[OpenGL 2D 2018 第14回]

「強敵」と書いて「とも」と読む

# 敵のバリエーションを増やす

## 敵・アイテムの出現処理を関数にする

今回は新たな敵を出現させようと思いますが、その前に、敵を出現させるプログラムを関数にしておきましょう。というのも、様々な機能を追加してきた結果、update関数が長くなって理解しづらくなってきたからです。  
関数が長くなってきたら、その一部を別の関数としてまとめると、プログラムを理解しやすくなります(長くなる前にそうできるともっと良いのですけどね)。

それでは、作成する関数の名前を決めて、プロトタイプ宣言を行いましょう。  
Main.cppに次のプログラムを追加してください。

/\*  
 \* プロトタイプ宣言.  
 \*/  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);  
 void playerBulletAndEnemyContactHandler(Actor\*, Actor\*);  
 void playerAndEnemyContactHandler(Actor\*, Actor\*);  
 void playerAndItemContactHandler(Actor\*, Actor\*);  
 void playerLaserAndEnemyContactHandler(Actor\*, Actor\*);  
 void stopPlayerLaser();  
**+**void generateObjectFromMap(float);

関数名はgenerateObjectFromMap(じぇねれーと・おぶじぇくと・ふろむ・まっぷ、「マップからオブジェクトを生成する」という意味)としました。敵の出現処理には経過時間が必要なので、引数にfloat型をひとつ設定するようにしています。

次に関数を定義します。  
レーザーを停止する関数の下に、次のプログラムを追加してください。

i->health = 0;  
 }  
 sePlayerLaser->Stop();  
 }  
 }  
**+**  
**+**/\*\*  
**+**\* マップデータを見て敵やアイテムを出現させる.  
**+**\*  
**+**\* @param deltaTime 前回の更新からの経過時間.  
**+**\*/  
**+**void generateObjectFromMap(float deltaTime)  
**+**{  
**+**}

関数を作成したら、update関数にある敵を出現させるプログラムを切り取って(Ctrl+X)、

// 敵の出現.  
 #if 1  
**-** const TiledMap::Layer& tiledMapLayer = enemyMap.GetLayer(0);  
**-** const glm::vec2 tileSize =  
**-** enemyMap.GetTileSet(tiledMapLayer.tilesetNo).size;  
**-** // 敵配置マップ参照位置の更新.  
**- .  
- .  
- .**  
**-** item->spr.Tweener(TA::Animate::Create(  
**-** TA::MoveBy::Create(16, glm::vec3(-1000, 0, 0))));  
**-** item->collisionShape = Rect(-16, -16, 32, 32);  
**-** item->health = 1;  
**-** }  
**-** }  
**-** }  
**-** }  
#else  
 // 出現までの時間が0以下になったら敵を出現させる.  
 enemyGenerationTimer -= deltaTime;  
 if (enemyGenerationTimer <= 0) {

作成した関数に貼り付けてください(Ctrl+V)。

void generateObjectFromMap(float deltaTime)  
{  
**+** const TiledMap::Layer& tiledMapLayer = enemyMap.GetLayer(0);  
**+** const glm::vec2 tileSize =  
**+** enemyMap.GetTileSet(tiledMapLayer.tilesetNo).size;  
**+** // 敵配置マップ参照位置の更新.  
**+** const float enemyMapScrollSpeed = 100; // 更新速度.  
**+** mapCurrentPosX += enemyMapScrollSpeed \* deltaTime;  
**+ .  
+ .  
+ .**  
**+** item->spr.Tweener(TA::Animate::Create(  
**+** TA::MoveBy::Create(16, glm::vec3(-1000, 0, 0))));  
**+** item->collisionShape = Rect(-16, -16, 32, 32);  
**+** item->health = 1;  
**+** }  
**+** }  
**+** }  
**+** }  
}

この関数を、先ほど切り取った位置で呼び出します。update関数に次のプログラムを追加してください。

// 敵の出現.  
 #if 1  
**+** generateObjectFromMap(deltaTime);  
 #else  
 // 出現までの時間が0以下になったら敵を出現させる.  
 enemyGenerationTimer -= deltaTime;  
 if (enemyGenerationTimer <= 0) {

これで敵を出現させるプログラムは関数になりました。  
プログラムが書けたら、**ビルドして実行してください。**関数を作る前と全く同じように敵が出現していれば成功です。

## 敵の情報(データ)を構造体にする

なにかのバリエーションを増やそうと思ったとき、これまではif文を使って分岐していました。  
もしそれぞれのif文の中に似たようなプログラムが作られるようなら、その部分を関数にすれば重複を減らすことができます。

実は、C/C++言語には他にも重複を減らす方法があります。異なっている部分を配列として管理するのです。例えば、さまざまな敵を出現させたい場合、まず必要なのは敵の画像を変えることです。画像を変えるためには、切り取る範囲、つまりRect型の値を変えることだけです。そこで、Rect型の配列を用意します。そうしておけば、あとは配列のどの部分を参照するかを選択するだけで、さまざまな画像を表示することができる、という仕組みです。

配列を作る前に、敵ごとに切り替える必要のあるデータを洗い出しましょう。まず画像の範囲がいりますね。画像が違うということは、衝突判定の範囲も違ってくるでしょうから、衝突判定の範囲も必要でしょう。画像が大きい敵は耐久力も高いでしょうから、health(ヘルス)も変えられるとよさそうです。  
とりあえず3つのデータが必要みたいですね。

さて、これらをひとつひとつ別の配列にした場合、敵を追加したり減らしたりしようと思ったときに、すべての配列に正しくデータを追加したり削除したりしなければなりません。そのような作業は面倒ですし、間違えやすいものです。こういう場合、データを構造体にまとめてしまい、その構造体を配列にします。そうすれば、敵の追加や削除は構造体単位で行うことになるので、間違えにくくなります。

では、ここまでで分かったことをプログラムに書き起こしましょう。  
この構造体はメイン画面で使うものなので、MainScene.hに定義することにします。  
MainScene.hに次のプログラムを追加してください。

/\*\*  
 \* @file MainScene.h  
 \*/  
**+**#include "Sprite.h" **+  
+**/\*\*  
**+**\* マップに配置されたオブジェクトのデータ.  
**+**\*/  
**+**struct MapObjectData {  
**+**Rect imageRect; // 画像の範囲.  
**+**Rect collisionShape; // 衝突判定の範囲.  
**+**int health; // 初期体力.  
**+**};  
  
 /\*\*  
 \* メイン画面で使用する構造体.  
 \*/

Rect構造体の定義はSprite.hにあります。ですから、Sprite.hをインクルードしなければならない点に注意してください。

ところで、どの構造体がどのタイルIDに対応するのかが分からないと、データを選ぶことができませんね。そこで、この構造体にタイルIDを持たせることにします。このタイルIDとマップデータから取得したタイルIDが一致したら、一致した構造体のデータを使って敵を出現させればいいのです。  
MapObjectData構造体に、次のプログラムを追加してください。

/\*\*  
\* マップに配置されたオブジェクトのデータ.  
\*/  
struct MapObjectData {  
**+**int id; // オブジェクトのID.  
Rect imageRect; // 画像の範囲.  
Rect collisionShape; // 衝突判定の範囲.  
int health; // 初期体力.  
};

