[OpenGL 2D 2018 第09回]

逃げる奴は皆敵だ！  
逃げない奴はよく訓練された敵だ！

# 敵の動きの作り方

## 2つの移動をつなげる

TweenAnimation::MoveByを使うことで、まっすぐ移動させることはできました。しかし、まっすぐ動くだけではあまり面白くありません。もっと複雑な動きは作れないのでしょうか。

実はTweenAnimationには2つ以上のMoveByを連続で実行させる仕組みが備わっています。これにはTweenAnimation::Sequence(とうぃーんあにめーしょん・しーけんす)という型を使います。  
TweenAnimation::SequenceにはAddというメンバ関数があります(メンバ関数は、「ある型専用の関数」なのでしたね)。この関数のパラメータとしてTweenAnimation::MoveBy::Createで作成した変数を設定すると、設定した順番でMoveByが実行されるようになっています。

試しに、SequenceとMoveByを使ってジグザグに移動する敵を作ってみましょう。  
update関数にある敵出現プログラムを次のように変更してください。

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, glm::vec3(-1000, 0, 0))));  
**+** namespace TA = TweenAnimation;  
**+** TA::SequencePtr seq = TA::Sequence::Create(2);  
**+** seq->Add(TA::MoveBy::Create(  
**+** 1, glm::vec3(-200, 100, 0), TA::EasingType::Linear));  
**+** seq->Add(TA::MoveBy::Create(  
**+** 1, glm::vec3(-200, -100, 0), TA::EasingType::Linear));  
**+** enemy->spr.Tweener(TA::Animate::Create(seq));  
 enemy->collisionShape = Rect(-16, -16, 32, 32);  
 enemy->health = 1;  
 enemyGenerationTimer = 2;  
 }

TweenAnimation::Sequence::Create関数には、「繰り返し回数」を指定できます。  
今回のプログラムでは、2つのMoveByを合わせてX座標を-400の位置まで移動させています。画面の横幅は800なので、これを2回繰り返せば画面を横断することができるでしょう。

また、今回は記述量を減らすために「using(ゆーじんぐ)宣言」を使用して「名前空間の別名」を設定しています。これは次のような構文になっています。

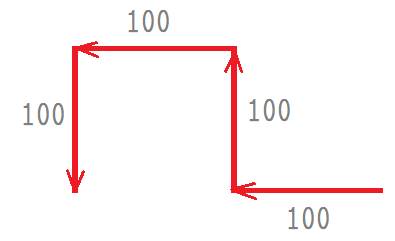
名前空間の別名  
 **↓**  
**using TA = TweenAnimation;**  
 **↑** **↑**  
using宣言 元の名前空間名

こうして別名を宣言をすることで、TweenAnimationのかわりにTAと書くことができるようになるのです。なお、別名が使えるのは「直前の波括弧に対応する閉じ波括弧まで」という点に気をつけてください。上記のプログラムでは直前の波括弧は「if (enemy != nullptr) {」の最後の部分に当たります。

**プログラムの変更が完了したら、ビルドして実行してください。**  
敵がジグザグに動いていたら成功です。

**[課題01]**  
MoveByのX移動量を-200から-100にしてください。それだけだと画面の途中で消えてしまうので、繰り返し回数を4回にしてください。

**[課題02]**  
下図のように、右、上、右、下の順番で1秒間に100ピクセルの速度で移動する動きを作成してください。



## XとYを別々に扱う

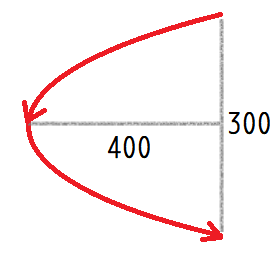
Xの移動量とYの移動量を別々に設定できると便利な場合があります。  
これにはTweenAnimation::Parallelize(とうぃーんあにめーしょん・ぱられらいず)と、MoveBy::Createの追加のパラメータを使います。  
update関数にある敵出現プログラムを次のように変更してください。

