[OpenGL 3D 2018 第10回]

シーンを継ぐもの

# 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.  
 \*/

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(いにしゃらいず) | 画面の表示に必要なデータを準備する。 |
| ProcessInput(ぷろせす・いんぷっと) | 入力を受けて内部状態を設定する。 |
| Update(あっぷでーと) | プログラムの状態を更新する。 |
| Render(れんだー) | モデルの表示などを行う。 |
| Finalize(ふぁいならいず) | 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」(めいん・げーむ・しーん)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()  
 {  
+ 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;  
+  
 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);

コピーしたプログラムのローカル変数は、メンバ変数として定義済みなので、ローカル変数を使わないように変更します。まずmeshListローカル変数を削除します。

bool MainGameScene::Initialize()  
 {  
- MeshList meshList;  
 if (!meshList.Allocate()) {  
 return 1;  
 }

次に、shaderProgram, pidLighting, progSimple, progFragmentローカル変数を削除し、メンバ変数で置き換えます。

if (!meshList.Allocate()) {  
 return 1;  
 }  
  
- const GLuint shaderProgram = Shader::BuildFromFile(  
- "Res/Simple.vert", "Res/Simple.frag");  
+ progSimple.Reset(Shader::BuildFromFile(  
+ "Res/Simple.vert", "Res/Simple.frag"));  
- const GLuint pidLighting = Shader::BuildFromFile(  
- "Res/FragmentLighting.vert", "Res/FragmentLighting.frag");  
+ progLighting.Reset(Shader::BuildFromFile(  
+ "Res/FragmentLighting.vert", "Res/FragmentLighting.frag"));  
- if (!shaderProgram || !pidLighting) {  
- return false;  
- }  
- Shader::Program progSimple(shaderProgram);  
- Shader::Program progFragmentLighting(pidLighting);  
   
 // テクスチャを作成する.

テクスチャ用のローカル変数も削除します。

-Texture::Image2D texId(Texture::CreateImage2D(  
+texId.Reset(Texture::CreateImage2D(  
 imageWidth, imageHeight, imageData, GL\_RGBA, GL\_UNSIGNED\_BYTE));  
-Texture::Image2D texTree(Texture::CreateImage2D(  
+texTree.Reset(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"));  
+texHouse.Reset(Texture::LoadImage2D("Res/House.tga"));  
+texRock.Reset(Texture::LoadImage2D("Res/Rock.tga"));  
+texHuman.Reset(Texture::LoadImage2D("Res/Human.tga"));  
  
 // ライトの設定.  
 Shader::LightList lights;

そして、ライトのローカル変数も削除します。

　　texHouse.Reset(Texture::LoadImage2D("Res/House.tga"));  
　　texRock.Reset(Texture::LoadImage2D("Res/Rock.tga"));  
　　texHuman.Reset(Texture::LoadImage2D("Res/Human.tga"));  
  
 // 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);

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.Get(0), lights.point.position[i],  
- glm::vec3(0, angleY, 0), glm::vec3(1.0f, -0.25f, 1.0f));  
- }  
- }  
  
 window.SwapBuffers();  
 }

次のように、MainGameScene.cppのRenderメンバ関数に貼り付けてください。

/\*\*  
 \* 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.Get(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。

# タイトル画面を作る

## タイトル画面で使うモデルと画像を用意する

ゲームにはタイトル画面があったほうがいいと思います。  
だから、タイトルシーンを追加しましょう。

タイトルシーンにも初期化や終了プログラムが必要ですし、入力を処理したり、表示状態を更新したり、画面を描画したりしなければなりません。つまり、基本的な構成はメインゲームシーンとほぼ同じなわけです。ですから、メインゲームシーン・クラスをコピーすることにします。クラス名は「TitleScene」(たいとる・しーん)としましょう。そのまんまですが、分かりやすい名前は重要です。

タイトル画面には、背景とロゴを表示することにします。それらを表示するために、長方形の板モデルを追加します。Resフォルダに「Plane.obj」(ぷれーん・おぶじぇ)というファイルを追加して、以下のモデルデータを書き込んでください。