構想体の名前はMapObjectData(まっぷ・おぶじぇくと・でーた)としました。「マップに置かれている物体の情報」のような意味になります。

ここで、敵のタイルIDを定義しておきましょう。MainScene.hに、次のプログラムを追加してください。

/\*\*  
 \* @file MainScene.h  
 \*/  
**+**  
**+**// マップに配置されたオブジェクトのID.  
**+**const int tileId\_SmallEnemy = 256;  
**+**const int tileId\_MediumEnemy = 191;  
**+**const int tileId\_LargeEnemy = 63;  
**+**const int tileId\_Boss = 75;  
**+**const int tileId\_PowerUpItem = 230;  
  
 /\*\*  
 \* マップに配置されたオブジェクトのデータ.  
 \*/

これまではenemyIdとpowerUpItemIdの2つだけでした。また、定義している場所も、それを使用しているプログラムのすぐ上でした。ですが、そのままでは、このタイルIDがほかの場所で必要になったときに使うことができません。  
そこで、定義する場所をMainScene.hに変更します。こうしておけば、MainScene.hをインクルードするだけでIDを使うことができます。

また、変数名にも少し工夫をして、Visual Studioからの支援を受けやすくしてみました。先頭の数文字を特徴的な名前で揃えておくと、その数文字を入力した時点で、同じ文字で始まる単語だけがリストアップされます。これは、最初の数文字さえ覚えておけば、残りのつづりを正確に覚えていなくても構わないことを意味します。覚えておくべきことが少なくなるので、それだけプログラムを書くのが楽になるわけです。

## 敵のデータを検索する

次に、敵データ構造体の配列を定義し、タイルIDに一致する構造体を取得する関数を作ります。  
さて、どこに関数を作りましょうか…。まず考えられるのはMain.cppに追加することです。とはいえ、Main.cppはすでにかなり行数が多いので、これ以上あれこれ追加するとさらに読みにくくなってしまいます。できれば他の場所に作りたいものです。  
ここで、今回追加したい関数について考えてみましょう。この関数は、敵の出現に関わるものです。そして、敵の出現は(今のところは)メイン画面だけに関係します。ということは、MainScene.cppを追加して、そこに関数を作成するのがいいのではないでしょうか。

早速始めましょう。まずはプロトタイプ宣言を書きます。  
MainScene.hに次のプログラムを追加してください。

/\*\*  
 \* メイン画面で使用する構造体.  
 \*/  
 struct MainScene  
 {  
 int stageNo;  
 };  
 bool initialize(MainScene\*);  
**+**  
**+**const MapObjectData\* findMapObjectData(int);

関数名はfindMapObjectData(ふぁいんど・まっぷ・おぶじぇくと・でーた)としました。

続いてMainScene.cppを追加します。ソリューションエクスプローラーのSrcフィルタを右クリックし「追加→新しい項目」を選んで「新しい項目の追加」ウィンドウを開きます。ファイルの種類で「C++ファイル」を選び、場所の末尾に「\Src」を追加し、名前を「MainScene.cpp」に変更します。変更したら、「追加」ボタンを押すとファイルが追加されます。ファイルの追加方法について、詳しくは以前の資料を参照してください。

それでは関数を定義しましょう。  
MainScene.cppに次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file MainScene.cpp  
**+**\*/  
**+**#include "MainScene.h"  
**+**  
**+**/\*\* **+**\* マップに配置されている物体のデータ. **+**\*/ **+**MapObjectData mapObjectDataList[] = {  
**+** { tileId\_SmallEnemy, Rect(480, 0, 32, 32), Rect(-16, -16, 32, 32), 1 },  
**+** { tileId\_MediumEnemy, Rect(464, 128, 48, 32), Rect(-24, -16, 48, 32), 5 },  
**+** { tileId\_LargeEnemy, Rect(448, 352, 64, 64), Rect(-32, -32, 64, 64), 30 },  
**+** { tileId\_Boss, Rect(320, 128, 128, 256), Rect(-48, -112, 96, 224), 200 },  
**+**};  
**+  
+**/\*\*  
**+**\* タイルIDに対応するMapObjectDataを取得する.  
**+**\*  
**+**\* @param tileId 検索するタイルID.  
**+**\*  
**+**\* @retval nullptr以外 タイルIDに対応するデータ.  
**+**\* @retval nullptr 対応するデータが見つからなかった.  
**+**\*/  
**+**const MapObjectData\* findMapObjectData(int tileId)  
**+**{  
**+** for (const MapObjectData\* i = std::begin(mapObjectDataList);  
**+** i != std::end(mapObjectDataList); ++i) {  
**+** if (i->id == tileId) {  
**+** return i;  
**+** }  
**+** }  
**+** return nullptr;  
**+**}

この関数は、dataという名前の配列を先頭から調べていきます。そして、引数として設定されたtileIdと一致するidの構造体を見つけたら、そのポインタを返して終了します。また、一致する構造体が見つからなかった場合はnullptr(ぬる・ぽいんたー)、つまり「何も指していないポインター」を返します。これによって、対応するデータが存在しないことが、findMapObjectData関数を呼び出したプログラムに伝えられます。

## 敵のバリエーションを増やす

それでは、作成した関数を使って、敵を出現させてみましょう。  
Main.cppのgenerateObjectFromMap関数を次のように変更してください。

// 次の列に到達したらデータを読む.  
 if (mapCurrentPosX - mapProcessedX >= tileSize.x) {  
 mapProcessedX += tileSize.x;  
 const int mapX = static\_cast<int>(mapProcessedX / tileSize.x);  
 for (int mapY = 0; mapY < tiledMapLayer.size.y; ++mapY) {  
**-** const int enemyId = 256; // 敵とみなすタイルID.  
**-** const int powerUpItemId = 230; // パワーアップアイテムのID.  
 const int tileId = tiledMapLayer.At(mapY, mapX);  
**-** if (tileId == enemyId) {  
**+** const MapOBjectData\* data = findMapObjectData(tileId);  
**+** if (data != nullptr) {  
 Actor\* enemy =  
 findAvailableActor(std::begin(enemyList), std::end(enemyList));  
 // 空いている構造体が見つかったら、それを使って敵を出現させる.  
 if (enemy != nullptr) {  
 const float y =  
**-** windowHeight \* 0.5f - static\_cast<float>(mapY \* tileSize.y);  
**+** (windowHeight – data->imageRect.size.y) \* 0.5f -  
**+** static\_cast<float>(mapY \* tileSize.y);  
**+** const float x = (windowWidth + data->imageRect.size.x) \* 0.5f;  
 enemy->spr = Sprite("Res/Objects.png",  
**-** glm::vec3(0.5f \* windowWidth, y, 0), Rect(480, 0, 32, 32));  
**+** glm::vec3(x, y, 0), data->imageRect);  
 enemy->spr.Animator(FrameAnimation::Animate::Create(tlEnemy));  
 namespace TA = TweenAnimation;  
 TA::SequencePtr seq = TA::Sequence::Create(4);  
 seq->Add(TA::MoveBy::Create(1, glm::vec3(0, 100, 0),  
 TA::EasingType::EaseInOut, TA::Target::Y));  
 seq->Add(TA::MoveBy::Create(1, glm::vec3(0, -100, 0),  
 TA::EasingType::EaseInOut, TA::Target::Y));  
 TA::ParallelizePtr par = TA::Parallelize::Create(1);  
 par->Add(seq);  
 par->Add(TA::MoveBy::Create(8, glm::vec3(-1000, 0, 0),  
 TA::EasingType::Linear, TA::Target::X));  
 enemy->spr.Tweener(TA::Animate::Create(par));  
**-** enemy->collisionShape = Rect(-16, -16, 32, 32);  
**-** enemy->health = 2;  
**+** enemy->collisionShape = data->collisionShape;  
**+** enemy->health = data->health;  
 }  
**-** } else if (tileId == powerUpItemId) {  
**+** } else if (tileId == tileId\_PowerUpItem) {  
 Actor\* item =  
 findAvailableActor(std::begin(itemList), std::end(itemList));

