[OpenGL 3D 2018 第12回]

やがて文字になる

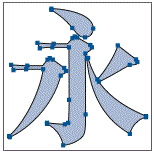
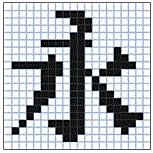
# 文字の表示方法を選ぶ

## アウトラインフォントとビットマップフォント

OpenGLはプリミティブを表示することしかできません。ですから、文字を表示する場合でもプリミティブを組み合わせて作らなければなりません。

現在、プリミティブを使って文字を表示する方法は二種類あります。  
ひとつは「アウトラインフォント」と呼ばれています。この方法では、文字を曲線の集合として定義し、ジオメトリシェーダーなどで曲線をたくさんのプリミティブの集合に変換して描画します。アウトラインフォントは、かなり大きく拡大してもキレイな文字を表示することができますが、あまり縮小すると見づらくなることが多いです。また、曲線からプリミティブへの変換をする計算を行うため、やや複雑なシェーダープログラムを必要とします。そのため、ビットマップフォントと比べるとたくさんの文字を表示するのは不得手です。

もうひとつは「ビットマップフォント」と呼ばれます。これは、文字を小さなピクセルの集合としてテクスチャにし、それをプリミティブに貼り付けて描画する方法です。テクスチャなので、あまり拡大するとギザギザが目立ってしまいますが、等倍で表示するときはアウトラインフォント以上に見やすい表示が可能です。また、大した計算をしなくてもいいので、シェーダープログラムも簡素なものになります。そのため、アウトラインフォントと比べて、たくさんの文字を表示することができます。



　　　　　　　［アウトラインフォント］　　　　　　　　　　　　　［ビットマップフォント］

## 表示方法の選択基準

どちらの方法を使うかは、コンピューターの性能やゲームの内容によります。古い世代のゲーム機には、アウトラインフォントを多用できるほどの性能がありませんでした。そのため、もっぱらビットマップフォントが使われました。  
ゲームのジャンルでいうと、ノベルゲームやアドベンチャーゲームなどのように、画面にあまり動きがなく、さまざまな大きさの文字を表示するゲームでは、アウトラインフォントのほうが見栄えがするでしょう。しかし、シューティングゲームやアクションゲームなどでは、ビットマップフォントの高速性が好まれる傾向にあります。なお、両者を組み合わせたハイブリッドな文字表示システムを作ることも可能です。ただし、メモリに両方の種類の文字データを読み込まなければならない点は注意が必要です。

さて、今回の文字表示には「ビットマップフォント」を使っていきます。  
理由は「作成するのが簡単だから」です。2D画像の表示はすでにタイトル画面でやっていますので、基本的には同じことをするだけです。

# 文字を表示する

## 得点の画像を用意する

まずはプレイヤーの得点を表示してみましょう。そのために、得点を示す「SCORE:」という画像と、０～９の数字の画像を用意してください。これらの画像は以下のURLから取得できます。

**SCORE画像: https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/Score.tga  
数字画像: https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/Number\_0.tga  
 ・  
 ・  
 ・  
 https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/Number\_9.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、みなさんのプロジェクトのResフォルダにコピーしておいてください。

次に、これらの画像を読み込むためのテクスチャ変数を追加します。  
MainGameScene .hを開き、MainGameSceneクラス定義に以下のプログラムを追加してください。

Texture::Image2D texId;  
 Texture::Image2D texTree;  
 Texture::Image2D texHouse;  
 Texture::Image2D texRock;  
 Texture::Image2D texHuman;  
 Texture::Image2D texBullet;  
 Texture::Image2D texZombie;  
  
+ // 情報表示用テクスチャ.  
+ Texture::Image2D texScore;  
+ Texture::Image2D texNumber[10];  
  
 Shader::Program progSimple;  
 Shader::Program progLighting;

続いて、これらの変数にテクスチャを読み込みましょう。  
MainGameScene::Initialize関数に、次のプログラムを追加してください。

texHouse.Reset(Texture::LoadImage2D("Res/House.tga"));  
 texRock.Reset(Texture::LoadImage2D("Res/Rock.tga"));  
 texHuman.Reset(Texture::LoadImage2D("Res/Human.tga"));  
 texBullet.Reset(Texture::LoadImage2D("Res/Bullet.tga"));  
 texZombie.Reset(Texture::LoadImage2D("Res/Zombie.tga"));  
  
**+** // 情報表示用テクスチャを読み込む.  
**+** texScore.Reset(Texture::LoadImage2D("Res/Score.tga"));  
**+** for (int i = 0; i < 10; ++i) {  
**+** const std::string filename =  
**+** std::string("Res/Number\_") + (char)('0' + i) + ".tga";  
**+** texNumber[i].Reset(Texture::LoadImage2D(filename.c\_str()));  
**+** }  
**+**  
 // 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));