#  
# Plane.obj  
#  
v -1.0 -1.0 0.0  
v 1.0 -1.0 0.0  
v 1.0 1.0 0.0  
v -1.0 1.0 0.0  
  
vt 0.0 0.0  
vt 1.0 0.0  
vt 1.0 1.0  
vt 0.0 1.0  
  
vn 0.0 0.0 1.0  
vn 0.0 0.0 1.0  
vn 0.0 0.0 1.0  
vn 0.0 0.0 1.0  
  
f 1/1/1 2/2/2 3/3/3  
f 3/3/3 4/4/4 1/1/1

作成したPlane.objをMeshList::Allocate関数で読み込みます。

AddFromObjFile("Res/Tree.obj");  
 AddFromObjFile("Res/House.obj");  
 AddFromObjFile("Res/Rock.obj");  
 AddFromObjFile("Res/Ground.obj");  
+ AddFromObjFile("Res/Plane.obj");  
  
 // VAOを作成する.  
 GLuint vbo = CreateVBO(tmpVertices.size() \* sizeof(Vertex), tmpVertices.data());  
 GLuint ibo = CreateIBO(tmpIndices.size() \* sizeof(GLushort), tmpIndices.data());

これでモデルの準備は完了です。背景とロゴの画像は以下のURLから取得できます。

**背景: https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/TitleBack.tga  
ロゴ: https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/TitleLogo.tga**

URLにアクセスしたら、右側にある「download」ボタンを押して画像をダウンロードし、プロジェクトのResフォルダにコピーしておいてください。

## TitleSceneクラスを定義する

それでは、Srcフォルダに「TitleScene.h」(たいとる・しーん・どっと・えいち)というファイルを追加してください。追加したTitleScene.hファイルを開き、次のプログラムを追加してください。

+/\*\*  
+\* @file TitleScene.h  
+\*/  
+#ifndef TITLESCENE\_H\_INCLUDED  
+#define TITLESCENE\_H\_INCLUDED  
+  
+#endif // TITLESCENE\_H\_INCLUDED

そして、MainGameScene.hのインクルード文とMainGameSceneクラスの定義をコピーして貼り付けてください。

#ifndef TITLESCENE\_H\_INCLUDED  
 #define TITLESCENE\_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 // TITLESCENE\_H\_INCLUDED

次に、「メインゲーム画面」を「タイトル画面」に、「MainGameScene」を「TitleScene」に変えます。

/\*\*  
-\* メインゲーム画面.  
+\* タイトル画面.  
 \*/  
-class MainGameScene  
+class TitleScene  
 {  
 public:  
- MainGameScene() = default;  
- ~MainGameScene() = default;  
+ TitleScene() = default;  
+ ~TitleScene() = default;

タイトル画面ではメインゲーム画面とは違うテクスチャを使うことになりますから、テクスチャ変数を変更します。それから、ライトは使わないつもりなので、progLighting, lights, angleYの3つの変数も消しましょう。

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;  
+ Texture::Image2D texLogo;  
+ Texture::Image2D texBackground;  
   
 Shader::Program progSimple;  
- Shader::Program progLighting;  
- Shader::LightList lights;  
-  
- float angleY = 0;  
 };  
  
 #endif // TITLESCENE\_H\_INCLUDED

それから、タイトル画面からメインゲーム画面に移るには、タイトル画面が終了したことを調べる機能が必要です。  
TitleSceneクラスに次のプログラムを追加してください。

void Update();  
 void Render();  
 void Finalize();  
+  
+ bool IsFinish() const;  
  
 private:  
 MeshList meshList;  
  
 Texture::Image2D texLogo;  
 Texture::Image2D texBackground;  
   
 Shader::Program progSimple;  
+  
+ float timer;  
+ bool isFinish;  
 };  
  
