ったのと同じ方法で、Resフォルダに「Simple.frag」というファイルを追加してください。  
ファイルを追加したら、Main.cppにあるfsCode変数を切り取ります(Ctrl+X)。

-/// フラグメントシェーダー.  
-static const char\* fsCode =  
-"#version 410 \n"  
-"layout(location=0) in vec4 inColor; \n"  
-"layout(location=1) in vec2 inTexCoord; \n"  
-"out vec4 fragColor; \n"  
-"uniform sampler2D texColor; \n"  
-"void main() { \n"  
-" fragColor = inColor \* texture(texColor, inTexCoord); \n"  
-"} \n";  
-  
  
/\*\*  
\* Vertex Buffer Objectを作成する.

そして、Simple.fragを開いて貼り付けてください(Ctrl+V)。

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

フラグメントシェーダーも、貼り付けただけではC++言語のプログラムのままです。GLSLとして使えるように、内容を修正しましょう。Simple.fragを次のように変更してください。

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

これでシェーダー・プログラムをファイルにすることができました。

## シェーダー・ファイルを読み込む

シェーダー・ファイルが作成できたら、Main.cppでそれを読み込んでみましょう。  
シェーダー・プログラム作成コードを次のように変更してください。

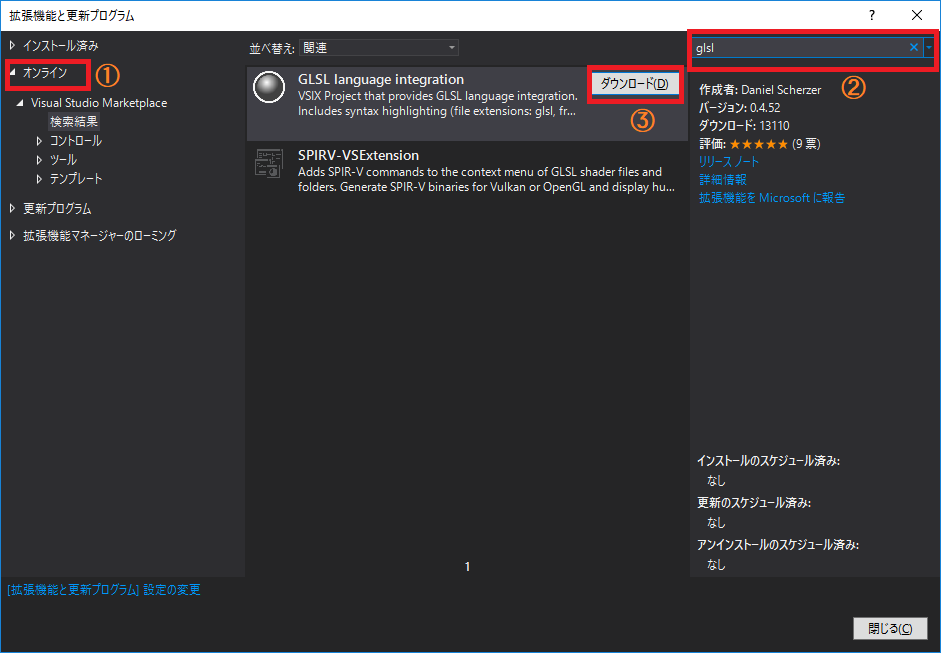
const GLuint vao = CreateVAO(vbo, ibo);  
-const GLuint shaderProgram = Shader::Build(vsCode, fsCode);  
+const GLuint shaderProgram =  
+ Shader::BuildFromFile("Res/Simple.vert", "Res/Simple.frag");  
 if (!vbo || !ibo || !vao || !shaderProgram) {  
 return 1;  
 }

変更したら、**ビルドして実行してください。**  
これまでどおりの画面が表示されれば成功です。

## シェーダー・ファイルを見やすくする

シェーダー・ファイルはただのテキスト・ファイルなので、単語に色をつけたり、間違った部分に赤色の波線をつけたりはしてくれません。シェーダー・ファイルを編集するにはとても不便です。  
ですが、安心してください。Visual Studioには、本来持っていない機能を追加するために「拡張機能」というものがあるんです(MODのようなものです)。これを使えば、シェーダー・ファイルに色を付けたり、間違った部分を指摘してくれるようにできます。