変更箇所が多いので、一箇所ずつ注意して書き換えてください。

プログラムが書けたら、**ビルドして実行してください。**  
…うーん。敵の種類、増えているんでしょうか？  
耐久力や衝突判定の大きさは変わっている気がするんですが、画像は変わっていないようです。

## 敵のアニメーションを増やす

画像が変化しない理由は次の行にあります。

enemy->spr.Animator(FrameAnimation::Animate::Create(tlEnemy));

ようするに、敵の種類に関係なく、同じアニメーションを設定しているのが問題なのです。  
試しに、この行の先頭に「//」を加えてコメント化してから、**ビルドして実行してください。**ちゃんと違う画像が表示されると思います。

確かに画像は変わりましたが、せっかく作ったアニメーションが使えないのでは、ゲームの見た目としてよろしくありません。やりたいことは、敵の種類によってアニメーションを変えることです。  
おそらくみなさんは

「なら、例の構造体(敵のデータの構造体)に、アニメーションのデータを追加すればいいのでは？」

と考えたことでしょう。  
そして、それは正しい考えです。

問題は「アニメーションデータをどうやって追加するか」という部分にあります。  
アニメーションを設定するときは「タイムライン」というものを使うのでした。そしてプログラムにおいては、タイムラインはFrameAnimation::TimelinePtr型の変数に格納されています。素直に考えるなら、MapObjectData構造体にFrameAnimation::TimelinePtr型のメンバを追加すればいいように思えます。そこまでは問題ありません。

しかし、タイムラインは初期化しなければ使えません。そこで、MapObjectData構造体の配列を初期化する関数を追加します。そして、この関数をプログラムを初期化しているときに呼び出すのです。

それではやっていきましょう。まずは構造体のメンバを追加し、プロトタイプ宣言を追加します。  
MainScene.hに次のプログラムを追加してください。

/\*\*  
\* マップに配置されたオブジェクトのデータ.  
\*/  
struct MapObjectData {  
int id; // オブジェクトのID.  
Rect imageRect; // 画像の範囲.  
Rect collisionShape; // 衝突判定の範囲.  
int health; // 初期体力.  
**+** FrameAnimation::TimelinePtr tl; // アニメーション・タイムライン.  
};  
  
 /\*\*  
 \* メイン画面で使用する構造体.  
 \*/  
 struct MainScene  
 {  
 int stageNo;  
 };  
 bool initialize(MainScene\*);  
  
**+**void initializeMapObjectDataList();  
 const MapObjectData\* findMapObjectData(int);

関数名はinitializeMapObjectDataList(いにしゃらいず・まっぷ・おぶじぇくと・でーた・りすと)としました。  
そしてこの関数を、初期化処理の途中で呼び出します。  
Main.cppに次のプログラムを追加してください。

// アニメーション・タイムラインの作成.  
 tlEnemy = FrameAnimation::Timeline::Create(enemyKeyFrames);  
 tlPlayer = FrameAnimation::Timeline::Create(playerKeyFrames);  
 tlBlast = FrameAnimation::Timeline::Create(blastKeyFrames);  
 tlEnemyBullet = FrameAnimation::Timeline::Create(enemyBulletKeyFrames);  
  
**+**initializeMapObjectDataList();  
  
 // タイトル画面を初期化する.  
 gamestate = gamestateTitle;  
 initialize(&titleScene);

次は、初期化関数で使用するアニメーションデータを用意します。  
MainScene.cppに次のプログラムを追加してください。なお、この部分を書くときはMain.cppにあるenemyKeyFrames配列をコピーしてくるといいでしょう。実際、smallEnemyKeyFrames配列は配列名が違うだけで、内容はenemyKeyFrames配列そのものです。

/\*\*  
 \* @file MainScene.cpp  
 \*/  
 #include "MainScene.h"  
**+**  
**+**// 敵のアニメーション.  
**+**const FrameAnimation::KeyFrame smallEnemyKeyFrames[] = {  
**+** { 0.000f, glm::vec2(480, 0), glm::vec2(32, 32) },  
**+** { 0.125f, glm::vec2(480, 96), glm::vec2(32, 32) },  
**+** { 0.250f, glm::vec2(480, 64), glm::vec2(32, 32) },  
**+** { 0.375f, glm::vec2(480, 32), glm::vec2(32, 32) },  
**+** { 0.500f, glm::vec2(480, 0), glm::vec2(32, 32) },  
**+**};  
**+**const FrameAnimation::KeyFrame mediumEnemyKeyFrames[] = {  
**+** { 0.000f, glm::vec2(464, 128), glm::vec2(48, 32) },  
**+** { 0.125f, glm::vec2(464, 160), glm::vec2(48, 32) },  
**+** { 0.250f, glm::vec2(464, 128), glm::vec2(48, 32) },  
**+**};  
**+**const FrameAnimation::KeyFrame largeEnemyKeyFrames[] = {  
**+** { 0.000f, glm::vec2(448, 352), glm::vec2(64, 64) },  
**+** { 0.125f, glm::vec2(448, 416), glm::vec2(64, 64) },  
**+** { 0.250f, glm::vec2(448, 352), glm::vec2(64, 64) },  
**+**};  
**+**const FrameAnimation::KeyFrame bossKeyFrames[] = {  
**+** { 0.000f, glm::vec2(320, 128), glm::vec2(128, 256) },  
**+** { 1.000f, glm::vec2(320, 128), glm::vec2(128, 256) },  
**+**};  
  
/\*\*  
\* マップに配置されている物体のデータ.  
\*/

データの用意ができたので関数を定義します。  
MainScene.cppに次のプログラムを追加してください。

{ tileId\_Boss, Rect(320, 128, 128, 256), Rect(-48, -112, 96, 224), 200 },  
};  
**+**  
**+**/\*\*  
**+**\* 物体のデータ配列を初期化する.  
**+**\*/  
**+**void initializeMapObjectDataList()  
**+**{  
**+** mapObjectDataList[0].tl =  
**+** FrameAnimation::Timeline::Create(smallEnemyKeyFrames);  
**+** mapObjectDataList[1].tl =  
**+** FrameAnimation::Timeline::Create(mediumEnemyKeyFrames);  
**+** mapObjectDataList[2].tl =  
**+** FrameAnimation::Timeline::Create(largeEnemyKeyFrames);  
**+** mapObjectDataList[3].tl =  
**+** FrameAnimation::Timeline::Create(bossKeyFrames);  
**+**}  
  
/\*\*  
\* タイルIDに対応するMapObjectDataを取得する.  
\*  
\* @param tileId 検索するタイルID.