 #endif // TITLESCENE\_H\_INCLUDED

これでTitleSceneクラスを定義することができました。

## 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「TitleScene.cpp」(たいとるしーん・しーぴーぴー)というファイルを作成し、次のプログラムを追加してください。

+/\*\*  
+\* @file TitleScene.cpp  
+\*/  
+#include "TitleScene.h"  
+#include <glm/gtc/matrix\_transform.hpp>  
+  
+/\*\*  
+\* 初期化.  
+\*/  
+bool TitleScene::Initialize()  
+{  
+ return true;  
+}  
+  
+/\*\*  
+\* 入力の反映.  
+\*/  
+void TitleScene::ProcessInput()  
+{  
+}  
+  
+/\*\*  
+\* 状態の更新.  
+\*/  
+void TitleScene::Update()  
+{  
+}  
+  
+/\*\*  
+\* 描画.  
+\*/  
+void TitleScene::Render()  
+{  
+}  
+  
+/\*\*  
+\* 終了.  
+\*/  
+void TitleScene::Finalize()  
+{  
+}  
+  
+/\*\*  
+\* タイトル画面が終わったかどうかを調べる.  
+\*  
+\* @retval true 終わった.  
+\* @retval false 終わっていない.  
+\*/  
+void TitleScene::IsFinish() const  
+{  
+ return isFinish;  
+}

次に、Initialize関数に以下のプログラムを追加してください。

/\*\*  
 \* 初期化.  
 \*/  
 bool TitleScene::Initialize()  
 {  
+ if (!meshList.Allocate()) {  
+ return false;  
+ }  
+  
+ progSimple.Reset(Shader::BuildFromFile("Res/Simple.vert", "Res/Simple.frag"));  
+  
+ texLogo.Reset(Texture::LoadImage2D("Res/TitleLogo.tga"));  
+ texBackGround.Reset(Texture::LoadImage2D("Res/TitleBack.tga"));  
+  
+ timer = 1.0f;  
+ isFinish = false;  
+  
 return true;  
}

今回作成するのは簡素なタイトル画面なので、読み込むデータはわずかです。  
timer変数は、タイトル画面がすぐに終了しないようにするために、一定時間入力を受け付けないようにするためのタイマーです。タイトル画面が開始してから1秒間待つようにしています。

## ProcessInput関数を作成する

ProcessInput関数では、入力があったらタイトル画面を終了状態にします。  
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 TitleScene::ProcessInput()  
 {  
+ GLFWEW::Window& window = GLFWEW::Window::Instance();  
+  
+ if (!isFinish && timer <= 0.0f) {  
+ if (window.IsKeyPressed(GLFW\_KEY\_ENTER)) {  
+ isFinish = true;  
+ }  
+ }  
 }

とりあえずEnterキーで終了するようにしてみました。

## Update関数を作成する

次はクラスの状態を更新する関数を作ります。Update関数に次のプログラムを追加してください。

/\*\*  
 \* 状態の更新.  
 \*/  
 void TitleScene::Update()  
 {  
+ GLFWEW::Window& window = GLFWEW::Window::Instance();  
+  
+ if (timer > 0.0f) {  
+ timer -= window.DeltaTime();  
+ }  
 }