数値の画像ファイル名は番号が違うだけです。そこで、for文を使って番号の部分だけを変更したファイル名を作成しています。

## 得点を表示する

それでは、読み込んだ画像を使って得点を表示しましょう。MainGameScene.cppを開き、MainGameScene::Render関数に、次のプログラムを追加してください。

RenderActorList(objectList, progLighting, meshList);  
 RenderActorList(enemyList, progLighting, meshList);  
 RenderActorList(playerBulletList, progLighting, meshList);  
   
 // 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.Use();  
 progSimple.BindTexture(0, texId.Get());  
 for (int i = 0; i < 8; ++i) {  
 progSimple.Draw(meshList.Get(4), lights.point.position[i],  
 glm::vec3(0, angleY, 0), glm::vec3(1.0f, -0.25f, 1.0f));  
 }  
  
**+** // 情報を表示.  
**+** {  
**+** glDisable(GL\_CULL\_FACE);  
**+** glDisable(GL\_DEPTH\_TEST);  
**+** glEnable(GL\_BLEND);  
**+** glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
**+**  
**+** progSimple.Use();  
**+  
+** // 正射影で描画するように行列を設定.  
**+** const glm::mat4x4 matProj = glm::ortho(  
**+** -400.0f, 400.0f, -300.0f, 300.0f, 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);  
**+  
+** const GLuint planeMeshId = meshList.Get(4); // この「4」は各自変更のこと.  
**+  
+** // スコアを表示.  
**+** progSimple.BindTexture(0, texScore.Get());  
**+** progSimple.Draw(planeMeshId,  
**+** glm::vec3(-120, 270, 0), glm::vec3(0), glm::vec3(96, 32, 1));  
**+  
+** const int maxScoreDigits = 8; // 表示する桁数. **+** int tmpScore = score;  
**+** for (int i = 0; i < maxScoreDigits; ++i) {  
**+** const int posX = -32 + 32 \* (maxScoreDigits – i);  
**+** const int number = tmpScore % 10;  
**+** tmpScore /= 10;  
**+** progSimple.BindTexture(0, texNumber[number].Get());  
**+** progSimple.Draw(planeMeshId,  
**+** glm::vec3(posX, 270, 0), glm::vec3(0), glm::vec3(32, 32, 1));  
**+** }  
**+** }  
**+** glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, 0);  
 }

基本的には、第10回で作成したタイトル画面を表示するプログラムと同じですが、今回はカリングも深度テストも無効化しています。カリングを無効にすると、上下反転、左右反転した画像を表示できるようになります(今回は特に使いませんが…)。また、深度テストを無効化すると、Zの値を常に0にすることができるようになるので、描画の順番だけを考えればよくなります。タイトル画面のときは既存の設定を流用しましたが、2D画像を表示するときはこの設定のほうが便利なことが多いと思います。

その後は、半透明合成機能を有効にし、正射影行列を設定します。  
そして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、第10回の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。上記のテキストではモデルのインデックスを4にしていますが、ここはみなさんのプログラムに合わせてインデックスを修正してください。

得点の表示は「点数を10で割り、その余りをインデックスにしてテクスチャを選択する」ということを、表示する桁数の回数だけ繰り返すことで実現しています。例えば123点を10で割ると「12あまり3」ですから、最初の桁に表示するのは「３」のテクスチャになります。次に12を10で割ると「1あまり2」ですから、2桁目には「２」のテクスチャを表示します。そして1を10で割ると「0あまり1」なので、3桁目には「１」のテクスチャを表示します。残りの5桁はずっと「0あまり0」なので、全て「０」のテクスチャが表示される、という仕組みです。

プログラムが書けたら、**ビルドして実行してください。**  
画面上部に得点が表示され、ゾンビに打ち込むたびに得点が増えていけば成功です。

## プレイヤーの体力を表示する

得点の次は、プレイヤーの体力を表示しましょう。  
まずは体力を示すテクスチャが必要ですね。この画像は以下のURLから取得できます。

**HP画像: https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/Score.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、みなさんのプロジェクトのResフォルダにコピーしておいてください。

次に、画像を読み込むためのテクスチャ変数を追加します。  
MainGameScene .hを開き、MainGameSceneクラス定義に以下のプログラムを追加してください。

// 情報表示用テクスチャ.  
 Texture::Image2D texScore;  
 Texture::Image2D texNumber[10];  
**+** Texture::Image2D texHP;  
  
 Shader::Program progSimple;  
 Shader::Program progLighting;  
 Shader::LightList lights;

