[OpenGL 2D 2018 第08回]

敵機殺すべし。慈悲はない。

# 敵を表示する

## 敵の構造体

宇宙に存在するのは自分だけ、というのはちょっとさびしいです。そこで、戦うべき相手を用意しましょう。

プレイヤーと同様に敵もスプライトを使って表示することにします。また、どれだけの攻撃を耐えられるかを示す耐久値や、攻撃を受けたとみなす範囲も決める必要がありそうです。こういった複数のデータをまとめて扱えると便利です。そこで、構造体を使うことにしましょう。  
Main.cppを開き、ウィンドウの大きさを示す変数定義の下に、次の構造体を追加して下さい。

const char title[] = "OpenGL2D 2018"; // ウィンドウタイトル.  
 const int windowWidth = 800; // ウィンドウの幅.  
 const int windowHeight = 600; // ウィンドウの高さ. **+**  
**+**/\*\*  
**+**\* 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).  
**+**};  
  
 SpriteRenderer renderer; // スプライト描画用変数.  
 Sprite sprBackground; // 背景用スプライト.  
 Sprite sprPlayer; // 自機用スプライト.

Actor(あくたー)というのは「俳優」「出演者」という意味です。ゲーム中に登場するものはなんであれ「出演者の一種」だと考えられますので、この名前にしてみました。

Actor構造体には3つのメンバーがあります。  
spr(spriteの略)は画像を表示するためのスプライトです。collisionShape(こりじょん・しぇいぷ)は衝突判定用の長方形の大きさです(collision(こりじょん)は「衝突」という意味です)。そしてhealth(へるす)は耐久値です。healthが0以下の場合、その敵は破壊されていることにします。そして、1以上の場合は生存していることにします。

## 敵の配列

プレイヤーと違い、敵は何機も出現する可能性があります。ということは、配列を使うのが良さそうですね。自機の移動速度の変数定義の下に、次のプログラムを追加してください。

SpriteRenderer renderer; // スプライト描画用変数.  
 Sprite sprBackground; // 背景用スプライト.  
 Sprite sprPlayer; // 自機用スプライト.  
 glm::vec3 playerVelocity; // 自機の移動速度.  
**+  
+**Actor enemyList[128]; // 敵のリスト.  
  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);

敵の配列はenemyList(えねみー・りすと)という名前にしてみました。とりあえず配列の大きさは128としています。つまり、同時に128機まで敵を出現させることができることになります。もっとたくさんの敵を同時に出現させたいと思ったときは、この数値を大きくしてください。

## 配列の初期化

敵の配列を作りましたが、配列の要素である128機の全てを常に出現させたいわけではありません。そして、出現していない敵は表示したくありません。これを実現するためには、配列のどの要素が出現している敵なのかを調べる方法が必要です。  
ところでActor構造体を作成した時、「healthが0以下の場合は破壊されている」というルールを定めました。このルールを拡張して「破壊されている敵は出現していない」という扱いにします。ゲームが始まったとき、敵は1機も出現していないでしょう。この状態を作るには、ルールに従って配列のすべての要素のhealthを0にしておけばいいわけです。  
これは次のようなプログラムになります。  
スプライトに画像を設定しているプログラムの下に、以下のプログラムを追加してください。

// スプライトに画像を設定.  
 sprBackground = Sprite("Res/UnknownPlanet.png");  
 sprPlayer = Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));  
**+  
+**// 敵の配列を初期化.  
**+**for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** i->health = 0;  
**+**}  
  
 // ゲームループ.  
 while (!window.ShouldClose()) {

「Actor\*(あくたー・ぽいんた)」という書き方はActor構造体の位置を表すためのC/C++の記述方法です。この書き方を使うと、添字がなくても配列の要素を読み書きできます。std::begin(えすてぃーでぃー・びぎん)は、配列の先頭の位置を取得する関数です。std::end(えすてぃーでぃー・えんど)は、配列の終端の位置を取得する関数です。「Actor\*」は配列の要素そのものではないので、要素のメンバを参照するには「.(ドット)」ではなく「->(アロー)」を使わなければなりません。

**[補足]**  
このように、型の名前に「\*(あすたりすく)」を付けたものを「ポインタ」といいます。ポインタは「変数の位置を示す変数」です。プログラムでは添字の代わりに配列を操作するために使ったりします。

ところで、このプログラムはhealthしか初期化していません。なぜかというと、healthが0以下の敵は画面に出現していないので、sprとcollisionShapeは使われないからです。sprとcollisionShapeは敵を出現させるときに設定します。