入力を受け付けない時間を計るタイマー変数から、経過時間だけ減らしていっています。

## Render関数を作成する

続いてクラスを描画する関数を作ります。Render関数に次のプログラムを追加してください。

/\*\*  
 \* 描画.  
 \*/  
 void TitleScene::Render()  
 {  
+ glEnable(GL\_DEPTH\_TEST);  
+ glEnable(GL\_CULL\_FACE);  
+  
+ // 半透明合成機能を有効にする.  
+ glEnable(GL\_BLEND);  
+ glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
+  
+ glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
+ glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
+  
+ meshList.BindVertexArray();  
+  
+ progSimple.Use();  
+  
+ // 正射影で描画するように行列を設定.  
+ const glm::mat4x4 matProj = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 1.0f, 500.0f);  
+ const glm::mat4x4 matView =  
+ glm::lookAt(glm::vec3(0, 0, 100), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
+ progSimple.SetViewProjectionMatrix(matProj \* matView);  
+  
+ // 背景とロゴをウィンドウの中心に描画.  
+ progSimple.BindTexture(0, texBackGround.Get());  
+ progSimple.Draw(meshList.Get(4),  
+ glm::vec3(400, 300, -1), glm::vec3(0), glm::vec3(400, 300, 1));  
+ progSimple.BindTexture(0, texLogo.Get());  
+ progSimple.Draw(meshList.Get(4),  
+ glm::vec3(400, 300, 0), glm::vec3(0), glm::vec3(400, 300, 1));  
 }

タイトル画面の描画では、いくつか新しい機能を使っています。  
まず、glEnable(じーえる・いねーぶる)関数にGL\_BLEND(じーえる・ぶれんど)を指定することで、OpenGLの**半透明機能**を有効にしています。そして、glBlendFunc(じーえる・ぶれんど・ふぁんく)関数で、**どのように合成するのか**を設定します。glBlendFunc関数の最初の引数は「フラグメントの色に掛ける値」で、2つめの引数は「描画先のピクセル色に掛ける値」です。合成方法はさまざまな設定が可能ですが、上記のプログラムにあるようにGL\_SRC\_ALPHA(じーえる・そーす・あるふぁ、フラグメントのアルファ値です)とGL\_ONE\_MINUS\_SRC\_ALPHA(じーえる・わん・まいなす・そーす・あるふぁ、1.0からフラグメントのアルファ値を引いた値です)の組み合わせを使うのが一般的です。この設定では、例えばアルファ値が0.7f(つまり不透明度が70%)のフラグメントを描画した場合、フラグメント色と描画先のピクセル色を7:3の割合で混ぜ合わせた色が描かれることになります。

タイトル画面のロゴには半透明の部分があるので、半透明機能を有効にしておく必要があるのです。

次に、行列を作成するときにglm::perspective(じーえるえむ・ぱーすぺくてぃぶ)ではなくglm::ortho(じーえるえむ・おると)を使っています。これは正射影行列を作り出す関数です。正射影行列は簡単に言うと**遠近法を無視して描画するための行列**になります。glm::ortho関数の引数は、表示する画面の大きさを指定します。ひとつめから順に、左端、右端、下端、上端、手前の端、奥の端、の位置になっています。  
つまり上記のプログラムでは、

|  |  |
| --- | --- |
| 横方向 | 0 ～ 800 |
| 縦方向 | 0 ～ 600 |
| 奥行方向 | 1 ～ 500 |

の大きさをもつ画面を作っているのです。横と縦はウィンドウの大きさと同じになっているわけですね。OpenGLの正射影行列は、ウィンドウと同じ大きさを指定するのが基本となります。

行列を設定したあとは、テクスチャを設定し、Draw関数でモデルを描画します。このとき、5.1節で作成したPlane.objを使って描画するようにしています。ここで、拡大率を(400, 300, 1)にしている理由は、正射影では、モデルの大きさはピクセル単位になるからです。Plane.objはX軸-1～+1、Y軸-1～+1、つまり2x2の大きさの正方形です。だから、横に400倍、縦に300倍して画面の中央に配置すると、ちょうど画面いっぱいに表示されるわけです。

**[注意！]** 上記のプログラムでは4番めのモデルを指定しています。これはこのテキストのプログラムではPlane.objを4番めのモデルとして読み込んでいるからです。読み込んだモデルの数が異なる場合は、この数値を適切な値に変更してください。

## Finalize関数を作成する

MainGameSceneクラスと同様に、現在のところタイトル画面の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。

## TitleSceneクラスを使う

それでは、タイトル画面を表示してみましょう。  
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 "TitleScene.h"  
 #include "MainGameScene.h"

次に、最初に作成するシーンをTitleSceneクラスに変更します。  
main関数のMainGameSceneクラスを作成するプログラムを、次のように変更してください。