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));  
 namespace TA = TweenAnimation;  
 TA::SequencePtr seq = TA::Sequence::Create(2);  
**-** seq->Add(TA::MoveBy::Create(  
**-** 1, glm::vec3(-200, 100, 0), TA::EasingType::Linear));  
**-** seq->Add(TA::MoveBy::Create(  
**-** 1, glm::vec3(-200, -100, 0), TA::EasingType::Linear));  
**-** enemy->spr.Tweener(TA::Animate::Create(seq));  
**+** 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 = 1;  
 enemyGenerationTimer = 2;  
 }

Sequenceと同様にTweenAnimation::Parallelize::Create関数にも繰り返し回数を指定できます。  
TweenAnimation::MoveBy::Create関数の4つめのパラメータには、TweenAnimation::Target(とうぃーんあにめーしょん・たーげっと)という型の値を設定します。X, Y, Zの組み合わせて、どの軸を移動の対象にするかを選択できます。例えばTweenAnimation::Target::Xを設定すると、YとZの移動は行われません。

**プログラムの変更が完了したら、ビルドして実行してください。**  
敵が波打つように移動していたら成功です。  
XとYを分けたことでトウィーニングの種類も別々に設定できるため、このような動きを作れるのです。

**[課題03]**  
下図のように、弧を描くように移動する動きを作成してください。移動速度は任意とします。



# フレーム・アニメーション

## トウィーニングとフレーム・アニメーションの違い

トウィーニングを使った敵の移動方法についてはひととおり説明しました。  
ところで、2D画像というのは「ある方向からの物体の見え方」を記録したものです。ですから、その物体が他の方向からどのように見えるかという情報はほとんど何も持っていません。そのため、トウィーニングではX軸あるいはY軸方向の回転を伴うようなアニメーションを行っても、平面が回転しているようにしか見えません。Z軸以外の回転を表現するには、フレーム・アニメーションを使う必要があります。

フレーム・アニメーション(セル・アニメーションとも)は「時間経過に従って、画像そのものを次々に切り替える」ことで、あたかも絵が動いているかのように見せる方法です。

## キーフレームとタイムライン

フレーム・アニメーションを行うには、「どの画像を」「どのくらいの時間」表示するかという情報が必要です。この情報を「キーフレーム(key frame)」といいます。そして、キーフレームを時間軸にそって並べたものを「タイムライン(timeline)」といいます。

タイムラインを作成するには、キーフレームの配列と、タイムラインを管理する変数を用意しなくてはなりません。敵リストの変数定義の下に、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.  
  
**+**// 敵のアニメーション.  
**+**const FrameAnimation::KeyFrame enemyKeyFrames[] = {  
**+** { 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) },  
**+**};  
**+**FrameAnimation::TimelinePtr tlEnemy;  
  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);

FrameAnimation::KeyFrame(ふれーむあにめーしょん・きーふれーむ)は、名前の通りキーフレームを表現するための型です。この型は構造体になっていて、表示開始時間、画像の表示したい範囲の右下座標、表示したい範囲の大きさ、の3つのメンバで構成されています。

FrameAnimation::TimelinePtr(ふれーむあにめーしょん・たいむらいん・ぽいんた)はタイムラインを管理するための型です。

## タイムラインを作成する

次に、キーフレームの配列からタイムラインを作成します。  
乱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。

random.seed(std::random\_device()()); // 乱数エンジンの初期化.  
**+**  
**+**tlEnemy = FrameAnimation::Timeline::Create(enemyKeyFrames);  
  
 // 使用する画像を用意.  
 sprBackground = Sprite("Res/UnknownPlanet.png");  
 sprPlayer = Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));

タイムラインを作成するにはFrameAnimation::Timeline::Create関数を使います。パラメータとしてキーフレーム配列を設定すると、その配列を元にタイムラインを作成してくれます。

## 敵をフレーム・アニメーションさせる

作成したタイムライン変数を使って敵をアニメーションさせましょう。  
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.Animator(FrameAnimation::Animate::Create(tlEnemy));  
 namespace TA = TweenAnimation;  
 TA::SequencePtr seq = TA::Sequence::Create(2);

スプライトにフレーム・アニメーションを設定するには、Animator関数を使います。この関数のパラメーターは、フレーム・アニメーションを制御するための変数です。この制御変数を作成するには、FrameAnimation::Animate::Create関数を使います。この関数のパラメーターはタイムライン変数です。

