[OpenGL 3D 2018 第10回]

このシーンがすごい

# テクスチャIDを管理するクラスを作成する

## Image2Dクラスを定義する

Shader::Programクラスを作成したことで、いつglDeleteProgram関数を実行するべきかを考えなくてもいいようになりました。同様に、テクスチャを管理するクラスを作成して、glDeleteTexture関数を実行するタイミングを気にかけずに済むようにしていきます。

Texture.hを開き、次のプログラムを追加してください。

namespace Texture {  
  
 GLuint CreateImage2D(GLsizei width, GLsizei height, const GLvoid\* data,  
 GLenum format, GLenum type);  
 GLuint LoadImage2D(const char\* path);  
  
**+**/\*\*  
**+**\* 2Dテクスチャ.  
**+**\*/  
**+**class Image2D  
**+**{  
**+**public:  
**+** Image2D() = default;  
**+** explicit Image2D(GLuint texId);  
**+** ~Image2D();  
**+**  
**+** void Reset(GLuint texId);  
**+** bool IsNull() const;  
**+** GLuint Get() const;  
**+**  
**+**private:  
**+** GLuint id = 0;  
**+**};  
**+**  
} // namespace Texture

2D画像を管理するクラスなので、「Image2D(いめーじ・つー・でぃー)」という名前にしました。

## メンバ関数を定義する

それではメンバ関数を定義していきましょう。Texture.cppを開き、LoadImage2D関数の定義の下に、次のプログラムを追加してください。

// 読み込んだ画像データからテクスチャを作成する.  
 return CreateImage2D(width, height, buf.data(), format, type);  
 }  
  
**+**/\*\*  
**+**\* コンストラクタ.  
**+**\*  
**+**\* @param texId テクスチャ・オブジェクトのID.  
**+**\*/  
**+**Image2D::Image2D(GLuint texId)  
**+**{  
**+** Reset(texId);  
**+**}  
**+**  
**+**/\*\*  
**+**\* デストラクタ.  
**+**\*/  
**+**Image2D::~Image2D()  
**+**{  
**+** glDeleteTextures(1, &id);  
**+**}  
**+**  
**+**/\*\*  
**+**\* テクスチャ・オブジェクトを設定する.  
**+**\*  
**+**\* @param texId テクスチャ・オブジェクトのID.  
**+**\*/  
**+**void Image2D::Reset(GLuint texId)  
**+**{  
**+** glDeleteTextures(1, &id);  
**+** id = texId;  
**+**}  
**+**  
**+**/\*\*  
**+**\* テクスチャ・オブジェクトが設定されているか調べる.  
**+**\*  
**+**\* @retval true 設定されている.  
**+**\* @retval false 設定されていない.  
**+**\*/  
**+**bool Image2D::IsNull() const  
**+**{  
**+** return id;  
**+**}  
**+**  
**+**/\*\*  
**+**\* テクスチャ・オブジェクトを取得する.  
**+**\*  
**+**\* @return テクスチャ・オブジェクトのID.  
**+**\*/  
**+**GLuint Image2D::Get() const  
**+**{  
**+** return id;  
**+**}  
**+**  
 } // namespace Texture

Reset(りせっと)は、あとからテクスチャIDを設定したり変更したりするための関数です。

## Image2Dクラスを使う

それでは、作成したImage2Dクラスを使って、glDeleteTextures関数を削除していきましょう。

Main.cppを開き、テクスチャを作成するプログラムを次のように変更してください。

-GLuint texId = Texture::CreateImage2D(imageWidth, imageHeight, imageData,  
- GL\_RGBA, GL\_UNSIGNED\_BYTE);  
-GLuint texTree = Texture::CreateImage2D(imageWidth, imageHeight, imageTree,  
- GL\_RGBA, GL\_UNSIGNED\_BYTE);  
-GLuint texHouse = Texture::LoadImage2D("Res/House.tga");  
-GLuint texRock = Texture::LoadImage2D("Res/Rock.tga");  
-GLuint texHuman = Texture::LoadImage2D("Res/human.tga");  
**+**Texture::Image2D texId(Texture::CreateImage2D(imageWidth, imageHeight,  
**+** imageData, GL\_RGBA, GL\_UNSIGNED\_BYTE));  
**+**Texture::Image2D texTree(Texture::CreateImage2D(imageWidth, imageHeight,  
**+** imageTree, GL\_RGBA, GL\_UNSIGNED\_BYTE));  
**+**Texture::Image2D texHouse(Texture::LoadImage2D("Res/House.tga"));  
**+**Texture::Image2D texRock(Texture::LoadImage2D("Res/Rock.tga"));  
**+**Texture::Image2D texHuman(Texture::LoadImage2D("Res/human.tga"));

