[OpenGL 2D 2018 第13回]

ぼくのかんがえたさいきょうの自機

# 自機を強化する

## ワイドショット

現在、プレイヤー機体に装備されている武装は、Aキーを押すたびに1発だけ発射される弾だけです。それに対し、敵は雲霞の勢いで襲ってきています。早急に武装を強化しなければ、たった一機でこれに対抗することはできないでしょう。

さて、どんなふうに強化しましょうか。高性能な誘導ミサイル、破壊力に優れた爆弾、敵を貫通するレーザー兵器など、強力な武装はいくらでも思いつきます。しかし、前線で戦う兵士が真に求めているものは、どんな状況でも対応できる汎用的で扱いやすい武器でしょう。

シューティングゲームの世界において、そんな即応型の武器は「ワイドショット(Wide shot)」などと呼ばれています。

## 複数の弾を同時に発射する

ワイドショットは、弾を複数の方向に発射するだけの装備です。弾を発射するプログラムは既にみなさんの手元にあるわけですから、そこから改造していけばよさそうです。必要な改造は、複数の弾を同時に発射することと、それぞれの弾の発射方向を変えることの2つです。

まずは複数の弾を発射するようにしてみましょう。  
同じことを何度も繰り返させたいときは「for文」を使うのでした。ということで、for文を使って弾を5発動時に発射するように改造します。processInput関数にある、Aキーを押したら弾を発射するプログラムを、次のように変更してください。

// 弾の発射.  
 if (gamepad.buttonDown & GamePad::A) {  
**+** for (int i = 0; i < 5; ++i) {  
 Actor\* bullet = findAvailableActor(  
 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.spr.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.  
 }  
**+** }  
**+** sePlayerShot->Play(); // 弾の発射音を再生.  
 }

なお、Audioシステムでは、同じ変数を同時に何度も再生することはできないようになっているので、発射音の再生はfor文の外に移動させました。

## 弾の発射方向を変える

for文を使うことで、同時に複数の弾が発射されるようにできました。しかし、全ての弾が同じ位置から、同じ方向へ、同じ速度で発射されています。これでは1発にしか見えません。どこも「ワイド」ではありません。

そこで、1発ごとに方向を変えることにしましょう。  
弾を発射するプログラムを、次のように変更してください。

// 弾の発射.  
 if (gamepad.buttonDown & GamePad::A) {  
 for (int i = 0; i < 5; ++i) {  
 Actor\* bullet = findAvailableActor(  
 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.spr.Position(), Rect(64, 0, 32, 16));  
**+** const float y[] = { 0, 100, -100, 200, -200 };  
 bullet->spr.Tweener(TweenAnimation::Animate::Create(  
**-** TweenAnimation::MoveBy::Create(1, glm::vec3(1200, 0, 0),  
**+** TweenAnimation::MoveBy::Create(1, glm::vec3(1200, y[i], 0),  
 TweenAnimation::EasingType::Linear)));  
 bullet->collisionShape = Rect(-8, -4, 16, 8);  
 bullet->health = 1;  
 }  
 }  
 sePlayerShot->Play(); // 弾の発射音を再生.  
 }

このプログラムでは、1発ごとにY方向の移動速度を変えることで、さまざまな方向に弾を発射しています。

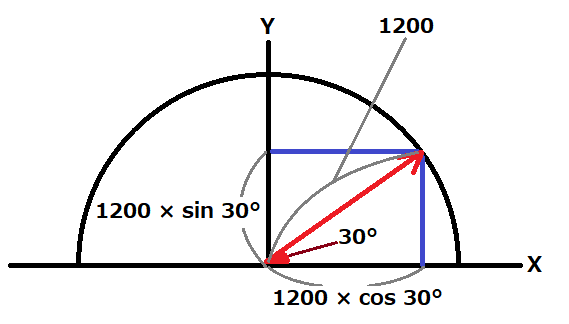
**プログラムが書けたら、ビルドして実行してください。**  
Aキーを押したとき、弾が同時に5方向へと発射されたら成功です。

## 三角関数

弾が広範囲(ワイド)に飛ぶようにはなりましたが、なんだか少し違和感がありませんか？  
その理由は、全ての弾が縦一列に並んで飛んでいるからです。現実の世界ならば、同じ地点から同じ速度で発射された物体は、どれも同じ距離だけ進みます。そのため、弾の分布は扇状(おうぎじょう)になるはずです。

縦一列になる原因は、Y方向の速度だけを変更していることです。つまり、Yの速度だけ速く飛んでしまっているのです。弾の速度を一定にするには、XとYの速度をかけ合わせた値が常に同じでなければなりません。ここで登場するのが三角関数です。

例えば速度1200の弾を30°の角度で発射する場合、XとYの速度はsin(サイン)とcos(コサイン)を使って次の図のように表現できます。



つまり、弾の「速度」と飛んでいく「角度(方向)」が与えられたとき、その弾のXとYの速度は

Xの速度 = 弾の速度 \* cos(角度)  
Yの速度 = 弾の速度 \* sin(角度)