では早速、機能を追加しましょう。メニューから「ツール→拡張機能と更新プログラム」を選択してください。すると、次のようなウィンドウが開きます。



左側のリストから「オンライン」を選択し(①)、右上の検索ボックスに「glsl」と入力してください(②)。  
すると、中央にいくつか候補が表示されると思います。その中から「GLSL language integration(じーえるえすえる・らんげーじ・いんてぐれーしょん)」を選択し、「ダウンロード」ボタンを押してください(③)。すると、小さなウィンドウが開いてダウンロードが始まります。

このウィンドウはダウンロードが終わったら自動的に閉じます。そうしたら、右下の「閉じる」ボタンで「拡張機能と更新プログラム」ウィンドウを閉じてください。ウィンドウを閉じたら、Visual Studioを終了してください。  
すると、次のようなウィンドウが表示されて、さきほど選択した拡張機能のインストールが始まります。



右下の「変更」ボタンを押すと、インストールが開始されます。インストールが終わるまでしばらく待ってください。  
インストールが完了すると、ウィンドウは次のような表示になります。



右下の「閉じる」ボタンを押してウィンドウを閉じます。そして、Visual Studioを起動してください。  
Visual Studioが起動したら、Simple.vertやSimple.fragを開いてみてください。  
いくつかの単語に色が付いていたら、インストールは成功しています。

# 明かりを灯す

## ライティング

木と家、それに岩が表示されたことで、画面が賑やかになってきましたね。ただ、ちょっと見た目が単調な気がしませんか？　なにかが足りないような…。参考にするために、現実世界に目を向けてみましょう。  
現実世界はさまざまな色であふれています。真っ白な壁でも、よく見ると微妙な陰影がついています。なぜなら、さまざまな角度から来た光に照らされているからです。現実世界では、光があるからこそ物体を見ることができるのです。

実は、みなさんが作った世界にも光はあります。詳しいことは後で説明しますが、みなさんが書いたシェーダーは「全体が均一な光で包まれている」状態を表現しているのです。だから木や家が表示されるんですね。しかし、この状態は陰影を表現することができません。頂点の色を調整することでそれらしい陰影を付けることはできますが、物体が回転した途端に御破算です。それから、現実世界の太陽などは、時刻によって異なる方向から光を投げかけてくれますよね。だからこそ、朝や昼、夕方や夜という一日の移り変わりがあるわけです。しかし、現在のシェーダーではこれを再現できません。そこで、もう少しだけ複雑な光の表現を作ることにします。

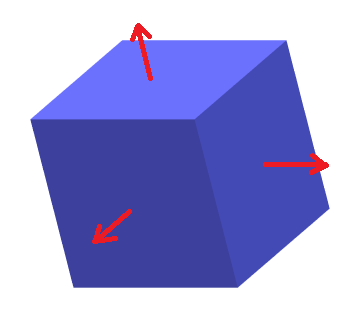
OpenGLの世界では「光」もまたデータの一種です。そして、「光のデータをもとに物体表面の明るさを計算する」ことによって、画面に表示する色が決まります。この工程を「ライティング」といいます。OpenGLの世界では、ライティングはシェーダーのお仕事です。ですから、光のデータをシェーダに送る必要があります。また、光が物体を照らす明るさは、光の位置と物体の表面の角度によって決まります。これを決めるために「物体の表面の角度」、という情報もシェーダーに送らなければなりません。

## 頂点データに法線(normal)を追加する

まず、「物体の表面の角度」をデータにします。いくつか方法がありますが、最も一般的なのは「頂点における表面の角度」を追加することです。「頂点における角度」なのですから、追加先はもちろん「頂点データ」です。  
ということで、頂点データに次のプログラムを追加してください。

/// 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.  
 };

「物体の表面の角度」をより詳しく説明すると、「物体表面のある点に接する平面(接平面)において、物体の外側へ向かう垂直な直線」ということになります。このような直線を「法線」と呼びます。

  
「赤い矢印が法線です」

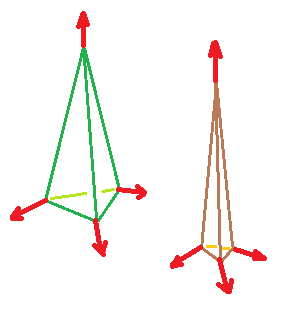
**[補足]** 「法線」という名前は、英語の「normal(ノーマル)」を翻訳したものです。ただ、「normal」の翻訳が何故「法線」になったのか、正確なところは伝わっていません。normalには「標準、規則、垂直」などの意味があるのですが、この単語が最初に舶来したとき、「規則→法」という翻訳がなされた結果、数学書に書かれているnormalにも同じ訳があてられたという説や、先に中国で訳されたものが入ってきたという説があります。とはいえ、いずれも噂に過ぎず、真相は不明です。

