[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) {  
+ // フラグメントからライトへ向かうベクトルを計算.  
+ vec3 lightVector = pointLight.position[i] - vPosition;  
+ vec3 lightDir = normalize(lightVector);  
+ // 面の傾きによる明るさの変化量を計算.  
+ float cosTheta = clamp(dot(vNormal, lightDir), 0.0, 1.0);  
+ // 距離による明るさの変化量を計算.  
+ float intensity = 1.0 / (1.0 + dot(lightVector, lightVector));  
+ // 変化量をかけ合わせて明るさを求め、合計に加算.  
+ lightColor += pointLight.color[i] \* cosTheta \* intensity;  
+ }  
+ }  
+  
+ outColor.rgb = vColor.rgb \* lightColor;  
  
 **// 環境光の計算.**  
 outColor.rgb += vColor.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、全てのライトの明るさを合計するlightColor(らいと・からー)という変数を用意しました。今後は、あるライトの明るさを計算するたびに、この変数に加算していくことにします。全てのライトから受ける光の量を合計したら、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の単位ベクトルを計算して返してくれます。

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。なお、除数に1.0を足しているのは、距離0のときの明るさの変化量が1になるようにするためです。

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下に、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);  
  
+ // ポイント・ライトの位置をモデル座標系に変換してGPUメモリに転送する.  
+ const glm::mat4 matInvModel = glm::inverse(matModelT \* matModelR);  
+ glm::vec3 pointLightPosOnModel[8];  
+ for (int i = 0; i < 8; ++i) {  
+ pointLightPosOnModel[i] = matInvModel \* glm::vec4(pointLightPos[i], 1);  
+ }  
+ glUniform3fv(locPointLightPos, 8, &pointLightPosOnModel[0].x);  
+  
 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。逆行列であることを示すため、変数名はmatInvModel(まっと・いんぶ・もでる)としました。Invはinverse(いんばーす)の略記です。

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]** 地面モデルを表示するプログラムに、ポイント・ライトの位置をモデル座標系に変換してGPUメモリに転送するプログラムを追加してください。

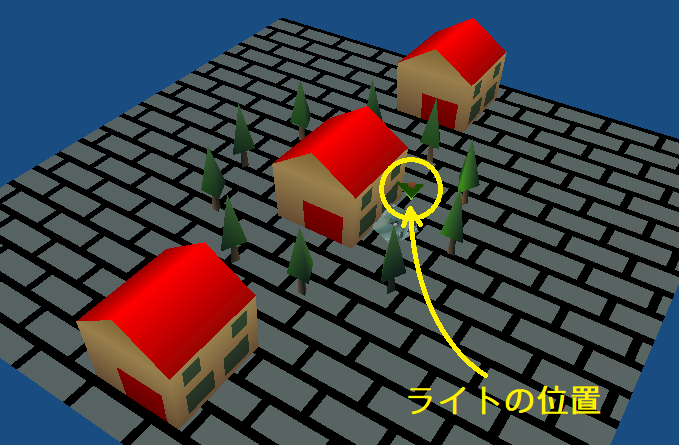
# キー入力

## ライトの位置を表示する

課題01が完了していれば、地面はポイント・ライトで照らされているはずです。しかし、ほとんど、あるいはまったくそうは見えないと思います。その理由は、頂点ライティングでは、頂点がライトの照らす範囲になければライティングされないからです。  
このことを調べるには、ポイント・ライトを移動させてみるのが簡単です。ただ、ポイント・ライトは直接目にすることはできませんので、一体どこから照らしているかが分かりにくいですよね。実際のゲーム制作でも、ライトの位置が分かるほうが便利です。そこで、ライトの位置に適当なモデルを表示してみましょう。  
Main.cppのメインループの末尾に、次のプログラムを追加してください。

glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
 glDrawElementsBaseVertex(meshList[2].mode, meshList[2].count,  
 GL\_UNSIGNED\_SHORT, meshList[2].indices, meshList[2].baseVertex);  
 }  
  
+ // 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.  
+ {  
+ // モデルのY軸回転角を更新.  
+ static float angleY = 0;  
+ angleY += glm::radians(1.0f);  
+ if (angleY > glm::radians(360.0f)) {  
+ angleY -= glm::radians(360.0f);  
+ }  
+  
+ // ライトの位置に木のモデルを表示.  
+ glUseProgram(shaderProgram);  
+ glActiveTexture(GL\_TEXTURE0);  
+ glBindTexture(GL\_TEXTURE\_2D, texId);  
+ for (int i = 0; i < 8; ++i) {  
+ const glm::mat4 matModelT = glm::translate(glm::mat4(1), pointLightPos[i]);  
+ const glm::mat4 matModelR = glm::rotate(glm::mat4(1),angleY, glm::vec3(0,1,0));  
+ const glm::mat4 matModelS = glm::scale(glm::mat4(1),glm::vec3(1, -0.25f, 1));  
+ const glm::mat4 matMVP = matProj \* matView \* matModelT \* matModelR \* matModelS;  
+ glUniformMatrix4fv(matMVPLoc, 1, GL\_FALSE, &matMVP[0][0]);  
+ glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
+ GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);  
+ }  
+ }  
+  
 glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, 0);  
  
 window.SwapBuffers();  
 }

プログラムが書けたら、**ビルドして実行してください。**  
ポイント・ライトの位置に、逆さまになった小さな木が表示されていたら成功です。



## GLFWを使ってキー入力を調べる

続いて、ライトの位置をキーボードで動かせるようにしてみましょう。GLFWライブラリには、どのキーが押されているかを簡単に調べられる関数が用意されています。まずは、この関数をGLFWEW::Windowから使えるようにします。GLFWEW.hを開き、Windowクラスに、次のプログラムを追加してください。

static Window& Instance();  
 bool Init(int w, int h, const char\* title);  
 bool ShouldClose() const;  
 void SwapBuffers() const;  
  