というふうに表せます。

sinとcosは、C/C++言語ではstd::sin、std::cosという関数で表現されています。また、C/C++言語の三角関数は、角度を度数法ではなく弧度法(ラジアン)で渡さなければなりません。  
弧度法では、度数法の180°を1πとして表現します。度数法から弧度法へ変換するには次の式を使います。

弧度法の角度 = 度数法の角度 ÷ 180 × π

それでは、これらの関数を使って弾の方向を設定してみましょう。

// 弾の発射.  
 if (gamepad.buttonDown & GamePad::A) {  
 for (int i = 0; i < 5; ++i) {  
 Actor\* bullet = findAvailableActor(  
 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.spr.Position(), Rect(64, 0, 32, 16));  
**-** const float y[] = { 0, 100, -100, 200, -200 };  
**+** const float angles[] = { 0, 7.5f, -7.5f, 15.0f, -15.0f };  
**+** const float radian = angles[i] / 180.0f \* 3.14f;  
**+** const float c = std::cos(radian);  
**+** const float s = std::sin(radian);  
 bullet->spr.Tweener(TweenAnimation::Animate::Create(  
**-** TweenAnimation::MoveBy::Create(1, glm::vec3(1200, y[i], 0),  
**+** TweenAnimation::MoveBy::Create(1, glm::vec3(1200 \* c, 1200 \* s, 0),  
 TweenAnimation::EasingType::Linear)));  
 bullet->collisionShape = Rect(-8, -4, 16, 8);  
 bullet->health = 1;  
 }  
 }  
 sePlayerShot->Play(); // 弾の発射音を再生.  
 }

**プログラムが書けたら、ビルドして実行してください。**  
分かりにくいですが、弾が扇状に発射されていたら成功です。

**[課題01]**  
同時に発射する弾の数を7発に増やしてください。angles配列に注意!

## 画像の角度

弾の発射角度に関わらず画像の中の弾は真横を向いたままです。  
画像を回転させて、方向と見ためを一致させると説得力が増します。スプライトの画像を回転させるには、Rotationメンバ関数を使います。  
上記のプログラムに、次のプログラムを追加してください。

const float angles[] = { 0, 7.5f, -7.5f, 15.0f, -15.0f };  
const float radian = angles[i] / 180.0f \* 3.14f;  
const float c = std::cos(radian);  
const float s = std::sin(radian);  
 bullet->spr.Tweener(TweenAnimation::Animate::Create(  
 TweenAnimation::MoveBy::Create(1, glm::vec3(1200 \* c, 1200 \* s, 0),  
 TweenAnimation::EasingType::Linear)));  
**+**bullet->spr.Rotation(radian);  
 bullet->collisionShape = Rect(-8, -4, 16, 8);  
 bullet->health = 1;

Rotationメンバ関数は、弧度法(ラジアン)で指定された角度だけ画像を回転させます。

**プログラムが書けたら、ビルドして実行してください。**弾の画像がまっすぐに飛ぶ方向を向いていれば成功です。

**[重要]**  
Rotationメンバ関数によって回転するのは画像だけです。衝突判定は回転しません。

## パワーアップ

ワイドショットは非常に強力です。しかし強力すぎて、ゲームを遊ぶ人に達成感を感じてもらうことが難しいという側面もあります。そこで、条件を満たすことで徐々にパワーアップしていくようにして、自分の力で強くなったと感じてもらえるようにしてみましょう。

条件はなんでもいいのですが、とりあえず得点によってパワーアップすることにしましょうか。  
武器にパワーアップのレベルを持たせて、レベルが1上がるごとに1つずつ発射される弾が増えていくようにします。

そうなると、武器のレベルを示す変数が必要ですね。  
スコアや音声用の変数を定義しているあたりに、次のプログラムを追加してください。

　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.  
  
　// 音声制御用変数.  
　Audio::SoundPtr bgm;  
　Audio::SoundPtr sePlayerShot;  
　Audio::SoundPtr seBlast;

次に、weaponLevel変数を初期化します。  
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。

score = 0;  
 timer = 0;  
**+**weaponLevel = weaponLevelMin;  
  
 // 敵配置マップを読み込む.  
 enemyMap.Load("Res/EnemyMap.json");  
 mapCurrentPosX = windowWidth;  
 mapProcessedX = windowWidth;

弾の発射数はfor文のループ回数で決められています。これまでは5や7といった定数を使っていましたが、これをweaponLevelに置き換えます。  
弾を発射するプログラムを次のように変更してください。

// 弾の発射.  
 if (gamepad.buttonDown & GamePad::A) {  
**-** for (int i = 0; i < 5; ++i) {  
**+** for (int i = 0; i < weaponLevel; ++i) {  
 Actor\* bullet = findAvailableActor(  
 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) {