## 敵の表示

次に敵を表示するプログラムを作ります。「表示」なのでrender関数に書きましょう。  
render関数に次のプログラムを追加してください。

renderer.BeginUpdate();  
 renderer.AddVertices(sprBackground);  
 renderer.AddVertices(sprPlayer);  
**+**for (const Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** if (i->health > 0) {  
**+** renderer.AddVertices(i->spr);  
**+** }  
**+**}  
 renderer.EndUpdate();  
 renderer.Draw(glm::vec2(windowWidth, windowHeight));

初期化のときと同様に「Actor\*」変数とstd::begin, std::endを使ってforループを作っています。  
「破壊されている敵は出現していない」というルールに従うと、表示する必要があるのは「healthが0より大きい敵」だけです。そこで、この条件をif文で表現しています。

## 乱数(ランダム)

敵というものは、背景や自機のようにただ画面に配置する、というわけにはいきません。  
大抵のシューティングゲームでは、自機が進むに従って、決められた位置に決められた順番で出現します。しかし、そういった出現データを作るには手間がかかります。そこで今回は、敵の出現位置を乱数で決めようと思います。

C++言語では、乱数を使うための関数や変数は「random」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。  
Main.cppの先頭に次のプログラムを追加してください。

#include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Sprite.h"  
**+**#include <random>  
  
 const char title[] = "OpenGL2D 2018"; // ウィンドウタイトル.  
 const int windowWidth = 800; // ウィンドウの幅.  
 const int windowHeight = 600; // ウィンドウの高さ.

次に、乱数を発生させる変数を追加します。  
ウィンドウの高さ変数の定義の下に、次のプログラムを追加してください。

const char title[] = "OpenGL2D 2018"; // ウィンドウタイトル.  
 const int windowWidth = 800; // ウィンドウの幅.  
 const int windowHeight = 600; // ウィンドウの高さ.  
**+**  
**+**std::mt19937 random; // 乱数を発生させる変数(乱数エンジン).  
  
 /\*\*  
 \* ゲームキャラクター構造体.  
 \*/

「乱数を発生させるための型、またはその型の変数」のことを「乱数エンジン」といいます。上記のプログラムでは、std::mt19937というのが「乱数を発生させるための型」の名前で、randomが「その型の変数」の名前です。

**[補足]**  
mt19937といのは「メルセンヌ・ツイスター」と呼ばれる乱数を発生させるためのルールを使った乱数エンジンです。  
そもそもコンピューターは計算機なので、サイコロのような真の乱数を作ることができません。そこで数学やコンピューターの研究者たちがさまざまな工夫をして、人間からみると乱数に見えるような計算式を考え出しました。このような計算式によって作り出される乱数を「疑似乱数(ぎじ・らんすう)」といいます。  
長年C言語で使われていた疑似乱数は、古いコンピューターでも簡単に計算できるにもかかわらず、それなりに乱数ぽい数値になる優れものでした。しかし、その簡単さゆえに真の乱数からは程遠く、例えば偶数と奇数が交互に出現するとか、乱数を「2の32乗-1」個作ると、また同じ順番の乱数が作られ始める(つまり長さが「2の32乗-1」しかない)という問題がありました。  
対してmt19937はその名の通り「2の19937乗-1」の長さを持ち、偶数と奇数が交互に出るようなこともありません。  
古い疑似乱数ほど簡単には計算できませんが、現代のコンピューターにとってはほとんど違いがないくらい高速に計算できます。そんなわけで、今のC++言語ではmt19937を使うのが定番となっています。

次に、乱数エンジンを初期化しましょう。  
スプライト描画用のrender変数を初期化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。

if (!renderer.Initialize(1024)) {  
 return 1;  
 }  
**+**  
**+**random.seed(std::random\_device()()); // 乱数エンジンの初期化.  
  