**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X軸回転するようにアニメーションしていれば成功です。

**[課題04]**自機のタイムラインを作成してアニメーションさせてください。

# 文字を表示する

## スコアを追加する

敵を破壊して敵が消えた時、得点が入るようにしてみましょう。  
まずは得点を記録する変数を追加します。  
自機の弾の配列を定義してい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。

Actor enemyList[128]; // 敵のリスト.  
 Actor playerBulletList[128]; // 自機の弾のリスト.  
**+  
+**int score; // プレイヤーの得点.  
  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);

次に得点を初期化します。  
自機の弾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。

// 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;  
 }  
**+**  
**+**score = 0;  
  
 // ゲームループ.  
 while (!window.ShouldClose()) {

最後に、敵を破壊したときに得点を増やします。  
自機の弾と敵の衝突判定プログラムに、次のプログラムを追加してください。

Rect enemyRect = enemy->collisionShape;  
 enemyRect.origin += glm::vec2(enemy->spr.Position());  
 if (detectCollision(&shotRect, &enemyRect)) {  
 bullet->health -= 1;  
 enemy->health -= 1;  
**+** score += 100; // 敵を破壊したら得点を増やす. break;  
 }

**プログラムが書けたら、ビルドして実行してください。**…得点が入っているのかどうか、さっぱり分かりませんね。

## フォントを初期化する

得点の変数を作ったはいいものの、実際に何点入っているのかが分からないのでは意味がありませんよね。そこで、得点の数字を画面に表示しようと思います。

数字などの文字を表示するには、FontRenderer(ふぉんと・れんだらー)という型の変数を使います。  
FontRenderer型はFont.hというファイルで定義されていますので、まずはこのファイルをインクルードしましょう。  
Sprite.hのインクルード文の下に、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 "Font.h"  
 #include <random>

Font.hをインクルードしたら、FontRenderer型の変数を追加します。  
SpriteRenderer型の変数を定義してい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。

SpriteRenderer renderer; // スプライト描画用変数.  
**+**FontRenderer fontRenderer; // フォント描画用変数.  
 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.

次に、追加した変数を初期化します。  
SpriteRendererをしょきか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 (!Texture::Initialize()) {  
 return 1;  
 }  
 if (!renderer.Initialize(1024)) {  
 return 1;  
 }  
**+**if (!fontRenderer.Initialize(1024, glm::vec2(windowWidth, windowHeight))) {  
**+** return 1;  
**+**}  
**+**if (!fontRenderer.LoadFromFile("Res/Font/makinas\_scrap.fnt")) {  
**+** return 1;  
**+**}  
  
 random.seed(std::random\_device()()); // 乱数エンジンの初期化.

FontRendererを初期化するにはInitialize(いにしゃらいず)関数を使います。  
この関数の最初のパラメーターは「画面に表示可能な文字数」で、2つめのパラメーターは表示する画面の大きさです。

FontRendererを使って文字を表示するには、初期化に加えて文字の情報を設定する必要があります。これを行うのがLoadFromFile関数です。この関数のパラメーターは文字情報が設定されたファイル名です。

**[補足]**  
今回のフォントには「もじワク研究 www.moji-waku.com」の「マキナスscrap(5)」を使用しています。

## スコアを文字に変換して表示する

文字を表示する準備ができたところで、得点を表示してみましょう。  
render関数のスプライト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。

renderer.EndUpdate();  
 renderer.Draw({ windowWidth, windowHeight });  
  
**+**fontRenderer.BeginUpdate();  
**+**char str[9];  
**+**snprintf(str, sizeof(str), ”%08d”, score);  
**+**fontRenderer.AddString(glm::vec2(-64 , 300), str);  
**+**fontRenderer.EndUpdate();  
**+**fontRenderer.Draw();  
  
 window.SwapBuffers();

フォントを描画する方法はスプライトの描画と似ています。まずBeginUpdate関数を呼び出し、次に必要なだけAddStringを使って文字を追加し、追加が終わったらEndUpdate関数を呼び出します。そしてDraw関数を呼び出せば、追加した文字が表示されるという仕組みです。