最後に、得点によって武器のレベルを更新します。とりあえず、2000点ごとに1段階ずつパワーアップさせてみましょう。  
メイン画面用のupdateに次のプログラムを追加してください。

sprPlayer.spr.Position(newPos);  
 }  
 sprPlayer.spr.Update(deltaTime);  
 }  
**+**  
**+** // 得点に応じて自機の武器を強化する.  
**+** weaponLevel = weaponLevelMin + score / 2000;  
**+** if (weaponLevel > weaponLevelMax) {  
**+** weaponLevel = weaponLevelMax;  
**+** }  
  
 // 敵の出現.  
 #if 1  
 const TiledMap::Layer& tiledMapLayer = enemyMap.GetLayer(0);

**プログラムが書けたら、ビルドして実行してください。**  
敵を倒していくごとに同時に発射される弾が増えていけば成功です。

# アイテムを出す

## アイテム用Actor配列を用意する

パワーアップの手段には、得点以外にも考えられます。  
よく使われるのは「アイテムを取得したらパワーアップする」という方法です。  
今度はこれを作ってみましょう。

さて、とにかく最初はアイテムを出現させることが必要です。  
そういうわけで、アイテム用のActor配列を用意します。

エフェクト用のActor配列の下に、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 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.  
 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.

**[課題02]**  
itemList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。

## マップデータを見てアイテムを出現させる

アイテムは既にマップデータに配置してあります。もし自分で敵配置マップを作成している場合は、そのマップに黄色の球にPと書かれたアイテムを配置してください。このアイテムのタイルIDは230です。  
これを利用して、敵と同様にアイテムを出現させてみましょう。

まずはアイテムのタイルIDを定義します。  
敵を出現させるプログラムを、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.  
**-** if (tiledMapLayer.At(mapY, mapX) == enemyId) {  
**+** 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) {  
 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) {

なお、タイルIDを比較するたびにAt関数を使うとプログラムが読みにくくなるので、一度だけ実行して結果をtileIdという変数に格納しておくことにしました。

次に、タイルIDがアイテムのIDだった場合にアイテムを出現させるプログラムを作ります。  
敵を出現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。

enemy->spr.Tweener(TA::Animate::Create(par));  
 enemy->collisionShape = Rect(-16, -16, 32, 32);  
 enemy->health = 2;  
 }  
**+**} else if (tileId == powerUpItemId) {  
**+** // アイテムを出現させる. **+** Actor\* item = findAvailableActor(std::begin(itemList), std::end(itemList));  
**+** if (item != nullptr) {  
**+** const float y =  
**+** windowHeight \* 0.5f - (float)(mapY \* tileSize.x);  
**+** item->spr = Sprite("Res/Objects.png", glm::vec3(0.5f \* windowWidth, y, 0),  
**+** Rect(160, 32, 32, 32));  
**+** 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;  
**+** }  
 }

**プログラムが書けたら、ビルドして実行してください。**ときどき「P」と書かれた画像が流れてきたら成功です。

## アイテムを取得する

無事アイテムを出現させることができたなら、次はそのアイテムを取得できるようにしましょう。  
やるべきことは、自機とアイテムの衝突判定を作ることです。

最初に、自機とアイテムの衝突を扱う関数を追加します。  
関数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。

/\*  
 \* 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\*);

そして、関数の定義を作成します。  
自機と敵の衝突を処理する関数の定義の下に、次のプログラムを追加してください。

blast->spr.Scale(glm::vec2(2, 2)); // 自機の爆発は少し大きめにする.  
 blast->health = 1;  
 seBlast->Play();// 爆発音を再生.  
 }  
 }  
 }  
  
**+**/\*\*  
**+**\* 自機とアイテムの衝突を処理する.  
**+**\*  
**+**\* @param player 自機のポインタ.  
**+**\* @param item アイテムのポインタ.  
**+**\*/  
**+**void playerAndItemContactHandler(Actor\* player, Actor\* item)  
**+**{  
**+** item->health = 0;  
**+**  
**+** // 自機の武器を強化する.  
**+** ++weaponLevel;  
**+** if (weaponLevel > weaponLevelMax) {  
**+** weaponLevel = weaponLevelMax;  
**+** }  
**+**}

次に、衝突判定を行う関数呼び出しを追加します。  
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。

// 自機と敵の衝突判定.  
 detectCollision(  
 &sprPlayer, &sprPlayer + 1,  
 std::begin(enemyList), std::end(enemyList),  
 playerAndEnemyContactHandler);  
**+** // 自機とアイテムの衝突判定.  
**+** detectCollision(  
**+** &sprPlayer, &sprPlayer + 1,  
**+** std::begin(itemList), std::end(itemList),  
**+** playerAndItemContactHandler);  
 }  
  