## 頂点データ配列に法線データを追加する

これで法線データを付け加えられるようになりました。  
とりあえず、木のモデルに法線を追加しましょう。頂点データ配列を次のように変更してください。

/// 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, 1.5f,-1.10f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.0f, 0.0f } },  
- { {-0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 0.5f, 0.0f } },  
- { { 0.75f, 1.5f, 0.40f}, {0.1f, 0.3f, 0.0f, 1.0f}, { 1.0f, 0.0f } },  
- { { 0.00f, 4.0f, 0.00f}, {0.2f, 0.1f, 0.0f, 1.0f}, { 0.0f, 1.0f } },  
- { { 0.00f, 0.0f,-0.37f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 0.0f, 0.0f } },  
- { {-0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 0.5f, 0.0f } },  
- { { 0.25f, 0.0f, 0.13f}, {0.5f, 0.3f, 0.2f, 1.0f}, { 1.0f, 0.0f } },  
+ { { 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

紙面の都合で頂点データの途中に改行を入れていますが、実際に書くときは途中の改行はしないほうが見やすくなると思います。木の法線はおおよそ次の図のようにしています。

法線は「方向を示すベクトル」です。方向を示すベクトルは、その長さが1でなければなりません。「長さが1のベクトル」のことを「単位ベクトル」といいます。単位ベクトルは次式を満たします。

一般的に、頂点の法線は「頂点に接する全ての面の法線の平均したもの」になります。実際に木の法線データは、頂点に接する面法線を平均して、長さがおおよそ1になるような値に設定してあります。

## VAOに法線データを追加する

頂点データのメンバ変数を増やしたので、増えたメンバをVAOに反映する必要があります。  
CreateVAO関数に次のプログラムを追加してください。

GLuint CreateVAO(GLuint vbo, GLuint ibo)  
 {  
 GLuint vao = 0;  
 glGenVertexArrays(1, &vao);  
 glBindVertexArray(vao);  
 glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  
 glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ibo);  
 const GLsizei stride = sizeof(Vertex);  
 glEnableVertexAttribArray(0);  
 glVertexAttribPointer(0, sizeof(Vertex::position) / sizeof(float),  
 GL\_FLOAT, GL\_FALSE, stride, (const GLvoid\*)(offsetof(Vertex, position)));  
 glEnableVertexAttribArray(1);  
 glVertexAttribPointer(1, sizeof(Vertex::color) / sizeof(float),  
 GL\_FLOAT, GL\_FALSE, stride, (const GLvoid\*)(offsetof(Vertex, color)));  
 glEnableVertexAttribArray(2);  
 glVertexAttribPointer(2, sizeof(Vertex::texCoord) / sizeof(float),  
 GL\_FLOAT, GL\_FALSE, stride, (const GLvoid\*)offsetof(Vertex, texCoord));  
+ glEnableVertexAttribArray(3);  
+ glVertexAttribPointer(3, sizeof(Vertex::normal) / sizeof(float),  
+ GL\_FLOAT, GL\_FALSE, stride, (const GLvoid\*)(offsetof(Vertex, normal)));  
 glBindVertexArray(0);  
 glDeleteBuffers(1, &vbo);  
 glDeleteBuffers(1, &ibo);  
 return vao;  
 }

## ライティング・シェーダーを作成する

今回は、新しくライティングを行うシェーダーを書くことにします。  
Simple.vertと同じ要領で、Resフォルダに「VertexLighting.vert(ばーてっくす・らいてぃんぐ・ばーと)」というファイルを追加してください。  
追加したVertexLighting.vertファイルを開き、Simple.vertの内容をコピーして貼り付けます。

+/\*\*  
+\* @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か。  
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 VertexLighting.vert  
 \*/  
 #version 410

