[OpenGL 2D 2018 第11回]

終わりと始まり

# 自機と敵の衝突

前回の「課題04」と「課題05」の回答編です。

まず、自機と敵の衝突を処理する関数を作成します。  
最初にプロトタイプ宣言を追加します。playerBulletAndEnemyContactHandler関数のプロトタイプ宣言の下に、次のプログラムを追加します。

using CollisionHandlerType = void(\*)(Actor\*, Actor\*);  
 void detectCollision(Actor\* first0, Actor\* last0,  
 Actor\* first1, Actor\* last1,  
 CollisionHandlerType function);  
 void playerBulletAndEnemyContactHandler(Actor\* bullet, Actor\* enemy);  
**+**void playerAndEnemyContactHandler(Actor\* player, Actor\* enemy);

次に定義を作ります。playerBulletAndEnemyContactHandler関数の定義の下に、次のプログラムを追加します(括弧の位置に注意してください)。

namespace TA = TweenAnimation;  
 blast->spr.Tweener(TA::Animate::Create(  
 TA::Rotation::Create(20 / 60.0f, glm::pi<float>() \* 0.5f)));  
 blast->health = 1;  
 }  
 }  
 }  
**+  
+**/\*\*  
**+**\* 自機と敵の衝突を処理する.  
**+**\*  
**+**\* @param player 自機のポインタ.  
**+**\* @param enemy 敵のポインタ.  
**+**\*/  
**+**void playerAndEnemyContactHandler(Actor \* player, Actor \* enemy)  
**+**{  
**+** const int　tmp = player->health;  
**+** player->health -= enemy->health;  
**+** enemy->health -= tmp;  
**+** // 敵が破壊されたなら得点を加算して爆発を表示.  
**+** if (enemy->health <= 0) {  
**+** score += 100;  
**+** Actor\* blast = findAvailableActor(  
**+** std::begin(effectList), std::end(effectList));  
**+** if (blast != nullptr) {  
**+** blast->spr = Sprite("Res/Objects.png", enemy->spr.Position());  
**+** blast->spr.Animator(FrameAnimation::Animate::Create(tlBlast));  
**+** namespace TA = TweenAnimation;  
**+** blast->spr.Tweener(TA::Animate::Create(  
**+** TA::Rotation::Create(20 / 60.0f, 1.5f)));  
**+** blast->health = 1;  
**+** }  
**+** }  
**+** // 自機が破壊されたなら爆発を表示.  
**+** if (player->health <= 0) {  
**+** Actor\* blast = findAvailableActor(  
**+** std::begin(effectList), std::end(effectList));  
**+** if (blast != nullptr) {  
**+** blast->spr = Sprite("Res/Objects.png", player->spr.Position());  
**+** blast->spr.Animator(FrameAnimation::Animate::Create(tlBlast));  
**+** namespace TA = TweenAnimation;  
**+** blast->spr.Tweener(TA::Animate::Create(  
**+** TA::Rotation::Create(20 / 60.0f, 1.5f)));  
**+** blast->spr.Scale(glm::vec2(2, 2)); // 自機の爆発は少し大きめにしてみました.  
**+** blast->health = 1;  
**+** }  
**+** }  
**+**}

最後に、detectCollision関数を使って自機と敵の衝突判定を追加します。

// 自機の弾と敵の衝突判定.  
 detectCollision(  
 std::begin(playerBulletList), std::end(playerBulletList),  
 std::begin(enemyList), std::end(enemyList),  
 playerBulletAndEnemyContactHandler);  
**+**  
**+** // 自機と敵の衝突判定.  
**+** detectCollision(  
**+** &sprPlayer, &sprPlayer + 1,  
**+** std::begin(enemyList), std::end(enemyList),  
**+** playerAndEnemyContactHandler  
**+** );  
}

以上です。

# タイトル画面

## 何度でも遊べるように

前回までのプログラムでは、自機がやられたらそれ以上の操作はできません。そのため、もう一度遊ぶためにはアプリケーションを再起動するしかありませんでした。  
これでは不便すぎます。そこで、タイトル画面を追加し、自機がやられたらタイトル画面に戻ってもう一度遊ぶことができるようにしてみましょう。

じゃんけんゲームのときは、「タイトルフラグ」を用意してタイトル画面と他の画面を分けていました。  
フラグを使った場面切り替えは簡単に使えて便利ですが、場面が増えるたびに新しいフラグを作る必要があります。また、複数のフラグのうちひとつだけがtrueになるように、全てのフラグの状態を注意深く管理しなければなりません。

フラグよりもいい方法として、各場面にIDを割り振り、現在実行中の場面のIDを格納する変数を作る、というものがあります。管理すべき変数はひとつだけなので、フラグのような注意は必要ありません。今回はこちらの方法で場面の切り替えを作っていくことにします。

## 場面を制御する変数を作る

まず、場面ごとのIDと、実行中の場面のIDを格納する変数を作ります。  
score変数の定義の下に、次のプログラムを追加してください。

float enemyGenerationTimer; // 次の敵が出現するまでの時間(単位:秒).  
 int score; // プレイヤーの得点.  
**+**  
**+**// ゲームの状態.  
**+**const int gamestateTitle = 0; // タイトル画面の場面ID.  
**+**const int gamestateMain = 1; // ゲーム画面の場面ID.  
**+**int gamestate; // 実行中の場面ID.  
  