続いてテクスチャを読み込みましょう。MainGameScene.cppを開き、MainGameScene::Initialize関数に、次のプログラムを追加してください。

// 情報表示用テクスチャを読み込む.  
 texScore.Reset(Texture::LoadImage2D("Res/Score.tga"));  
 for (int i = 0; i < 10; ++i) {  
 const std::string filename =  
 std::string("Res/Number\_") + (char)('0' + i) + ".tga";  
 texNumber[i].Reset(Texture::LoadImage2D(filename.c\_str()));  
 }  
**+** texHP.Reset(Texture::LoadImage2D("Res/HP.tga"));  
  
 // 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));

これで体力を表示する準備ができました。  
それでは、体力を表示しましょう。MainGameScene::Render関数に、次のプログラムを追加してください。

const int maxScoreDigits = 8; // 表示する桁数. int tmpScore = score;  
 for (int i = 0; i < maxScoreDigits; ++i) {  
 const int posX = -32 + 32 \* (maxScoreDigits – i);  
 const int number = tmpScore % 10;  
 tmpScore /= 10;  
 progSimple.BindTexture(0, texNumber[number].Get());  
 progSimple.Draw(planeMeshId,  
 glm::vec3(posX, 270, 0), glm::vec3(0), glm::vec3(32, 32, 1));  
 }  
**+  
+** // プレイヤーの体力を表示.  
**+** progSimple.BindTexture(0, texHP.Get());  
**+** progSimple.Draw(planeMeshId,  
**+** glm::vec3(-336, -270, 0), glm::vec3(0), glm::vec3(48, 32, 1));  
**+  
+** const int maxHealthDigits = 2; // 表示する桁数. **+** int tmpHealth = player.health;  
**+** for (int i = 0; i < maxHealthDigits; ++i) {  
**+** const int posX = -300 + 32 \* (maxHealthDigits – i);  
**+** const int number = tmpHealth % 10;  
**+** tmpHealth /= 10;  
**+** progSimple.BindTexture(0, texNumber[number].Get());  
**+** progSimple.Draw(planeMeshId,  
**+** glm::vec3(posX, -270, 0), glm::vec3(0), glm::vec3(32, 32, 1));  
**+** }  
 }  
 glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, 0);  
 }

このプログラムは得点の表示とほとんど同じです。表示する座標が左下になり、表示する数値が2桁の体力になっているだけです。プログラムが書けたら、**ビルドして実行してください。**  
左下に体力が表示されていたら成功です。

# ステージクリア・ゲームオーバーを作成する

## 画像を用意する

現在のところ、全てのゾンビを倒してしまうと、それ以上ゾンビは増えません。また、プレイヤーの体力が0になるとプレイヤーが倒れますが、それだけです。操作ができなくなったりはしません。  
その都度実行しなおせばまた遊べるので、一応ゲームの形にはなっていると言えなくもない、かもしれませんが、ちょっと微妙ですよね。

そこで、全てのゾンビを倒したら「ステージクリア」として新たにゾンビを出現させ、また、プレイヤーの体力が0になったら「ゲームオーバー」としてタイトル画面に戻るようにします。それぞれの状態になったときは、それと分かるような画像を表示することにしましょう。

ということで、まずはステージクリアとゲームオーバーの画像を用意します。  
この画像は以下のURLから取得できます。

**ステージクリア:** [**https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/StageClear.tga**](https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/StageClear.tga) **ゲームオーバー:  
 https://github.com/tn-mai/OpenGL3D2018/blob/master/Res/GameOver.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、みなさんのプロジェクトのResフォルダにコピーしておいてください。

次に、画像を読み込むためのテクスチャ変数を追加します。  
MainGameScene .hを開き、MainGameSceneクラス定義に以下のプログラムを追加してください。

// 情報表示用テクスチャ.  
 Texture::Image2D texScore;  
 Texture::Image2D texNumber[10];  
 Texture::Image2D texHP;  
+ Texture::Image2D texStageClear;  
+ Texture::Image2D texGameOver;  
  
 Shader::Program progSimple;  
 Shader::Program progLighting;  
 Shader::LightList lights;

続いてテクスチャを読み込みましょう。MainGameScene.cppを開き、MainGameScene::Initialize関数に、次のプログラムを追加してください。

// 情報表示用テクスチャを読み込む.  
 texScore.Reset(Texture::LoadImage2D("Res/Score.tga"));  
 for (int i = 0; i < 10; ++i) {  
 const std::string filename =  
 std::string("Res/Number\_") + (char)('0' + i) + ".tga";  
 texNumber[i].Reset(Texture::LoadImage2D(filename.c\_str()));  
 }  
 texHP.Reset(Texture::LoadImage2D("Res/HP.tga"));  
**+** texStageClear.Reset(Texture::LoadImage2D("Res/StageClear.tga"));  
**+** texGameOver.Reset(Texture::LoadImage2D("Res/GameOver.tga"));  
  