// スプライトに画像を設定.  
 sprBackground = Sprite("Res/UnknownPlanet.png");  
 sprPlayer = Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));

seed(しーど)というのが乱数エンジンを初期化する関数です。パラメーターに設定された値によって、異なる順番で乱数を作り出すようになります。  
パラメーターにはstd::random\_device型の変数の結果を設定しています(括弧が2つある点に注意してください)。

**[補足]**  
random\_deviceは真の乱数を発生します。コンピューター自身は真の乱数を計算できませんが、コンピューターに接続された機械が発するノイズなどを計測することで、外部から真の乱数を取得することは可能です。random\_deviceはそのような方法でほぼ真の乱数といえるものを作り出します。  
しかし、擬似乱数の計算と比べると、計測は非常に時間がかかります。そのため、疑似乱数の初期化にのみ使用することで、双方の利点を活用しているのです。

## 敵の出現間隔

今回は敵を一定間隔で出現させることにします。そのために、次の出現までも時間を測る変数が必要です。敵の配列の定義の下に、次のプログラムを追加してください。

SpriteRenderer renderer; // スプライト描画用変数.  
 Sprite sprBackground; // 背景用スプライト.  
 Sprite sprPlayer; // 自機用スプライト.  
  
 Actor enemyList[128]; // 敵のリスト.  
**+**float enemyGenerationTimer; // 次の敵が出現するまでの時間(単位:秒).  
  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);

次にこの変数を初期化します。敵配列を初期化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。

for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 i->health = 0;  
 }  
 **+**enemyGenerationTimer = 2;  
  
 // ゲームループ.  
 while (!window.ShouldClose()) {

このときに設定した値が最初の敵が出現するまでの時間になります。0秒だといきなりすぎるので2秒にしてみました。

## 敵の出現

ようやく敵を出現させられます。敵の出現は「ゲーム状態の更新」の一部だと考えられますので、update関数で行うことにしましょう。  
update関数に次のプログラムを追加してください。ちょっと長いですが、頑張って書いてくださいね。

sprPlayer.Position(newPos);  
 }  
 sprPlayer.Update(deltaTime);  
  
**+** // 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) {  
**+** // 空いている(破壊されている)敵構造体を検索. **+** Actor\* enemy = nullptr;  
**+** for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** if (i->health <= 0) {  
**+** enemy = i;  
**+** break;  
**+** }  
**+** }  
**+** // 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 std::uniform\_real\_distribution<float> y\_distribution(  
**+** -0.5f \* windowHeight, 0.5f \* windowHeight); **+** enemy->spr = Sprite("Res/Objects.png",  
**+** glm::vec3(0.5f \* windowWidth, y\_distribution(random), 0),  
**+** Rect(480, 0, 32, 32));  
**+** enemy->collisionShape = Rect(-16, -16, 32, 32);  
**+** enemy->health = 1;  
**+** // 次の敵が出現するまでの時間を設定する.  
**+** enemyGenerationTimer = 2;  
**+** }  
**+** }  
 }  
  
 /\*\*  
 \* ゲームの状態を描画する.

このプログラムは大きく3つの部分に分けられます。

最初は敵の出現判定で、enemyGenerationTimerの計算と一番外側のif文がそれに当たります。  
この部分はenemyGenerationTimerを経過時間だけへらし、0以下になったら敵を出現させるプログラムが実行されるように制御しています。

２つめの部分はfor文で、空いている敵構造体を検索しています。「空いている」というのは、ここでは「破壊されている」つまり「healthが0以下になっている」状態を指しています。for文の使い方は初期化や描画のときと同じくActor\*とstd::begin, std::endを使っています。  
このプログラムの特徴はfor文の上にある「Actor\* enemy = nullptr;」という文にあります。「Actor\*」はポインタといって「変数の位置を示す変数」です。でも、この変数に設定しているnullptr(ぬる・ぽいんた)はなんなんでしょう？　これは、ポインタが「どの変数も指していない」ことを示すもの(値)なんです。  
for文の中でhealthが0以下の構造体が見つかったら、その位置「i」をこのenemy変数に設定してbreakしています。もしhealthが0以下の構造体が見つからなかったらenemy変数の値はnullptrのままになります。

3つめの部分が敵を出現させるプログラムです。先のfor文では、空いている構造体が見つかったら、その位置をenemy変数に設定していました。見つからなかった場合は敵を出現させることができませんから、if文を使ってenemy変数がnullptrではない場合だけ出現させるようにしています。  
敵を出現させるには、画像を設定してhealthを1にします。  
画像はスプライトとして設定しますので、やりかたは背景や自機と同じです。ただし、出現する座標だけランダムにしたいです。これには先に初期化した乱数エンジンを使います。mt19937は0～4294967295の範囲の整数(これは32bitの整数の範囲です)を結果として返します。そこで、この数値を必要な範囲に変換してあげなければいけません。幸いというか当然というか、C++言語にはこの変換を行ってくれる型が用意されています。それが「std::uniform\_real\_distribution(えすてぃーでぃー・ゆにふぉーむ・りある・でぃすとりびゅーしょん)」です。