 /\*\*  
 \* ゲームの状態を描画する.  
 \*

最後に、得点によるパワーアップを削除します(もし完全に消してしまうのが忍びない場合は、行の先頭に「//」を書いてコメント化するといいでしょう)。

sprPlayer.spr.Position(newPos);  
 }  
 sprPlayer.spr.Update(deltaTime);  
 }  
**-**  
**-** // 得点に応じて自機の武器を強化する.  
**-** weaponLevel = weaponLevelMin + score / 2000;  
**-** if (weaponLevel > weaponLevelMax) {  
**-** weaponLevel = weaponLevelMax;  
**-** }  
  
 // 敵の出現.  
 #if 1  
 const TiledMap::Layer& tiledMapLayer = enemyMap.GetLayer(0);

**プログラムが書けたら、ビルドして実行してください。**アイテムを取得するごとに武器のレベルが上がっていたら成功です。

## アイテムを取得したときに音を鳴らす

アイテムを取得したときに効果音を鳴らすと、「取得した！」あるいは「パワーアップした！」という事実をより効果的にプレイヤーに知らせることができます。  
まずは音声制御用変数を追加しましょう。  
音声制御用変数を定義している部分に、次のプログラムを追加してください。

// 音声制御用変数.  
 Audio::SoundPtr bgm;  
 Audio::SoundPtr sePlayerShot;  
 Audio::SoundPtr seBlast;  
**+**Audio::SoundPtr sePowerUp;

次に、音声を準備します。  
音声を準備している部分に、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");  
**+**sePowerUp = audio.Prepare("Res/Audio/GetItem.xwm");  
bgm = audio.Prepare("Res/Audio/Neolith.xwm");

最後に、アイテムを取得したときに音声を鳴らします。  
自機とアイテムの衝突を処理する関数に、次のプログラムを追加してください。

/\*\*  
\* 自機とアイテムの衝突を処理する.  
\*  
\* @param player 自機のポインタ.  
\* @param item アイテムのポインタ.  
\*/  
void playerAndItemContactHandler(Actor\* player, Actor\* item)  
{  
 item->health = 0;  
**+** sePowerUp->Play();  
  
 // 自機の武器を強化する.  
 ++weaponLevel;  
 if (weaponLevel > weaponLevelMax) {  
 weaponLevel = weaponLevelMax;  
 }  
}

**プログラムが書けたら、ビルドして実行してください。**アイテムを取得したときに音声が再生されていたら成功です。

# 更に自機を強化する

## 武器を切り替える

ワイドショットを手に入れたことで、敵との戦闘はずいぶんと楽になりました。しかし、敵も手をこまぬいているだけではないでしょう。近い将来、より強力な兵器をたずさえて反撃してくるはずです。そのような兵器を前にしたとき、ワイドショットだけでは心もとないと感じることがあるかもしれません。そこで、自機にはもうひとつの武器、あらゆる敵をなぎ倒すレーザー兵器を装備しておこうと思います。

最初に、ワイドショットとレーザーを切り替えられるようにしましょう。  
まずは選択中の武器を示す変数を作ります。この変数に格納されている値を見て、ボタンが押されたときに発射する武器を決めます。武器のレベルを示す変数の定義の下に、次のプログラムを追加してください。

const 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.  
**+**  
**+**const int weaponTypeWideShot = 0; // 広範囲ショット.  
**+**const int weaponTypeLaser = 1; // レーザー.  
**+**int weaponType; // 選択中の武器.  
  
 // 音声制御用変数.

次に、作成した変数を初期化しましょう。  
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。

score = 0;  
 timer = 0;  
 weaponLevel = weaponLevelMin;  
**+**weaponType = weaponTypeWideShot;  
  
 // 敵配置マップを読み込む.  
 enemyMap.Load("Res/EnemyMap.json");  
 mapCurrentPosX = windowWidth;  
 mapProcessedX = windowWidth;

武器の切り替えはBボタン(Sキー)で行うことにしましょう。  
メイン画面用の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;  
 }  
**+**  
**+**// 武器の切り替え.  
**+**if (gamepad.buttonDown & GamePad::B) {  
**+** if (weaponType == weaponTypeWideShot) {  
**+** weaponType = weaponTypeLaser;  
**+** } else {  
**+** weaponType = weaponTypeWideShot;  
**+** }  
**+**}  
  
 // 弾の発射.  
 if (gamepad.buttonDown & GamePad::A) {  
 for (int i = 0; i < weaponLevel; ++i) {

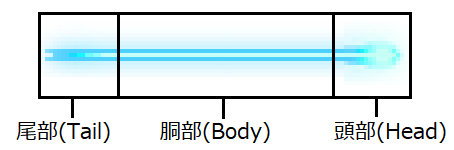
次に、weaponTypeに応じて使える武器が変わるようにします。  
弾を発射するプログラムを、次のように変更してください。