 // 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));

これで画像の用意はO.K.です。

## 状態変数を追加する

ところで、どのような条件でこれらの画像を表示したらいいでしょう。  
例えばステージクリアは？  
「もちろん、全部のゾンビを倒したときさ！」  
それじゃあ、ゲームオーバーは？  
「プレイヤーの体力が0以下になったときさ。当然だよ！」  
それじゃあ、プレイヤーの体力が0以下になると同時に、全部のゾンビを倒したときは？  
「…えっ、そんなことあるの？」

プログラム次第ではあるかもしれません。考えてみてください。弾を発射すると同時にゾンビの噛まれる、などというシチュエーションは十分にありえますよね。そうなるとステージクリアしているのにゲームオーバーでもある、というやっかいな状態になってしまいます。こういう状態はとても扱いづらいので、一般的にはどちらかを優先的に選ぶようにプログラムを書きます。そして、ゲームがプレイヤーを楽しませるものである以上、大抵はプレイヤーが有利になるほうが選ばれます。この場合は「ステージクリア」です。

同時と言っても、実際のプログラムでは「まずプレイヤーの弾の衝突判定をして、次に敵の攻撃の判定をする」というように、順番に実行しています。これは、みなさんも既に理解していることでしょう。ですから、それぞれのプログラムが好き勝手に状態を変更しないように、実行される順番を適切に管理すれば「ひとたびステージクリアかゲームオーバーになったら、他の状態には変化しない」ということが実現できるはずです。

さて、現在のゲームが置かれた状態を知りたいとき、その都度「ゾンビを全部倒したかどうか」と「プレイヤーの体力が0かどうか」を調べるのは面倒です。そこで、ゲームがどの状態にあるかを示す変数を追加することにしましょう。この変数は「通常プレイ中」、「ステージクリア」、「ゲームオーバー」の3つの状態を表すことができるものとします。  
MainGameScene.hを開き、MainGameSceneクラスの定義に次のプログラムを追加してください。

std::vector<Actor\*> enemies;  
  
 std::mt19937 random;  
  
+ // ゲーム状態.  
+ enum class State {  
+ play, // 通常プレイ中.   
+ stageClear, // ステージクリア.  
+ gameOver, // ゲームオーバー.  
+ };  
+ State state = State::play;  
+  
 int stageNo = 1;  
 int score = 0;  
 float enemySpeed = 1.0f; // ゾンビの移動速度.  
 float enemyPoppingInterval = 15.0f; // ゾンビ出現間隔.  
 float enemyPoppingTimer = 0.0f; // ゾンビ出現タイマー.

ゲーム状態はState(すてーと)という名前の列挙型として定義しました。

「列挙型」は、プログラムを読みやすくするために、数値に名前をつけることができます。列挙型を定義するときは、カンマで区切って付けたい名前を並べます。このとき、先頭には0番という番号が振られ、以降は1,2,3…と順番に番号が振られます。  
自分で番号を付けたい場合は「play = 10」のようにします。なお、名前に直接番号を振ると、以降の名前にはその番号からの連番(11,12,13,…)が振られます。

この「列挙型」、どこが便利なんでしょう。int型ではだめなんでしょうか？  
それが、だめなんです。もちろん、int型に名前をつける方法でも、列挙型と同様のことはできるでしょう。ですが、結局はint型でしかないため、数値を書ける場所ならどこにでも書けてしまいます。これは、名前を間違えたり、書く場所を間違える原因になりえます。  
対して、C++の列挙型はintとは異なる型なので、その型を受け付ける場所にしか書くことができません。書ける場所を制限できるため、int型のような間違いを起こしにくいわけです。

## ゲーム状態を変更する

それでは、ゲーム状態を変化させていきましょう。まずはステージクリアからです。  
MainGameScene::Update関数に、次のプログラムを追加してください。

++enemyKilled;  
 } else {  
 score += 10;  
 }  
 });  
  
+ // ステージクリア判定.  
+ if (state == State::play && enemyKilled >= enemyTotal) {  
+ state = State::stageClear;  
+ }  
+  
 // ゾンビの攻撃.  
 for (auto& actor : enemies) {  
 if (actor->health <= 0) {  
 continue;  
 }