/// 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()) {  
+ TitleScene\* pTitleScene = new TitleScene;  
+ MainGameScene\* pMainGameScene = nullptr;  
+ if (!pTitleScene || !pTitleScene->Initialize()) {  
+ delete pTitleScene;  
 return 1;  
 }

シーンを切り替えるたびに変数を作り直せるように、ポインタ変数を使うように変更しています。

続いて、メインループのProcessInput, Update, Renderメンバ関数の呼び出しを、次のように変更してください。

// メインループ.  
 window.InitTimer();  
 while (!window.ShouldClose()) {  
 window.UpdateTimer();  
  
- mainGameScene.ProcessInput();  
- mainGameScene.Update();  
- mainGameScene.Render();  
+ if (pTitleScene) {  
+ if (pTitleScene->IsFinish()) {  
+ glFinish();  
+ pTitleScene->Finalize();  
+ delete pTitleScene;  
+ pTitleScene = nullptr;  
+ pMainGameScene = new MainGameScene;  
+ if (!pMainGameScene || !pMainGameScene->Initialize()) {  
+ return 1;  
+ }  
+ } else {  
+ pTitleScene->ProcessInput();  
+ pTitleScene->Update();  
+ pTitleScene->Render();  
+ }  
+ } else if (pMainGameScene) {  
+ pMainGameScene->ProcessInput();  
+ pMainGameScene->Update();  
+ pMainGameScene->Render();  
+ }  
 window.SwapBuffers();  
 }  
 mainGameScene.Finalize();

変更したプログラムでは、if文を使って、タイトル画面を実行しているときと、メインゲーム画面を実行しているときで、異なるクラスのメンバ関数を呼び出すようにしています。TitleScene::IsFinish関数がtrueを返すと、タイトル画面を破棄してメインゲーム画面を作ります。このときglFinish(じーえる・ふぃにっしゅ)というOpenGLの関数を使っています。これは「これまでにGPUに送られた全てのOpenGLの処理が終了するのを待つ」関数です。GPUがまだモデルやテクスチャを使っているのにそれらを削除すると、それ以降GPUが正しく処理を行えなくなってしまいます。そこで、glFinishを実行してGPUの処理が終了するのを待っているのです。

最後に、Finalizeメンバ関数の呼び出しを、次のように変更してください。

window.SwapBuffers();  
 }  
- mainGameScene.Finalize();  
+ if (pTitleScene) {  
+ pTitleScene->Finalize();  
+ delete pTitleScene;  
+ }  
+ if (pMainGameScene) {  
+ pMainGameScene->Finalize();  
+ delete pMainGameScene;  
+ }  
  
 return 0;  
 }

どのタイミングでゲームが終了されるか分からないので、両方の変数に対して終了処理を実行するプログラムになっています。

これでタイトル画面は完成です。**ビルドして実行してください。**  
タイトル画面が表示され、Enterキーを押したときにメインゲーム画面へ切り替われば成功です。

# クラスの利点を享受する

## 基底クラスを作る

タイトル画面を追加しただけなのですが、main関数がごちゃごちゃしてしまいました。それというのも、タイトル画面もメインゲーム画面も「画面を表示する」という点では同じことをしているのに、クラスが異なるだけでメンバ関数を呼び分けているからです。C++では、いくつかのクラスの機能を「基底クラス」と「派生クラス」に分離することによって、よりスマートな書き方ができるようになります。

「基底クラス」では、すべての派生クラスで共通して使われるメンバ関数を宣言します。そして、「派生クラス」でその定義を行います。これは、派生クラスによって定義を変えられることを意味します。プログラムでは、基底クラスを使ってメンバ関数を実行します。実際にどの派生クラスを使っているのかを考える必要はありません。

まずは基底クラスを作りましょう。Srcフォルダに「Scene.h」という名前のヘッダーファイルを追加してください。Scene.hを開き、次のプログラムを追加してください。