// 弾の発射.  
**-**if (gamepad.buttonDown & GamePad::A) {  
**+**if ((weaponType == weaponTypeWideShot) &&  
**+** (gamepad.buttonDown & GamePad::A)) {  
 for (int i = 0; i < weaponLevel; ++i) {  
 Actor\* bullet = findAvailableActor(  
 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.spr.Position(), Rect(64, 0, 32, 16));  
 const float angles[] = { 0, 7.5f, -7.5f, 15.0f, -15.0f };  
 const float radian = angles[i] / 180.0f \* 3.14f;  
 const float c = std::cos(radian);  
 const float s = std::sin(radian);  
 bullet->spr.Tweener(TweenAnimation::Animate::Create(  
 TweenAnimation::MoveBy::Create(1, glm::vec3(1200 \* c, 1200 \* s, 0),  
 TweenAnimation::EasingType::Linear)));  
 bullet->spr.Rotation(radian);  
 bullet->collisionShape = Rect(-8, -4, 16, 8);  
 bullet->health = 1;  
 }  
 }  
 sePlayerShot->Play(); // 弾の発射音を再生.  
 }  
**+**// レーザーの発射.  
**+**if ((weaponType == weaponTypeLaser) &&(gamepad.buttons & GamePad::A)) {  
**+**}

レーザーはボタンを押しっぱなしで発射できるように、buttonDown変数ではなくbuttons変数を調べるようにしています。

## レーザー用Actor配列を追加する

レーザーは、下図のように3つのスプライトで表現することにします。



胴部は、Y方向のスケールを変更して長さを変化させることで、ひとつのスプライトだけで表現することができるでしょう。そういうわけで、レーザーのためのActor配列を追加しなければなりません。  
Actor配列を定義している部分に、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.  
 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.

**[課題03]**  
playerLaserList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。

## レーザーを発射する

Actorの準備が整ったところで、レーザーを発射するプログラムを作っていきます。  
メイン画面用のprocessInput関数に、次のプログラムを追加してください。

// レーザーの発射.  
 if ((weaponType == weaponTypeLaser) && (gamepad.buttons & GamePad::A)) {  
**+** if (playerLaserList[0].health <= 0) {  
**+** const glm::vec3 posFiringPoint = sprPlayer.spr.Position();  
**+** playerLaserList[0].spr = Sprite("Res/Objects.png", posFiringPoint,  
**+** Rect(96, 0, 32, 16));  
**+** playerLaserList[0].collisionShape = Rect(-8, -4, 16, 8);  
**+** playerLaserList[0].health = 1;  
**+** playerLaserList[1].spr = Sprite("Res/Objects.png", posFiringPoint,  
**+** Rect(112, 0, 32, 16));  
**+** playerLaserList[1].health = 1;  
**+** playerLaserList[1].collisionShape = Rect(-8, -4, 16, 8);  
**+** playerLaserList[2].spr = Sprite("Res/Objects.png", posFiringPoint,  
**+** Rect(128, 0, 32, 16));  
**+** playerLaserList[2].collisionShape = Rect(-8, -4, 16, 8);  
**+** playerLaserList[2].health = 1;  
**+** }  
**+**}

0番目のActorがレーザーの尾部(Tail)、1番めが胴部(Body)、2番めが頭部(Head)です。  
最初に自機の座標をposFiringPoint(ぽす・ふぁいありんぐ・ぽいんと=発射口の座標)という変数に格納しています。レーザー用のActorはたかだか3つだけです。この程度なら「sprPlayer.spr.Position()」を3回書いてもよかったでしょう。しかし、変数として名前をつけたほうが、プログラムが読みやすくなると思います。

さて、レーザーは常に自機と同じY座標に表示しようと思います。トウィーニングは決められた動きをさせるだけなので、こういった場合には使えません。つまり、レーザー用の更新プログラムを作成しなければならない、ということです。  
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。

// Actorの更新.  
 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);  
  
**+**// レーザーの移動.  
**+**if (playerLaserList[0].health > 0) {  
**+** // レーザー(頭部)の移動.  
**+** const float laserSpeed = 1600.0f;  
**+** const glm::vec3 posFiringPoint = sprPlayer.spr.Position();  
**+** glm::vec3 posHead = playerLaserList[2].spr.Position();  
**+** posHead.x += laserSpeed \* deltaTime;  
**+** if (posHead.x > windowWidth \* 0.5f) {  
**+** posHead.x = windowWidth \* 0.5f;  
**+** }  
**+** posHead.y = posFiringPoint.y;  
**+** playerLaserList[2].spr.Position(posHead);  
**+**  
**+** // レーザー(胴部)の移動.  
**+** const float halfHeadSize = playerLaserList[2].spr.Rectangle().size.x \* 0.5f;  
**+** const float halfTailSize = playerLaserList[0].spr.Rectangle().size.x \* 0.5f;  
**+** const float bodySize = playerLaserList[1].spr.Rectangle().size.x;  
**+** const float bodyLength =  
**+** posHead.x - posFiringPoint.x – halfHeadSize - halfTailSize;  
**+** glm::vec3 posBody = playerLaserList[1].spr.Position();  
**+** posBody.x = posFiringPoint.x + (posHead.x – posFiringPoint.x) \* 0.5f;  
**+** posBody.y = posFiringPoint.y;  
**+** playerLaserList[1].spr.Position(posBody);  
**+** playerLaserList[1].spr.Scale(glm::vec2(bodyLength / bodySize, 1));  
**+** playerLaserList[1].collisionShape =  
**+** 　 Rect(-bodyLength \* 0.5f, -4, bodyLength, 8);  
**+**  
**+** // レーザー(尾部)の移動.  
**+** playerLaserList[0].spr.Position(posFiringPoint);  
**+**}  
  