これでmapObjectDataListの初期化は完了です。  
ようやく準備ができたので、実際にアニメーションを設定します。  
Main.cppのgenerateObjectFromMap関数を次のように変更してください。

const float x = (windowWidth + data->imageRect.size.x) \* 0.5f;  
enemy->spr = Sprite("Res/Objects.png",  
 glm::vec3(x, y, 0), data->imageRect);  
**-**enemy->spr.Animator(FrameAnimation::Animate::Create(tlEnemy));  
**+**enemy->spr.Animator(FrameAnimation::Animate::Create(data->tl));  
 namespace TA = TweenAnimation;  
 TA::SequencePtr seq = TA::Sequence::Create(4);  
 seq->Add(TA::MoveBy::Create(1, glm::vec3(0, 100, 0),  
 TA::EasingType::EaseInOut, TA::Target::Y));

プログラムが書けたら、**ビルドして実行してください。**  
敵の種類によって異なるアニメーションが動いていれば成功です。

## 行動を設定するプログラムを関数にする

ここまでで、敵の見た目や耐久力などを変えられるようになりました。  
次は敵の行動を変えられるようにしましょう。

さて、敵の行動はトウィーニングによって設定しています。ここで問題になるのは、トウィーニングの設定は、単にデータを切り替えるだけというわけにはいかず、プログラムを書かなければならないことです(できないわけではありませんが、それを可能にするにはたくさんの別のプログラムを書かなければなりません)。

ちょっと視点を変えてみましょう。プログラムの一部を関数にするのは、それほど難しくありません(よね？)。行動設定プログラムを関数にしておけば、敵のデータには「どの関数を使って行動を設定するか」という情報を持たせられる気がします。こういうときは「関数のポインタ」の出番です。

関数のポインタを作るときは、まず「関数の型」を決めるのでした。  
今回作る関数は、「トウィーニングを制御する変数」を作って返してくれる関数にしようと思います。トウィーニング変数を表す型は「TweenAnimation::TweenPtr(とうぃーん・あにめーしょん・とうぃーん・ぽいんたー)」です。引数はなくてもいいでしょう。

それでは、MapObjectData構造体に次のプログラムを追加してください。  
「関数の型」を定義するには「using宣言」を使うのでした。=(イコール)の左に「型の名前」、右に「戻り値や引数の型」を書きます。

/\*  
 \* マップに配置されたオブジェクトのデータ.  
 \*/  
 struct MapObjectData {  
 int type; // オブジェクトの種類.  
 int id; // オブジェクトのID.  
 Rect imageRect; // 画像の範囲.  
 Rect collisionShape; // 衝突判定の範囲.  
 int health; // 初期体力.  
 FrameAnimation::TimelinePtr tl; // アニメーション・タイムライン.  
**+**  
**+** using CreateTweenFunc = TweenAnimation::TweenPtr(\*)(); // 行動設定関数の型.  
**+** CreateTweenFunc createTween; // 行動を設定する関数.  
 };

関数の型の名前はCreateTweenFunc(くりえいと・とうぃーん・ふぁんく)としました。「トウィーニング変数を作る関数」というような意味です。また、関数のポインタの名前はcreateTween(くりえいと・とうぃーん)としました。

次に、実際に行動を設定する関数を作っていきます。  
いつものように、プロトタイプ宣言から始めましょう。  
MainScene.cppに次のプログラムを追加してください。

/\*\*  
 \* @file MainScene.cpp  
 \*/  
 #include "MainScene.h"  
**+**  
**+**// プロトタイプ宣言.  
**+**TweenAnimation::TweenPtr createCurveTween();  
  
 // 敵のアニメーション.  
 const FrameAnimation::KeyFrame smallEnemyKeyFrames[] = {

そして定義を作ります。  
findMapObjectData関数定義の下に、次のプログラムを追加してください。

if (i->id == tileId) {  
 return i;  
 }  
 }  
 return nullptr;  
 }  
**+**  
**+**/\*\*  
**+**\* 敵の移動パターンを作成する(蛇行タイプ).  
**+**\*/  
**+**void createCurveTween()  
**+**{  
**+**}

トウィーニングを設定するプログラムは、Main.cppのgenerateObjectFromMap関数からコピー(Ctrl+C)して、次のようにsetCurveTweener関数に貼り付けてください(Ctrl+V)。

/\*\*  
\* 敵の移動パターンを作成する(蛇行タイプ).  
\*/  
 void createCurveTween()  
 {  
**+** namespace TA = TweenAnimation;  
**+** TA::SequencePtr seq = TA::Sequence::Create(4);  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(0, 100, 0),  
**+** TA::EasingType::EaseInOut, TA::Target::Y));  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(0, -100, 0),  
**+** TA::EasingType::EaseInOut, TA::Target::Y));  
**+** TA::ParallelizePtr par = TA::Parallelize::Create(1);  
**+** par->Add(seq);  
**+** par->Add(TA::MoveBy::Create(8, glm::vec3(-1000, 0, 0),  
**+** TA::EasingType::Linear, TA::Target::X));  
 }

さて、この関数はトウィーニング変数を返さなければなりませんが、貼り付けただけではそうなっていません。そこで、createCurveTween関数の末尾に、次のプログラムを追加してください。

TA::ParallelizePtr par = TA::Parallelize::Create(1);  
 par->Add(seq);  
 par->Add(TA::MoveBy::Create(8, glm::vec3(-1000, 0, 0),  
 TA::EasingType::Linear, TA::Target::X));  
**+** return par;  
 }

コピーしたプログラムが最終的に作っているとウィーニング変数はparですから、これを返しておきます。

それでは、定義した関数を敵データ配列に設定しましょう。  
mapObjectDataList配列を、次のように変更してください。

/\*\*\* マップに配置されている物体のデータ.\*/MapObjectData mapObjectDataList[] = {  
**-** { tileId\_SmallEnemy, Rect(480, 0, 32, 32), Rect(-16, -16, 32, 32), 1 },  
**-** { tileId\_MediumEnemy, Rect(464, 128, 48, 32), Rect(-24, -16, 48, 32), 5 },  
**-** { tileId\_LargeEnemy, Rect(448, 352, 64, 64), Rect(-32, -32, 64, 64), 30 },  
**-** { tileId\_Boss, Rect(320, 128, 128, 256), Rect(-48, -112, 96, 224), 200 },  
**+** { tileId\_SmallEnemy, Rect(480, 0, 32, 32), Rect(-16, -16, 32, 32), 1,  
**+** nullptr, createCurveTween },  
**+** { tileId\_MediumEnemy, Rect(464, 128, 48, 32), Rect(-24, -16, 48, 32), 5,  
**+** nullptr, createCurveTween },  
**+** { tileId\_LargeEnemy, Rect(448, 352, 64, 64), Rect(-32, -32, 64, 64), 30,  
**+** nullptr, createCurveTween },  
**+** { tileId\_Boss, Rect(320, 128, 128, 256), Rect(-48, -112, 96, 224), 200,  
**+** nullptr, createCurveTween },  
};

最後に、追加した関数のポインタを使って敵の行動を設定します。  
Main.cppのgenerateObjectFromMap関数を次のように変更してください。

const float x = (windowWidth + data->imageRect.size.x) \* 0.5f;  
enemy->spr = Sprite("Res/Objects.png",  
 glm::vec3(x, y, 0), data->imageRect);  
 enemy->spr.Animator(FrameAnimation::Animate::Create(tlEnemy));  
**-**namespace TA = TweenAnimation;  
**-**TA::SequencePtr seq = TA::Sequence::Create(4);  
**-**seq->Add(TA::MoveBy::Create(1, glm::vec3(0, 100, 0),  
**-** TA::EasingType::EaseInOut, TA::Target::Y));  
**-**seq->Add(TA::MoveBy::Create(1, glm::vec3(0, -100, 0),  
**-** TA::EasingType::EaseInOut, TA::Target::Y));  
**-**TA::ParallelizePtr par = TA::Parallelize::Create(1);  
**-**par->Add(seq);  
**-**par->Add(TA::MoveBy::Create(8, glm::vec3(-1000, 0, 0),  
**-** TA::EasingType::Linear, TA::Target::X));  
**-**enemy->spr.Tweener(TA::Animate::Create(par));  
**+**enemy->spr.Tweener(TweenAnimation::Animate::Create(data->createTween()));  
 enemy->collisionShape = data->collisionShape;  
 enemy->health = data->health;