[補足]  
uniform = 一様(いちよう)な  
real = 実数(じっすう)、連続(れんぞく)  
distribution = 分布(ぶんぷ)  
→ uniform\_real\_distribution = 連続一様分布

この型は乱数エンジンの結果を指定した範囲に変換する機能を持っています。  
今回は画面の高さ全体のどこかの座標が選ばれるようにしてみました。

衝突判定用の長方形もここで設定しておきます。画像の大きさが32x32なので、衝突判定も同じ大きさにしています。なお、この長方形はスプライトの中心を原点(0,0)とします。

最後に、次の敵の出現までの時間を再設定しています。

## 敵の更新

出現している敵は、継続的に状態を更新しなければなりません。  
敵出現プログラムの下に、次のプログラムを追加してください。

// 次の敵が出現するまでの時間を設定する.  
 enemyGenerationTimer = 2;  
 }  
 }  
**+** // 敵の更新.  
**+** for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** if (i->health > 0) {  
**+** i->spr.Update(deltaTime);  
**+** }  
**+** }  
 }

更新はスプライト専用のUpdate関数を実行するだけです。

これで敵の表示は完成です。ビルドして実行してください。  
画面の右側に敵が表示されたら成功です。

**[課題01]**  
乱数エンジンを使って、次の敵の出現までの時間がランダムになるようにしてください。  
範囲は0.2秒から2秒とします。

# 敵を移動させる

## トウィーニング

画面の右側に敵が出現していますが、それだけです。もしプレイヤーが弾を撃てるようになったら、なすすべもなくやられてしまうでしょう。せっかく出現したのにそれではあまりにもかわいそうです。  
そこで、敵を移動させてあげましょう。  
敵を発生させるプログラムに、次のプログラムを追加してください。

if (enemy != nullptr) {  
 const std::uniform\_real\_distribution<float> y\_distribution(  
 -0.5f \* windowHeight, 0.5f \* windowHeight);  
 enemy->spr = Sprite("Res/Objects.png",  
 glm::vec3(0.5f \* windowWidth, y\_distribution(random), 0),  
 Rect(480, 0, 32, 32));  
**+** enemy->spr.Tweener(TweenAnimation::Animate::Create(  
**+** TweenAnimation::MoveBy::Create(  
**+** 5.0f, glm::vec3(-1000, 0, 0), TweenAnimation::EasingType::Linear)));  
 enemy->collisionShape = Rect(-16, -16, 32, 32);  
 enemy->health = 1;  
// 次の敵が出現するまでの時間を設定する.  
 enemyGenerationTimer = 2;  
 }

自機とは異なり、敵はキー操作などで動作を変えることはありませんから、決まった動きをさせられれば十分です。スプライトにはそのための機能があるのでそれを使います。その機能は「トウィーニング」といいます。

スプライトにトウィーニングを設定するにはTweener(とうぃーなー)というスプライト専用の関数を使います。Tweenerのパラメーターはトウィーニングを制御する変数です。この変数を作成するにはTweenAnimation::Animate::Create(とうぃーんあにめーしょん・あにめいと・くりえいと)関数を使います。この関数のパラメーターは「どのような動きをさせるか」を決めるための変数です。動かし方によっていくつかのバリエーションがあるのですが、今回は「指定された距離を移動する」機能を持つMoveBy(むーぶ・ばい)という型を使います。

MoveBy型の変数を作成するにはTweenAnimation::MoveBy::Create(とうぃーんあにめーしょん・むーぶ・ばい・くりえいと)という関数を使います。この関数のパラメーターには「移動にかかる時間」、「移動する距離」、「補間方法」の3つを設定します。今回は「5秒かけて」、「距離(-1000, 0, 0)まで」、「Linear(りにあ)補完で移動」するようにしてみました。

書けたらビルドして実行してください。  
敵が右に出現し、左に向かって移動したら成功です。

## 移動を終えた敵を破壊する

5秒かけて(-1000, 0, 0)の距離を移動した敵は停止します。画面の横幅は800ドットなので、敵は画面外に出ていることでしょう。しかし、自動的にいなくなったりはしないので、画面に映らない位置でじっとしていることになります。そのままでは128個の配列を使い切って、二度と敵が出現しなくなってしまいます。