次に、glDeleteTextures関数呼び出しを削除します。main関数の末尾を次のように変更してください。

window.SwapBuffers();  
 }  
  
- glDeleteTextures(1, &texHuman);  
- glDeleteTextures(1, &texRock);  
- glDeleteTextures(1, &texHouse);  
- glDeleteTextures(1, &texTree);  
- glDeleteTextures(1, &texId);  
-  
 return 0;  
 }

**[課題01]** 全てのProgram::BindTexture関数について、テクスチャIDの指定を「texId.Get()」のようにImage2D::Get関数を使うように置き換えてください。

課題01を終えたらビルドして実行してください。Image2Dクラスを使う前と全く同じように表示されていたら成功です。

# LightList構造体に初期化機能を追加する

LightList構造体のメンバを都度初期化するのは面倒です。そこで、初期化を行う関数を追加しましょう。Shader.hを開き、LightList構造体に次のプログラムを追加してください。

/\*\*  
 \* ライトをまとめた構造体.  
 \*/  
 struct LightList  
 {  
 AmbientLight ambient;  
 DirectionalLight directional;  
 PointLight point;  
 SpotLight spot;  
  
**+** void Init();  
 };

次にShader.cppを開き、BuildFromFile関数定義の下に、Initメンバ関数の定義を追加してください。

const std::vector<GLchar> vsCode = ReadFile(vsPath);  
 const std::vector<GLchar> fsCode = ReadFile(fsPath);  
 return Build(vsCode.data(), fsCode.data());  
 }  
  
**+**/\*\*  
**+**\* ライトリストを初期化する.  
**+**\*  
**+**\* 全ての光源の明るさを0にする.  
**+**\*/  
**+**void LightList::Init()  
**+**{  
**+** ambient.color = glm::vec3(0);  
**+** directional.color = glm::vec3(0);  
**+** for (int i = 0; i < 8; ++i) {  
**+** point.color[i] = glm::vec3(0);  
**+** }  
**+** for (int i = 0; i < 4; ++i) {  
**+** spot.color[i] = glm::vec3(0);  
**+** }  
**+**}  
**+** /\*\*  
 \* コンストラクタ.  
 \*  
 \* @param id プログラム・オブジェクトのID.  
 \*/

これで初期化機能が追加されました。

# Shader::Programクラスの変更

## メンバ関数を追加する

現在のShader::Programクラスを作成するには、引数付きのコンストラクタを使うしかありません。引数がなくても作成できないと、クラスのメンバ変数などで使うには不便です。そこで、いくつかのメンバ関数を追加します。Shader.hを開き、Programクラスを次のように変更してください。

/\*\*  
 \* シェーダー・プログラム.  
 \*/  
 class Program  
 {  
 public:  
**+** Program();  
 explicit Program(GLuint programId);  
 ~Program();  
  
**+** void Reset(GLuint programId);  
**+** bool IsNull() const;  
 void Use();  
 void BindVertexArray(GLuint);  
 void BindTexture(GLuint, GLuint);  
 void SetLightList(const LightList&);  
 void SetViewProjectionMatrix(const glm::mat4&);  
 void Draw(const Mesh&,  
 const glm::vec3& t, const glm::vec3& r, const glm::vec3& s);  
  
 private:  
**-** GLuint id; // プログラムID.  
**+** GLuint id = 0; // プログラムID.  
  