 // 敵の出現を制御するためのデータ.  
 TiledMap enemyMap;  
 float mapCurrentPosX;  
 float mapProcessedX;

タイトル画面とゲーム画面の2つの場面についてIDを設定しています。

## タイトル画面用の構造体を作る

タイトル画面には、背景とタイトルロゴを表示したいと思います。  
これらは表示するだけなのでActor型ではなくSprite型を使うことにします。  
さて、これまでと同様にタイトル画面で使う変数を定義していってもいいのですが、だらだらと定義してしまうと、どの変数がどの画面で必要なのかが分かりにくくなってしまいます。

そこで、タイトル画面専用の構造体を作り、タイトル画面で必要な変数は全てその構造体のメンバーとして定義することにします。こうしておけば「タイトル画面用の構造体のメンバーならタイトル画面で使う変数」、「メンバーでなければ他の場面で使う変数」だと、すぐに分かりますよね。  
「場面」に対応する英語は「Scene(しーん)」ですから、構造体の名前は「TitleScene(たいとる・しーん)」としましょうか。それでは、gamestate変数の定義の下に、次のプログラムを追加してください。

// ゲームの状態.  
const int gamestateTitle = 0; // タイトル画面の場面ID.  
const int gamestateMain = 1; // ゲーム画面の場面ID.  
 int gamestate; // 実行中の場面ID.  
**+**  
**+**/\*\*  
**+**\* タイトル画面用の構造体.  
**+**\*  
**+**\* タイトル画面で使用する変数などを格納する.  
**+**\*/  
**+**struct TitleScene  
**+**{  
**+** Sprite bg;  
**+** Sprite logo;  
**+** const int modeStart = 0; // タイトル表示待ちモード.  
**+** const int modeTitle = 1; // 入力受付モード.  
**+** const int modeNextState = 2; // ゲーム開始待ちモード.  
**+** int mode; // 実行中のモード.  
**+** float timer; // モード切り替えで使用するタイマー.  
**+**};  
**+**bool initialize(TitleScene\*);  
**+**void finalize(TitleScene\*);  
**+**void processInput(GLFWEW::WindowRef, TitleScene\*);  
**+**void update(GLFWEW::WindowRef, TitleScene\*);  
**+**void render(GLFWEW::WindowRef, TitleScene\*); **+**TitleScene titleScene;  
  
 // 敵の出現を制御するためのデータ.  
 TiledMap enemyMap;  
 float mapCurrentPosX;  
 float mapProcessedX;

タイトル画面にも状態を持たせています。なぜなら、ボタンを連打されたとき、タイトルを表示する間もなくゲームが始まってしまうのは避けたいですし、ボタンが押されたらすぐにゲームを始めるのではなく、画面に変化を与えて「押された」ことに対する反応をユーザーに返したいからです。そこで、「タイトル表示待ち」、「入力受付」、「ゲーム開始待ち」の3つのモードを作ることにしました。実行中のモードはmode変数に格納します。timer変数は、モードを切り替えるために使います。

さて、ゲームの動作を制御するためにprocessInput, update, renderという3つの関数を作りましたね。これらの関数の中に、直接、タイトル画面を制御するプログラムを書いてしまうと、プログラムの流れが分かりづらくなってしまう恐れがあります。そこで、タイトル画面専用のprocessInput, update, renderという関数を作ります。そして、先に作ったほうの関数の中でgamestate変数の値を調べ、それがgamestateTitleだった場合のみ、タイトル画面用の関数を呼び出すことにします。  
これらの関数は、既存の関数と同様にWindowRef型のパラメーターを受け取りますが、それに加えて、TitleScene\*型の変数も受け取るようにしています。

これら3つの関数に加えて、タイトル画面の初期設定をするinitialize(いにしゃらいず)関数と、タイトル画面の後始末をするfinalize(ふぁいならいず)関数も作ります。これも、既存のプログラムとタイトル画面用のプログラムを分けるためです。

**[補足]**  
本当はupdate関数やrender関数の中に直接プログラムを書いてもらって、そのあとで関数として分離するようにしたかったのですが、全体に長くなりそうなので最初から関数として作ることにしました。

## initialize関数

それではinitialize関数から作っていきましょう。  
自機と敵の衝突を処理する関数の下に、次のプログラムを追加してください。

namespace TA = TweenAnimation;  
 blast->spr.Tweener(TA::Animate::Create(  
 TA::Rotation::Create(20 / 60.0f, glm::pi<float>() \* 0.5f)));  
 blast->spr.Scale(glm::vec2(2, 2)); // 自機の爆発は少し大きめにしてみました.  
 blast->health = 1;  
 }  
 }  
 }  
**+**  
**+**/\*\*  
**+**\* タイトル画面用の構造体の初期設定を行う.  
**+**\*  
**+**\* @param scene タイトル画面用構造体のポインタ.  
**+**\*  
**+**\* @retval true 初期化成功.  
**+**\* @retval false 初期化失敗.  
**+**\*/  
**+**bool initialize(TitleScene\* scene)  
**+**{  
**+** scene->bg = Sprite("Res/UnknownPlanet.png");  
**+** scene->logo = Sprite("Res/Title.png", glm::vec3(0, 100, 0));  
**+** scene->mode = scene->modeStart;  
**+** scene->timer = 0.5f; // 入力を受け付けない期間(秒).  
**+** return true;  
**+**}