ゲーム状態が倒した数が敵の総数を超えていたらステージクリアです。

次はゲームオーバーです。  
同じくMainGameScene::Update関数に、次のプログラムを追加してください。

if (zombie->isAttacking) {  
 // ゾンビの正面左右45度の範囲にターゲットがいたら、ターゲットの体力を1減らす.  
 const glm::vec3 vFront =  
 glm::rotate(glm::mat4(1), zombie->rotation.y, glm::vec3(0, 1, 0)) \*  
 glm::vec4(0, 0, -1, 1);  
 const glm::vec3 vTarget = zombie->target->position - zombie->position;  
 const float angle =  
 std::acos(glm::dot(vFront,glm::normalize(vTarget)));  
 if (std::abs(angle) < glm::radians(45.0f) &&  
 glm::length(vTarget) < 1.5f) {  
 --zombie->target->health;  
 }  
 }  
 }  
+  
+ // ゲームオーバー判定.  
+ if (state == State::play && player.health <= 0) {  
+ state = State::gameOver;  
+ }  
 }  
  
/\*  
\* 描画.  
\*/

## 状態に応じて画像を表示する

状態が分かるようになったので、これを使って画像を表示しましょう。  
MainGameScene::Render関数に、次のプログラムを追加してください。

const int maxHealthDigits = 2; // 表示する桁数. int tmpHealth = player.health;  
 for (int i = 0; i < maxHealthDigits; ++i) {  
 const int posX = -300 + 32 \* (maxHealthDigits – i);  
 const int number = tmpHealth % 10;  
 tmpHealth /= 10;  
 progSimple.BindTexture(0, texNumber[number].Get());  
 progSimple.Draw(planeMeshId,  
 glm::vec3(posX, -270, 0), glm::vec3(0), glm::vec3(32, 32, 1));  
 }  
**+**  
**+** // ステージクリア・ゲームオーバー表示.  
**+** if (state == State::stageClear) {  
**+** progSimple.BindTexture(0, texStageClear.Get());  
**+** progSimple.Draw(planeMeshId,  
**+** glm::vec3(0), glm::vec3(0), glm::vec3(350, 60, 1));  
**+** } else if (state == State::gameOver) {  
**+** progSimple.BindTexture(0, texGameOver.Get());  
**+** progSimple.Draw(planeMeshId,  
**+** glm::vec3(0), glm::vec3(0), glm::vec3(300, 60, 1));  
**+** }  
 }  
 glActiveTexture(GL\_TEXTURE0);  
 glBindTexture(GL\_TEXTURE\_2D, 0);  
 }

これで、ステージクリアとゲームオーバーの画像が表示されるようになったはずです。

## 入力を制御する

ステージクリアやゲームオーバーのときは、プレイヤーが移動や攻撃をしないようにしましょう。  
MainGameScene::ProcessInput関数を、次のように変更してください。

void MainGameScene::ProcessInput()  
 {  
+ if (state == State::play) {  
 // プレイヤーを移動する.  
 player.velocity = glm::vec3(0);  
 if (window.IsKeyPressed(GLFW\_KEY\_A)) {  
 player.velocity.x = -1;  
 } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
 player.velocity.x += 1;  
 }  
 if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
 player.velocity.z = -1;  
 } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
 player.velocity.z = 1;  
 }  
 if (player.velocity.x || player.velocity.z) {  
 player.velocity = glm::normalize(player.velocity);  
   
 // ショットボタンが押されていなければ方向転換.  
 if (!window.IsKeyPressed(GLFW\_KEY\_SPACE)) {  
 player.rotation.y = std::atan2(-player.velocity.z, player.velocity.x);  
 player.rotation.y -= glm::radians(90.0f);  
 }  
 const float speed = 10.0f;  
 player.velocity \*= speed;  
 }  
  
 // ショットボタンが押されていたら弾を発射.  
 if (window.IsKeyPressed(GLFW\_KEY\_SPACE)) {  
 if (playerBulletTimer <= 0) {  
 Actor\* bullet = FindAvailableActor(playerBullets);  
 if (bullet) {  
 const int meshId = 6; // 弾のメッシュID.  
 const float speed = 40.0f; // 弾の移動速度(m/秒).  
 const glm::mat4 matRotY =  
 glm::rotate(glm::mat4(1), player.rotation.y, glm::vec3(0, 1, 0));  
 bullet->Initialize(meshId, texBullet.Get(), 1,  
 player.position, player.rotation, glm::vec3(1));  
 bullet->velocity = matRotY \* glm::vec4(0, 0, -speed, 1);  
 bullet->colLocal =  
 { glm::vec3(-0.25f, -0.25f, -0.25f), glm::vec3(1, 1, 1) };  
  
 playerBulletTimer = 1.0f / 8.0f; // 秒間8連射.  
 }  
 }  
 } else {  
 playerBulletTimer = 0.0f;  
 }  
+ }  
 }