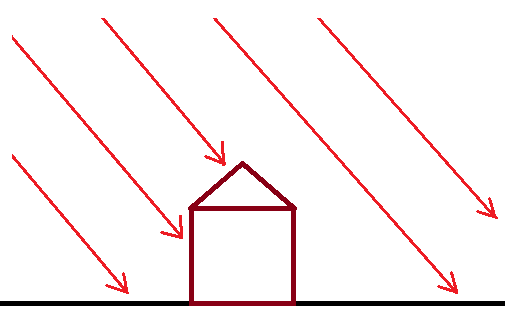
次に、法線を受け取る入力変数を追加します。  
テクスチャ座標の入力変数定義の下に、次のプログラムを追加してください。

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

次に、光のデータを定義します。  
テクスチャ座標の出力変数定義の下に、次のプログラムを追加してください。

layout(location=0) out vec4 outColor;  
 layout(location=1) out vec2 outTexCoord;  
+  
+// 指向性ライト.  
+struct DirectionalLight  
+{  
+ vec3 direction;  
+ vec3 color;  
+};  
+  
+uniform DirectionalLight directionalLight;  
  
 uniform mat4x4 matMVP;  
  
 /\*\*  
 \* 頂点シェーダー.  
 \*/

光を放つ物体を「ライト」あるいは「光源(こうげん)」と呼びます。ライトにも様々な種類がありますが、今回は「Directional light(でぃれくしょなる・らいと、指向性ライト)」と呼ばれるものを追加していきます。  
「指向性ライト」は、太陽や月のように、「非常に遠くにあって、強力な光を放つ物体」を表現するためのライトです。単一の方向に平行な光線を放つので、「平行光源」とも呼ばれます。

  
「赤い矢印が光線です」

指向性ライトはふたつのパラメーターから作ることができます。ひとつは光の方向、もうひとつは光の色です。上記のプログラムでは、それぞれdirection(でぃれくしょん、「方向」という意味)とcolor(カラー)という2つのメンバ変数を定義しています。

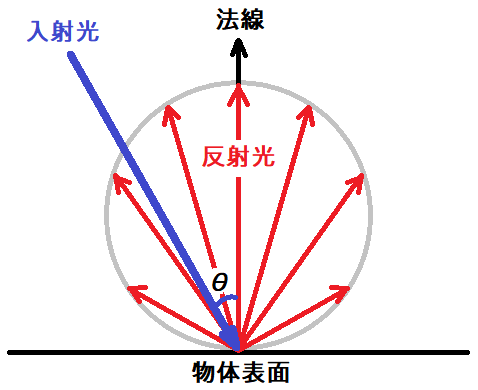
**[補足]** GLSLの構造体はおおむねC++言語のものと同じですが、以下のような違いがあります。  
- 無名構造体を定義することはできません。  
- 構造体内で構造体を定義することはできません。  
 しかし、グローバルに定義した構造体を、別の構造体のメンバにすることは可能です。  
- 構造体は様々な場所で使う事ができますが、シェーダの入出力(in/out修飾子のついた変数)としては使えません。  
- 初期化構文として、C++クラスのコンストラクタのように「()」を使うことができます。

それでは、用意したデータを使って物体の表面の明るさを計算しましょう。  
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()  
{  
- outColor = vColor;  
+ // ランバート反射による拡散反射光の計算.  
+ float cosTheta = clamp(dot(vNormal, -directionalLight.direction), 0.0, 1.0);  
+ outColor.rgb = (vColor.rgb \* directionalLight.color \* cosTheta);  
+ outColor.a = vColor.a;  
+  
 outTexCoord = vTexCoord;  
 gl\_Position = matMVP \* vec4(vPosition, 1.0);  
 }

コンピューター・グラフィックスでは、物体の表面の明るさも計算で算出します。しかし、現実世界を再現しようとすると膨大な計算が必要で、ゲームのように1/60秒で画面を描き出すことは不可能です(Q: CG映画の1画面分の計算にどれだけの時間がかかるでしょう？　A: 数万個のCPUを使って、短くて数時間、長ければ数日)。  
そんなわけで、簡単な計算でそれっぽい見た目になる方法が考え出されました。  
上記のプログラムでは、そんな方法のひとつである「ランバート反射」を使っています。

ランバート反射は、「物体のある点で反射した光(反射光)の強さは、その点の法線と光源方向のなす角θの余弦(コサイン)と正比例する」という法則(「ランベルトの余弦則」といいます)を利用した計算方法です。  
ランバート反射は次の図で表されます。