この関数は、背景とタイトルロゴの画像を設定し、最初のモードをmodeStart、タイマーを0.5秒に設定しています。

## finalize関数

続いてfinalize関数を作ります。  
先ほど書いたinitialize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 initialize(TitleScene\* scene)  
{  
 scene->bg = Sprite("Res/UnknownPlanet.png");  
 scene->logo = Sprite("Res/Title.png", glm::vec3(0, 100, 0));  
 scene->mode = scene->modeStart;  
 scene->timer = 0.5f; // 入力を受け付けない期間(秒).  
 return true;  
}  
**+**  
**+**/\*\*  
**+**\* タイトル画面の終了処理を行う.  
**+**\*  
**+**\* @param scene タイトル画面用構造体のポインタ.  
**+**\*/  
**+**void finalize(TitleScene\* scene)  
**+**{  
**+** scene->bg = Sprite();  
**+** scene->logo = Sprite();  
**+**}

finalize関数では、背景とロゴのスプライトを空のスプライトに置き換えています。こうすることで、他の画面では使用しない画像やアニメーションが解放されます。結果として、それらが使っていたメモリは別の用途で使えるようになります。

## processInput関数

次はprocessInput関数です。finalize関数定義の下に、次のプログラムを追加してください。

void finalize(TitleScene\* scene)  
{  
 scene->bg = Sprite();  
 scene->logo = Sprite();  
}  
**+**  
**+**/\*\*  
**+**\* タイトル画面のプレイヤー入力を処理する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\* @param scene タイトル画面用構造体のポインタ.  
**+**\*/  
**+**void processInput(GLFWEW::WindowRef window, TitleScene\* scene)  
**+**{  
**+** window.Update();  
**+** // 入力受付モードになるまでなにもしない.  
**+** if (scene->mode != scene->modeTitle) {  
**+** return;  
**+** }  
**+** // AまたはSTARTボタンが押されたら、ゲーム開始待ちモードに移る.  
**+** const GamePad gamepad = window.GetGamePad();  
**+** if (gamepad.buttonDown & (GamePad::A | GamePad::START)) {  
**+** scene->mode = scene->modeNextState;  
**+** scene->timer = 2.0f;  
**+** }  
**+**}

この関数は、「入力受付モード」のときのみプレイヤーの入力を受け付けます。そして、ボタンが押されたら実行中のモードを「ゲーム開始待ちモード」に変更します。

## update関数

update関数を作ります。2.6で作成したprocessInput関数定義の下に、次のプログラムを追加してください。

if (gamepad.buttonDown & (GamePad::A | GamePad::START)) {  
 scene->mode = scene->modeNextState;  
 scene->timer = 2.0f;  
 }  
}  
**+**  
**+**/\*\*  
**+**\* タイトル画面を更新する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\* @param scene タイトル画面用構造体のポインタ.  
**+**\*/  
**+**void update(GLFWEW::WindowRef window, TitleScene\* scene)  
**+**{  
**+** const float deltaTime = window.DeltaTime();  
**+**  
**+** scene->bg.Update(deltaTime);  
**+** scene->logo.Update(deltaTime);  
**+  
+**　　// タイマーが0以下になるまでカウントダウン.  
**+** if (scene->timer > 0) {  
**+** scene->timer -= deltaTime;  
**+** return;  
**+** }  
**+  
+** if (scene->mode == scene->modeStart) {  
**+** scene->mode = scene->modeTitle;  
**+** } else if (scene->mode == scene->modeNextState) {  
**+** finalize(scene); // タイトル画面の後始末. **+** gamestate = gamestateMain;  
**+  
+** // ゲームの初期設定を行う.  
**+  
+** //スプライトに画像を設定.  
**+** sprBackground = Sprite("Res/UnknownPlanet.png");  
**+** sprPlayer.spr =  
**+** Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));  
**+** sprPlayer.collisionShape = Rect(-24, -8, 48, 16);  
**+** sprPlayer.health = 1;  
**+**  
**+** initializeActorList(std::begin(enemyList), std::end(enemyList));  
**+** initializeActorList(  
**+** std::begin(playerBulletList), std::end(playerBulletList));  
**+** initializeActorList(std::begin(effectList), std::end(effectList));  
**+  
+** score = 0;  
**+  
+** // 敵配置マップを読み込む.  
**+** enemyMap.Load("Res/EnemyMap.json”);  
**+** mapCurrentPosX = windowWidth;  
**+** mapProcessedX = windowWidth;  
**+** }  
**+**}

この関数のポイントは、「ゲーム開始待ちモード」の処理です。このとき、main関数でやっているのと全く同じようにゲームの初期設定をしています。このタイミングで初期設定を行うことにより、前回のゲーム状態を引き継ぐことなく最初からゲームを始めることができるのです。  
「ゲームの初期設定を行う」というコメントの下にあるプログラムは、main関数からコピペするとよいでしょう。