数値を文字列に変換するにはsnprintf関数を使うのが簡単です。  
この関数はprintfに似ていますが、文字を画面ではなくパラメーターとして設定した変数に書き込む点が違っています。

# 敵配置マップ

## ランダムの先へ

攻撃を仕掛けるのに部隊の出現位置をサイコロに任せる指揮官はいないでしょう。  
そこで、敵の出現位置やタイミングをランダム以外の方法で制御してみましょう。

通常、指揮官は地図を見ながら部隊の配置を考えます。そして、ゲームプログラムにおいて、地図に相当する最も簡単な構造は２次元配列でしょう。ということで、早速プログラムを書いていきたいところですが、少し待ってください。プログラムに２次元配列を直接書き込んでしまうと、配置を考え直すたびにビルドしなおさなければなりません。加えて、部隊が出現したときにどのような画面になるかのイメージもつかみにくいです。これを解決するために、敵の配置をサポートしてくれるツールを使い、そのツールで作ったデータを読み込んで、敵を出現させることにします。

## TiledMapを初期化する

ツールが作ったデータを読み込むには、TiledMap型の変数を使います。  
この型はTIledMap.hというファイルで定義されていますので、まずはTiledMap.hをインクルードしましょう。Font.hのインクルード文の下に、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 "Font.h"  
**+**#include "TiledMap.h"  
 #include <random>

次に、TiledMap型の変数を定義します。  
フレーム・アニメーション用の変数を定義してい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。

{ 0.500f, glm::vec2(480, 0), glm::vec2(32, 32) },  
 };  
 FrameAnimation::TimelinePtr tlEnemy;  
**+**  
**+**// 敵の出現を制御するためのデータ.  
**+**TiledMap enemyMap;  
**+**float mapCurrentPosX;  
**+**float mapProcessedX;  
  
　void processInput(GLFWEW::WindowRef);  
　void update(GLFWEW::WindowRef);  
　void render(GLFWEW::WindowRef);  
　bool detectCollision(const Rect\* lhs, const Rect\* rhs);

mapCurrentPosXとmapProcessedXは、敵配置データをどこまで処理したかを記録するための変数です。

変数の用意ができたら、敵配置データを読み込みましょう。  
スコア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。

score = 0;  
**+**  
**+**// 敵配置マップを読み込む.  
**+**enemyMap.Load("Res/EnemyMap.json");  
**+**mapCurrentPosX = mapProcessedX = windowWidth;  
  
 // ゲームループ.  
 while (!window.ShouldClose()) {

敵配置データを読み込むには、TiledMapのLoadメンバ関数を使います。パラメータは敵配置ファイル名です。データの処理位置を記録する変数は、画面の横幅で初期化しています(何故こうしているかは後述)。

## プログラムの切り替えを可能にする

敵配置データを使うプログラムを書くと、ランダムバージョンの敵出現プログラムを消さなくてはなりません。せっかく書いたものを消すのは忍びないので、プリプロセッサの機能を使ってランダムバージョンを一時的に無効化することにします。  
敵出現プログラムに、次のプログラムを追加してください。

// 敵の出現.  
**+**#if 1  
**+**  
**+**#else  
 enemyGenerationTimer -= deltaTime;  
 if (enemyGenerationTimer < 0) {  
 Actor\* enemy = nullptr;  
 for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 if (i->health <= 0) {  
 enemy = i;  
 break;  
 }  
 }  
 if (enemy != nullptr) {  
  
 …省略…  
  
 }  
 }  
**+**#endif

「#if ～ #else ～ #endif」は「プリプロセッサ・ディレクティブ」と呼ばれるものの一種です。  
#ifに書いた条件が真(0以外)であれば#ifから#elseのあいだにあるプログラムだけが、偽(0)であれば#elseから#endifのあいだにあるプログラムだけが有効になり、残りの部分はコンピューターから無視されるようになります。

## 敵配置データを使う

それでは、敵配置データを使って敵を出現させてみましょう。  
#if 1と#elseのあいだに次のプログラムを追加してください。なお、空いている構造体の検索や敵のアニメーションを設定する部分などはランダムバージョンと全く同じです。「コピー&貼り付け」をうまく使ってください。