そこで、移動を終えた敵には消えてもらうことにしましょう。  
敵の状態を更新するプログラムに、次のプログラムを追加してください。

// 敵の更新.  
 for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 if (i->health > 0) {  
 i->spr.Update(deltaTime);  
**+** if (i->spr.Tweener()->IsFinished()) {  
**+** i->health = 0;  
**+** }  
 }  
 }

トウィーニングを制御する型には、「移動が完了したかどうか」を調べるIsFinished(いず・ふぃにっしゅど)関数があります。これを使えば移動が終わったかどうかが分かります。また、Tweener関数をパラメーターなしで実行すると、トウィーニング制御変数のポインタを返すようになっていますから、このポインタ経由でIsFinished関数を実行しています。ポインタ変数から要素のメンバを参照するには「->(あろー)」を使うのでしたね。

IsFinished関数は、移動が完了していればtrue、まだ移動中ならfalseを返しますので、trueだった場合だけhealthを0にすることで敵を破壊しています。

プログラムが書けたら、ビルドして実行してください。  
見た目には変化がありませんが、移動を終えた敵は破壊されているはずです。

**[課題02]**  
敵の移動距離を(-500, 0, 0)に短縮して、移動を終えた敵が消えることを確認してください。  
確認できたら移動距離を(-1000, 0, 0)に戻しておいてください。

# 自機から弾を発射する

## 弾の配列

今度は自機から弾を発射できるようにしましょう。  
敵と同様に、弾も何発も発射されるものです。ですから配列を使うのがいいでしょう。  
敵の配列の下に、次のプログラムを追加してください。

SpriteRenderer renderer; // スプライト描画用変数.  
 Sprite sprBackground; // 背景用スプライト.  
 Sprite sprPlayer; // 自機用スプライト.  
 glm::vec3 playerVelocity; // 自機の移動速度.  
Actor enemyList[128]; // 敵のリスト.  
**+**Actor playerBulletList[128]; // 自機の弾のリスト.  
 float enemyGenerationTimer; // 次の敵が出現するまでの時間(単位:秒).  
  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);

自機の弾の構造体は敵と同じものを使いまわします。

## 弾の配列の初期化

続いて配列を初期化します。  
敵の配列を初期化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。

// 敵の配列を初期化.  
for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 i->health = 0;  
} **+**// 自機の弾の配列を初期化.  
**+**for (Actor\* i = std::begin(playerBulletList);  
**+** i != std::end(playerBulletList); ++i) {  
**+** i->health = 0;  
**+**}  
 enemyGenerationTimer = 2;  
  
 // ゲームループ.  
 while (!window.ShouldClose()) {

紙面の都合でfor文の途中に改行を入れていますが、実際に書き写す時には改行はしなくて結構です。

## 弾の表示

弾の表示も敵の場合と同じです。  
敵を表示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。

for (const Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 if (i->health > 0) {  
 renderer.AddVertices(i->spr);  
 }  
}  
**+**for (const Actor\* i = std::begin(playerBulletList);  
**+** i != std::end(playerBulletList); ++i) {  
**+** if (i->health > 0) {  
**+** renderer.AddVertices(i->spr);  
**+** }  
**+**}  
 renderer.EndUpdate();  
 renderer.Draw(glm::vec2(windowWidth, windowHeight));

## 弾を発射する

弾の発射はprocessInput関数で行います。なぜなら、弾が発射されるのは「プレイヤーが発射ボタンを押した時」だからです。  
processInput関数に、次のプログラムを追加してください。

if (playerVelocity.x || playerVelocity.y) {  
 playerVelocity = glm::normalize(playerVelocity) \* 400.0f;  
 }  
  
**+** // 弾の発射.  
**+** if (gamepad.buttonDown & GamePad::A) {  
**+** // 空いている弾の構造体を検索. **+** Actor\* bullet = nullptr;  
**+** for (Actor\* i = std::begin(playerBulletList);  
**+** i != std::end(playerBulletList); ++i) {  
**+** if (i->health <= 0) {  
**+** bullet = i;  
**+** break;  
**+** }  
**+** }  
**+** // 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;  
**+** }  
**+** }  
 }  
  
 /\*\*  
 \* ゲームの状態を更新する.