ご覧の通り、ゲーム状態がState::playのときだけ操作可能にしてみました。

さらに、ステージクリア状態のときはEnterキーで次のステージに進むようにしましょう。  
MainGameScene::ProcessInput関数に、次のプログラムを追加してください。

playerBulletTimer = 1.0f / 8.0f; // 秒間8連射.  
 }  
 }  
 } else {  
 playerBulletTimer = 0.0f;  
 }  
**+** } else if (state == State::stageClear) {  
**+** player.velocity.x = player.velocity.z = 0;  
**+** if (window.IsKeyDown(GLFW\_KEY\_ENTER)) {  
**+** ++stageNo;  
**+** player.position = glm::vec3(8, 0, 8);  
**+** state = State::play;  
**+  
+** // より多くの敵を、より早く出現させる. **+** enemyTotal = 10 + stageNo \* 10;  
**+** enemyLeft = enemyTotal;  
**+** enemyKilled = 0;  
**+** enemyPoppingInterval = 15.0f - (float)(stageNo - 1) \* 2.0f;  
**+** if (enemyPoppingInterval < 5.0f) {  
**+** enemyPoppingInterval = 5.0f;  
**+** }  
**+** enemyPoppingTimer = 0;  
**+  
+** // 敵の移動速度を上げる. **+** enemySpeed = 1.0f + (float)(stageNo - 1) \* 0.2f;  
**+** }  
 }  
 }

漫然とステージが進むだけではつまらないので、ステージが進むごとに、より多くの敵を倒さなければならないようにしてみました。

それから、最初のステージからゾンビが多すぎるので、ちょっと減らしましょう。  
MainGameScene.hを開き、ゾンビの初期数を変更してください。

int stageNo = 1;  
 int score = 0;  
 float enemySpeed = 1.0f; // ゾンビの移動速度.  
 float enemyPoppingInterval = 15.0f; // ゾンビ出現間隔.  
 float enemyPoppingTimer = 0.0f; // ゾンビ出現タイマー.  
**-** int enemyTotal = 100; // 敵の総数.  
**-** int enemyLeft = 100; // 未登場の敵の数. 敵を出現させるたびに減少していく.  
**+** int enemyTotal = 20; // 敵の総数.  
**+** int enemyLeft = 20; // 未登場の敵の数. 敵を出現させるたびに減少していく.  
 int enemyKilled = 0; // 殺した敵の数. この数値がenemyTotalと等しくなったらステージクリア.  
 };  
  
 #endif // MAINGAMESCENE\_H\_INCLUDED

プログラムが書けたら、**ビルドして実行してください。**  
ゾンビを全滅させたらステージクリア画像が表示され、Enterキーを押すことで次のステージが始まれば成功です。

**［課題01］**画面にステージ番号を表示してください。

# より汎用的なシーン遷移(せんい)

## シーンの切替方法を変更する

タイトル画面からメインゲーム画面に移るとき、IsFinished関数でタイトル画面の状態を調べていました。これはこれでうまく機能しますが、あるシーンから別のシーンへと自由に移り変わるには、「シーンが終わったかどうか」のような単純な条件では情報不足です。  
そこで、より多くの情報を伴ったシーンの切替方法に置き換えましょう。

まず、SrcフォルダにScene.hというヘッダーファイルを追加してください。そして追加した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;  
**+  
+** // シーン切り替え用の関数.  
**+** const std::string& NextScene() const { return nextScene; }  
**+** void NextScene(const char\* name) { nextScene = name; }  
**+**  
**+**private:  
**+** std::string nextScene;  
**+**};  
**+**  
**+**#endif // SCENE\_H\_INCLUDED

作成したScene(しーん)クラスのメンバ関数を見ると、タイトル画面クラスやメインゲーム画面クラスのものと同じ名前になっています。でも、先頭に「virtual(ばーちゃる)」が付いていたり、末尾に「= 0(いこーる・ぜろ)」が付いていたりします。  
virtualは「仮想関数」を示すキーワードでしたね。そして仮想関数は「役割によって異なる動作をする関数」なのでした。末尾にある「= 0」も、仮想関数に関係する構文です。これがついている仮想関数は「純粋仮想関数(じゅんすいかそうかんすう)」と呼ばれます。その意味は

**「この仮想関数は、派生クラスで必ず定義しなければならない」**

というものです。

実は仮想関数だからといって、必ずしも派生クラスで再定義する必要はないのです。必要なものだけ再定義できる、という自由が認められているわけです。しかし純粋仮想関数は必ず再定義しなければなりません。そのかわり、といってはなんですが、基底クラスの定義を省略してもよいことになっています。派生クラスで必ず再定義されるのですから、基底クラスに定義がなくても構わないのです。