 // uniform変数の位置.  
- GLint locMatMV;  
- GLint locPointLightPos;  
- GLint locPointLightCol;  
- GLint locDirLightDir;  
- GLint locDirLightCol;  
- GLint locAmbLightCol;  
- GLint locSpotLightPos;  
- GLint locSpotLightDir;  
- GLint locSpotLightCol;  
**+** GLint locMatMVP = -1;  
**+** GLint locPointLightPos = -1;  
**+** GLint locPointLightCol = -1;  
**+** GLint locDirLightDir = -1;  
**+** GLint locDirLightCol = -1;  
**+** GLint locAmbLightCol = -1;  
**+** GLint locSpotLightPos = -1;  
**+** GLint locSpotLightDir = -1;  
**+** GLint locSpotLightCol = -1;  
  
- glm::mat4 matVP;  
**+** glm::mat4 matVP = glm::mat4(1);  
 LightList lights;  
 };

いくつかのメンバ関数の宣言を追加し、メンバ変数には初期値を設定するようにしました。

次に、メンバ関数を定義しましょう。Shader.cppを開き、LightList::Init関数定義の下に、次のプログラムを追加してください。

for (int i = 0; i < 4; ++i) {  
 spot.color[i] = glm::vec3(0);  
 }  
 }  
  
**+**/\*\*  
**+**\* コンストラクタ.  
**+**\*/  
**+**Program::Program()  
**+**{  
**+** lights.Init();  
**+**}  
**+**  
 /\*\*  
 \* コンストラクタ.  
 \*

コンストラクタの次は、Resetメンバ関数とIsNullメンバ関数を定義します。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()  
 {  
 glDeleteProgram(id);  
 }  
  
**+**/\*\*  
**+**\* プログラム・オブジェクトを設定する.  
**+**\*  
**+**\* @param id プログラム・オブジェクトのID.  
**+**\*/  
**+**void Program::Reset(GLuint programId)  
**+**{  
**+** glDeleteProgram(id);  
**+** id = programId;  
**+** if (id == 0) {  
**+** locMatMVP = -1;  
**+** locPointLightPos = -1;  
**+** locPointLightCol = -1;  
**+** locDirLightDir = -1;  
**+** locDirLightCol = -1;  
**+** locAmbLightCol = -1;  
**+** locSpotLightPos = -1;  
**+** locSpotLightDir = -1;  
**+** locSpotLightCol = -1;  
**+** return;  
**+** }  
**+**  
**+** locMatMVP = glGetUniformLocation(id, "matMVP");  
**+** locPointLightPos = glGetUniformLocation(id, "pointLight.position");  
**+** locPointLightCol = glGetUniformLocation(id, "pointLight.color");  
**+** locDirLightDir = glGetUniformLocation(id, "directionalLight.direction");  
**+** locDirLightCol = glGetUniformLocation(id, "directionalLight.color");  
**+** locAmbLightCol = glGetUniformLocation(id, "ambientLight.color");  
**+** locSpotLightPos = glGetUniformLocation(id, "spotLight.posAndInnerCutOff");  
**+** locSpotLightDir = glGetUniformLocation(id, "spotLight.dirAndCutOff");  
**+** locSpotLightCol = glGetUniformLocation(id, "spotLight.color");  
**+**  
**+** const GLint texColorLoc = glGetUniformLocation(id, "texColor");  
**+** if (texColorLoc >= 0) {  
**+** glUseProgram(id);  
**+** glUniform1i(texColorLoc, 0);  
**+** glUseProgram(0);  
**+** }  
**+**}  
**+**  
**+**/\*\*  
**+**\* プログラム・オブジェクトが設定されているか調べる.  
**+**\*  
**+**\* @retval true 設定されている.  
**+**\* @retval false 設定されていない.  
**+**\*/  
**+**bool Program::IsNull() const  
**+**{  
**+** return 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()

Reset関数のうち、glGetUniformLocation関数を使う部分は引数付きコンストラクタと全く同じです。コピー&貼り付けを使ってください。

Reset関数が初期化の機能を備えているので、引数月コンストラクタはReset関数を呼び出すように変更します。引数付きコンストラクタを次のように変更してください。