余弦を使うことで、「光が垂直に入射してきたときに最も明るく反射し、表面に対して入射角度がきつくなるほど反射が弱くなる」という、現実の多くの物体が持つ性質を表すことができるわけですね。

**[補足]** 「ランベルトの余弦則」と「ランバート反射」は、これを発見したドイツの数学者「ヨハン・ハインリヒ・ランベルト(Johann Heinrich Lambert)」の名前にちなんでなづけられました。微妙な名前の違いは、分野による読み方の慣習によります。数学徒はドイツ語読みで「ランベルト」、物理学徒は英語読みで「ランバート」と呼んだ、ということですね。

上記のプログラムにおいて、ランバート反射はdot関数で表されています。  
「dot(どっと)」は内積を計算するGLSLの関数です。この式では、「2つの単位ベクトルがなす角をθ(しーた)としたとき、ベクトルの内積はcosθ(こす・しーた)と等しい」という事実を利用しています。なお、物体から見た光源方向は、光の向きの真逆になるため、光の向きを逆にしている(マイナスをつけて符号を反転している)ことに注意してください。

ランバート反射の式の外側にある「clamp(くらんぷ)」もGLSLの関数です。この関数は、数値が設定された最小値と最大値を越えないように制限します。上記のプログラムの場合、最小値は0.0で最大値は1.0にしています。光が物体表面の反対側から入射している場合、cosθは負の値になります。しかし、マイナスの光というのは存在しません。ですから、clamp関数を使って0を下回らないようにしているわけです。

ランバート反射の強さが得られたら、あとはその強さとライトの色、そして物体の色の3つを掛け合わせることで最終的に表示する色が決まります。乗算なので、色成分が0の場合はどんなに強い光を当てても0のままです。現実の物体は、たとえ純粋な赤色に見えるものでも、実際にはわずかに緑や青の色成分が含まれます。現在最も黒い物質と呼ばれる「ベンタブラック」ですら光の0.04%は反射します。色成分が0になるのはブラックホールくらいでしょう。

**表示する色の計算式:**  
 表示する色 = 物体の色 × ライトの色 × cos(ライトの向きと法線がなす角θ)

**[課題01]** (1.0, 1.0, 0.7)という色の光が、(0.5, 0.2, 0.1)という色の物体に対して垂直方向に入射したときの、最終的な色を求めなさい。計算にはWindows 10の電卓アプリや(左上の「三」をクリックして「関数電卓」を選ぶとよい)、スマホの電卓アプリなどを使うとよいでしょう。

**[課題02]** 課題01の光と物体について、光が45°、90°、120°の角度で入射したときの最終的な色を、小数点以下第２位まで求めなさい。

## シェーダー・ファイルを読み込む

それでは、作成したVertexLighting.vertからプログラム・オブジェクトを作成しましょう。  
Main.cppに次のプログラムを追加してください。

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

ライティング用のフラグメントシェーダーには、Simple.fragを使い回すことにします。

作成したら破棄する義務があります。  
main関数の終わり付近に、次のプログラムを追加してください。

glDeleteTextures(1, &texHouse);  
 glDeleteTextures(1, &texId);  
+glDeleteProgram(progLighting);  
 glDeleteProgram(shaderProgram);  
 glDeleteVertexArrays(1, &vao);  
  
 return 0;

## uniform変数の位置を取得する

作成したプログラム・オブジェクトのuniform変数の位置を取得します。  
shaderProgram用のuniform変数の位置を取得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 GLint texColorLoc = glGetUniformLocation(shaderProgram, "texColor");  
 if (texColorLoc >= 0) {  
 glUniform1i(texColorLoc, 0);  
 }  
+  
+ const GLint locMatMVP = glGetUniformLocation(progLighting, "matMVP");  
+ const GLint locDirLightDir =  
+ glGetUniformLocation(progLighting, "directionalLight.direction");  
+ const GLint locDirLightCol =  
+ glGetUniformLocation(progLighting, "directionalLight.color");  
+ if (locDirLightDir < 0 || locDirLightCol < 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);

「matMVP」というuniform変数名はshaderProgramのものと同じです。しかしuniform変数はプログラム・オブジェクトごとに定義が作られます。ですから、プログラム・オブジェクトが異なれば、変数名が同じでも別ものとして扱われます。別の変数なので、場所も個別に取得しなければなりません。