プログラムが書けたら、**ビルドして実行してください。**敵が蛇行しながら進んできたら(つまり、変更前と全く同じように動いていたら)成功です。

## 敵によって行動を変える

敵の行動を関数で作れるようになったので、今度は敵の種類によって違う行動をさせてみましょう。  
MainScene.cppのプロトタイプ宣言に、次のプログラムを追加してください。

/\*\*  
 \* @file MainScene.cpp  
 \*/  
 #include "MainScene.h"  
  
 // プロトタイプ宣言.  
 TweenAnimation::TweenPtr createCurveTween();  
**+**TweenAnimation::TweenPtr createUTurnTween();  
**+**TweenAnimation::TweenPtr createSwayTween();  
**+**TweenAnimation::TweenPtr createBossTween();  
  
 // 敵のアニメーション.  
 const FrameAnimation::KeyFrame smallEnemyKeyFrames[] = {

そして、追加した関数をmapObjectDataListに設定します。  
mapObjectDataListを次のように変更してください。

/\*\*\* マップに配置されている物体のデータ.\*/MapObjectData mapObjectDataList[] = {  
 { tileId\_SmallEnemy, Rect(480, 0, 32, 32), Rect(-16, -16, 32, 32), 1,  
**-** nullptr, createCurveTween },  
**+** nullptr, createCurveTween },  
 { tileId\_MediumEnemy, Rect(464, 128, 48, 32), Rect(-24, -16, 48, 32), 5,  
**-** nullptr, createCurveTween },  
**+** nullptr, createUTurnTween },  
 { tileId\_LargeEnemy, Rect(448, 352, 64, 64), Rect(-32, -32, 64, 64), 30,  
**-** nullptr, createCurveTween },  
**+** nullptr, createSwayTween },  
 { tileId\_Boss, Rect(320, 128, 128, 256), Rect(-48, -112, 96, 224), 200,  
**-** nullptr, createCurveTween },  
**+** nullptr, createBossTween },  
};

最後に、関数の定義を行います。  
createCurveTween関数の定義の下に、次のプログラムを追加してください。

TA::ParallelizePtr par = TA::Parallelize::Create(1);  
 par->Add(seq);  
 par->Add(TA::MoveBy::Create(8, glm::vec3(-1000, 0, 0),  
 TA::EasingType::Linear, TA::Target::X));  
 return par;  
 }  
**+**  
**+**/\*\*  
**+**\* 敵の移動パターンを作成する(Uターンタイプ).  
**+**\*/  
**+**TweenAnimation::TweenPtr createUTurnTween()  
**+**{  
**+** namespace TA = TweenAnimation;  
**+** TA::SequencePtr seq = TA::Sequence::Create(1);  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(-100, 0, 0),  
**+** TA::EasingType::EaseOut));  
**+** seq->Add(TA::Wait::Create(2));  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(100, 0, 0),  
**+** TA::EasingType::EaseIn));  
**+** return seq;  
**+**}  
**+**  
**+**/\*\*  
**+**\* 敵の移動パターンを設定する(行ったり来たりするタイプ).  
**+**\*/  
**+**TweenAnimation::TweenPtr createSwayTween()  
**+**{  
**+** namespace TA = TweenAnimation;  
**+** TA::SequencePtr seq = TA::Sequence::Create(1);  
**+** seq->Add(TA::MoveBy::Create(2, glm::vec3(-200, 0, 0),  
**+** TA::EasingType::EaseOut));  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(0, 100, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** for (int i = 0; i < 2; ++i) {  
**+** seq->Add(TA::MoveBy::Create(2, glm::vec3(0, -200, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** seq->Add(TA::MoveBy::Create(2, glm::vec3(0, 200, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** }  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(0, -100, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** seq->Add(TA::MoveBy::Create(4, glm::vec3(-800, 0, 0),  
**+**  TA::EasingType::EaseIn));  
**+** return seq;  
**+**}  
**+**  
**+**/\*\*  
**+**\* 敵の移動パターンを設定する(ボス用).  
**+**\*/  
**+**TweenAnimation::TweenPtr createBossTween()  
**+**{  
**+** namespace TA = TweenAnimation;  
**+** TA::SequencePtr seq = TA::Sequence::Create(1);  
**+** seq->Add(TA::MoveBy::Create(2, glm::vec3(-200, 0, 0),  
**+**  TA::EasingType::EaseOut));  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(0, 100, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** for (int i = 0; i < 8; ++i) {  
**+** seq->Add(TA::MoveBy::Create(2, glm::vec3(0, -200, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** seq->Add(TA::MoveBy::Create(2, glm::vec3(0, 200, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** }  
**+** seq->Add(TA::MoveBy::Create(1, glm::vec3(0, -100, 0),  
**+**  TA::EasingType::EaseInOut));  
**+** seq->Add(TA::MoveBy::Create(4, glm::vec3(-800, 0, 0),  
**+**  TA::EasingType::EaseIn));  
**+** return seq;  
**+**}

プログラムが書けたら、**ビルドして実行してください。**  
敵の種類によって異なる動きをしていたら成功です。

# 敵の攻撃をプログラムする

## 敵の弾用のActor配列を追加する

敵の種類が増えて、しかもそれぞれが違った動きを見せるようになりました。しかし、いまだに彼らの攻撃手段は体当たりだけです。ワイドショットとレーザーを装備した自機にとって、このような相手を倒すのは、赤子の手をひねるようなものです。このままではちょっと張り合いがなさすぎて、ゲームとしての面白みに欠けますね。  
そこで、プレイヤーに本気を出してもらうために、敵も弾を撃ってくるようにしましょう。

まずは敵の弾用のActor配列を追加します。  
Main.cppに次のプログラムを追加してください。

Actor enemyList[128]; // 敵のリスト.  
 Actor playerBulletList[128]; // 自機の弾のリスト.  
 Actor playerLaserList[3]; // 自機のレーザーのリスト.  
 Actor effectList[128]; // 爆発などの特殊効果用スプライトのリスト.  
 Actor itemList[64]; // アイテム用スプライトのリスト.  
**+**Actor enemyBulletList[256]; // 敵の弾のリスト.  
  
 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.  
 float timer; // シーン切り替えで使用するタイマー.  
  
 const int weaponLevelMin = 1; // 自機の武器強化の最低段階.  
 const int weaponLevelMax = 7; // 自機の武器強化の最高段階.  
 int weaponLevel; // 自機の武器強化段階.