/\*\*  
 \* コンストラクタ.  
 \*  
 \* @param id プログラム・オブジェクトのID.  
 \*/  
 Program::Program(GLuint id)  
 {  
- locMatMVP = glGetUniformLocation(id, "matMVP");  
- locPointLightPos = glGetUniformLocation(id, "pointLight.position");  
- locPointLightCol = glGetUniformLocation(id, "pointLight.color");  
- locDirLightDir = glGetUniformLocation(id, "directionalLight.direction");  
- locDirLightCol = glGetUniformLocation(id, "directionalLight.color");  
- locAmbLightCol = glGetUniformLocation(id, "ambientLight.color");  
- locSpotLightPos = glGetUniformLocation(id, "spotLight.posAndInnerCutOff");  
- locSpotLightDir = glGetUniformLocation(id, "spotLight.dirAndCutOff");  
- locSpotLightCol = glGetUniformLocation(id, "spotLight.color");  
-  
- const GLint texColorLoc = glGetUniformLocation(id, "texColor");  
- if (texColorLoc >= 0) {  
- glUseProgram(id);  
- glUniform1i(texColorLoc, 0);  
- glUseProgram(0);  
- }  
+ lights.Init();  
+ Reset(id);  
 }

デフォルトコンストラクタの追加によって、プログラムIDが設定されない状態があり得るようになってしまいました。そのため、プログラムIDが設定されていないのに描画が実行され得ます。対策として、プログラムIDが0のときは何もしないようにします。Program::Draw関数に次のプログラムを追加してください。

void Program::Draw(const Mesh& mesh, const glm::vec3& t, const glm::vec3& r,  
 const glm::vec3& s)  
 {  
+ if (id == 0) {  
+ return;  
+ }  
+  
 // モデル行列を計算する.  
 const glm::mat4 matScale = glm::scale(glm::mat4(1), s);  
 const glm::mat4 matRotateX = glm::rotate(glm::mat4(1), r.x,  
 glm::vec3(1, 0, 0));

これでプログラムIDが設定されていなくても安心ですね。

ここでビルドを行い、エラーが出ないことを確認してください。エラーが出ていたらそれを修正し、エラーが出なくなるまでビルドしてください。実行はしなくて構いません。

# 機能を関数に分ける

## MainGameSceneクラスを定義する

これまで、表示内容に関わるプログラムのほとんどはmain関数に書いてきました。いろいろな機能を追加して規模も大きくなってきたことですし、そろそろ機能ごとに関数に分けようと思います。  
各機能は次の5つの関数に分配していきます。

Initialize 画面の表示に必要なデータを準備する。

ProcessInput 入力を受けて内部状態を設定する。

Update プログラムの状態を更新する。

Render モデルの表示などを行う。

Finalize Initialize関数で準備したデータを破棄する。

これらの関数は、2Dシューティングのテキストでも同様の名前で作成していました。ですから、見覚えがあると思います。今回は「MainGameScene」(メインゲーム・シーン)という名前のクラスを作成し、上記の関数はそのメンバ関数として作って行こうと思います。

まずはクラスを作成します。  
Srcフォルダに「MainGameScene.h」という名前のヘッダーファイルを追加してください。  
追加したファイルを開き、次のプログラムを追加してください。

+/\*\*  
+\* @file MainGameScene.h  
+\*/  
+#ifndef MAINGAMESCENE\_H\_INCLUDED  
+#define MAINGAMESCENE\_H\_INCLUDED  
+#include "GLFWEW.h"  
+#include "Texture.h"  
+#include "Shader.h"  
+#include "MeshList.h"  
+  
+/\*\*  
+\* メインゲーム画面.  
+\*/  
+class MainGameScene  
+{  
+public:  
+ MainGameScene() = default;  
+ ~MainGameScene() = default;  
+  
+ bool Initialize();  
+ void ProcessInput();  
+ void Update();  
+ void Render();  
+ void Finalize();  
+  
+private:  
+ MeshList meshList;  
+  
+ Texture::Image2D texId;  
+ Texture::Image2D texTree;  
+ Texture::Image2D texHouse;  
+ Texture::Image2D texRock;  
+ Texture::Image2D texHuman;  
+  
+ Shader::Program progSimple;  
+ Shader::Program progLighting;  
+ Shader::LightList lights;  
+  
+ float angleY = 0;  
+};  
+  
+#endif // MAINGAMESCENE\_H\_INCLUDED

Main.cppからプログラムを移してくることを見越して、変数を定義しています。  
なお、texTreeは木のテクスチャ、texRockは岩のテクスチャですが、みなさんのプログラムでは違う名前になっているかもしれません。その場合はtexTree, texRockをみなさん自身が書いた変数名に変更してください。また、これら以外のオリジナルのテクスチャを定義している場合も、それに合わせて上記のテクスチャ等の定義を追加、または変更してください。このあとに出てくるプログラムでも同様に、適宜変更や追加を行ってください。