+/\*\*  
+\* @file Scene.h  
+\*/  
+#ifndef SCENE\_H\_INCLUDED  
+#define SCENE\_H\_INCLUDED  
+#include <string>  
+  
+/\*\*  
+\* シーンの基底クラス.  
+\*/  
+class Scene  
+{  
+public:  
+ Scene() = default;  
+ virtual ~Scene() = default;  
+  
+ virtual bool Initialize() = 0;  
+ virtual void ProcessInput() = 0;  
+ virtual void Update() = 0;  
+ virtual void Render() = 0;  
+ virtual void Finalize() = 0;  
+  
+ const std::string& NextScene() const { return nextScene; }  
+ void NextScene(const char\* name) { nextScene = name; }  
+  
+private:  
+ std::string nextScene;  
+};  
+  
+#endif // SCENE\_H\_INCLUDED

C++言語では、基底クラスを作るとき、派生クラスで定義しなければならないメンバ関数には「virtual(ばーちゃる)」というキーワードを付ける決まりになっています。virtualキーワードのついた関数のことを「仮想関数(かそうかんすう)」と呼びます。さらに、仮想関数の宣言の最後に「= 0(いこーる・ぜろ)」を書くと、基底クラスでは定義を書かなくてよくなります。そのかわり、必ず派生クラスで定義を書かなくてはなりません。「= 0」を付与し、派生クラスで定義を書かなければならない仮想関数のことを「純粋仮想関数(じゅんすいかそうかんすう)」と呼びます。

NextScene関数は、TitleScene::IsFinish関数の代わりとして追加しました。IsFinish関数では「終了したかどうか」しか分からないので、例えばタイトル画面にオプションやゲーム終了を選ぶ機能を付けるようなことが難しいからです。

## TitleSceneクラスをSceneクラスの派生クラスにする

基底クラスの次は派生クラスを作ります。といっても、やることは既存のTitleSceneとMainGameSceneをSceneの派生クラスに仕立て上げるだけです。まずはTitleScene.hを開き、Scene.hをインクルードします。

#include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Shader.h"  
 #include "MeshList.h"  
+#include "Scene.h"

次に、TitleSceneクラスの定義を以下のように変更してください。

-class TitleScene  
+class TitleScene : public Scene  
 {  
 public:  
 TitleScene() = default;  
- ~TitleScene() = default;  
+ virtual ~TitleScene() = default;  
  
- bool Initialize();  
- void ProcessInput();  
- void Update();  
- void Render();  
- void Finalize();  
+ virtual bool Initialize() override;  
+ virtual void ProcessInput() override;  
+ virtual void Update() override;  
+ virtual void Render() override;  
+ virtual void Finalize() override;  
  
 bool IsFinish() const;

派生クラスを作るには、クラス定義の最初のクラス名の後ろに「: public 基底クラス名」と書くだけです。  
C++言語の仕様では、派生クラスのメンバ関数は、基底クラスの同名の関数が仮想関数なら、virtualキーワードを書かなくても自動的に仮想関数になります。しかし、どれが仮想関数なのかが分かりづらいので、一般的にはvirtualを書くことが多いです。また、末尾に追加したoverride(おーばーらいど)というキーワードは「この関数は基底クラスの仮想関数に対応します」ということを明言するためのものです。overrideを書いておくと、もし単語のつづりを間違えるなどで基底クラスと名前が一致しなければエラーになります。つまりoverrideは打ち間違えを防ぐための機能です。

すべての派生クラスの元になるクラスだから「基底クラス」、そして、「基底クラス」から派生して作られるから「派生クラス」というわけです。C++言語ではこの関係を**「派生クラスは基底クラスを継承(けいしょう)する」**といいます。

**[課題02]** MainGameSceneクラスをSceneクラスの派生クラスにしてください。

## タイトル画面終了時に次のシーンを設定する

タイトル終了時にNextScene関数を実行します。TitleScene::ProcessInput関数を次のように変更してください。

if (!isFinish && timer <= 0.0f) {  
 if (window.IsKeyPressed(GLFW\_KEY\_ENTER)) {  
 isFinish = true;  
+ NextScene("MainGameScene");  
 }  
 }