+ bool IsKeyPressed(int key) const;  
  
 private:  
 Window();  
 ~Window();

関数名はIsKeyPressed(いず・きー・ぷれすど=「キーは押されていますか？」)としました。

次に、関数の定義を追加します。GLFWEW.cppを開き、SwapBuffers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::SwapBuffers() const  
 {  
 glfwPollEvents();  
 glfwSwapBuffers(window);  
 }  
  
+/\*\*  
+\* キーが押されているか調べる.  
+\*  
+\* @param key 調べたいキー・コード(GLFW\_KEY\_Aなど).  
+\*  
+\* @retval true キーが押されている.  
+\* @retval false キーが押されていない.  
+\*   
+\* 使用できるキー・コードの一覧はglfw3.hにあります(「keyboard」で検索).  
+\*/  
+bool Window::IsKeyPressed(int key) const  
+{  
+ return glfwGetKey(window, key) == GLFW\_PRESS;  
+}  
+  
} // namespace GLFWEW

キーが押されているかを調べるにはglfwGetKey(じーえる・えふ・だぶりゅー・げっと・きー)関数を使います。  
glfwGetKeyは、引数にキー・コードと呼ばれるキーの種類を設定すると、そのキーが押されているかどうかを返してくれる関数です。キーが押されていればGLFW\_PRESS(じーえる・えふ・だぶりゅー・ぷれす)、押されていなければGLFW\_RELEASE(じーえる・えふ・だぶりゅー・りりーす)が返されます。

今回作成する関数は、glfwGetKey関数から返された値をそのまま返すのではなく、bool型に変換してから返すようにしました。「押されていればtrueを返す」ようにしたいので、戻り値をGLFW\_PRESSと比較し、その結果を返しています。

## ライトを動かす

これで、押されているキーを調べられるようになりました。それでは、ライトを動かすプログラムを書きましょう。  
Main.cppを開き、ポイント・ライトの位置にモデルを表示するプログラムに、次のプログラムを追加してください。

// 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;  
+ if (window.IsKeyPressed(GLFW\_KEY\_A)) {  
+ pointLightPos[0].x -= speed;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
+ pointLightPos[0].x += speed;  
+ }  
+ if (window.IsKeyPressed(GLFW\_KEY\_LEFT\_SHIFT)) {  
+ if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
+ pointLightPos[0].y += speed;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
+ pointLightPos[0].y -= speed;  
+ }  
+ } else {  
+ if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
+ pointLightPos[0].z -= speed;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
+ pointLightPos[0].z += speed;  
+ }  
+ }  
+  
 // モデルのY軸回転角を更新.  
 static float angleY = 0;  
 angleY += glm::radians(1.0f);  
 if (angleY > glm::radians(360.0f)) {  
 angleY -= glm::radians(360.0f);  
 }

Window::IsKeyPressed関数でW/A/S/Dキーの状態を調べ、押されているキーに応じて0番目のポイント・ライトの位置を変えています。さらに、左シフトキーが押されている場合はW/Sキーの移動方向を切替えています。

## 視点を固定する

現在のプログラムでは、視点の位置を移動させています。しかし、何かを動かそうとする時に、視点が固定されていないと、操作がとても難しくなります。いい加減、画面が回り続けるのにも飽きてきたことでしょうし、とりあえず視点を固定してしまいましょう。  
メインループにある視点を移動させるプログラムを、次のように変更してください。

glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
  
-// 視点を回転移動させる.  
-static float degree = 0.0f;  
-degree += 0.01f;  
-if (degree >= 360.0f) { degree -= 360.0f; }  
-const glm::vec3 viewPos = glm::rotate(glm::mat4(1),  
- glm::radians(degree), glm::vec3(0, 1, 0)) \* glm::vec4(-20, 30, -30, 1);  
+// 視点を設定する.  
+const glm::vec3 viewPos(20, 30, 30);  
  
 glUseProgram(progLighting);

プログラムが書けたら、**ビルドして実行してください。**  
W/A/S/Dキーを押したとき、ポイント・ライトが移動していたら成功です。  
また、左のシフトキーを押しながらW/Sキーを押すと、ライトを垂直方向に移動させることができます。

**[課題02]** ポイント・ライトを動かして、地面の明るさが変化する位置を見つけてください。

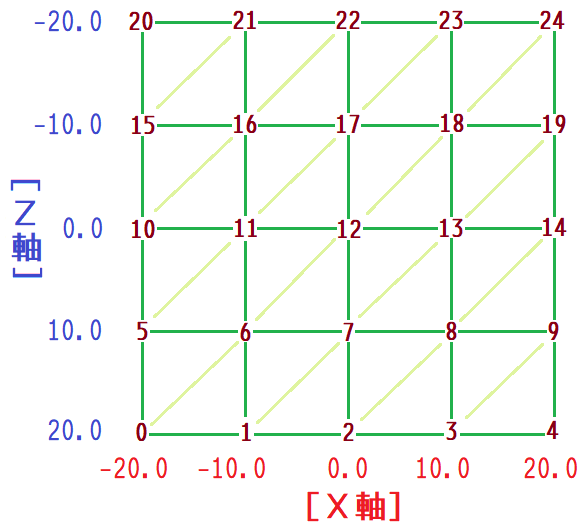
## 頂点の数を増やす

ライトを地面の端のほうに移動させると、ようやく地面に光があたるようになります。2.1節で説明したように、頂点ライティングでは、文字どおり頂点だけがライティングの対象になっているからです。つまり、頂点ライティングは現実の光を非常に荒く近似したものなのです。

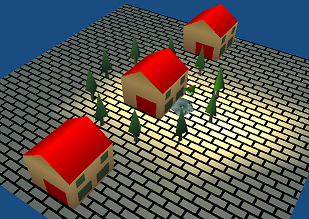
これは例えば、一日の気温の変化を知りたいとき、0時と12時と24時の気温を直線で結ぶようなものです。大まかな変化は分かりますが、気温が最も下がる早朝や、最も上がる14時～15時の情報は得られません。より正確な変化を知りたければ、3時間ごとや1時間ごとの気温を調べなければなりません。