// 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.  
**+** 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;  
**+** }  
**+** // 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) {  
**+** // 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 float y =  
**+** windowHeight \* 0.5f - static\_cast<float>(mapY \* tileSize.x);  
**+** enemy->spr = Sprite("Res/Objects.png",  
**+** glm::vec3(0.5f \* windowWidth, y, 0), Rect(480, 0, 32, 32));  
**+** 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 = 1;  
**+** }  
**+** }  
**+** }  
**+** }  
 #else  
 enemyGenerationTimer -= deltaTime;  
 if (enemyGenerationTimer < 0) {

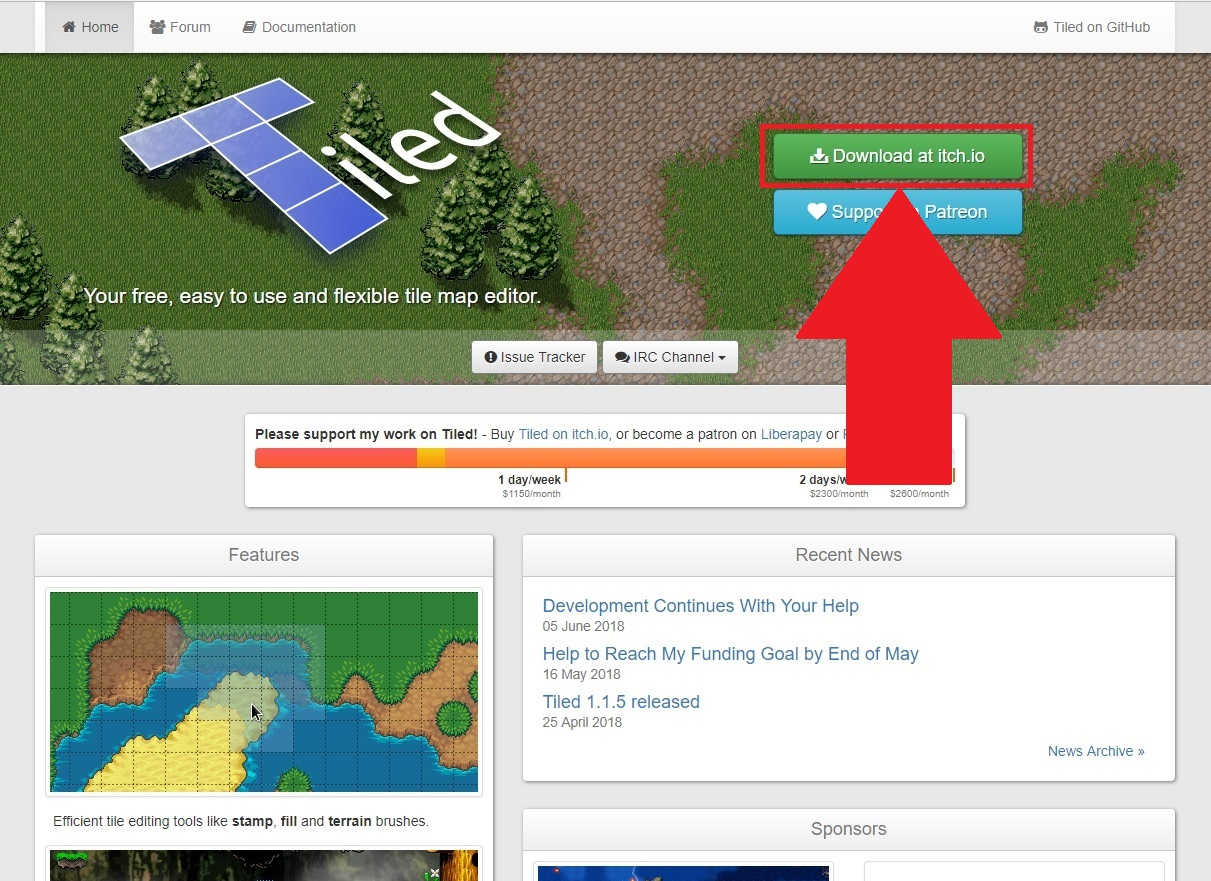
プログラムが書けたら、ビルドして実行してください。  
敵が隊列を組んで現れたら成功です。