さて、純粋仮想関数にはもうひとつの効果があります。それは

**「純粋仮想関数を持つクラスは、そのクラスの変数を定義できない」**

というものです。  
以下の例を見てください。

class X {  
 virtual void Func() = 0;  
};  
  
class Y : public X {  
 virtual void Func() override {}  
};  
  
int main() {  
 X x; // エラー!  
 Y y; // OK.  
 X\* px = &y; // OK.  
}

クラスXはFuncメンバ関数が純粋仮想関数なので、変数xを定義しようとするとエラーになります。対して、クラスYは派生クラスですから、変数yを定義することができます。さらに、クラスXのポインタ変数pxは問題なく定義でき、派生クラスの変数yのアドレスを代入することができます。

こうして、基底クラスのポインタを用いて派生クラスを抽象的に扱うことから、純粋仮想関数を持つクラスのことを「**抽象クラス**(ちゅうしょう・くらす)」と呼びます。

## MainGameSceneをSceneの派生クラスにする

さて、作成したSceneクラスをどうするかというと、これを既存のシーンクラスの基底クラスにします。  
あるクラスを派生クラスにするには基底クラスの定義が必要なので、ヘッダファイルをインクルードしましょう。MainGameScene.hを開き、次のプログラムを追加してください。

#ifndef MAINGAMESCENE\_H\_INCLUDED  
 #define MAINGAMESCENE\_H\_INCLUDED  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Shader.h"  
 #include "MeshList.h"  
**+**#include "Scene.h"  
 #include "Actor.h"  
 #include <random>

次に、MainGameSceneクラスの定義を次のように変更してください。

/\*\*  
 \* メインゲーム画面.  
 \*/  
**-**class MainGameScene  
**+**class MainGameScene : public Scene  
{  
public:  
 MainGameScene() = default;  
**-** ~MainGameScene() = default;  
**+** virtual ~MainGameScene() = 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;  
  
 private:  
 MeshList meshList;  
  
 Texture::Image2D texId;  
 Texture::Image2D texTree;

元のプログラムからの変更点は、クラス名の定義のあとに基底クラスの指定を追加していること、メンバ関数にvirtualとoverrideを追加していること、の2点だけです。  
この変更によって、MainGameSceneクラスはSceneクラスの派生クラスになりました。

## TitleSceneをSceneの派生クラスにする

同様に、タイトル画面クラスもSceneクラスの派生クラスにします。  
TitleScene.hを開き、ヘッダファイルをインクルードしてください。

/\*\*  
 \* @file TitleGameScene.h  
 \*/  
 #ifndef TITLESCENE\_H\_INCLUDED  
 #define TITLESCENE\_H\_INCLUDED  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Shader.h"  
 #include "MeshList.h"  
**+**#include "Scene.h"

そして、TitleSceneクラスの定義を次のように変更してください。

/\*\*  
 \* タイトル画面.  
 \*/  
**-**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;  
private:  
 MeshList meshList;  
  
 Texture::Image2D texLogo;  
 Texture::Image2D texBackGround;

変更点はMainGameSceneクラスと同様です。  
この変更によって、TitleSceneクラスもSceneクラスの派生クラスになりました。

## シーンに対応する文字列を決める

Sceneクラスを使ってシーンを切り替える手順は以下のとおり:

1. 各シーンに対応する文字列を決める。
2. シーンを切り替えるタイミングがきたら、NextSceneメンバ関数に対応するシーンの文字列を設定する。
3. メインループでNextScene関数に設定された文字列を調べ、対応するシーンに切り替える。

まずはシーンに対応する文字列を決めましょう。しかし「シーンに対応する文字列」では長すぎます。これからは単に「シーン名」と呼ぶことにします。  
さて、シーン名はつまり「シーンを簡潔に表す名前」です。現在、クラスの名前はまさにシーンを表していますから、「シーン名＝クラス名」とするのが簡単そうです。というわけで、タイトル画面のシーン名は”TitleScene”、メインゲーム画面のシーン名は”MainGameScene”としましょう。

## NextScene関数の呼び出しを追加する

次にシーンを切り替えたいタイミングでScene::NextScene関数を実行します。  
まずはタイトル画面からメインゲーム画面への切り替えを作成しましょう。TitleScene.cppを開き、TitleScene::ProcessInput関数に、次のプログラムを追加してください。

void TitleScene::ProcessInput()  
 {  
 GLFWEW::Window& window = GLFWEW::Window::Instance();  
  
 if (!isFinish && timer <= 0.0f) {  
 if (window.IsKeyPressed(GLFW\_KEY\_ENTER)) {  
+ NextScene("MainGameScene");  
 isFinish = true;  
 }  
 }  
 }