同じことが頂点ライティングにも言えます。ですから、面積あたりの頂点の数を増やせば増やすほど、正しいライティングに近づいていくわけです。

**[課題03]** 地面のモデルを次の図のように変更し、ライトを動かして、変更した地面がどのように照らされるかを確認してください。



課題03を完了すると、わずか4頂点の地面では見られなかった明るい地面が見られると思います。もっとせまい間隔で頂点を追加していけば、より正確なライティングが行えるようになるでしょう。



しかし、コンピューターの計算能力は有限です。頂点シェーダーでライティングをするだけのために、平らな面の頂点を増やし続けるわけにはいきません。かといって頂点を減らすと、ライティングは不正確で不自然なものになってしまいます。

残念なことですが、頂点シェーダーでライティングを行う限り、正確で自然なライティングを実現することは困難です。

# 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にあたります。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。そのため、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。

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。

# フラグメント・ライティング

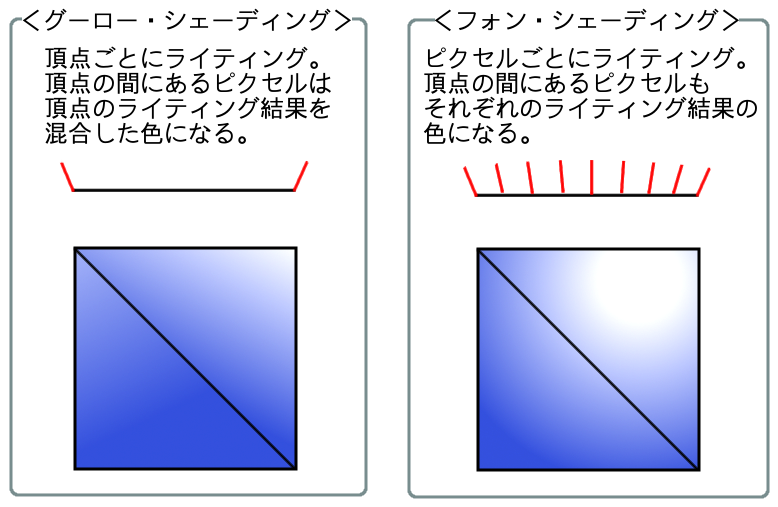
## 頂点シェーダによるライティングの問題点

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、重大な問題がありました。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！

この問題は「頂点を増やす」ことで多少は軽減できます。しかし、頂点シェーダーでライティングを行う限り、この問題を本当に解決することはできません。そこで、ライティングの舞台をフラグメントシェーダに移します。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。

## フォン・シェーディング

「頂点間の法線を補間し、フラグメント単位でライティングを行う」技法のことを、考案者の名前を取って「Phong shading(フォン・シェーディング)」と呼びます。また、前回作成した「頂点単位でライティングを行い、その結果を補間してフラグメントの色を決める」技法のことを、これも考案者の名前から「Gouraud shading(グーロー・シェーディング)」と呼びます。



## 頂点シェーダの追加

フォン・シェーディングには、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=3) 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);  
 }

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。しかし、追加されたプログラムはフォン・シェーディングを行う上で

頂点シェーダーの変更は以上です。

## 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(vNormal, -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) {  
+ // フラグメントからライトへ向かうベクトルを計算.  
+ vec3 lightVector = pointLight.position[i] - inPosition;  
+ vec3 lightDir = normalize(lightVector);  
+ // 面の傾きによる明るさの変化量を計算.  
+ float cosTheta = clamp(dot(vNormal, lightDir), 0.0, 1.0);  
+ // 距離による明るさの変化量を計算.  
+ float intensity = 1.0 / (1.0 + dot(lightVector, lightVector));  
+ // 変化量をかけ合わせて明るさを求め、合計に加算.  
+ lightColor += pointLight.color[i] \* cosTheta \* intensity;  
+ }  
+ }  
+  
 fragColor = inColor \* texture(texColor, inTexCoord);  
 }

これを、フラグメント・シェーダー用に修正していきます。

## ラスタライザは法線の長さを変えてしまう

ライティングには法線が必要です。そのために頂点シェーダーの出力に法線を追加したわけですが、それだけでは正しい法線にはなりません。その原因は、グラフィックス・パイプラインのラスタライザにあります。  
ラスタライザがフラグメント・シェーダーの入力変数に入れる値を計算するとき、頂点間の距離に応じて頂点シェーダーの出力変数を補間します。実は、ここに問題があるんです。ラスタライザは「線形補間」と呼ばれる方法で計算を行うのですが、この計算はベクトルの要素ごとに行われます。そのため、線形補間を行うとベクトルの長さが変わってしまうのです。

頂点Aの法線Anが(0.0, 1.0, 0.0)で、頂点Bの法線Bnが(0.71, 0.0, 0.71)だった場合を考えてみましょう。   
フラグメントFがAとBのちょうど中間にある場合、ラスタライザはFの法線Fnを次のように計算します。

**Fn = An \* 0.5 + Bn \* 0.5  
 = (0.0, 1.0, 0.0) \* 0.5 + (0.71, 0.0, 0.71) \* 0.5  
 = (0.0, 0.5, 0.0) + (0.355, 0.0, 0.355)  
 = (0.355, 0.5, 0.355)**

さて、Fnの長さはどうなったでしょう。3Dベクトルの長さは「√(x^2 + y^2 + z^2)」で求まります。

**|Fn| = √(x^2 + y^2 + z^2)  
 = √(0.355^2 + 0.5^2 + 0.355^2)  
 = √(0.126 + 0.25 + 0.126)  
 = √(0.502)  
 = 0.7085...**