# Tiled Map Editor

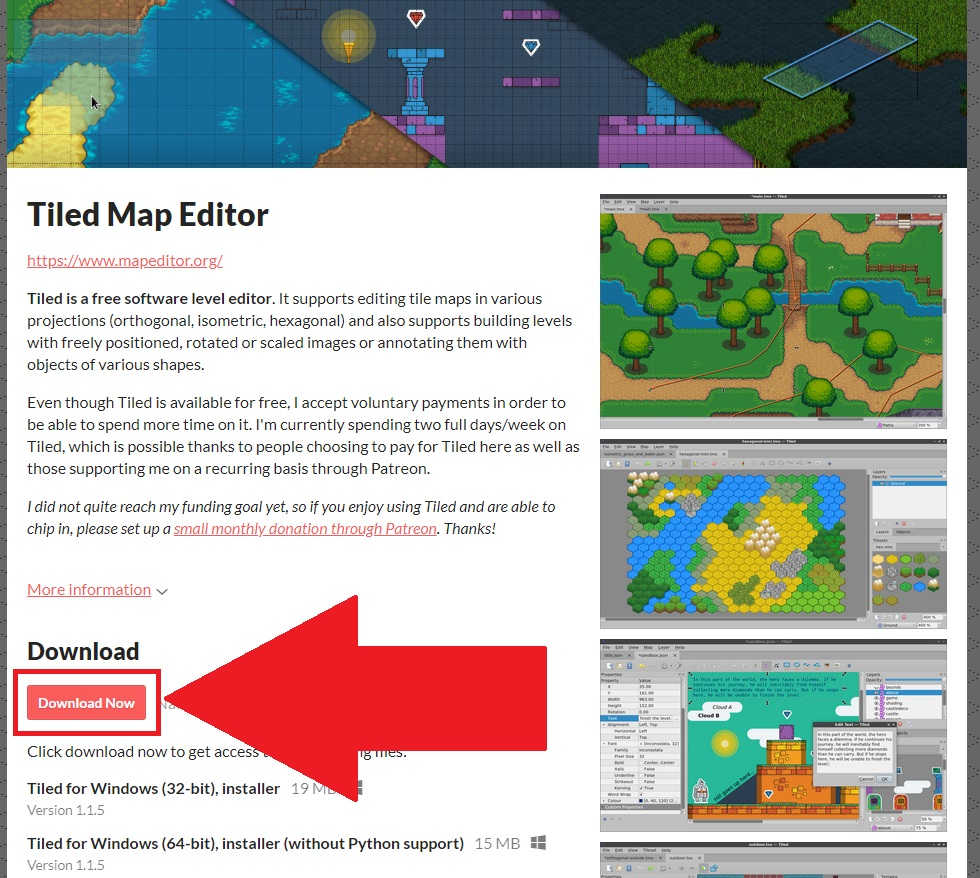
## インストーラーをダウンロードする

今回、敵配置データを作るために使用したツールは「Tiled Map Editor(たいるど・まっぷ・えでぃたー)」というものです。みなさんの手で敵の配置をいじれるように、このツールをインストールしてもらおうと思います。