敵の弾用のActor配列の名前はenemyBulletList(えねみー・ばれっと・りすと)とし、要素数は256個としました。

**[補足]**  
256という個数に大した根拠はありません。敵の数の倍も画面に表示できれば十分だろうと考えただけです。

**[課題01]**  
enemyBulletListの初期化、更新、描画を行う関数の呼び出しを、適切な位置に追加してください。

## Actor構造体にタイマー変数を追加する

敵は、自機めがけて弾を発射するようにしたいと思います。ただ、メインループのたびに弾を発射していては、レーザービームのようになり、どうやっても避けられない攻撃になってしまいます。100%勝てないものはゲームとは呼べません。ということで、一定時間ごとに弾を発射するようにしましょう。

時間の経過を計測するには、時刻を記録する変数が必要です。時刻記録用の配列を作る方法もありますが、今回はActor構造体に追加してしまいます。敵じゃなくても時間経過を計れると便利なこともあるでしょうから。きっと。多分。

Actor.hに定義されているActor構造体に、次のプログラムを追加してください。

/\*\*  
 \* ゲームキャラクター構造体.  
 \*/  
 struct Actor  
 {  
 Sprite spr; // 画像表示用スプライト.  
 Rect collisionShape; // 衝突判定の位置と大きさ.  
 int health; // 耐久力(0以下なら破壊されている).  
**+** float timer; // 多目的タイマー.  
 };

timerメンバ変数はどこかのタイミングで初期化しなければなりません。そのタイミングですが、敵を出現させたときがいいでしょう。  
generateObjectFromMap関数に、次のプログラムを追加してください。

enemy->spr.Animator(FrameAnimation::Animate::Create(tlEnemy));  
enemy->spr.Tweener(TweenAnimation::Animate::Create(data->createTween()));  
 enemy->collisionShape = data->collisionShape;  
 enemy->health = data->health;  
**+** enemy->timer = 0;  
 }  
 } else if (tileId == tileId\_PowerUpItem) {

## 弾を発射する

敵が弾を発射するプログラムも関数にしましょう。  
Main.cppのプロトタイプ宣言に、次のプログラムを追加してください。

void playerBulletAndEnemyContactHandler(Actor\*, Actor\*);  
 void playerAndEnemyContactHandler(Actor\*, Actor\*);  
 void playerAndItemContactHandler(Actor\*, Actor\*);  
 void playerLaserAndEnemyContactHandler(Actor\*, Actor\*);  
 void playerAndEnemyBulletContactHandler(Actor\*, Actor\*);  
 void stopPlayerLaser();  
 void generateObjectFromMap(float);  
**+**void updateEnemies(float);  
  
 /\*\*  
 \* メイン画面用の構造体の初期設定を行う.  
 \*

関数名はupdateEnemies(あっぷでーと・えねみーず)としました。  
この関数はメイン画面用のupdate関数で呼び出します。  
メイン画面用のupdate関数に、次のプログラムを追加してください。

#endif  
 // Actorの更新.  
**+** updateEnemies(deltaTime);  
 updateActorList(std::begin(enemyList), std::end(enemyList), deltaTime);  
 updateActorList(  
 std::begin(playerBulletList), std::end(playerBulletList), deltaTime);  
 updateActorList(  
 std::begin(playerLaserList), std::end(playerLaserList), deltaTime);  
 updateActorList(std::begin(effectList), std::end(effectList), deltaTime);  
 updateActorList(std::begin(itemList), std::end(itemList), deltaTime);  
 updateActorList(  
 std::begin(enemyBulletList), std::end(enemyBulletList), deltaTime);

最後に、関数を定義します。  
generateObjectFromMap関数の定義の下に、次のプログラムを追加してください。

namespace TA = TweenAnimation;  
 item->spr.Tweener(TA::Animate::Create(  
 TA::MoveBy::Create(16, glm::vec3(-1000, 0, 0))));  
 item->collisionShape = Rect(-16, -16, 32, 32);  
 item->health = 1;  
 }  
 }  
 }  
 }  
 }  
**+**  
**+**/\*\*  
**+**\* 敵の状態を更新する.  
**+**\*  
**+**\* @param deltaTime 前回の更新からの経過時間.  
**+**\*/  
**+**void updateEnemies(float deltaTime)  
**+**{  
**+** const glm::vec3 playerPos = sprPlayer.spr.Position();  
**+** for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** if (i->health <= 0) {  
**+** continue;  
**+** }  
**+**  
**+** // 一定時間ごとに弾を発射.  
**+** const float shotInterval = 2.0f;  
**+** i->timer += deltaTime;  
**+** if (i->timer < shotInterval) {  
**+** continue;  
**+** }  
**+** i->timer -= shotInterval;  
**+**  
**+** // 空いている敵の弾を検索.  
**+** Actor\* bullet = findAvailableActor(  
**+** std::begin(enemyBulletList), std::end(enemyBulletList));  
**+** if (bullet == nullptr) {  
**+** continue;  
**+** }  
**+**  
**+** // 自機の方向を計算.  
**+** const glm::vec3 distance = playerPos - i->spr.Position();  
**+** const float radian = std::atan2(distance.y, distance.x);  
**+** const float c = std::cos(radian);  
**+** const float s = std::sin(radian);  
**+**  
**+** // 自機に向かって弾を発射.  
**+** bullet->spr =  
**+** Sprite("Res/Objects.png", i->spr.Position(), Rect(464, 0, 16, 16));  
**+** namespace TA = TweenAnimation;  
**+** bullet->spr.Tweener(TA::Animate::Create(  
**+** TA::MoveBy::Create(8, glm::vec3(1200.0f \* c, 1200.0f \* s, 0))));  
**+** bullet->spr.Rotation(radian + 3.14f);  
**+** bullet->collisionShape = Rect(-4, -4, 8, 8);  
**+** bullet->health = 1;  
**+** }  
**+**}

敵から自機へ向かう角度を計算するには、敵から自機までの距離distance(でぃすたんす)を計算したあと、そのY成分とX成分をatan2(えー・たん・つー)という関数に引き渡します。  
atan2はXとYから角度を計算してくれる関数です。  
角度さえ手に入ればあとは簡単。コサインとサインを計算し、それを使って弾を移動させるだけです。  
それから、Rotationメンバ関数にradianではなくradian+3.14fを設定している理由ですが、これは敵の弾の画像が左向き、つまり180°回転した画像になっているからです。

プログラムが書けたら、**ビルドして実行してください。**  
敵が自機に向かって弾を発射してきたら成功です。

## 画面外にいるときは弾を撃たせない

先ほどのプログラムでは、敵が画面左に消えた後も弾を撃ってきます。倒せない位置から弾を撃つという行為は、プレイヤーに不公平感を与えてしまい、よくありません。そこで、敵が画面外に出ていたら弾を発射しないようにしましょう。  
updateEnemies関数に、次のプログラムを追加してください。

if (i->timer < shotInterval) {  
 continue;  
 }  
 i->timer -= shotInterval;  
**+**  
**+**// 画面外にいるときは発射しない.  
**+**const glm::vec3 enemyPos = i->spr.Position();  
**+**if (enemyPos.x <= windowWidth \* -0.5f || enemyPos.x > windowWidth \* 0.5f) {  
**+** continue;  
**+**}  
**+**if (enemyPos.y <= windowHeight \* -0.5f || enemyPos.y > windowHeight \* 0.5f) {  
**+** continue;  
**+**}  
  
 if (i->id == tileId\_SmallEnemy) {  
 // 空いている敵の弾を検索.  
 Actor\* bullet = findAvailableActor(  
 std::begin(enemyBulletList), std::end(enemyBulletList));