長さは約0.7です。ずいぶん短くなってしまいましたね…。  
このように、線形補間はベクトルの長さを変えてしまいます。

ここで、(0.0, 1.0, 0.0)と(0.71, 0.0, 0.71)が法線ではなく、色の情報だった場合を考えてみましょう。  
(0.0, 1.0, 0.0)=緑色、(0.71, 0.0, 0.71)=濃い紫色、(0.355, 0.5, 0.355) = 濃い灰緑色となり、光の三原色の混ざり方として何の問題もありません。

ラスタライザによる線形補間は、「色」のように要素ごとに独立した情報の場合は望ましい挙動ですが、「法線」のように単位ベクトルであることが重要な情報にとっては悪夢なのです。

## 法線を単位ベクトルに戻す

法線が単位ベクトルでなければ、正しいライティングはできません。そこで、フラグメント・シェーダーに法線を単位ベクトルに戻す処理を加えて、この問題を修正します。  
フラグメント・シェーダーのmain関数に、次のプログラムを追加してください。

/\*\*  
 \* フラグメントシェーダー.  
 \*/  
 void main()  
 {  
+ vec3 normal = normalize(inNormal);  
+  
 // 指向性ライトの明るさを計算する.  
 float cosTheta = clamp(dot(vNormal, -directionalLight.direction), 0.0, 1.0);  
 vec3 lightColor = directionalLight.color \* cosTheta;

normalize(のーまらいず)関数は、引数に設定されたベクトルの単位ベクトルを返す関数でしたね。この関数を使えば、あっという間に単位ベクトルの出来上がりです。簡単ですね。

**[補足]** 法線を表すnormalと正規化を行うnormalize関数は、どちらもnormalという単語が元になっています。しかし、全く違うものを表しているので、混同しないように気をつけてください。

## vNormalをnormalに書き換える

頂点シェーダーからコピーしてきたプログラムは、法線にvNormal変数を使っています。この変数はフラグメントシェーダーには存在しないので、さきほど計算したnormal変数で置き換えましょう。  
フラグメントシェーダーのmain関数を、次のように変更してください。

/\*\*  
 \* フラグメントシェーダー.  
 \*/  
 void main()  
 {  
 vec3 normal = normalize(inNormal);  
  
 // 指向性ライトの明るさを計算.  
- float cosTheta = clamp(dot(vNormal, -directionalLight.direction), 0.0, 1.0);  
+ float cosTheta = clamp(dot(normal, -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.  
 for (int i = 0; i < 8; ++i) {  
 if (dot(pointLight.color[i], pointLight.color[i]) != 0) {  
 // フラグメントからライトへ向かうベクトルを計算.  
 vec3 lightVector = pointLight.position[i] - inPosition;  
 vec3 lightDir = normalize(lightVector);  
 // 面の傾きによる明るさの変化量を計算.  
- float cosTheta = clamp(dot(vNormal, lightDir), 0.0, 1.0);  
+ float cosTheta = clamp(dot(normal, lightDir), 0.0, 1.0);  
 // 距離による明るさの変化量を計算.  
 float intensity = 1.0 / (1.0 + dot(lightVector, lightVector));  
 // 変化量をかけ合わせて明るさを求め、合計に加算.  
 lightColor += pointLight.color[i] \* cosTheta \* intensity;  
 }  
 }  
  
 fragColor = inColor \* texture(texColor, inTexCoord);  
 }

## 明るさを反映する

最後に、計算したライトの明るさを反映します。  
フラグメントシェーダーのmain関数の末尾に、次のプログラムを追加してください。

// 変化量をかけ合わせて明るさを求め、合計に加算.  
 lightColor += pointLight.color[i] \* cosTheta \* intensity;  
 }  
　 }  
  
+　　// アンビエントライトの明るさを計算.  
+ lightColor += ambientLight.color;  
+  
　 fragColor = inColor \* texture(texColor, inTexCoord);  
+ fragColor.rgb \*= lightColor;  
 }

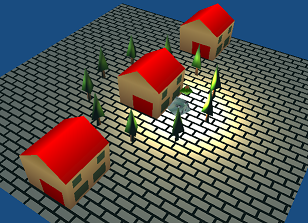
これでフラグメントシェーダーの修正は完了です。  
グーロー・シェーディングとフォン・シェーディングは、プリミティブのどの部分に注目して計算を行うかが違うだけで、明るさの計算式は同じです。だから、コピーしたプログラムを少し変更するだけで完成するんですね。

## フォン・シェーディング・シェーダーを読み込む

それでは、作成したフォン・シェーディング用のファイルを使ってみましょう。  
頂点ライティング・シェーダーの代わりに、フォン・シェーディング・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;  
 }

プログラムが書けたら、**ビルドして実行してください。**  
ポイント・ライトの光が、プリミティブの境界に影響されずに円形の範囲を照らしていたら成功です。  
WASDキーでライトを動かして、グーロー・シェーディングとの違いを確認してください。  
地面だけでなく木の照らされ方の違いにも注目してください。

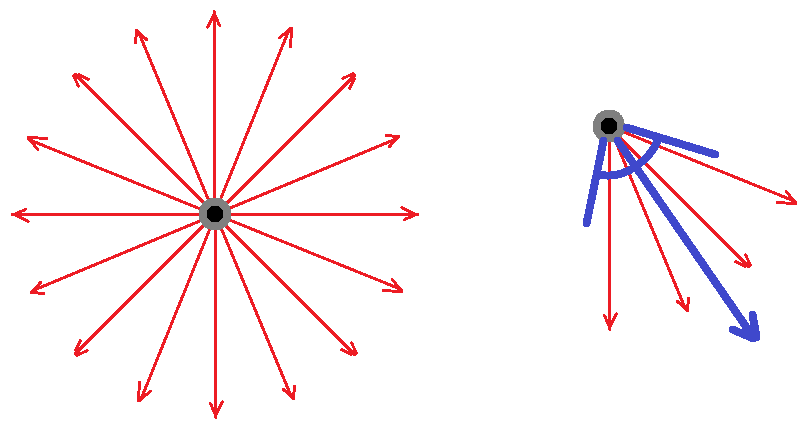


# スポット・ライト

## ポイント・ライト + 方向 + 角度 = スポット・ライト

ポイント・ライトは扱いやすい光源ですが、現実世界で見かけるライトの多くは、それほど単純ではありません。大抵は、照らす範囲を制限するための覆いや反射板が取り付けられています。この章では、そういった明かりのひとつである「スポット・ライト」を追加します。