## Sceneクラスを使う

それではSceneクラスを使うようにmain関数を書き換えましょう。  
まずはmain関数のTitleSceneクラスを作成するプログラムを、次のように変更してください。

GLFWEW::Window& window = GLFWEW::Window::Instance();  
 if (!window.Init(800, 600, "OpenGL Tutorial")) {  
 return 1;  
 }  
  
- TitleScene\* pTitleScene = new TitleScene;  
- MainGameScene\* pMainGameScene = nullptr;  
- if (!pTitleScene || !pTitleScene->Initialize()) {  
+ Scene\* pScene = new TitleScene;  
+ if (!pScene || !pScene->Initialize()) {  
 delete pScene;  
 return 1;  
 }

TitleScene用とMainGameScene用に2つのポインタ変数を用意していましたが、クラスの継承を使うことで基底クラスのポインタ変数だけで十分になりました。なぜなら、基底クラスのポインタ変数には、その基底クラスを継承しているあらゆる派生クラスのポインタを代入することができるからです。

続いて、メインループを次のように変更してください。

// メインループ.  
 window.InitTimer();  
 while (!window.ShouldClose()) {  
 window.UpdateTimer();  
- if (pTitleScene) {  
- if (pTitleScene->IsFinish()) {  
- glFinish();  
- pTitleScene->Finalize();  
- delete pTitleScene;  
- pTitleScene = nullptr;  
- pMainGameScene = new MainGameScene;  
- if (!pMainGameScene || !pMainGameScene->Initialize()) {  
- return 1;  
- }  
- } else {  
- pTitleScene->ProcessInput();  
- pTitleScene->Update();  
- pTitleScene->Render();  
- }  
- } else if (pMainGameScene) {  
- pMainGameScene->ProcessInput();  
- pMainGameScene->Update();  
- pMainGameScene->Render();  
- }  
+ const std::string& nextScene = pScene->NextScene();  
+ if (!nextScene.empty()) {  
+ glFinish();  
+ if (nextScene == "MainGameScene") {  
+ pScene->Finalize();  
+ delete pScene;  
+ pScene = new MainGameScene;  
+ }  
+ if (!pScene || !pScene->Initialize()) {  
+ return 1;  
+ }  
+ }  
+ pScene->ProcessInput();  
+ pScene->Update();  
+ pScene->Render();  
 window.SwapBuffers();  
 }  
 mainGameScene.Finalize();

クラスの継承を使う前は、if文を使ってタイトル画面とメインゲーム画面のプログラムを分けていたため、やや複雑なプログラムになっていました。それが、仮想関数を使うことで簡素になっています。仮想関数は、ポインタ変数が指している実体に応じて処理が切り替わります。「条件によって分岐する」という点ではif文と同じですが、仮想関数の場合は条件が「クラスの違い」に限られます。言い換えると**「クラスによって違うことをさせたい場合は、if文ではなく仮想関数を使うとプログラムが簡単になる」**ということです。

最後に、Finalizeメンバ関数の呼び出しを、次のように変更してください。

window.SwapBuffers();  
 }  
- if (pTitleScene) {  
- pTitleScene->Finalize();  
- delete pTitleScene;  
- }  
- if (pMainGameScene) {  
- pMainGameScene->Finalize();  
- delete pMainGameScene;  
- }  
+ if (pScene) {  
+ pScene->Finalize();  
+ delete pScene;  
+ }  
  
 return 0;  
 }

ここのプログラムも継承を使うことで、pSceneがどのようなシーンを指していようとも、同じ方法で終了させられるようになっています。

これで継承を使ったプログラムは完成です。**ビルドして実行してください。**  
継承を使う以前と同じようにタイトル画面が表示され、Enterキーを押すとメインゲーム画面に切り替われば成功です。

**[課題03]** メインゲーム画面でEnterキーを押したら、タイトル画面に切り替わる機能を追加してください。