プログラムが書けたら、**ビルドして実行してください。**画面左に消えた敵が弾を撃ってこなければ成功です。

**[課題02]**  
自機と敵の弾の衝突判定を作成してください。

## Actor構造体にIDを追加する

敵の種類によって、異なる弾の撃ち方をさせたい場合、処理中のActorの種類が分からなければなりません。しかし、現在のActor構造体には種類を示す変数がないので、種類を見分けることができなくなっています。そこで、Actor構造体にIDを追加します。  
Actor.hに次のプログラムを追加してください。

/\*\*  
 \* ゲームキャラクター構造体.  
 \*/  
 struct Actor  
 {  
 Sprite spr; // 画像表示用スプライト.  
 Rect collisionShape; // 衝突判定の位置と大きさ.  
 int health; // 耐久力(0以下なら破壊されている).  
 float timer; // 多目的タイマー.  
**+** int id; // 識別用ID.  
 };

追加したメンバ変数には、敵を出現させるときに値を設定します。

generateObjectFromMap関数に、次のプログラムを追加してください。

enemy->spr.Animator(FrameAnimation::Animate::Create(tlEnemy));  
enemy->spr.Tweener(TweenAnimation::Animate::Create(data->createTween()));  
 enemy->collisionShape = data->collisionShape;  
 enemy->health = data->health;  
enemy->timer = 0;  
**+** enemy->id = data->id;  
 }  
 } else if (tileId == tileId\_PowerUpItem) {

## 敵の種類によって弾の撃ち方を変える

追加したIDメンバを使って、弾の撃ち方を変えてみましょう。  
updateEnemies関数を、次のように変更してください。

// 一定時間ごとに弾を発射.  
 const float shotInterval = 2.0f;  
 i->timer += deltaTime;  
 if (i->timer < shotInterval) {  
 continue;  
 }  
 i->timer -= shotInterval;  
**+  
+**// 大型の敵の攻撃. **+**if (i->id < tileId\_LargeEnemy) {  
**+** for (int n = 0; n < 2; ++n) { **+** Actor\* bullet = findAvailableActor(  
**+** std::begin(enemyBulletList), std::end(enemyBulletList));  
**+** if (bullet == nullptr) {  
**+** continue;  
**+** }  
**+** glm::vec3 pos = i->spr.Position();  
**+** pos.y += (float)(16 - n \* 32);  
**+** bullet->spr =  
**+** Sprite("Res/Objects.png", pos, Rect(464, 0, 16, 16));  
**+** namespace TA = TweenAnimation;  
**+** bullet->spr.Tweener(TA::Animate::Create(  
**+** TA::MoveBy::Create(4, glm::vec3(-1200, 0, 0))));  
**+** bullet->spr.Scale(glm::vec2(3, 1));  
**+** bullet->collisionShape = Rect(-12, -4, 24, 8);  
**+** bullet->health = 1;  
**+** }  
**+** continue;  
**+**}  
**+  
+**// それ以外の敵の攻撃.  
 // 空いている敵の弾を検索.  
 Actor\* bullet = findAvailableActor(  
std::begin(enemyBulletList), std::end(enemyBulletList));  
 if (bullet == nullptr) {  
 continue;  
 }

プログラムが書けたら、**ビルドして実行してください。**  
緑色の大きめの敵が、他の敵とは違う攻撃をしてきたら成功です。

**[課題03]**  
IDがtileId\_MediumEnemyの敵の攻撃を考えて、プログラムしてください。

**[課題04]**  
IDがtileId\_Bossの敵の攻撃を考えて、プログラムしてください。

# ステージクリア

## ステージクリア処理をプログラムする

現在のプログラムは、敵出現マップの左端からスタートし、右端に到達したらまた左端に戻すことで、無限に敵を出現させています。そういうゲームもありだと思いますが、今回はボスが死んだらステージクリアというルールに変更しようと思います。

まずはステージクリア判定に使う変数を追加しましょう。  
Main.cppに、次のプログラムを追加してください。

// ゲームの状態.  
 int gamestate; // 実行中の場面ID.  
**+**Actor\* boss; // ボスActorのポインタ.  
**+**bool isStagePassed; // ステージをクリアしていればtrue.  
  
 TitleScene titleScene;  
 MainScene mainScene;  
 GameOverScene gameOverScene;  
  
 // 敵の出現を制御するためのデータ.  
 TiledMap enemyMap;  
 float mapCurrentPosX;  
 float mapProcessedX;  
**+**bool isEndOfMap; // マップの終端に到達したらtrue.  
  
 // 自機のアニメーション.  
 const FrameAnimation::KeyFrame playerKeyFrames[] = {

作成した変数は、メイン画面用のinitialize関数で初期化します。  
メイン画面用のinitialize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;  
**+**isEndOfMap = false;  
**+**boss = nullptr;  
**+**isStagePassed = false;  
  
 // 音声を準備する.  
 Audio::EngineRef audio = Audio::Engine::Instance();  
 seBlast = audio.Prepare("Res/Audio/Blast.xwm");

## マップの終端を判定する

マップの端に到達したら、それ以上敵を出現させないようにします。  
generateObjectFromMap関数に、次のプログラムを追加してください。

void generateObjectFromMap(float deltaTime)  
 {  
**+** if (isEndOfMap) {  
**+** return;  
**+** }  
 const TiledMap::Layer& tiledMapLayer = enemyMap.GetLayer(0);  
 const glm::vec2 tileSize =  
 enemyMap.GetTileSet(tiledMapLayer.tilesetNo).size;  
 // 敵配置マップ参照位置の更新.  
 const float enemyMapScrollSpeed = 100; // 更新速度.  
 mapCurrentPosX += enemyMapScrollSpeed \* deltaTime;  
 if (mapCurrentPosX >= tiledMapLayer.size.x \* tileSize.x) {  
**-** // 終端を超えたら先頭にループ.  
**-** mapCurrentPosX = 0;  
**-** mapProcessedX = 0;  
**+** // 終端を超えたら停止.  
**+** isEndOfMap = true;  
**+** return;  
 }  
 // 次の列に到達したらデータを読む.  
 if (mapCurrentPosX - mapProcessedX >= tileSize.x) {  
 mapProcessedX += tileSize.x;

これまでは、マップの更新座標を示す変数mapCurrentPosXがマップの右端(マップのX方向のタイル数 \* タイルのX方向のサイズ)を超えた場合、更新座標mapCurrentPosXと、更新処理済み座標mapProcessedXを0にして、左端に戻していました。  
今回の変更では、右端を超えた場合はisEndOfMap変数をtrueにしてから、すぐに関数を終了しています。その結果、以降の関数呼び出しでは、関数の先頭にあるif文によって、関数はすぐに終了します(isEndOfMap変数がtrueなので)。そうすることで、右端に到達したら、それ以上敵を出現させないようにしているわけです。

## ボスのポインタを設定する

次に、「ボスのポインタ」を設定します。  
「ボスのポインタ」は、ボスが出現しているかどうか、出現したあとで死んでいるかどうかを判定するために使います。ステージクリアのルールは「ボスが死んだら」なので、ボスの状態を知るためにこのポインタが必要になるのです。

設定は簡単で、出現させようとした敵がボスだったときに、その敵のポインタをボスのポインタに代入するだけです。generateObjectFromMap関数に、次のプログラムを追加してください。

enemy->health = data->health;  
 enemy->timer = 0;  
 enemy->id = data->id;  
**+** // ボスのポインタを設定する. **+** if (data->id == tileId\_Boss && boss == nullptr) {  
**+** boss = enemy;  
**+** }  
 }  
 } else if (tileId == tileId\_PowerUpItem) {