スポット・ライトは、ポイント・ライトをベースにして、照らす範囲の制限を追加したものです。ですから、ポイント・ライトと同様に色と位置を持っています。それに加えて、光を放射する方向と、放射角を持ちます。この追加のパラメーターによって、懐中電灯のように円錐状の範囲だけを照らような表現を行うわけです。

  
[ポイント・ライト(左)とスポット・ライト(右)]

## スポット・ライト変数の追加

まずはフラグメントシェーダーにスポット・ライト用の構造体と変数を追加します。  
FragmentLighting.fragを開き、次のプログラムを追加してください。

// ポイント・ライト.  
 struct PointLight  
 {  
 vec3 position[8];  
 vec3 color[8];  
 };  
  
+// スポットライト  
+struct SpotLight  
+{  
+ vec4 dirAndCutOff[4]; // 光の方向(xyzに入れる)とcos(放射角)(wに入れる).  
+ vec3 position[4];  
+ vec3 color[4];  
+};  
+  
 uniform AmbientLight ambientLight;  
 uniform DirectionalLight directionalLight;  
 uniform PointLight pointLight;  
+uniform SpotLight spotLight;  
 uniform mat4x4 matMVP;

方向と放射角は、dirAndCutOff(でぃあ・あんど・かっと・おふ)という名前でひとつのvec4変数にまとめました。変数名はdirectionの略称dirと、放射角を表すcut off(カット・オフ)という名詞から名付けました。このようにデータをまとめておくと、位置を取得する手間が少なくなりますし、転送にかかる時間も短くなります。

なお、スポット・ライトはポイント・ライトより計算に手間がかかるので、最大4個にしてみました。

## 明るさを計算するプログラムの追加

データを定義したので、スポット・ライトの明るさを計算を追加しましょう。  
FragmentLighting.fragのmain関数に、次のプログラムを追加してください。

// 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) {  
 // フラグメントからライトへ向かうベクトルを計算.  
 vec3 lightVector = pointLight.position[i] - inPosition;  
 vec3 lightDir = normalize(lightVector);  
 // 面の傾きによる明るさの変化量を計算.  
 float cosTheta = clamp(dot(normal, lightDir), 0.0, 1.0);  
 // 距離による明るさの変化量を計算.  
 float intensity = 1.0 / (1.0 + dot(lightVector, lightVector));  
 // 変化量をかけ合わせて明るさを求め、合計に加算.  
 lightColor += pointLight.color[i] \* cosTheta \* intensity;  
 }  
 }  
  
+ // スポット・ライトの明るさを計算.  
+ for (int i = 0; i < 4; ++i) {  
+ if (dot(spotLight.color[i], spotLight.color[i]) != 0) {  
+ // フラグメントからライトへ向かうベクトルを計算.  
+ vec3 lightVector = spotLight.position[i] - inPosition;  
+ vec3 lightDir = normalize(lightVector);  
+ // 面の傾きによる明るさの変化量を計算.  
+ float cosTheta = clamp(dot(normal, lightDir), 0.0, 1.0);  
+ // 距離による明るさの変化量を計算.  
+ float intensity = 1.0 / (1.0 + dot(lightVector, lightVector));  
+ // 放射角による明るさの変化量を計算.  
+ float spotCosTheta = dot(lightDir, -spotLight.dirAndCutOff[i].xyz);  
+ float cutOff = step(spotLight.dirAndCutOff[i].w, spotCosTheta);  
+ // 変化量をかけ合わせて明るさを求め、合計に加算.  
+ lightColor += spotLight.color[i] \* cosTheta \* intensity \* cutOff;  
+ }  
+ }  
+  
 　　// アンビエントライトの明るさを計算.  
 lightColor += ambientLight.color;  
  
　 fragColor = inColor \* texture(texColor, inTexCoord);  
 fragColor.rgb \*= lightColor;  
}

スポット・ライトも、ポイント・ライトと同様に、光が浅い角度で当たるほど、またライトが遠くにあるほど明るさが減少します。これはポイント・ライトと全く同じ式で計算できます。それに加えて、放射角をこえると急速に明るさが減少していきます。これを行っているのがspotCosTheta(すぽっと・こす・しーた)とcutOff(かっと・おふ)の計算です。  
spotCosThetaは、「ライトの位置とフラグメントを結ぶベクトルと、ライトの方向がなす角θ」のコサインです(2つの単位ベクトルのなす角のコサインを求めるには内積を使うのでしたね)。ライトの方向はdirAndCutOffのxyz要素に格納されているので、この部分だけを取り出しています。この値が放射角のコサイン(dirAndCutOffのw要素に格納されています)より小さくなったら、光が当たらなくなります。なお、ここで使っている「変数.xyz」のような書き方はGLSLのベクトル型が持つ機能です。この機能を使うと、ベクトル型に含まれる要素を最大4個まで好きな順序で並び替えてfloat, vec2, vec3, vec4の値を作り出すことができます。「変数.zz」や「変数.zyx」のようなことができるわけですね。上のプログラムでは、この機能を使ってxyz要素をvec3型として取り出しています。  
spotCosThetaと放射角を比較して、光が当たる範囲かどうかを計算するために、今回は「step(すてっぷ)」というGLSLの関数を使っています。step関数は2つの引数edge(えっじ)とxを受け取ります。そして、「edge > x」が成立するなら0、不成立なら1を返します。上記のプログラムでは「edge=放射角」、「x=θ」ですから、「放射角>θ」であればスポット範囲外なので0、そうでなければ範囲内なので1となります。