## render関数

最後に定義するのはrender関数です。  
2.7で作成したupdate関数の定義の下に、次のプログラムを追加してください。

score = 0;  
 enemyMap.Load("Res/EnemyMap.json”);  
 mapCurrentPosX = mapProcessedX = windowWidth;  
 }  
}  
**+**  
**+**/\*\*  
**+**\* タイトル画面を描画する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\* @param scene タイトル画面用構造体のポインタ.  
**+**\*/  
**+**void render(GLFWEW::WindowRef window, TitleScene\* scene)  
**+**{  
**+** renderer.BeginUpdate();  
**+** renderer.AddVertices(scene->bg);  
**+** renderer.AddVertices(scene->logo);  
**+** renderer.EndUpdate();  
**+** renderer.Draw(glm::vec2(windowWidth, windowHeight));  
**+**  
**+** fontRenderer.BeginUpdate();  
**+** if (scene->mode == scene->modeTitle) {  
**+** fontRenderer.AddString(glm::vec2(-80, -100), "START");  
**+** } else if (scene->mode == scene->modeNextState) {  
**+** // ゲーム開始待ちのときは文字を点滅させる.  
**+** if ((int)(scene->timer \* 10) % 2) {  
**+** fontRenderer.AddString(glm::vec2(-80, -100), "START");  
**+** }  
**+** }  
**+** fontRenderer.EndUpdate();  
**+** fontRenderer.Draw();  
**+**  
**+** window.SwapBuffers();  
**+**}

タイトル画面用の描画関数がやっていることは、既存のrender関数とさほど違いません。  
ただ、文字を点滅させるためのプログラムは分かりにくいかもしれません。  
update関数で実行モードを「ゲーム開始待ちモード」に変更する時、タイマーに2秒という数値を設定しました。これが「ゲームが開始されるまでの待ち時間」になります。  
点滅プログラムではこのタイマーを流用し、「タイマーを10倍して整数に変換した値が2で割り切れないとき」だけ文字を表示するようにしています。タイマーは2秒で始まるので10倍すると20です。20は2で割り切れるので、文字は表示されません。カウントダウンが進み、タイマーが1.94秒になったとします。10倍して整数にすると19です。19は2で割り切れませんから、文字が表示されます。さらにカウントダウンが進み、1.88秒になったとします。10倍して整数にすると18です。18は2で割り切れますから、再び文字が消えます。  
このようにして、0.1秒ごとに文字を消したり表示したりすることで点滅させているわけです。

**[補足]**  
点滅用のタイマーを用意してもよかったのですが、あまり凝った演出をするつもりもなかったので流用で済ませました。  
凝ったことをするなら文字列を画像にして、トウィーニングなどを使って演出を加えるほうがいいでしょう。

## 初期化関数を呼び出す

作成した関数を呼び出すプログラムを書いていきます。  
ゲームの初期設定を行うプログラムを、次のように変更してください。

tlEnemy = FrameAnimation::Timeline::Create(enemyKeyFrames);  
 tlPlayer = FrameAnimation::Timeline::Create(playerKeyFrames);  
 tlBlast = FrameAnimation::Timeline::Create(blastKeyFrames);  
  
**-**// スプライトに画像を設定.  
**-**sprBackground = Sprite("Res/UnknownPlanet.png");  
**-**sprPlayer.spr =  
**-** Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));  
**-**sprPlayer.collisionShape = Rect(-24, -8, 48, 16);  
**-**sprPlayer.health = 1;  
**-**  
**-**initializeActorList(std::begin(enemyList), std::end(enemyList));  
**-**initializeActorList(std::begin(playerBulletList), std::end(playerBulletList));  
**-**initializeActorList(std::begin(effectList), std::end(effectList));  
**-**  
**-**enemyGenerationTimer = 2;  
**-**score = 0;  
**-**  
**-**// 敵配置マップを読み込む.  
**-**enemyMap.Load("Res/EnemyMap.json");  
**-**mapCurrentPosX = mapProcessedX = windowWidth;  
**+**// タイトル画面を初期化する.  
**+**gamestate = gamestateTitle;  
**+**initialize(&titleScene);  
  
 // ゲームループ.  
 while (!window.ShouldClose()) {

ゲームの初期設定はタイトル画面用のupdate関数にコピペ済みなので、main関数からは削除します。

## 入力処理関数を呼び出す

続いてprocessUpdate関数を変更します。processInput関数に次のプログラムを追加してください。

/\*\*  
 \* プレイヤーの入力を処理する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void processInput(GLFWEW::WindowRef window)  
 {  
**+** if (gamestate == gamestateTitle) {  
**+** processInput(window, &titleScene);  
**+** return;  
**+** }  
  
 window.Update();

gamestate変数がgamestateTitleだった場合のみ、タイトル画面用のprocessInput関数を呼び出しています。

## 更新関数を呼び出す

次は更新関数です。update関数に次のプログラムを追加してください。

/\*\*  
 \* ゲームの状態を更新する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void update(GLFWEW::WindowRef window)  
 {  
**+** if (gamestate == gamestateTitle) {  
**+** update(window, &titleScene);  
**+** return;  
**+** }  
  
 const float deltaTime = window.DeltaTime();

## 描画関数を呼び出す

最後は描画関数の呼び出しです。render関数に次のプログラムを追加してください。

/\*\*  
 \* ゲームの状態を描画する..  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void render(GLFWEW::WindowRef window)  
 {  
**+** if (gamestate == gamestateTitle) {  
**+** render(window, &titleScene);  
**+** return;  
**+** }  
  
 renderer.BeginUpdate();  
 renderer.AddVertices(sprBackground);

## タイトル画面に戻る処理を追加する

自機がやられたら、タイトル画面に戻るようにします。  
update関数にある、タイトル画面用update関数を呼び出すプログラムの下に、次のプログラムを追加してください。