今回はシェーダー・プログラムで構造体を使いましたが、構造体の変数名だけで位置を取得することはできません。かわりに、「構造体変数名.メンバ変数名」という形で、メンバ変数ごとに位置を取得します。  
位置を示す変数名「locDirLightDir(ろく・でぃあ・らいと・でぃあ)」は「location of Directional Light Direction(ろけーしょん・おぶ・でぃれくしょなる・らいと・でぃれくしょん)」を省略したものです。そのまま変数名にするには、ちょっと長すぎるので…。それから、位置を示す変数名は「loc」で始まるようにしてみました。

## 複数のプログラム・オブジェクトを使う

法線データを設定したのは木のモデルだけです。家や岩などは法線を設定していないので、正しいライティングができません。そこで、木を描画するときだけライティング用のシェーダーを使い、他のモデルは既存のシェーダーを使うようにしましょう。特に難しいことはありません。モデルを描画する前にglUseProgramでシェーダーを切り替えるだけです。  
メインループ内で使用するプログラム・オブジェクトを選択している部分を、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(shaderProgram);  
+glUseProgram(progLighting);  
  
 // 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));

そして、木のモデルを描き終わったところでshaderProgramを設定します。  
木のモデルを描画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。

for (float i = 0; i < treeCount; ++i) {  
 const float x = std::cos(3.14f \* 2 / treeCount \* i) \* radius;  
 const float z = std::sin(3.14f \* 2 / treeCount \* i) \* radius;  
 const glm::mat4x4 matModel = glm::translate(glm::mat4(1), glm::vec3(x, 0, z));  
 const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
 glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
 glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
 }  
  
+glUseProgram(shaderProgram);  
  
 {  
 glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, texHouse);

これで、プログラム・オブジェクトの切り替えは完了です。

## 光のデータをGPUメモリに転送する

progLightingに、指向性ライトのデータを設定しましょう。  
メインループの手前に、次のプログラムを追加してください。

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 dirLightDir = glm::normalize(glm::vec3(1,-1,-1)); // 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.  
+  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);  
 glEnable(GL\_CULL\_FACE);

指向性ライトの方向は、原点から「右に5m、下に2m、前に2m」に移動した位置を指すようなベクトルです。ただし、方向としてのベクトルは「単位ベクトル(長さが1のベクトル)」になっている必要があります。長さが異なるベクトル同士で比較や計算をすると、正しい結果が得られないからです。そこで、glm::normalize(じーえるえむ・のーまらいず)関数を使います。この関数は、引数に設定したベクトルの向きはそのままで、長さだけ1にしたベクトルを返します。

**[補足]** 方向ベクトル同士の計算は長さが揃ってさえいればいいので、長さが0.3や20でも計算自体は可能です。しかし、単位ベクトルならば、長さによる乗算や除算を無視できるので都合がいいのです。

光の色は頂点の色と同じく「赤、緑、青」の成分で設定します。今回の場合、すべての成分が1.0fなので「真っ白な光」になります。

続いて、座標変換行列を作成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。

// 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));  
  
+// ライトをGPUメモリに転送する.  
+glUniform3fv(locDirLightDir, 1, &dirLightDir.x);  
+glUniform3fv(locDirLightCol, 1, &dirLightCol.x);  
+  
 glBindVertexArray(vao);  
  
 glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, texId);

uniform変数に値を設定する方法は、これまで見てきたものと大差ありません。今回の転送先はvec3なので、glUniform3fv(じーえる・ゆにふぉーむ・さん・えふ・ぶい)関数を使っているくらいです。

## 木のMVP行列を設定する

プログラム・オブジェクトが異なると、uniform変数の位置も異なります。先ほど、木を描画するときのプログラム・オブジェクトを変更したので、MVP行列の転送先も変更しなくてはなりません。  
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。

for (float i = 0; i < treeCount; ++i) {  
 const float x = std::cos(3.14f \* 2 / treeCount \* i) \* radius;  
 const float z = std::sin(3.14f \* 2 / treeCount \* i) \* radius;  
 const glm::mat4x4 matModel = glm::translate(glm::mat4(1), glm::vec3(x, 0, z));  
 const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
- glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
+ glUniformMatrix4fv(locMatMVP, 1, GL\_FALSE, &matMVP[0][0]);  
 glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
 }