 // 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);

レーザーの更新では、ヘッド、ボディ、テイルのそれぞれに異なるプログラムを書いています。  
ヘッドは発射されたら1600ドット/秒の速度で画面右側に移動させます。さらに、移動した先が画面右端を超えていた場合、それ以上右には移動しないようにしています。  
テイルはレーザーの発射地点に当たる画像なので、発射口の位置がそのままテイルの座標になります。

ボディは少し複雑です。Y座標には、他の2つと同様に発射口のY座標を使います。X座標は、ヘッドと発射口(これはテイルの位置でもあります)のちょうど中間に来るようにしています。ボディとして表示するべき長さ(bodyLength(ぼでぃ・れんぐす))は、ヘッドと発射口の距離から、ヘッドとテイルの半分の長さを引いたものになります。半分の長さを引いている理由は、ヘッドにもテイルにもそれぞれ画像が割り当てられているので、その画像と重ならないようにするためです。そして、この「表示するべき長さ」を「ボディ画像の大きさ」で割ったものを、X方向の拡大率にしています。  
そして、ボディはX方向の長さが変化するわけですから、衝突判定の大きさもそれに合わせて変化させます。

これで、レーザーを発射できるようになったはずです。  
**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、Sキーを押して武器をレーザーに切り替え、Aキーを押してレーザーを発射してみてください。  
…あれれ？エラーが出て、プログラムが停止してしまいましたね。  
ぬるぽ？

## updateActorList関数を修正する

(ガッ)  
レーザーを発射するとゲームが停止してしまう原因は、updateActorList関数の中にあります。  
現在、この関数は次のようになっていると思います。さて、どこが良くなかったのでしょうか？

void updateActorList(Actor\* first, Actor\* last, float deltaTime)  
{  
 for (Actor\* i = first; i != last; ++i) {  
 if (i->health > 0) {  
 i->spr.Update(deltaTime);  
 if (i->spr.Tweener()->IsFinished()) {  
 i->health = 0;  
 }  
 }  
 }  
}

思い出してほしいのですが、レーザー用のActor変数にはトウィーニングを設定していませんでした。ところが、上記のupdateActorList関数では「i->spr.Tweener()->IsFinished()」の部分でトウィーニング用の変数を参照しています。つまりコンピューターは、レーザー用のActorを更新しようとしたときにトウィーニング用の変数を見つけられなくて、パニックを起こしていたわけです。

そこで修正方法ですが、要はトウィーニング用の変数の有無が分かればいいわけです。  
updateActorList関数を次のように変更してください。

void updateActorList(Actor\* first, Actor\* last, float deltaTime)  
 {  
 　for (Actor\* i = first; i != last; ++i) {  
 　 if (i->health > 0) {  
 　 i->spr.Update(deltaTime);  
**-** if (i->spr.Tweener()->IsFinished()) {  
**+** if (i->spr.Tweener() && i->spr.Tweener()->IsFinished()) {  
 　 i->health = 0;  
 　}  
 　}  
　 }  
　}

Tweener関数の戻り値がヌルポインタでないときに限り、IsFinished関数を呼び出すようにしています。

**プログラムが書けたら、ビルドして実行してください。**Sキーを押して武器をレーザーに切り替え、Aキーを押し、レーザーが発射されれば成功です。

## レーザーを停止する

普通の弾と違い、レーザーはひとたび発射されたら止める方法はありません。  
さすがに危険すぎるので(レーザーを出したまま、母艦や基地に戻るわけにはいきませんよね？)、Aキーが押されていないときは停止するようにしましょう。

メイン画面用のprocessInput関数に、次のプログラムを追加してください。

playerLaserList[2].spr = Sprite("Res/Objects.png", posFiringPoint,  
 Rect(128, 0, 32, 16));  
 playerLaserList[2].collisionShape = Rect(-8, -4, 16, 8);  
 playerLaserList[2].health = 1;  
 }  
**+**} else {  
**+** // レーザーを停止する.  
**+** if (playerLaserList[0].health > 0) {  
**+** for (Actor\* i = std::begin(playerLaserList);  
**+** i != std::end(playerLaserList); ++i) {  
**+** i->spr = Sprite();  
**+** i->health = 0;  
**+** }  
**+** }  
}