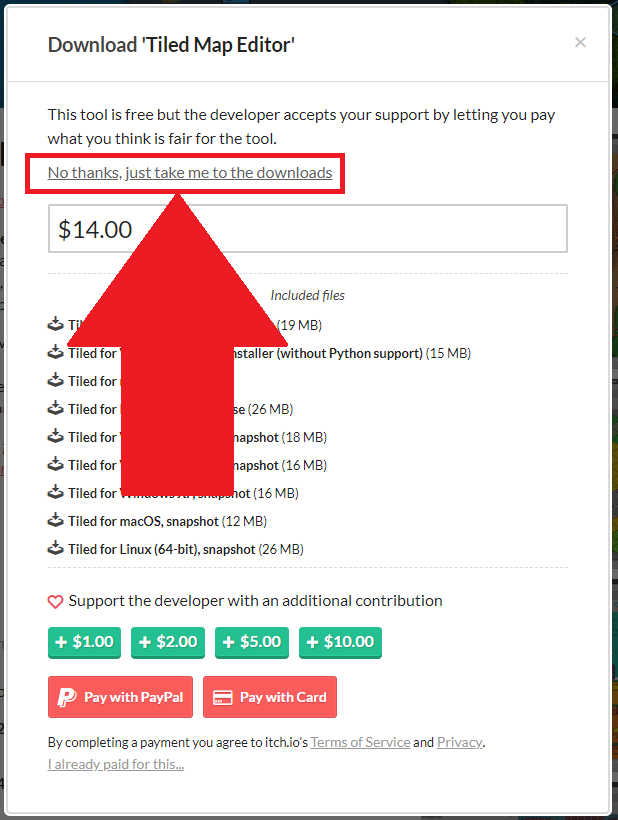
ブラウザを起動し、アドレスバーに「www.mapeditor.org」と入力するか、「tiled map editor」で検索してTiled Map Editorの公式サイトを表示して下さい。



サイトが表示されたら、右上にある「Download at itch.io(だうんろーど・あっと・いっち・あいおー)」ボタンをクリックします。

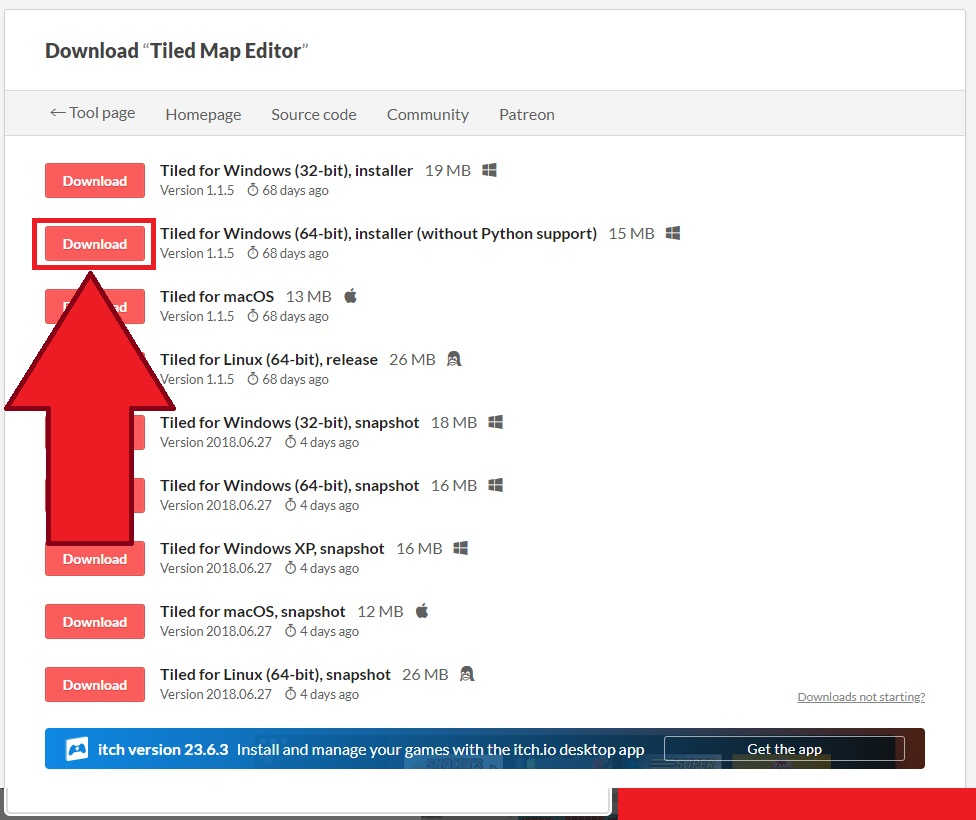


「Download Now(だうんろーど・なう)」をクリックします。  
すると画面中央に寄付をお願いするウィンドウが開きます。



寄付しない場合は「No thanks, just take me to the downloads(のー・さんくす, じゃすと・ていく・みー・とぅ・ざ・だうんろーず)」と書かれている文章をクリックします。