プログラムが書けたら、**ビルドして実行してください。**すこし分かりにくいですが、光が入射してくる方向が明るく、そうでない方向が暗く表示されていれば成功です。



**[課題03]** 指向性ライトの色を、オレンジ色にしてください。

**[課題04]** XおよびZ軸方向に±10mの大きさを持つ法線付き地面モデルを作成してください。地面モデルの法線は全て上向き(0, 1, 0)とし、色は適当に決めてください。そして、作成した地面モデルを原点に配置し、progLightingを使って描画してください。

# 指向性ライトを回転に対応させる

## 木の向きを変えてみる

唐突ですが、全ての木が同じ角度で生えているのは不自然ですよね。そこで、1本ごとに植える角度を変えてみまましょう。木を植えるプログラムを、次のように変更してください。

for (float i = 0; i < treeCount; ++i) {  
- const float x = std::cos(3.14f \* 2 / treeCount \* i) \* radius;  
- const float z = std::sin(3.14f \* 2 / treeCount \* i) \* radius;  
- const glm::mat4x4 matModel = glm::translate(glm::mat4(1), glm::vec3(x, 0, z));  
- const glm::mat4x4 matMVP = matProj \* matView \* matModel;  
+ 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]);  
 glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
 }

回転させる角度の計算を何度も書くのは、いざ何かを書き換えようとしたときに変更箇所が増えて面倒です。そこで、回転角の計算結果をtheta(しーた)という変数に代入し、この変数を使うように変えました。  
モデルを回転させるための行列を作るにはglm::rotate()を使います。最初の引数は元になる行列、2つめの引数が弧度法による回転角で、3つめが回転軸です。なお、回転角に5を掛けているのは、thetaだけだと角度の違いが分かりにくかったので適当に何倍かしてみたところ、5倍くらいがちょうどよさそうだったからです。

プログラムが書けたら、**ビルドして実行してください。**  
…分かりにくいですが、光の当たり方がばらばらになっているように見えます。  
回転させただけなのに、どうしてこんなことになるんでしょう。  
それは、回転させたのはモデルの座標だけで、法線の向きが変化していないからです。

## 指向性ライトの向きをモデル座標系に変換する

物体が回転すれば、その表面の向きも回転します。ということは、法線の向きが変わるはずですね。しかし、現在のシェーダーは、法線になんの計算も加えていません。そのため、明るさを計算してから回転させた状態になってしまうのです。

この問題の解決方法は簡単です。法線も回転させればいいのです。  
さっそくプログラムを変更…といきたいところですが、ちょっとだけ待ってください。  
その昔、頭がいい人は考えました。

「頂点はたくさんあるから、法線の回転もたくさんしなくちゃならないぞ。計算時間がもったいないな…。」  
「いや待て、逆に考えるんだ。法線ではなくて、光の向きを回転させちゃえばいいさ、と考えるんだ。」

そうなんです。物体の表面の立場からすると、木が回転したから明るさが変わったのか、光の向きが回転したから明るさが変わったのかは、結果が同じならどちらでも構わないのです。これは一種のトリックなので、すぐには腑に落ちないと思います。「確かにそのとおりだ」となるまでじっくり考えてください。

この方針に従って、プログラムを書きましょう。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::mat3(matModelR)) \* dirLightDir;  
+ glUniform3fv(locDirLightDir, 1, &dirLightDirOnModel.x);  
+  
 glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
 }

ライトの向きを逆回転させるには、回転行列の逆行列を作り、それをライトの向きベクトルに掛けます。  
これは、「ワールド座標系で定義されたライトの向きを、モデル座標系から見た向きに変換する」操作になります。「逆行列」というのは、「元の行列に掛け合わせると単位行列になる行列」のことです。任意の行列Aと単位行列Eに対してAB=BA=Eとなるような行列Bがあるとき、「BはAの逆行列である」といいます。実数の逆数の行列バージョンですね。  
さて、逆行列を得るにはglm::inverse(じーえるえむ・いんばーす)関数を使います(inverse=逆の、反対の)。また、回転行列を表すには3x3の大きさがあれば十分なので、glm::mat3コンストラクタを使ってmat4からmat3に変換しています。なお、回転行列が4x4なのは、ビュー行列やプロジェクション行列と掛け合わせるためです。行列の掛け算は大きさを揃える必要があるのです。

向きを計算したら、glUniform3fv関数でuniform変数に転送します。  
こうしてモデルごとに向きを計算し直すことで、ライトがモデルの回転に対応するようになるわけです。