このプログラムの基本的な構造は、敵を出現させるプログラムと同じです(見比べてみてください)。  
配列から空いている構造体を検索し、見つかったら弾の画像、移動方法、耐久値を設定しています。

## 弾の更新

発射された弾は、継続的に状態を更新しなければなりません。  
update関数にある敵の状態を更新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。

// 敵の更新.  
 for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 if (i->health > 0) {  
 i->spr.Update(deltaTime);  
 if (i->spr.Tweener()->IsFinished()) {  
 i->health = 0;  
 }  
 }  
 }  
**+** // 自機の弾の更新.  
**+** for (Actor\* i = std::begin(playerBulletList);  
**+** i != std::end(playerBulletList); ++i) {  
**+** if (i->health > 0) {  
**+** i->spr.Update(deltaTime);  
**+** if (i->spr.Tweener()->IsFinished()) {  
**+** i->health = 0;  
**+** }  
**+** }  
**+** }  
 }

自機の弾の更新は、敵の更新とほぼ同じです。

プログラムが書けたら、ビルドして実行してください。  
Aキーを押して自機から弾が発射されたら成功です。

# 自機の弾と敵の衝突判定

## 衝突検出関数

敵を表示し、自機の弾も発射できるようになりました。しかし今のところ、両者はぶつかることなくすり抜けてしまいます。それは衝突判定を行っていないからです。  
まずは衝突判定用の長方形同士がぶつかっているかどうかを判定する関数を書きます。自機の弾と敵は複数存在するので衝突判定はfor文を使って書くことになりますが、その中に衝突判定を埋め込んでしまうとプログラムが見づらくなるからです。

まずはプロトタイプ宣言を行います。  
render関数の宣言付近に、次のプログラムを追加してください。

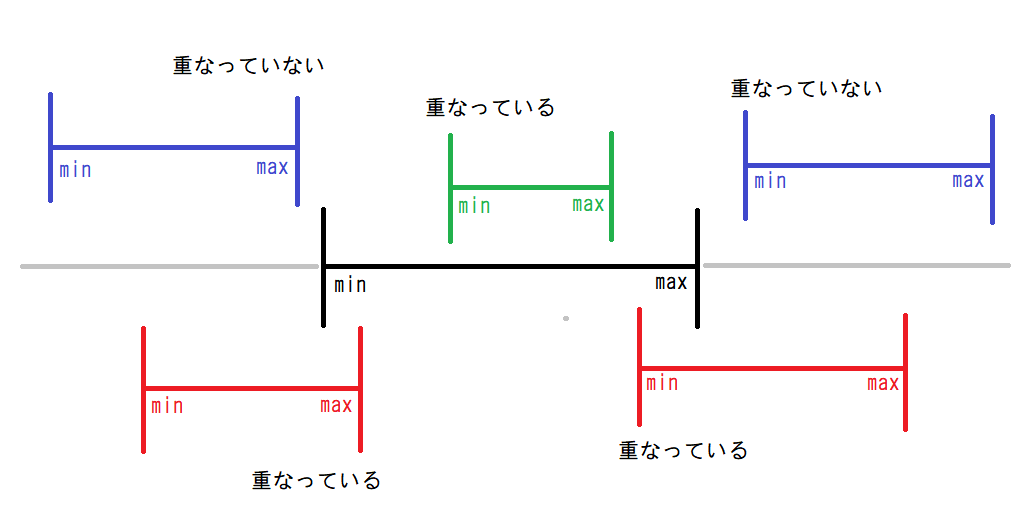
**+**/\*  
**+**\* プロトタイプ宣言.  
**+**\*/  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);  
**+**bool detectCollision(const Rect\*, const Rect\*);

今回作成する関数の名前はdetectCollision(でぃてくと・こりじょん)としました(detectは「検出する・見抜く」という意味の動詞、collisionは「衝突・不一致」という意味の名詞です)。  
ついでに、プログラムを読みやすくするためのコメントも追加してみました。