レーザーを撃つのを止めたとき、全てのレーザー用Actorのhealthを0にして未使用の状態にしています。

**プログラムが書けたら、ビルドして実行してください。**  
レーザーを発射したあと、Aキーを離したときにレーザーが消えれば成功です。

## レーザー発射音を鳴らす

レーザーにも音を付けましょう。  
まずは音声制御用変数を作成するのでしたね。  
音声制御用変数を定義している部分に、次のプログラムを追加してください。

// 音声制御用変数.  
 Audio::SoundPtr bgm;  
 Audio::SoundPtr sePlayerShot;  
**+**Audio::SoundPtr sePlayerLaser;  
 Audio::SoundPtr seBlast;  
 Audio::SoundPtr sePowerUp;

そして、レーザー用音声を準備します。  
音声を準備している部分に、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");  
**+**sePlayerLaser = audio.Prepare("Res/Audio/Laser.xwm");  
 sePowerUp = audio.Prepare("Res/Audio/GetItem.xwm");  
 bgm = audio.Prepare("Res/Audio/Neolith.xwm");

音声を準備したら、それを鳴らしましょう。  
レーザーを発射している部分に、次のプログラムを追加してください。

playerLaserList[2].spr = Sprite("Res/Objects.png", posFiringPoint,  
 Rect(128, 0, 32, 16));  
 playerLaserList[2].collisionShape = Rect(-8, -4, 16, 8);  
 playerLaserList[2].health = 1;  
**+** sePlayerLaser->Play(Audio::Flag\_Loop);  
 }  
 } else {  
 // レーザーを停止する.  
 if (playerLaserList[0].health > 0) {  
**+** sePlayerLaser->Stop(); for (Actor\* i = std::begin(playerLaserList);  
 i != std::end(playerLaserList); ++i) {  
 i->spr = Sprite();  
 i->health = 0;  
 }  
 }  
}

レーザーはAキーを押している間ずっと発射されつづけるので、音声もループ再生させています。そして、選択されている武器がレーザーでなくなったり、Aキーが離されたときに停止させています。

**プログラムが書けたら、ビルドして実行してください。**  
レーザーを発射したとき、音が再生されていたら成功です。

## レーザーを停止する(自機が破壊された場合)

ここでちょっと試してほしいことがあります。レーザーを発射した状態で敵の横や後ろから体当たりをして、自機を破壊してください。  
やってみると、自機が破壊されたにもかかわらず、レーザーが発射されたままで、レーザー発射音も鳴り続けているのではないでしょうか。

こうなってしまう原因は、processInput関数の中でレーザーを停止する仕組みにあります。  
メイン画面用のprocessInput関数は、おおよそ次のような構造をしています。

void processInput(GLFWEW::WindowRef window)  
{  
 **/\* [A] タイトル画面およびゲームオーバー画面の処理. \*/**  
 if (gamestate == gamestateTitle) {  
 processInput(window, &titleScene);  
 return;  
 } else if (gamestate == gamestateGameOver) {  
 processInput(window, &gameOverScene);  
 return;  
 }  
  
 **/\* [B] メイン画面用の処理. \*/**  
 window.Update();  
  
 if (sprPlayer.health <= 0) {  
 **/\* [C] 自機が破壊されているときの処理. \*/**  
 } else {  
 **/\* [D] 自機が生存しているときの処理(自機の移動、武器の切り替え、弾・レーザーの発射など) \*/**  
 }  
}

このなかで、レーザーを停止する処理は、「[D]自機が生存しているときの処理」に組み込まれています。そのため、自機が破壊されてしまった場合、レーザーを停止することはできません。

**[補足]**  
[C]と[D]の順序が逆になっている人もいると思います。その場合は、みなさん自身が書いたprocessInput関数の中で、自機のhealthが0以下の場合の処理が[C]、0より大きい場合の処理が[D]だと考えてください。

今回の場合、やりたいことは「自機が破壊されている場合にもレーザーを停止させる」です。つまり、[D]にある「レーザーを停止させる」プログラムを、[C]にも追加すればいいはずです。ただ、同じプログラムを複数の場所にコピペするのはあまり良いプログラミング作法とは言えません。ですから、まずは「レーザーを停止させるプログラム」を関数にしましょう。

最初に行うのは、プロトタイプ宣言を書くことです。  
プロトタイプ宣言をしている部分に、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 stopPlayerLaser();

関数名はstopPlayerLaser(すとっぷ・ぷれいやー・れーざー)としてみました。

続いて、関数本体を定義します。  
自機とアイテム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。

// 自機の武器を強化する.  
 ++weaponLevel;  
 if (weaponLevel > weaponLevelMax) {  
 weaponLevel = weaponLevelMax;  
 }  
 }  
**+**  
**+**/\*\*  
**+**\* 自機のレーザーを停止する.  
**+**\*/  
**+**void stopPlayerLaser()  
**+**{  
**+**}