気をつけるべき点として、「ボス」とみなされるのは、tileId\_Bossを持つタイルのうち、最初に出現したものだけだということです。もし2体以上のボスを出現させても、2体目以降はステージクリアには関係しません。

**[補足]**  
2体以上のボスと同時に戦い、全てを倒さないとステージクリアにならない、というようなステージを作る場合、「ボスのポインタ」を倒さなければならないボス敵の数だけ用意します。

## ステージクリアを判定する

それでは、ステージクリア判定をプログラムしましょう。  
update関数に、次のプログラムを追加してください。

} else if (gamestate == gamestateGameOver) {  
 update(window, &gameOverScene);  
 return;  
 }  
**+**  
**+** // ステージクリア判定.  
**+** if (!isStagePassed && boss && (boss->health <= 0)) {  
**+** isStagePassed = true;  
**+** boss = nullptr;  
**+** timer = 2;  
**+** }  
**+** if (isStagePassed) {  
**+** timer -= deltaTime;  
**+** if (timer <= 0) {  
**+** stopPlayerLaser();  
**+** initialize(&mainScene);  
**+** return;  
**+** }  
**+** }  
  
 // 自機の移動.  
 if (sprPlayer.health > 0) {

ステージクリアの条件は3つあります。

1. まだステージをクリアしていない  
2. ボスが登場している  
3. ボスのhealthが0以下

この3つの条件をすべて満たすとステージクリアとなり、isStagePassed変数にtrueを設定して、ステージをクリアした状態にしています。同時に、次のステージが始まるまでの待機時間として、タイマー変数に2秒を設定しています。このタイマー変数は、ゲームオーバー用のものを流用しています。ゲームオーバーとステージクリアが同時に起こることはないでしょうから、流用しても問題にはならないと思います。

追加した2つ目のif文では、次のステージが始まるまでのカウントダウンをしています。  
タイマーが0以下になったらメイン画面を初期化し直して、次のステージを始めます。

## ステージクリアの文字を表示する

ステージをクリアしたことが分かるように文字を表示します。  
メイン画面用のrender関数に、次のプログラムを追加してください。

// 文字列を表示する.  
 fontRenderer.BeginUpdate();  
 char str[9];  
 snprintf(str, sizeof(str), "%08d", score);  
 fontRenderer.AddString(glm::vec2(-100, 300), str);  
**+**if (isStagePassed) {  
**+** fontRenderer.AddString(glm::vec2(-100, 32), "STAGE CLEAR!");  
**+**}  
 fontRenderer.EndUpdate();  
 fontRenderer.Draw();

ステージクリアしているかどうかはisStagePassed変数を見ればわかりますので、この変数がtrueの場合のみ、「STAGE CLEAR!」という文字を表示させています。

**[課題05]**  
ステージクリア状態になったとき、自機と敵の衝突判定、自機と敵の弾の衝突判定を行わないようにしてください。

## 得点や自機のパワーアップ状態を引き継ぐ

新しいステージが始まると、前のステージで得たスコアやパワーアップ状態が初期化されてしまいます。  
このような仕様は、継続して遊ぶモチベーションを低下させてしまいます。  
そこで、ステージクリアしてもスコアやパワーアップ状態が引き継がれるようにします。

まず、MainScene構造体にステージ番号を示す変数を追加しましょう。  
MainScene.hに次のプログラムを追加してください。

/\*\*  
 \* メイン画面で使用する構造体.  
 \*/  
 struct MainScene  
 {  
**+** int stageNo; // 現在のステージ番号.  
 };  
 bool initialize(MainScene\*);

次に、このメンバ変数を初期化します。初期化のタイミングですが、タイトル画面でゲームが開始される瞬間がいいでしょう。

ということで、TitleScene.cppのupdate関数に、次のプログラムを追加してください。

// 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;  
**+** mainScene.stageNo = 1; // ステージ1から開始.  
 initialize(&mainScene);  
 }

そして、ステージクリアのタイミングでこの番号を増やします。  
メイン画面用のupdate関数に、次のプログラムを追加してください。

// ステージクリア判定.  
 if (!isStagePassed && boss && (boss->health <= 0)) {  
 isStagePassed = true;  
 boss = nullptr;  
 timer = 2;  
 }  
 if (isStagePassed) {  
 timer -= deltaTime;  
 if (timer <= 0) {  
 stopPlayerLaser();  
**+** ++mainScene.stageNo;  
 initialize(&mainScene);  
 return;  
 }  
 }

最後に、当初の目的であるスコアとパワーアップの引き継ぎをプログラムします。  
メイン画面用のinitialize関数を、次のように変更してください。

initializeActorList(std::begin(itemList), std::end(itemList));  
 initializeActorList(std::begin(enemyBulletList), std::end(enemyBulletList));  
  
 enemyGenerationTimer = 2;  
**-**score = 0;  
**-**weaponLevel = weaponLevelMin;  
**-**weaponType = weaponTypeWideShot;  
**+**if (scene->stageNo == 1) {  
**+** score = 0;  
**+** weaponLevel = weaponLevelMin;  
**+** weaponType = weaponTypeWideShot;  
**+**}  
 timer = 0;  
  
 // 敵配置マップを読み込む.  
 enemyMap.Load("Res/EnemyMap.json");

プログラムが書けたら、**ビルドして実行してください。**  
ステージをクリアしたときに、スコアとパワーアップ状態が引き継がれていたら成功です。

## ステージ番号を表示する

ついでなので、現在プレイ中のステージ番号を表示しておきましょう。  
メイン画面用のrender関数に、次のプログラムを追加してください。

// 文字列を表示する.  
 fontRenderer.BeginUpdate();  
**-**char str[9];  
**+**char str[64];  
 snprintf(str, sizeof(str), "%08d", score);  
 fontRenderer.AddString(glm::vec2(-100, 300), str);  
**+**  
**+**snprintf(str, sizeof(str), "STAGE: %02d", mainScene.stageNo);  
**+**fontRenderer.Scale(glm::vec2(0.5f, 0.5f));  
**+**fontRenderer.Color(glm::vec4(0.75f, 0.75f, 1, 1));  
**+**fontRenderer.Thickness(0);  
**+**fontRenderer.AddString(glm::vec2(-384, 300), str);  
**+**fontRenderer.Scale(glm::vec2(1, 1));  
**+**fontRenderer.Color(glm::vec4(1, 1, 1, 1));  
**+**fontRenderer.Thickness(0.33f);  
**+**  
 if (isStagePassed) {  
 fontRenderer.AddString(glm::vec2(-100, 32), "STAGE CLEAR!");  
 }

今回は、フォントの描画で使える機能をいくつか使ってみました。  
Scale(すけーる)メンバ関数は、フォントの大きさを変えます。初期状態はX,Yともに1です。  
Color(からー)メンバ関数は、フォントの色を変えます。フォントの色はglm::vec4型のパラメータで、赤、緑、青、不透明度の4つを0.0～1.0の範囲で設定するようになっています。初期状態は全て1です。  
Thickness(ちっくねす)メンバ関数は、フォントの太さを変えます。0が最も太く、1が最も細くなります。太さの初期状態は0.33です。  
これらの設定は引き継がれますので、フォントを書くたびに設定するか、設定を変更して書いたあとで、初期値に戻すかしなければなりません。上記のプログラムでは、ステージ番号を書いたあとで初期状態に戻しています。

**[課題06]**  
STAGE CLEAR!の文字に色を付け、大きさを変えてください。色と大きさは好きに決めて構いません。