/\*\*  
 \* ゲームの状態を更新する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void update(GLFWEW::WindowRef window)  
 {  
 if (gamestate == gamestateTitle) {  
 update(window, &titleScene);  
 return;  
**+** } else if (gamestate == gamestateMain) {  
**+** // 自機が破壊されていたらタイトル画面に戻る. **+** if (sprPlayer.health <= 0) {  
**+** gamestate = gamestateTitle;  
**+** initialize(&titleScene);  
**+** return;  
**+** }  
 }  
  
 const float deltaTime = window.DeltaTime();

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たら、AキーかEnterキーを押すとゲームが開始されます。自機がやられたらタイトル画面に戻ってくること、そのままもう一度ゲームを開始するとスコアが0に戻っていて、敵の出現パターンが最初から再生されること、などを確認してください。

# ゲームオーバー画面

## ゲームオーバー画面用のIDを追加する

ところで、自機が破壊されたらいきなりタイトル画面に戻るのは、唐突すぎる気がしませんか？  
多くのゲームでは、「全ての自機が破壊された」などの理由でゲームを続けることができなくなったとき、「ゲームオーバー画面」というものを経由させることで、プレイヤーにゲームが終了したことを伝えるようにしています。先達の例にならって、わたしたちもゲームオーバー画面を付け加えてみましょう。

ゲームオーバー画面の作り方は、おおむねタイトル画面と同じです。  
まずゲームオーバー画面用の場面IDを追加します。

// ゲームの状態.  
const int gamestateTitle = 0; // タイトル画面の場面ID.  
const int gamestateMain = 1; // ゲーム画面の場面ID.  
**+**const int gamestateGameOver = 2; // ゲームオーバー画面の場面ID.  
int gamestate; // 実行中の場面ID.

## ゲームオーバー画面用の構造体を作る

次に、ゲームオーバー画面用の構造体と、関数のプロトタイプ宣言を追加します。  
タイトル画面用のプロトタイプ宣言の下に、次のプログラムを追加してください。

void processInput(GLFWEW::WindowRef, TitleScene\*);  
 void update(GLFWEW::WindowRef, TitleScene\*);  
 void render(GLFWEW::WindowRef, TitleScene\*);  
TitleScene titleScene;  
**+**  
**+**/\*\*  
**+**\* ゲームオーバー画面で使用する構造体.  
**+**\*/  
**+**struct GameOverScene  
**+**{  
**+** Sprite bg;  
**+** float timer;  
**+**};  
**+**bool initialize(GameOverScene\*);  
**+**void finalize(GameOverScene\*);  
**+**void processInput(GLFWEW::WindowRef, GameOverScene\*);  
**+**void update(GLFWEW::WindowRef, GameOverScene\*);  
**+**void render(GLFWEW::WindowRef, GameOverScene\*);  
**+**GameOverScene gameOverScene;  
  
 // 敵の出現を制御するためのデータ.  
 TiledMap enemyMap;

ゲームオーバー画面はタイトルよりシンプルで、背景の上に文字を表示するだけにする予定です。モード分けもなく、ボタンが押されたらすぐにタイトル画面に戻るようにします。  
そういうわけで、構造体のメンバーも2つだけです。

## initialize関数

続いて関数を作っていきます。  
タイトル画面用のrender関数の定義の下に、次のプログラムを追加してください。

} else if (scene->mode == scene->modeNextState) {  
 if ((int)(scene->timer \* 10) % 2) {  
 fontRenderer.AddString(glm::vec2(-80, -100), "START");  
 }  
 }  
 fontRenderer.EndUpdate();  
 fontRenderer.Draw();  
  
 window.SwapBuffers();  
 }  
**+**  
**+**/\*\*  
**+**\* ゲームオーバー画面の初期設定を行う.  
**+**\*  
**+**\* @param scene ゲームオーバー画面用構造体のポインタ.  
**+**\*  
**+**\* @retval true 初期化成功.  
**+**\* @retval false 初期化失敗.  
**+**\*/  
**+**bool initialize(GameOverScene\* scene)  
**+**{  
**+** scene->bg = Sprite("Res/UnknownPlanet.png");  
**+** scene->timer = 0.5f; // 入力を受け付けない期間(秒).  
**+** return true;  
**+**}

背景画像を設定し、入力を開始するまでの時間を設定しているだけです。

## finalize関数

次は後始末をする関数を作ります。  
先ほど書いたinitialize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 initialize(GameOverScene\* scene)  
{  
 scene->bg = Sprite("Res/UnknownPlanet.png");  
 scene->timer = 0.5f; // 入力を受け付けない期間(秒).  
 return true;  
}  
**+**  
**+**/\*\*  
**+**\* ゲームオーバー画面の終了処理を行う.  
**+**\*  
**+**\* @param scene ゲームオーバー画面用構造体のポインタ.  
**+**\*/  
**+**void finalize(GameOverScene\* scene)  
**+**{  
**+** scene->bg = Sprite();  
**+**}

後始末は、背景用スプライトを空のスプライトで置き換えるだけです。  
タイトル画面用のfinalize関数で説明したように、このプログラムは不要になったメモリを解放します。