これで放射角による明るさの変化量を手に入れたので、全ての数値をかけ合わせてスポット・ライトの明るさを求め、lightColorに加算します。

## スポット・ライト変数の位置を取得する

次に、スポット・ライト変数の位置を取得します。  
Main.cppを開き、ポイント・ライトの位置を取得してい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。

if (texColorLoc >= 0) {  
 glUniform1i(texColorLoc, 0);  
 }  
  
 const GLint locMatMVP = glGetUniformLocation(progLighting, "matMVP");  
 const GLint locAmbLightCol =  
 glGetUniformLocation(progLighting, "ambientLight.color");  
 const GLint locDirLightDir =  
 glGetUniformLocation(progLighting, "directionalLight.direction");  
 const GLint locDirLightCol =  
 glGetUniformLocation(progLighting, "directionalLight.color");  
 const GLint locPointLightPos =  
 glGetUniformLocation(progLighting, "pointLight.position");  
 const GLint locPointLightCol =  
 glGetUniformLocation(progLighting, "pointLight.color");  
+const GLint locSpotLightDir =  
+ glGetUniformLocation(progLighting, "spotLight.dirAndCutOff");  
+const GLint locSpotLightPos =  
+ glGetUniformLocation(progLighting, "spotLight.position");  
+const GLint locSpotLightCol =  
+ glGetUniformLocation(progLighting, "spotLight.color");  
 if (locDirLightDir < 0 || locDirLightCol < 0 || locAmbLightCol < 0 ||  
- locPointLightPos < 0 || locPointLightCol < 0) {  
+ locPointLightPos < 0 || locPointLightCol < 0 ||  
+ locSpotLightDir < 0 || locSpotLightPos < 0 || locSpotLightCol < 0) {  
 std::cerr << "ERROR: uniform変数の位置を取得できません.\n";  
 return 1;  
 }

よく似た変数名が増えてきましたので、プログラムを書くときは注意してください。

## スポット・ライトのデータを作成する

次に、転送するデータを準備します。  
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;  
+glm::vec4 spotLightDirAndCutOff[4] = {};  
+glm::vec3 spotLightPos[4] = {};  
+glm::vec3 spotLightCol[4] = {};  
+const glm::vec3 tmpSpotLightDir = glm::normalize(glm::vec3(-1,-2,-2));  
+spotLightDirAndCutOff[0] = glm::vec4(tmpSpotLightDir, std::cos(glm::radians(20.0f));  
+spotLightPos[0] = glm::vec3(-6, 6, 8);  
+spotLightCol[0] = glm::vec3(0.4f, 0.8f, 1.0f) \* 200.0f;  
+  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);  
 glEnable(GL\_CULL\_FACE);

スポット・ライトは4個まで作れますが、とりあえず1個だけ用意します。  
位置(spotLightPos(すぽっと・らいと・ぽす))と色(spotLightCol(すぽっと・らいと・こる))はポイント・ライトと同じなので難しいところはないと思います。ただ、spotLightDirAndCutOff(すぽっと・らいと・でぃあ・あんど・かっと・おふ)は、方向と放射角がまとめてあるせいで理解しにくいかもしれません。xyz要素には方向を代入し、w要素にはcos(放射角)を代入しているのがポイントです。

続いて、ポイント・ライトの色をGPUメモリに転送してい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。

// ライトをGPUメモリに転送する.  
　glUniform3fv(locAmbLightCol, 1, &ambLightCol.x);  
 glUniform3fv(locDirLightDir, 1, &dirLightDir.x);  
 glUniform3fv(locDirLightCol, 1, &dirLightCol.x);  
 glUniform3fv(locPointLightCol, 8, &pointLightCol[0].x);  
+glUniform3fv(locSpotLightCol, 4, &spotLightCol[0].x);  
  
 glBindVertexArray(vao);  
  
 glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, texId);

スポット・ライトは4個なので、転送するデータの数も4になっている点に注意してください。また、指向性ライト、ポイント・ライトと同様に、方向と位置はモデル座標系に変換してから転送しなければなりません。そのため、ここでは色だけを転送しています。

## スポット・ライトの座標をモデル座標系に変換する

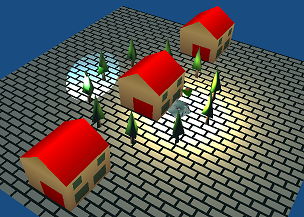
それでは、スポット・ライトの方向と位置をGPUメモリに転送していきましょう。  
木を植えるプログラムに、次のプログラムを追加してください。

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::mat3 matInvRotate = glm::inverse(glm::mat3(matModelR));  
- const glm::vec3 dirLightDirOnModel =  
- glm::inverse(glm::mat3(matModelR)) \* dirLightDir;  
+ const glm::vec3 dirLightDirOnModel = matInvRotate \* dirLightDir;  
 glUniform3fv(locDirLightDir, 1, &dirLightDirOnModel.x);  
  
 // ポイント・ライトの位置をモデル座標系に変換してGPUメモリに転送する.  
 const glm::mat4 matInvModel = glm::inverse(matModelT \* matModelR);  
 glm::vec3 pointLightPosOnModel[8];  
 for (int i = 0; i < 8; ++i) {  
 pointLightPosOnModel[i] = matInvModel \* glm::vec4(pointLightPos[i], 1);  
 }  
 glUniform3fv(locPointLightPos, 8, &pointLightPosOnModel[0].x);  
  