そして、processInput関数からレーザーを停止させるプログラムをコピーし(Ctrl+C)、stopPlayerLaser関数に貼り付けます(Ctrl+V)。

/\*\*  
\* 自機のレーザーを停止する.  
\*/  
void stopPlayerLaser()  
{  
**+** if (playerLaserList[0].health > 0) {  
**+** for (Actor\* i = std::begin(playerLaserList);  
**+** i != std::end(playerLaserList); ++i) {  
**+** i->spr = Sprite();  
**+** i->health = 0;  
**+** }  
**+** sePlayerLaser->Stop();  
**+** }  
}

これでstopPlayerLaser関数は完成です。  
次はコピー元のプログラムをこの関数で置き換えましょう。  
processInput関数の中のレーザーを停止させるプログラムを、次のように変更してください。

playerLaserList[2].spr = Sprite("Res/Objects.png", posFiringPoint,  
 Rect(128, 0, 32, 16));  
 playerLaserList[2].collisionShape = Rect(-8, -4, 16, 8);  
 playerLaserList[2].health = 1;  
 }  
} else {  
**-** // レーザーを停止する.  
**-** if (playerLaserList[0].health > 0) {  
**-** for (Actor\* i = std::begin(playerLaserList);  
**-** i != std::end(playerLaserList); ++i) {  
**-** i->spr = Sprite();  
**-** i->health = 0;  
**-** }  
**-** sePlayerLaser->Stop();  
**-** }  
**+** stopPlayerLaser();  
}

そして最後に、自機が破壊されている場合にもstopPlayerLaser関数を追加します。元々はこれが目的でしたね。

if (sprPlayer.health <= 0) {  
 playerVelocity = glm::vec3(0, 0, 0);  
**+** stopPlayerLaser();  
 } else {

**プログラムが書けたら、ビルドして実行してください。**  
レーザーを発射したまま敵に体当たりして、レーザーが消えてレーザー発射音が停止したら成功です。

## レーザーと敵の衝突判定

レーザーの発射と停止が問題なくできるようになったので、いよいよ敵との衝突判定をつけようと思います。  
衝突を処理する関数のプロトタイプ宣言から始めましょう。  
プロトタイプ宣言をしている部分に、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 playerLaserAndEnemyContactHandler(Actor\*, Actor\*);  
 void playerAndEnemyContactHandler(Actor\*, Actor\*);  
 void playerAndItemContactHandler(Actor\*, Actor\*);  
 void stopPlayerLaser();

そして、関数の定義を作成します。  
自機の弾と敵の衝突を処理する関数定義の下に、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, 1.5f)));  
 blast->health = 1;  
 seBlast->Play();// 爆発音を再生.  
 }  
 }  
 }  
**+**  
**+**/\*\*  
**+**\* 自機のレーザーと敵の衝突を処理する.  
**+**\*  
**+**\* @param laser 自機のレーザーのポインタ.  
**+**\* @param enemy 敵のポインタ.  
**+**\*/  
**+**void playerLaserAndEnemyContactHandler(Actor\* laser, Actor\* enemy)  
**+**{  
**+** enemy->health -= laser->health;  
**+** if (enemy->health <= 0) {  
**+** score += 100;  
**+** // 爆発を表示する.  
**+** 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.  
**+** }  
**+** }  
**+**}  
  
 /\*\*  
 \* 自機と敵の衝突を処理する.  
 \*  
 \* @param player 自機のポインタ.

レーザーは敵を貫通する設定にしたいので、敵のhealthだけ減らしています。  
そこから下は、自機の弾と敵の衝突判定と全く同じなので「コピー&貼り付け」を使うのがいいでしょう。

関数が出来たら、それを使って衝突判定をプログラムします。  
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。

// 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);  
**+**// 自機のレーザーと敵の衝突判定.  
**+**detectCollision(  
**+** std::begin(playerLaserList), std::end(playerLaserList),  
**+** std::begin(enemyList), std::end(enemyList),  
**+** playerLaserAndEnemyContactHandler);  
 // 自機と敵の衝突判定.  
 detectCollision(  
 &sprPlayer, &sprPlayer + 1,  
 std::begin(enemyList), std::end(enemyList),  
 playerAndEnemyContactHandler);

**プログラムが書けたら、ビルドして実行してください。**レーザーで敵を倒すことが出来たら成功です。

**[課題04]**  
爆発を表示するプログラムを関数にしてください。爆発を表示するプログラムは何箇所も存在しますが、それらはほとんど同じものです。できる範囲でいいので、それらを作成した関数で置き換えてください。

**[課題05]**  
武器のレベルに応じてレーザーを強化する方法を考えてください(実際にプログラムできる必要はありません)。