## Initialize関数を作成する

それではメンバ関数を定義していきましょう。Srcフォルダに「MainGameScene.cpp」というファイルを追加してください。追加したファイルを開き、次のプログラムを追加してください。

+/\*\*  
+\* @file MainGameScene.cpp  
+\*/  
+#include "MainGameScene.h"  
+#include <glm/gtc/matrix\_transform.hpp>  
+  
+/\*\*  
+\* 初期化.  
+\*/  
+bool MainGameScene::Initialize()  
+{  
+ return true;  
+}  
+  
+/\*\*  
+\* 入力の反映.  
+\*/  
+void MainGameScene::ProcessInput()  
+{  
+}  
+  
+/\*\*  
+\* 状態の更新.  
+\*/  
+void MainGameScene::Update()  
+{  
+}  
+  
+/\*\*  
+\* 描画.  
+\*/  
+void MainGameScene::Render()  
+{  
+}  
+  
+/\*\*  
+\* 終了.  
+\*/  
+void MainGameScene::Finalize()  
+{  
+}

それでは、Main.cppからMainGameScene.cppへ、プログラムを移動させましょう。  
まずは初期化プログラムを移動します。  
Main.cppにおいて、初期化に相当するのは以下の5つです。

1. GLFWEWの初期化  
2. メッシュリストの作成  
3. シェーダーの作成  
4. テクスチャの作成  
5. ライトの初期化

このうち、「1. GLFWEWの初期化」は最初に一度だけ行えばいいのでmain関数に残します。残りの2,3,4,5は、シーンごとに違うデータを使う可能性があります。そこで、この4つを移動させることにしましょう。では、さっそくですがMain.cppのmain関数から次のプログラムを切り取り、

if (!window.Init(800, 600, "OpenGL Tutorial")) {  
 return 1;  
 }  
  
-MeshList meshList;  
-if (!meshList.Allocate()) {  
- return 1;  
-}  
-  
-const GLuint shaderProgram = Shader::BuildFromFile(  
- "Res/Simple.vert", "Res/Simple.frag");  
--const GLuint fragmentLightingId = Shader::BuildFromFile(  
- "Res/FragmentLighting.vert", "Res/FragmentLighting.frag");  
-if (!shaderProgram || !fragmentLightingId) {  
- return 1;  
-}  
-Shader::Program progSimple(shaderProgram);  
-Shader::Program progFragmentLighting(fragmentLightingId);  
-  
-// テクスチャを作成する.  
-  
 .  
 .  
 .  
-  
-// ライトの設定.  
-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;  
-lights.spot.posAndInnerCutOff[0] =  
- glm::vec4(-6, 6, 8, std::cos(glm::radians(15.0f)));  
-lights.spot.dirAndCutOff[0] = glm::vec4(  
- glm::normalize(glm::vec3(-1,-2,-2)), std::cos(glm::radians(20.0f)));  
-lights.spot.color[0] = glm::vec3(0.4f, 0.8f, 1.0f) \* 200.0f;  
-  
 // メインループ.  
 window.InitTimer();  
 while (!window.ShouldClose()) {

次のようにMainGameScene.cppのMainGameScene::Initialize関数に貼り付けてください。

/\*\*  
 \* 初期化.  
 \*/  
 bool MainGameScene::Initialize()  
 {  
+ if (!meshList.Allocate()) {  
+ return 1;  
+ }  
+ progSimple.Reset(Shader::BuildFromFile(  
+ "Res/Simple.vert", "Res/Simple.frag"));  
+ progLighting.Reset(Shader::BuildFromFile(  
+ "Res/FragmentLighting.vert", "Res/FragmentLighting.frag"));  
+  
+ // テクスチャを作成する.  
+  
 .  
 .  
 .  
+  
+ // ライトの設定.  
+ 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;  
+ lights.spot.posAndInnerCutOff[0] =  
+ glm::vec4(-6, 6, 8, std::cos(glm::radians(15.0f)));  
+ lights.spot.dirAndCutOff[0] = glm::vec4(  
+ glm::normalize(glm::vec3(-1, -2, -2)), std::cos(glm::radians(20.0f)));  
+ lights.spot.color[0] = glm::vec3(0.4f, 0.8f, 1.0f) \* 200.0f;  
+  
 return true;  
}  
  