+ // スポット・ライトの方向と位置をモデル座標系に変換してGPUメモリに転送する.  
+ glm::vec4 spotLightDirOnModel[4];  
+ glm::vec3 spotLightPosOnModel[4];  
+ for (int i = 0; i < 4; ++i) {  
+ const glm::vec3 invDir = matInvRotate \* glm::vec3(spotLightDirAndCutOff[i]);  
+ spotLightDirOnModel[i] = glm::vec4(invDir, spotLightDirAndCutOff[i].w);  
+ spotLightPosOnModel[i] = matInvModel \* glm::vec4(spotLightPos[i], 1);  
+ }  
+ glUniform4fv(locSpotLightDir, 4, &spotLightDirOnModel[0].x);  
+ glUniform3fv(locSpotLightPos, 4, &spotLightPosOnModel[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、およびポイント・ライトの位置の計算と同じです。  
スポット・ライトは4個あるので、ポイント・ライトと同様にfor文で処理します。

**[課題04]** 地面モデルを表示するプログラムに、スポット・ライトの方向と位置をモデル座標系に変換してGPUメモリに転送するプログラムを追加してください。

これでスポット・ライトを追加することができました。  
課題04が終わったら、**ビルドして実行してください。**青白いスポット・ライトが追加されていたら成功です。



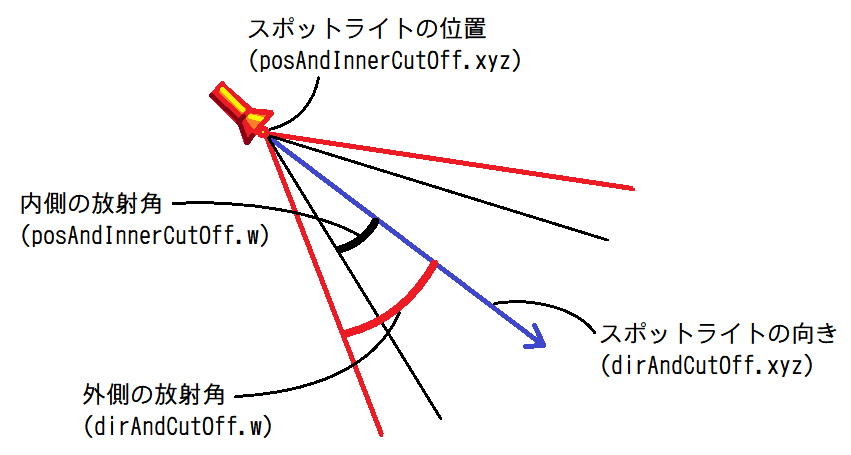
## スポット・ライトの縁を柔らかくする

スポット・ライトの当たっている部分の境界がはっきりしすぎている気がしませんか？　少しだけ縁(ふち)を柔らかくしてみましょう。

最初に、どの程度柔らかく表示するかを設定できるようにします。  
FragmentLighting.fragのスポットライト構造体を、次のように変更してください。

// ポイント・ライト.  
 struct PointLight  
 {  
 vec3 position[8];  
 vec3 color[8];  
 };  
  
 // スポットライト  
 struct SpotLight  
 {  
 vec4 dirAndCutOff[4]; // 光の方向(xyzに入れる)とcos(放射角)(wに入れる).  
- vec3 position[4];  
+ vec4 posAndInnerCutOff[4]; // 光の位置(xyzに入れる)とcos(減衰開始角)(wに入れる).  
 vec3 color[4];  
 };  
  
 uniform AmbientLight ambientLight;  
 uniform DirectionalLight directionalLight;  
 uniform PointLight pointLight;  
 uniform SpotLight spotLight;  
 uniform mat4x4 matMVP;

posAndInnerCutOff (ぽす・あんど・いんなー・かっと・おふ)変数は、スポットライトの座標(position)と柔らかくなり始める放射角(inner cut off)をまとめたものです。CutOff(かっと・おふ)とInnerCutOff(いんなー・かっと・おふ)は次の図で示すような関係になっています。

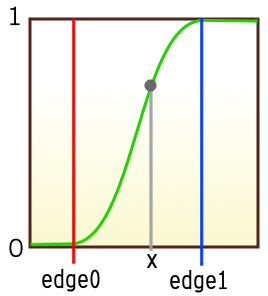


図から分かるようにInnerCutOffは常にCuttOffより小さな角度になります。

それでは、InnerCutOffをライティングに反映していきましょう。  
スポット・ライトのカット・オフを計算するプログラムを、次のように変更してください。

// スポット・ライトの明るさを計算.  
 for (int i = 0; i < 4; ++i) {  
 if (dot(spotLight.color[i], spotLight.color[i]) != 0) {  
 // フラグメントからライトへ向かうベクトルを計算.  
- vec3 lightVector = spotLight.position[i] - inPosition;  
+ vec3 lightVector = spotLight.posAndInnerCutOff[i].xyz - inPosition;  
 vec3 lightDir = normalize(lightVector);  
 // 面の傾きによる明るさの変化量を計算.  
 float cosTheta = clamp(dot(inNormal, lightDir), 0.0, 1.0);  
 // 距離による明るさの変化量を計算.  
 float intensity = 1.0 / (1.0 + dot(lightVector, lightVector));  
 // 放射角による明るさの変化量を計算.  
 float spotCosTheta = dot(lightDir, -spotLight.dirAndCutOff[i].xyz);  
- float cutOff = step(spotLight.dirAndCutOff[i].w, spotCosTheta);  
+ float cutOff = smoothstep(  
+ spotLight.dirAndCutOff[i].w, spotLight.posAndInnerCutOff[i].w, spotCosTheta);  
 // 変化量をかけ合わせて明るさを求め、合計に加算.  
 lightColor += spotLight.color[i] \* cosTheta \* intensity \* cutOff;  
 }  
 }

縁を柔らかくするには、step関数ではなくsmoothstep(すむーす・すてっぷ)関数を使います。smoothstep関数は、edge0, edge1, xという3つの引数を受け取り、xの値に応じて次の結果を返します。図の赤いラインがedge0、青いラインがedge1、緑のラインがxの取りうる値です。edge0からedge1にかけて、徐々に数値が1に近づいているのが分かりますね。



x <= edge0 : 0を返します  
 x >= edge1 : 1を返します  
edge0 < x < edge1 : xの値に応じて0～1の間の値を返します