## processInput関数

次は入力を処理する関数を作ります。  
先ほど書いたfinalize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 finalize(GameOverScene\* scene)  
{  
 scene->bg = Sprite();  
}  
**+**  
**+**/\*\*  
**+**\* ゲームオーバー画面のプレイヤー入力を処理する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\* @param scene ゲームオーバー画面用構造体のポインタ.  
**+**\*/  
**+**void processInput(GLFWEW::WindowRef window, GameOverScene\* scene)  
**+**{  
**+** window.Update();  
**+** if (scene->timer > 0) {  
**+** return; **+** }  
**+** const GamePad gamepad = window.GetGamePad();  
**+** if (gamepad.buttonDown & (GamePad::A | GamePad::START)) {  
**+** finalize(scene); // ゲームオーバー画面の後始末. **+** // タイトル画面に戻る.  
**+** gamestate = gamestateTitle;  
**+** initialize(&titleScene);  
**+** }  
**+**}

タイマーが0以下ならプレイヤーの入力を調べ、AキーかEnterキーが押されていたらゲームオーバー画面を終了してタイトル画面を起動しています。

## update関数

続いて更新を行う関数を作ります。  
先ほど書いたprocessInput関数の定義の下に、次のプログラムを追加してください。

gamestate = gamestateTitle;  
 initialize(&titleScene);  
 }  
}  
**+**  
**+**/\*\*  
**+**\* ゲームオーバー画面を更新する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\* @param scene ゲームオーバー画面用構造体のポインタ.  
**+**\*/  
**+**void update(GLFWEW::WindowRef window, GameOverScene\* scene)  
**+**{  
**+** const float deltaTime = window.DeltaTime();  
**+** if (scene->timer > 0) {  
**+** scene->timer -= deltaTime;  
**+** }  
**+** scene->bg.Update(deltaTime);  
**+**}

この関数が行うことは、タイマーと背景スプライトの更新だけです。

## render関数

最後に作成するのは描画関数です。  
先ほど書いたupdate関数の定義の下に、次のプログラムを追加してください。

if (scene->timer > 0) {  
 scene->timer -= deltaTime;  
 }  
 scene->bg.Update(deltaTime);  
}  
**+**  
**+**/\*\*  
**+**\* ゲームオーバー画面を更新する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\* @param scene ゲームオーバー画面用構造体のポインタ.  
**+**\*/  
**+**void render(GLFWEW::WindowRef window, GameOverScene\* scene)  
**+**{  
**+** renderer.BeginUpdate();  
**+** renderer.AddVertices(scene->bg);  
**+** renderer.EndUpdate();  
**+** renderer.Draw(glm::vec2(windowWidth, windowHeight));  
**+**  
**+** fontRenderer.BeginUpdate();  
**+** fontRenderer.AddString(glm::vec2(-112, 16), "GAME OVER");  
**+** fontRenderer.EndUpdate();  
**+** fontRenderer.Draw();  
**+**  
**+** window.SwapBuffers();  
**+**}

背景スプライトと「GAME OVER」という文字を表示するだけです。

## 初期化関数を呼び出す

それでは、作成した関数の呼び出しを書いていきましょう。  
update関数内のタイトル画面に戻る処理を、次のように変更してください。

void update(GLFWEW::WindowRef window)  
 {  
 if (gamestate == gamestateTitle) {  
 update(window, &titleScene);  
 return;  
 } else if (gamestate == gamestateMain) {  
**-** // 自機が破壊されていたらタイトル画面に戻る. **+** // 自機が破壊されていたらゲームオーバー画面を表示する. if (sprPlayer.health <= 0) {  
**-** gamestate = gamestateTitle;  
**-** initialize(&titleScene);  
**+** gamestate = gamestateGameOver;  
**+** initialize(&gameOverScene);  
 return;  
 }  
 }

## 入力処理関数を呼び出す

続いてprocessUpdate関数を変更します。  
processInput関数に次のプログラムを追加してください。

/\*\*  
 \* プレイヤーの入力を処理する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void processInput(GLFWEW::WindowRef window)  
 {  
 if (gamestate == gamestateTitle) {  
 processInput(window, &titleScene);  
 return;  
**+** } else if (gamestate == gamestateGameOver) {  
**+** processInput(window, &gameOverScene);  
**+** return;  
 }

## 更新関数を呼び出す

次は更新関数です。update関数に次のプログラムを追加してください。

/\*\*  
 \* ゲームの状態を更新する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void update(GLFWEW::WindowRef window)  
 {  
 if (gamestate == gamestateTitle) {  
 update(window, &titleScene);  
 return;  
 } else if (gamestate == gamestateMain) {  
// 自機が破壊されていたらゲームオーバー画面を表示する. if (sprPlayer.health <= 0) {  
 gamestate = gamestateGameOver;  
 initialize(&gameOverScene);  
 return;  
 }  
**+** } else if (gamestate == gamestateGameOver) {  
**+** update(window, &gameOverScene);  
**+** return;  
 }  
  
 const float deltaTime = window.DeltaTime();

この場合のように、複数の括弧の内側にプログラムを追加するときは、追加するプログラムがどの括弧に対応しているかを考えながら、注意深く書くようにしてください。

## 描画関数を呼び出す

最後は描画関数の呼び出しです。render関数に次のプログラムを追加してください。

/\*\*  
 \* ゲームの状態を描画する..  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void render(GLFWEW::WindowRef window)  
 {  
 if (gamestate == gamestateTitle) {  
 render(window, &titleScene);  
 return;  
**+** } else if (gamestate == gamestateGameOver) {  
**+** render(window, &gameOverScene);  
**+** return;  
 }  
  
 renderer.BeginUpdate();  
 renderer.AddVertices(sprBackground);