続いて関数を定義します。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。

window.swapBuffers();  
 } **+**/\*\*  
**+**\* 2つの長方形の衝突状態を調べる.  
**+**\*  
**+**\* @param lhs 長方形その1.  
**+**\* @param rhs 長方形その2.  
**+**\*  
**+**\* @retval true 衝突している.  
**+**\* @retval false 衝突していない.  
**+**\*/  
**+**bool detectCollision(const Rect\* lhs, const Rect\* rhs)  
**+**{  
**+** return  
**+** lhs->origin.x < rhs->origin.x + rhs->size.x &&  
**+** lhs->origin.x + lhs->size.x > rhs->origin.x &&  
**+** lhs->origin.y < rhs->origin.y + rhs->size.y &&  
**+** lhs->origin.y + lhs->size.y > rhs->origin.y;  
**+**}

このプログラムがなにをしているのかは、次の図で解説します。



この図は上記のプログラムがどのように衝突を判定しているかをで表現したものです。簡単にするためにx軸についてのみ説明します。中央の黒い横線をlhs(長方形その1)としたとき、周囲の横線はそれぞれがrhs(長方形その2)の状態を示しています。minは長方形の左側の辺(origin.x)、maxは右側の辺(origin.x + size.x)の座標だと考えてください。

lhsの右側の辺(黒い線のmax)がrhsの左側の辺(右の青い線のmin)より小さい、つまり左側にある場合は重なる部分はありません。同様に、lhsのminがrhsのmaxより大きい場合も重なりません。  
lhsのmaxがrhsのmin以上かつlhsのminがrhsのmax以下の場合は、どこかが重なっています。

これをx軸とy軸の両方に行っているのが上記のコードです。どちらかひとつでも重なっていない軸があれば衝突していません。両方の軸において重なっている場合のみ衝突していると判定されます。

## 衝突判定

それでは、detectCollision関数を使って衝突判定を書いていきましょう。  
update関数の中にある、自機の弾を更新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。

// 自機の弾の更新.  
 for (Actor\* i = std::begin(playerBulletList);  
 i != std::end(playerBulletList); ++i) {  
 if (i->health > 0) {  
 i->spr.Update(deltaTime);  
 if (i->spr.Tweener()->IsFinished()) {  
 i->health = 0;  
 }  
 }  
 }  
**+**  
**+** // 自機の弾と敵の衝突判定.  
**+** for (Actor\* bullet = std::begin(playerBulletList);  
**+** bullet != std::end(playerBulletList); ++bullet) {  
**+** if (bullet->health <= 0) {  
**+** continue;  
**+** }  
**+** Rect shotRect = bullet->collisionShape;  
**+** shotRect.origin += glm::vec2(bullet->spr.Position());  
**+** for (Actor\* enemy = std::begin(enemyList);  
**+** enemy != std::end(enemyList); ++enemy) {  
**+** if (enemy->health <= 0) {  
**+** continue;  
**+** }  
**+** Rect enemyRect = enemy->collisionShape;  
**+** enemyRect.origin += glm::vec2(enemy->spr.Position());  
**+** if (detectCollision(&shotRect, &enemyRect)) {  
**+** bullet->health -= 1;  
**+** enemy->health -= 1;  
**+** break;  
**+** }  
**+** }  
**+** }  
 }

衝突判定プログラムは、自機の弾のための外側のfor文と、敵のための内側のfor文による二重ループになっています。また、自機の弾も敵もhealthが0以下の場合は、continue文を使って処理をスキップするようにしてあります。すでに破壊されているものに衝突することはない、というわけです。  
どちらのhealthも1以上だった場合、detectCollision関数で弾の衝突判定用長方形と敵の衝突判定用長方形が衝突しているかを調べます。そして、もし衝突していれば、弾と敵の両方のhealthを1減らすようにします。現在のプログラムでは、自機の弾(あるいは敵)を作成したとき、healthに1を設定していますから、この処理によってどちらも一撃で破壊されるはずです。

ここまで書けたら、ビルドして実行してください。  
Aキーで発射した弾が敵に当たった時、弾も敵も消えていたら成功です。

**[課題03]**  
update関数で敵を出現させる時、敵のhealthに2以上の値を設定し、弾を当てた時にどうなるか確認してください。

**[課題04]**  
自機の弾を発射する時、自機の弾のhealthに2以上の値を設定してください。そして、弾が的に当たった時にどうなるかを確認してください。

**[課題05]**  
弾と敵が衝突したとき、弾と敵のhealthが相殺されるようにしてください(敵のhealthから弾のhealthを引き、弾のhealthから敵のhealthを引く)。  
ヒント: 少なくとも1個のローカル変数を定義する必要があるでしょう。