/\*\*  
\* 入力の反映.  
\*/

貼り付けが済んだら細部を調整しましょう。  
main関数の場合、戻り値は0が正常終了でそれ以外はエラーでした。しかし、Initialize関数の戻り値はtrueが正常終了でfalseがエラーを表します。当然ながら、main関数からコピーしたプログラムはmain関数の流儀で戻り値を決めていますから、Initialize関数の流儀に書き換えなくてはなりません。というわけで、貼り付けたプログラムのreturn文を次のように変更してください。

MeshList meshList;  
 if (!meshList.Allocate()) {  
- return 1;  
+ return false;  
 }  
  
 const GLuint shaderProgram = Shader::BuildFromFile(  
 "Res/Simple.vert", "Res/Simple.frag");  
 const GLuint pidLighting = Shader::BuildFromFile(  
 "Res/FragmentLighting.vert", "Res/FragmentLighting.frag");  
 if (!shaderProgram || !pidLighting) {  
- return 1;  
+ return false;  
 }  
 Shader::Program progSimple(shaderProgram);  
 Shader::Program progFragmentLighting(pidLighting);

これでInitialize関数は完成です。

## ProcessInput関数を作成する

次に、入力を扱うProcessInput関数を作成します。  
現在のところ、入力に関係するのはライトの位置を移動させるプログラムだけです。  
Main.cppのメインループにある、次のプログラムを切り取り、

progSimple.Use();  
 progSimple.SetViewProjectionMatrix(matProj \* matView);  
  
 // 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)) {  
- lights.point.position[0].x -= speed;  
- } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
- lights.point.position[0].x += speed;  
- }  
- if (window.IsKeyPressed(GLFW\_KEY\_LEFT\_SHIFT)) {  
- if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
- lights.point.position[0].y += speed;  
- } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
- lights.point.position[0].y -= speed;  
- }  
- } else {  
- if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
- lights.point.position[0].z -= speed;  
- } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
- lights.point.position[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);  
 }

MainGameScene.cppのProcessInput関数に貼り付けてください

/\*\*  
 \* 入力の反映.  
 \*/  
 void MainGameScene::ProcessInput()  
 {  
+ // 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)) {  
+ lights.point.position[0].x -= speed;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
+ lights.point.position[0].x += speed;  
+ }  
+ if (window.IsKeyPressed(GLFW\_KEY\_LEFT\_SHIFT)) {  
+ if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
+ lights.point.position[0].y += speed;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
+ lights.point.position[0].y -= speed;  
+ }  
+ } else {  
+ if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
+ lights.point.position[0].z -= speed;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
+ lights.point.position[0].z += speed;  
+ }  
+ }  
 }  
  
 /\*\*  
 \* 状態の更新.  
 \*/

貼り付けただけの状態では、window変数がないためエラーになってしまいます。ですから、定義を追加しましょう。ProcessInput関数に次のプログラムを追加してください。

/\*\*  
 \* 入力の反映.  
 \*/  
 void MainGameScene::ProcessInput()  
 {  
+ GLFWEW::Window& window = GLFWEW::Window::Instance();  
+  
 // 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)) {  
 lights.point.position[0].x -= speed;  
 } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
 lights.point.position[0].x += speed;  
 }

これでProcessInput関数の作成は完了です。

## Update関数を作成する

続いて、Update関数を作成しましょう。Main.cppのメインループから、次のプログラムを切り取って、

// 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);  
- }  
  
 // 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.  
 progSimple.BindTexture(0, texId.Get());  
 for (int i = 0; i < 8; ++i) {  
 progSimple.Draw(meshList[0], lights.point.position[i],  
 glm::vec3(0, angleY, 0), glm::vec3(1.0f, -0.25f, 1.0f));  
 }

MainGameScene.cppのUpdate関数に貼り付けてください。

/\*\*  
 \* 状態の更新.  
 \*/  
 void MainGameScene::Update()  
 {  
+ // 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);  
+ }  
 }  
  
 /\*\*  
 \* 描画.  
 \*/

angleY変数はMainGameSceneクラスのメンバ変数として定義しているので、static変数は削除しましょう。Update関数を次のように変更してください。

void MainGameScene::Update()  
 {  
 // 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);  
 }  
 }