これでゲームオーバー画面が完成しました。  
**ビルドして実行してください。**  
そして、自機が破壊されたときにゲームオーバー画面が表示されること、ゲームオーバー画面でAキーかEnterキーを押すとタイトル画面に戻ること、の2点を確認してください。

**[課題01]**  
自機が破壊されたあと、2秒待ってからゲームオーバー画面に移るようにしてください。  
まず、タイトル画面やゲームオーバー画面のプログラムを参考にして、タイマー変数を追加する必要があるでしょう。次に、自機のhealthを減らしたとき、それが0以下になったらタイマーを2秒にセットします。そして、タイマーが0以下になったらゲームオーバー画面を表示するようにif文を追加します。

# 音を鳴らす

## ヘッダファイルのインクルード

タイトル画面やゲームオーバー画面が増えて、ちょっとゲームらしくなってきました。でも、まだなにか味気ないですね…。きっとその理由は、音声がないからです。

音声を再生する機能はAudio.h(おーでぃお・どっと・えいち)というファイルで宣言されています。まずはこのファイルをインクルードしましょう。

#include "Sprite.h"  
 #include "Font.h"  
 #include "TiledMap.h"  
**+**#include "Audio.h"  
 #include <random>

## 音声制御用変数を追加する

Audio.hを使用する音声の再生では、音声制御用の変数を用意することになっています。そこで、次は音声制御用変数を追加します。スコア変数の定義の下に、次のプログラムを追加してください。

float enemyGenerationTimer; // 次の敵が出現するまでの時間(単位:秒).  
 int score; // プレイヤーのスコア.  
 float timer; // シーン切り替え用のタイマー(課題01のタイマー変数).  
**+  
+**// 音声制御用変数. **+**Audio::SoundPtr bgm;  
**+**Audio::SoundPtr sePlayerShot;  
**+**Audio::SoundPtr seBlast;  
  
 // ゲームの状態.  
 const int gamestateTitle; // タイトル画面用の場面ID.

## 音声再生システムを初期化する

音声を鳴らす前に、再生システムを初期化しておかなければなりません。  
main関数に次のプログラムを追加してください。

// アプリケーションの初期化.  
 GLFWEW::WindowRef window = GLFWEW::Window::Instance();  
 if (!window.Initialize(windowWidth, windowHeight, title)) {  
 return 1;  
 }  
**+** // 音声再生システムの初期化.  
**+** Audio::EngineRef audio = Audio::Engine::Instance();  
**+** if (!audio.Initialize()) {  
**+** return 1;  
**+** }  
 if (!Texture::Initialize()) {  
 return 1;  
 }

音声再生システムはAudio::Engine型のオブジェクトです。この型は、直接変数を作ることが禁止されています(C++ではそういうことができるのです)。そのため、音声を操作するときはAudio::Engine::Instance(おーでぃお・えんじん・いんすたんす)という関数を使って、変数を取得しなければなりません。

システムの初期化にはInitialize(いにしゃらいず)関数を使います。初期化に成功するとtrue、失敗するとfalseが帰ってきますので、失敗していた場合は1を返してプログラムを終了しています。

## 再生システムを更新する

音声再生システムは、定期的に更新処理を実行しなければなりません。  
更新するにはUpdate(あっぷでーと)関数を呼び出します。ゲームループ中のどの場所で呼び出してもいいのですが、直接ゲームループに追加するのが簡単でしょう。

main関数に次のプログラムを追加してください。

// ゲームループ.  
 while (!window.ShouldClose()) {  
 processInput(window);  
 update(window);  
 render(window);  
**+** audio.Update();  
 }  
 Texture::Finalize();

## 再生システムを破棄する

Audio.hによる音声の再生では、プログラムを終了する前に再生システムを破棄しなければなりません。これにはDestroy(ですとろい)関数を使います。  
ゲームループの下に、次のプログラムを追加してください。

// ゲームループ.  
 while (!window.ShouldClose()) {  
 processInput(window);  
 update(window);  
 render(window);  
 audio.Update();  
 }  
 Texture::Finalize();  
**+**audio.Destroy();

## 音声を準備する

音声はハードディスクやSSDなどにファイルとして保存されています。再生するためにはメモリに読み込んで、すぐに再生できるように準備しておかなければなりません。音声ファイルをメモリに読み込むにはPrepare(ぷりぺあ)関数を使います。Prepareは「準備する」という意味の英単語です。  
タイトル画面用のupdate関数に、次のプログラムを追加してください。

// 敵配置マップを読み込む.  
 enemyMap.Load("Res/EnemyMap.json”);  
 mapCurrentPosX = windowWidth;  
 mapProcessedX = windowWidth;  
**+  
+** // 音声を準備する. **+** Audio::EngineRef audio = Audio::Engine::Instance();  
**+** seBlast = audio.Prepare("Res/Audio/Blast.xwm");  
**+** sePlayerShot = audio.Prepare("Res/Audio/PlayerShot.xwm");  
**+** bgm = audio.Prepare("Res/Audio/Neolith.xwm"); }  
}

音声ファイル名をパラメーターに設定してPrepare関数を実行すると、音声操作用のオブジェクトが返されます。返されたオブジェクトを事前に作成した変数に格納しておきます。この変数は音声の再生や停止の際に必要となります。