ゲームオーバーになったときにタイトル画面に戻る処理も追加しましょう。  
MainGameScene.cppを開き、MainGameScene::ProcessInput関数に、次のプログラムを追加してください。

**+** // 敵の移動速度を上げる. **+** enemySpeed = 1.0f + (float)(stageNo - 1) \* 0.2f;  
**+** }  
**+** } else {  
**+** player.velocity.x = player.velocity.z = 0;  
**+** if (window.IsKeyDown(GLFW\_KEY\_ENTER)) {  
+ NextScene("TitleScene");  
**+** }  
 }  
 }  
  
 /\*\*  
 \* 状態更新.  
 \*/

これでゲームオーバーになったとき、Enterキーを押すとタイトル画面に戻るようにできます。

## 設定された文字列に応じてシーンを切り替える

それではシーンの切り替えを作成しましょう。  
まずはMain.cppを開き、TitleScene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;  
 }  
  
**-** TitleScene\* pTitleScene = new TitleScene;  
**-** MainGameScene\* pMainGameScene = nullptr;  
**-** if (!pTitleScene || !pTitleScene->Initialize()) {  
**-** delete pTitleScene;  
**+** Scene\* pScene = new TitleScene;  
**+** if (!pScene || !pScene->Initialize()) {  
**+** delete pScene;  
 return 1;  
 }

次に、メインループを以下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) {  
- pTitleScene->ProcessInput();  
- pTitleScene->Update();  
- pTitleScene->Render();  
- if (pTitleScene->IsFinish()) {  
- pTitleScene->Finalize();  
- delete pTitleScene;  
- pTitleScene = nullptr;  
- pMainGameScene = new MainGameScene;  
- if (pMainGameScene && !pMainGameScene->Initialize()) {  
- break;  
- }  
- }  
- } else if (pMainGameScene) {  
- pMainGameScene->ProcessInput();  
- pMainGameScene->Update();  
- pMainGameScene->Render();  
- }  
**+** pScene->ProcessInput();  
**+** pScene->Update();  
**+** pScene->Render();  
**+** // NextSceneが設定されていたら、シーンを切り替える.  
**+** if (!pScene->NextScene().empty()) {  
**+** const std::string sceneName = pScene->NextScene(); **+** pScene->Finalize();  
**+** delete pScene;  
**+** pScene = nullptr; **+** if (sceneName == "TitleScene") {  
**+** pScene = new TitleScene;  
**+** } else if (sceneName == "MainGameScene") {  
**+** pScene = new MainGameScene;  
**+** }  
**+** if (!pScene || !pScene->Initialize()) {  
**+** break;  
**+** }  
**+** }  
  
 window.SwapBuffers();  
 }  
 if (pTitleScene) {

これまではシーンごとに異なる関数を呼び出さなければならなかったため、メインループが煩雑になっていました。しかし、2つのシーンをSceneの派生クラスにしたことにより、実際に実行中のシーンのクラスに関わらず、同じSceneクラスへのポインタで扱うことができるようになっているわけです。

最後に、メインループのあとの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。

if (!pScene || !pScene->Initialize()) {  
 break;  
 }  
 }  
  
 window.SwapBuffers();  
 }  
**-** if (pTitleScene) {  
**-** pTitleScene->Finalize();  
**-** delete pTitleScene;  
**-** }  
**-** if (pMainGameScene) {  
**-** pMainGameScene->Finalize();  
**-** delete pMainGameScene;  
**-** } **+** if (pScene) {  
**+** pScene->Finalize();  
**+** delete pScene;  
**+** }  
 return 0;  
 }

こちらも、クラスごとに別々のプログラムだったものが、ひとつのプログラムだけに減らせています。  
このように、派生クラスは基底クラスのポインタ(または参照)と共に用いることで、その真価を発揮します。

プログラムが書けたら、ビルドして実行してください。  
タイトル画面でEnterキーを押してメインゲーム画面に移動できること、適当にゾンビにじゃれつかれてゲームオーバーになったとき、Enterキーを押してタイトル画面へ戻れること、の2つが確認できたら成功です。

**［課題02］**メインゲーム画面のState列挙型に「一時停止状態」を追加し、Enterキーが押されたときにゲームの進行が一時停止状態になるようにしてください。そして、一時停止状態のときにEnterキーが押されたら、ゲームが再開されるようにしてください。

**［課題03］**一時停止中は画面に「一時停止」などの文章を表示してください。  
文章の画像は適当に作成してください。