プログラムが書けたら、**ビルドして実行してください。**  
これも分かりにくいですが、光の向きを回転させる前と比べて、光の当たり方が統一されているように見えていれば成功です。

**[課題05]** 地面モデルの表示プログラムに、指向性ライトの向きをモデル座標系に変換してGPUメモリに転送するプログラムを追加してください。

# 環境光(Ambient Light)

## 光が当たらない部分は見えない？

指向性ライトを導入したことで、光の色や当たり具合を調整できるようになりました。  
ただ、光が当たっていない部分は真っ黒になってしまいます。光が当たらないのだから、当然のことに思えますよね。でも、現実世界はそんなに単純ではありません。

現実の世界では、太陽に照らされていない部分が真っ黒になったりはしません。光は周囲の様々な物体で反射するので、直接光が当たらない部分も、そういった間接的な光で照らされているからです。  
全ての光の反射を計算できれば現実のような見た目にできるのですが、それには膨大な計算が必要で、とても1/60秒では足りません。そこで「すべての物体はあらゆる間接光の平均値で照らされている」とみなすことで、この計算を簡単にする方法が考えられました。これが「環境光(かんきょうこう)」、あるいは「Ambient Light(あんびえんと・らいと)」と呼ばれるものです(Ambientは「周囲」や「環境」といった意味です)。

**[補足]** 環境光はあまりに単純な近似なので、現代のコンピューター・グラフィックスではほとんど使われていません。しかし、現代的な技法には、さまざまな前提条件の設定と複雑な数式が必要になります。まずは基礎的な技法から学習していきましょう。

## ライティング・シェーダーに環境光を導入する

それでは、環境光を追加していきましょう。まずはシェーダーに環境光のデータを定義します。  
VertexLighting.vertを開き、次のプログラムを追加してください。

layout(location=0) out vec4 outColor;  
 layout(location=1) out vec2 outTexCoord;  
+  
+// 環境光.  
+struct AmbientLight  
+{  
+ vec3 color;  
+};  
  
 // 指向性ライト.  
 struct DirectionalLight  
 {  
 vec3 direction;  
 vec3 color;  
 };  
  
+uniform AmbientLight ambientLight;  
 uniform DirectionalLight directionalLight;  
  
 uniform mat4x4 matMVP;

環境光は空間全体を均一な明るさで照らします。そのため、方向は持たず、色を表すメンバ変数だけを定義しています。

続いて、環境光を物体に当てましょう。  
VertexLighting.vertのmain関数に、次のプログラムを追加してください。

void main()  
 {  
 // ランバート反射による拡散反射光の計算.  
 float cosTheta = clamp(dot(vNormal, -directionalLight.direction), 0.0, 1.0);  
 outColor.rgb = (vColor.rgb \* directionalLight.color \* cosTheta);  
+  
+ // 環境光の計算.  
+ outColor.rgb += ambientLight.color;  
 outColor.a = vColor.a;  
  
 outTexCoord = vTexCoord;  
 gl\_Position = matMVP \* vec4(vPosition, 1.0);  
 }

これでシェーダーの変更は完了です。

## uniform変数の位置を取得する

続いて、追加した環境光のuniform変数の位置を取得します。  
指向性ライトのuniform変数の位置を取得している部分に、次のプログラムを追加してください。

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");  
- if (locDirLightDir < 0 || locDirLightCol < 0) {  
+ if (locDirLightDir < 0 || locDirLightCol < 0 || locAmbLightCol < 0) {  
 std::cerr << "ERROR: uniform変数の位置を取得できません.\n";  
 return 1;  
 }

環境光の位置を示す変数名は「location of Ambient Light Color(ろけーしょん・おぶ・あんびえんと・らいと・からー)」を略して「locAmbLightCol(ろく・あんぶ・らいと・こる)」としました。

## 環境光のデータを転送する

環境光のuniform変数の位置を取得したら、次はその変数にデータを転送します。  
まずライトのデータを追加します。指向性ライトを定義している部分に、次のプログラムを追加してください。

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.

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

これで、指向性ライトに加えて環境光が世界を照らすようになりました。  
プログラムが書けたら、**ビルドして実行してください。**  
分かりにくいですが、木が全体的にわずかに明るくなり、以前は真っ黒だった部分にほんの少しだけ色が付いていたら成功です。

**[課題06]** 環境光の色を、暗めの紫色(あるいは好きな色)に変更してください。