## 音声をループ再生する

音声を再生するには、音声操作用の変数に対してPlay(ぷれい)関数を呼び出します。  
音声を準備するプログラムの下に、次のプログラムを追加してください。

// 音声を準備する.  
 Audio::EngineRef audio = Audio::Engine::Instance();  
 seBlast = audio.Prepare("Res/Audio/Blast.xwm");  
 sePlayerShot = audio.Prepare("Res/Audio/PlayerShot.xwm");  
 bgm = audio.Prepare("Res/Audio/Neolith.xwm");  
**+**// BGMをループ再生する.  
**+**bgm->Play(Audio::Flag\_Loop);

Play関数はパラメーターとして再生方法を制御するフラグを設定することができます。何も設定しなければ普通に一度だけ再生されます。Audio.hを使った音声の再生では、Audio::Flag\_Loop(おーでぃお・ふらぐ・るーぷ)という値を設定すると、無限にループ再生してくれます。このフラグはBGMなどを再生するときに便利です。

プログラムが書けたら**ビルドして実行してください。**  
ゲームを開始したときにBGMが再生されたら成功です。

**[補足]**  
再生可能なファイル形式はxwm, wav, mp3, wmaなどです。

## 音声を再生する

次に弾の発射音を再生します。  
弾を発射するプログラムに、次のプログラムを追加してください。

Actor\* bullet = fundAvailableActor(  
std::begin(playerBulletList), std::end(playerBulletList));  
// 空いている構造体が見つかったら、それを使って弾を発射する. if (bullet != nullptr) {  
 bullet->spr = Sprite("Res/Objects.png",  
 sprPlayer.Position(), Rect(64, 0, 32, 16));  
 bullet->spr.Tweener(TweenAnimation::Animate::Create(  
TweenAnimation::MoveBy::Create(1, glm::vec3(1200, 0, 0),  
TweenAnimation::EasingType::Linear)));  
 bullet->collisionShape = Rect(-8, -4, 16, 8);  
 bullet->health = 1;  
**+** sePlayerShot->Play(); // 弾の発射音を再生.  
 }  
 }

通常の再生は、パラメーター無しでPlay関数を呼び出すだけです。

敵の爆発音も再生しましょう。  
自機の弾と敵の衝突処理関数に、次のプログラムを追加してください。

void playerBulletAndEnemyContactHandler(Actor \* bullet, Actor \* enemy)  
 {  
 bullet->health -= 1;  
 enemy->health -= 1;  
 if (enemy->health <= 0) {  
 score += 100;  
 Actor\* blast = findAvailableActor(  
 std::begin(effectList), std::end(effectList));  
 if (blast != nullptr) {  
 blast->spr = Sprite("Res/Objects.png", enemy->spr.Position());  
 blast->spr.Animator(FrameAnimation::Animate::Create(tlBlast));  
 namespace TA = TweenAnimation;  
 blast->spr.Tweener(TA::Animate::Create(  
 TA::Rotation::Create(20 / 60.0f, 1.5f)));  
 blast->health = 1;  
**+** seBlast->Play();// 爆発音を再生. }  
 }  
 }

プログラムが書けたら**ビルドして実行してください。**  
弾を発射するときと敵を破壊したときに音声が再生されたら成功です。

## 音声を停止する

音声を停止するにはStop(すとっぷ)関数を使います。  
ゲーム中はBGMがループ再生されていますので、ゲームオーバー画面に移る前に停止させましょう。  
ループフラグを使って再生した場合、Stop関数で停止させるまで再生は止まりません。

update関数でゲームオーバー画面用のinitialize関数を呼び出している付近に、次のプログラムを追加してください。

if (gamestate == gamestateTitle) {  
 update(window, &titleScene);  
 return;  
} else if (gamestate == gamestateMain) {  
// 自機が破壊されていたらゲームオーバー画面を表示する. if (sprPlayer.health <= 0) {  
 if (timer >= 0) {  
 timer -= window.DeltaTime();  
 } else {  
**+** bgm->Stop(); // BGMを停止する. gamestate = gamestateGameOver;  
 initialize(&gameOverScene);  
 return;  
 }  
 }  
}

プログラムが書けたら、**ビルドして実行してください。**自機が破壊されてゲームオーバー画面に移るときに、BGMが停止したら成功です。

## 変数を使わない再生

ちょっとした音声を再生するためだけに変数を用意し、それを管理するのは面倒です。  
そんなときは、Prepareの戻り値に対して直接Play関数を実行することで、変数を管理することなく再生させることが可能です。このやり方で、タイトル画面でゲームを開始するときの音声を再生してみましょう。タイトル画面用のprocessInput関数に、次のプログラムを追加してください。

// AまたはSTARTボタンが押されたら、ゲーム開始待ちモードに移る.  
 const GamePad gamepad = window.GetGamePad();  
 if (gamepad.buttonDown & (GamePad::A | GamePad::START)) {  
 scene->mode = scene->modeNextState;  
 scene->timer = 2.0f;  
**+** Audio::Engine::Instance().Prepare("Res/Audio/Start.xwm")->Play(); }

見ての通り、変数を使わずにPlay関数を使っています。

プログラムが書けたらビルドして実行し、タイトル画面でAキーまたはEnterキーを押したときに、音声が再生されることを確認してください。

**[課題02]**  
自機が破壊されたときに音声を再生してください。

**[課題03]**  
タイトル画面とゲームオーバー画面において、ゲーム中とは異なるBGMを再生してください。  
BGMは適当に用意してください。なお、再生可能なファイル形式はxwm, wav, mp3, wmaなどです。