今回のプログラムでは、edge0が外側の放射角(CutOff)、edge1が内側の放射角(InnerCutOff)にあたります。フラグメントがライトの方向からedge1度を超えた位置にある場合、その位置がedge0度に近いほど明るさが弱まり、edge0度より外側には全く光が当たりません。

ところで、uniform変数の名前を変えたので、変数の位置を取得するプログラムを修正しなければなりません。  
Main.cppを開き、スポット・ライト変数の位置を取得するプログラムを、次のように変更してください。。

　const GLint locMatMVP = glGetUniformLocation(progLighting, "matMVP");  
 const GLint locAmbLightCol =  
 glGetUniformLocation(progLighting, "ambientLight.color");  
 const GLint locDirLightDir =  
 glGetUniformLocation(progLighting, "directionalLight.direction");  
 const GLint locDirLightCol =  
 glGetUniformLocation(progLighting, "directionalLight.color");  
 const GLint locPointLightPos =  
 glGetUniformLocation(progLighting, "pointLight.position");  
 const GLint locPointLightCol =  
 glGetUniformLocation(progLighting, "pointLight.color");  
 const GLint locSpotLightDir =  
 glGetUniformLocation(progLighting, "spotLight.dirAndCutOff");  
 const GLint locSpotLightPos =  
- glGetUniformLocation(progLighting, "spotLight.position");  
+ glGetUniformLocation(progLighting, "spotLight.posAndInnerCutOff");  
 const GLint locSpotLightCol =  
 glGetUniformLocation(progLighting, "spotLight.color");  
 if (locDirLightDir < 0 || locDirLightCol < 0 || locAmbLightCol < 0 ||  
 locPointLightPos < 0 || locPointLightCol < 0 ||  
 locSpotLightDir < 0 || locSpotLightPos < 0 || locSpotLightCol < 0) {  
 std::cerr << "ERROR: uniform変数の位置を取得できません.\n";  
 return 1;  
 }

## 内側の放射角を設定する

次に、転送するデータにInnerCutOffを設定します。  
ライトを設定しているプログラムを、次のように変更してください。

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;  
 glm::vec4 spotLightDirAndCutOff[4] = {};  
-glm::vec3 spotLightPos[4] = {};  
+glm::vec4 spotLightPosAndInnerCutOff[4] = {};  
 glm::vec3 spotLightCol[4] = {};  
 const glm::vec3 tmpSpotLightDir = glm::normalize(glm::vec3(-1,-2,-2));  
 spotLightDirAndCutOff[0] = glm::vec4(tmpSpotLightDir, std::cos(glm::radians(20.0f));  
-spotLightPos[0] = glm::vec3(-6, 6, 8);  
+spotLightPosAndInnerCutOff[0] = glm::vec4(-6, 6, 8, std::cos(glm::radians(15.0f));  
 spotLightCol[0] = glm::vec3(0.4f, 0.8f, 1.0f) \* 200.0f;  
  
 // メインループ.  
 while (!window.ShouldClose()) {  
 glEnable(GL\_DEPTH\_TEST);  
 glEnable(GL\_CULL\_FACE);

内側の放射角を格納するために、spotLightPosAndInnerCutOff変数の型がvec4に変わっていることに注意しましょう。位置と内側の放射角がまとめてあるので理解しにくいかもしれませんが、xyz要素には位置を代入し、w要素には内側の放射角を代入していることを確認しながら進めてください。  
内側の放射角は15度にしてみました。外側の放射角が20度なので、柔らかい縁は5度の幅を持つことになります。

## スポット・ライトの内側の放射角をGPUメモリに転送する

それでは、スポット・ライトの方向と位置を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のスポット・ライトのデータをGPUメモリに転送するプログラムを、次のように変更してください。

// スポット・ライトの方向と位置をモデル座標系に変換してGPUメモリに転送する.  
 glm::vec4 spotLightDirOnModel[4];  
- glm::vec3 spotLightPosOnModel[4];  
+ glm::vec4 spotLightPosOnModel[4];  
 for (int i = 0; i < 4; ++i) {  
 const glm::vec3 invDir = matInvRotate \* glm::vec3(spotLightDirAndCutOff[i]);  
 spotLightDirOnModel[i] = glm::vec4(invDir, spotLightDirAndCutOff[i].w);  
+ const glm::vec3 pos = glm::vec3(spotLightPosAndInnerCutOff[i]);  
- spotLightPosOnModel[i] = matInvModel \* glm::vec4(spotLightPos[i], 1);  
+ spotLightPosOnModel[i] = matInvModel \* glm::vec4(pos, 1);  
+ spotLightPosOnModel[i].w = spotLightPosAndInnerCutOff[i].w;  
 }  
 glUniform4fv(locSpotLightDir, 4, &spotLightDirOnModel[0].x);  
- glUniform3fv(locSpotLightPos, 4, &spotLightPosOnModel[0].x);  
+ glUniform4fv(locSpotLightPos, 4, &spotLightPosOnModel[0].x);  
  
 glDrawElementsBaseVertex(meshList[0].mode, meshList[0].count,  
 GL\_UNSIGNED\_SHORT, meshList[0].indices, meshList[0].baseVertex);

GLSLではvec4型の変数vに対して「v.xyz」と書けば、xyzをvec3型の値として利用できます。しかし、GLMライブラリにこの機能はありません。代わりにglm::vec4(spotLightPosAndInnerCutOff)のように書くことでxyz要素を取り出すことができます。この方法で位置を変換したら、内側の放射角は変換の必要がないので、ただwに代入するだけです。

**[課題05]** 地面モデルを表示するプログラムにある、スポットライトの位置を変換してGPUメモリに転送する処理を、spotLightPosAndInnerCutOffを使うように修正してください。内側の放射角も転送してください。

課題05が終わったら、**ビルドして実行してください。**  
スポット・ライトの縁がぼやけていたら成功です。