これでUpdate関数は完成です。

## Render関数を作成する

Update関数の次は、Render関数を作成します。現在、メインループに残されたプログラムはほぼすべてが描画に関わるものなので、これをまるごと切り取って貼り付けます。  
Main.cppのメインループから、次のプログラムを切り取り、

// 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);  
-  
- glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
- glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
-  
- // 視点を設定する.  
- const glm::vec3 viewPos(20, 30, 30);  
-  
 .  
 .  
 .  
-  
- // 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.  
- {  
- progSimple.BindTexture(0, texId.Get());  
- for (int i = 0; i < 8; ++i) {  
- progSimple.Draw(meshList[0], lights.point.position[i],  
- glm::vec3(0, angleY, 0), glm::vec3(1.0f, -0.25f, 1.0f));  
- }  
- }  
  
 window.SwapBuffers();  
 }

/\*\*  
 \* 描画.  
 \*/  
 void MainGameScene::Render()  
 {  
+ const float deltaTime = (float)window.DeltaTime();  
+  
+ glEnable(GL\_DEPTH\_TEST);  
+ glEnable(GL\_CULL\_FACE);  
+  
+ glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
+ glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
+  
+ // 視点を設定する.  
+ const glm::vec3 viewPos(20, 30, 30);  
+  
 .  
 .  
 .  
+  
+ // 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.  
+ {  
+ progSimple.BindTexture(0, texId.Get());  
+ for (int i = 0; i < 8; ++i) {  
+ progSimple.Draw(meshList[0], lights.point.position[i],  
+ glm::vec3(0, angleY, 0), glm::vec3(1.0f, -0.25f, 1.0f));  
+ }  
+ }  
 }  
  
 /\*\*  
 \* 終了.  
 \*/

このプログラムにもwindow変数が不足しているので補いましょう。  
Render関数に次のプログラムを追加してください。

/\*\*  
 \* 描画.  
 \*/  
 void MainGameScene::Render()  
 {  
+ GLFWEW::Window& window = GLFWEW::Window::Instance();  
 const float deltaTime = (float)window.DeltaTime();  
  
 glEnable(GL\_DEPTH\_TEST);  
 glEnable(GL\_CULL\_FACE);

これでRender関数も完成です。

## Finalize関数を作成する

現在のところ、シーンの終了時に特別にやるべきことは何もありません。  
ですからFinalize関数はすでに完成しています。

## MainGameSceneクラスを使う

それでは、作成したMainGameSceneクラスを使ってMain.cppを修正していきましょう。  
まずはインクルード文を書き換えます。  
Main.cppのインクルード文を次のように変更してください。

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

ほとんどのプログラムをMainGameSceneクラスに移動した結果、必要なヘッダーファイルは元からあるGLFWEW.hと、今回追加するMainGameScene.hだけになりました。

次に、MainGameSceneクラスの変数を定義し、メンバ関数呼び出しを追加します。  
main関数に次のプログラムを追加してください。

/// エントリーポイント.  
 int main()  
 {  
 GLFWEW::Window& window = GLFWEW::Window::Instance();  
 if (!window.Init(800, 600, "OpenGL Tutorial")) {  
 return 1;  
 }  
  
+ MainGameScene mainGameScene;  
+ if (!mainGameScene.Initialize()) {  
+ return 1;  
+ }  
+  
 // メインループ.  
 window.InitTimer();  
 while (!window.ShouldClose()) {  
 window.UpdateTimer();  
+  
+ mainGameScene.ProcessInput();  
+ mainGameScene.Update();  
+ mainGameScene.Render();  
+  
 window.SwapBuffers();  
 }  
+ mainGameScene.Finalize();  
  
 return 0;  
 }

追加したプログラムでは、まずMainGameSceneクラスの変数mainGameSceneを定義しています。  
そしてシーンのInitialize関数を実行し、それが失敗したら(戻り値がfalseなら)1を返して終了します。これがシーンの初期化処理になります。次に、メインループ中では、ProcessInput関数、Update関数、Render関数を順番に実行します。これがシーンを実行する部分です。最後に、メインループを抜けたらFinalize関数を実行し、シーンを終了させています。

プログラムが書けたら、**ビルドして実行してください**。MainGameSceneクラスを作る前と同じように表示されていたら成功です。