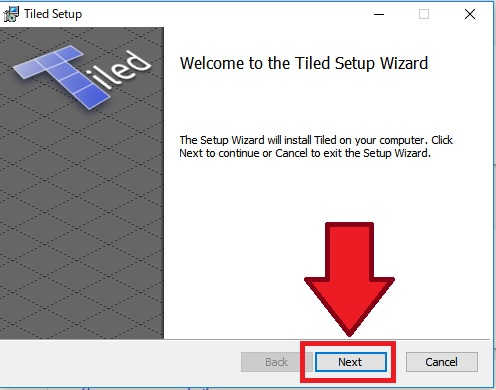
すると、次のような環境ごとに違うファイルをダウンロードするページが開きます。



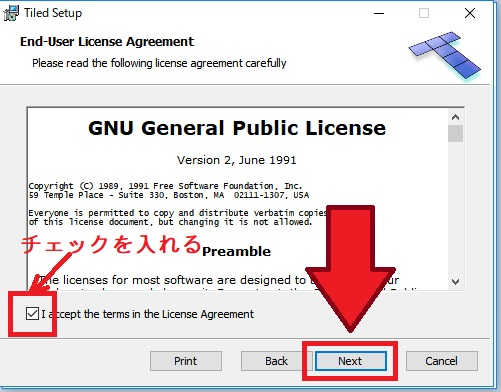
「Tield for Windows(64-bit)」という文章の左にある「Download」ボタンをクリックします。  
すると、ファイルのダウンロード先を選択するウィンドウが開きますので、適当な場所を選んでOKを押してダウンロードしてください。

## Tiled Map Editorのインストール

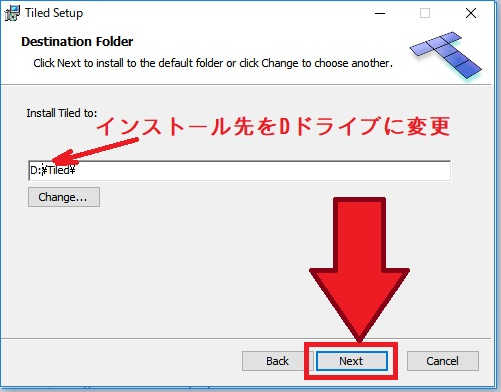
ダウンロードが完了したら、ダウンロードしたファイルを開いてください。  
すると次のようなウィンドウが表示されます。



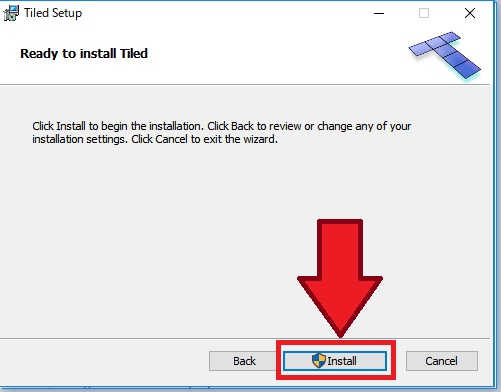
「Next」をクリックしてください。  
すると、ライセンス条項への同意を求められます。



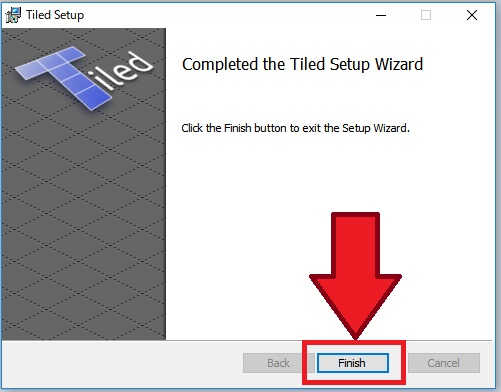
左下のチェックボックスをクリックしてチェックを入れ、「Next」をクリックしてください。  
次はインストール先の選択です。



インストール先をDドライブに変更して「Next」をクリックしてください。



「Install」ボタンをクリックするとインストールが始まります。  
インストールが完了すると、次の画面が表示されます。



「Finish」ボタンをクリックして、インストーラーを終了してください。

これでTiled Map Editorのインストールは完了です。  
敵配置ファイルを開いて適当にいじってみてください。