[OpenGL 2D 2018 第07回]

動けよ我が機体、とプレイヤーは言った

# OpenGLで絵を描くには

## テクスチャ

コンピューターグラフィックスではさまざまな画像を扱います。画像はプログラムの指示によってコンピューター内の空いているメモリに読み込まれたあと、必要に応じて表示用のメモリにコピーされます。  
OpenGLでは、こういった画像データを「テクスチャ(texture(てくすちゃ))」と呼びます。

## 頂点

画像を画面に表示するためには「画面のどの位置に」「画像のどの部分を」表示するか、という情報が必要となります。OpenGLでは、この情報のことを「頂点(vertex(ばーてっくす))」と呼びます。  
現代の一般的なコンピューターグラフィックスでは、3つの頂点からなる三角形を基本単位として画像を表示します。この三角形のことを「ポリゴン(polygon(ぽりごん)」と呼びます。

**[補足]**  
ポリゴンは多角形という意味なので三角形とは限りません。事実、古いグラフィックスシステムなどでは四角形以上の多角形を扱えました。ところが、より高性能なグラフィックス表示能力が求められた結果、コンピューターは三角形を表示する能力に特化していきました。あらゆる多角形は三角形の組み合わせで表現できるので、問題にはならないと考えられたからです。その結果、現代のコンピューターは実際には三角形しか扱えないにもかかわらず、その呼び方はポリゴンのままになっています。  
なお、三角形であることを明示したい場合は「triangle(とらいあんぐる)」と呼びます。

テクスチャと頂点の2種類を用意すればOpenGLで絵を描くことができます。  
2Dゲームで使われるような長方形の画像は「スプライト(sprite)」と呼ばれ、1つのスプライトは4つの頂点で構成されます(実際には長方形を対角線で分割した2つの三角形が描かれます)。  
2Dゲームでは、このスプライトを自由に使って画面を作っていくことになります。

# 初期化

「初期化」とは変数や配列に値を設定して使えるようにしたり、OSにプログラムを動かすための準備をさせることです。実はじゃんけんゲームでもapplication関数が呼ばれる前に初期化をしていました。  
今回のプログラムでは「初期化」もみなさんに書いてもらいます。といっても、全てを書いてもらうのはとても大変なので、難しい部分はこちらで行い「初期化」ではどんなことをするのか、という点に絞って書いていってもらおうと思います。

## OpenGL用のウィンドウを準備する

絵を描くためには画像を表示できるウィンドウが必要となります。  
初期化で最初に行うのは、このウィンドウの作成です。  
とはいえ、正攻法でウィンドウを作成しようとすると数十行のプログラムを書く必要があります。みなさんにはまだ難しい部分もありますので、今回は簡単にウィンドウを作成できる関数を用意しました。

この関数はGLFWEW.h(じーえるえふだぶりゅーいーだぶりゅー.h)というファイルの中に定義していますので、まずはこれをインクルードしましょう。それから、ウィンドウに関するいくつかの情報を変数として定義しておきます。  
Main.cppを開き、次のプログラムを追加してください。

/\*\*  
 \* @file Main.cpp  
 \*/  
**+**#include "GLFWEW.h"  
  
**+**const char windowTitle[] = "OpenGL2D 2018"; // タイトルバーに表示される文章.  
**+**const int windowWidth = 800; // ウィンドウの描画領域の幅.  
**+**const int windowHeight = 600; // ウィンドウの描画領域の高さ.  
  
 /\*\*  
 \* プログラムのエントリーポイント.  
 \*/

続いてウィンドウを作成します。  
main関数の内側に、次のプログラムを追加してください。

/\*\*  
 \* プログラムのエントリーポイント.  
 \*/  
 int main()  
 {  
**+** // アプリケーションの初期化.  
**+** GLFWEW::WindowRef window = GLFWEW::Window::Instance();  
**+** if (!window.Initialize(windowWidth, windowHeight, windowTitle)) {  
**+** return 1;  
**+** }  
 return 0;  
 }

このプログラムはまずGLFWEW::Window::Instance(じーえるえふだぶりゅーいーだぶりゅー・うぃんどう・いんすたんす)関数でウィンドウ用の変数を取得し、次にGLFWEW::Window::Initialize(じーえるえふだぶりゅーいーだぶりゅー・うぃんどう・いにしゃらいず)関数を実行してウィンドウを作成します(Initialize(いにしゃらいず)は「初期化する」という意味の英単語です)。  
Initialize関数はウィンドウの描画領域の幅と高さ、タイトルバーに表示する文章の3つのパラメータを必要とします。これらはさきほど変数として用意したものを設定しています。また、作成に成功した場合はtrue、失敗した場合はfalseが返ってくるので、falseの場合は終了コード1でプログラムを終了させています。

**[補足]**  
一般的なプログラムでは、正常にプログラムが実行された場合、main関数の最後にある「return 0;」が実行されることにより終了コード0で終了します。上記のように、正常に実行できなかった場合は1などの0以外の数値を返すようにします。これは必須ではありませんが、コンピューターの世界では長年の慣習になっていますので合わせておくべきでしょう。

## 型や関数にラベルを付ける(名前空間)

関数名の途中にある「::」は+や==のような演算子のひとつで「スコープ解決演算子」というものです。  
プログラムはさまざまな機能を持つ変数や関数を組み合わせて作られます。プログラムが大きく複雑になってくると、ある変数に関係する関数がどれなのかが分かりにくくなっていきます。そこで、関係のある変数や関数にラベルをつけて、その関係性を示すことが考えられました。

例えば、じゃんけんゲームでは画像を配置したり移動したりするために、次のような関数がありました。

- set\_image  
- move\_image  
- scale\_image  
 など…

これらは全て画像、つまりimage(イメージ)を操作する関数ですから、imageという名前のラベルをつけることにします。さらに関数名の「\_image」という部分はラベルと重複するので消します。  
こうしてラベルを付けた関数は次のようになるでしょう。

|  |  |
| --- | --- |
| ラベルを使わない関数名 | ラベルを使った関数名 |
| set\_image | image::set |
| move\_image | image::move |
| scale\_image | image::scale |
| など… |  |

関数名だけを見るとあまりラベルを付ける意味ないように思えますね。しかし実は大きな違いがあります。これはC++言語の機能なので、コンパイラはこれらの関数が関連しているものだと認識できます。これによって、インテリセンス等がラベル名→関数名の順番で補完してくれるようになるため、より効率よくプログラムを書けるのです。

そんなわけで、GLFWEW::WindowRef(じーえるえふだぶりゅーいーだぶりゅー・うぃんどうれふ)というのは「GLFWEWというラベルの付いたWindowRefという型」というように読めるわけです(なおRefというのはReference(れふぁれんす)の略で「参照」という意味があります。が、今はあまり気にしないでください)。  
ラベルは何段でも付けられます。また、型もラベルの一種とみなされますので、例えばGLFWEW::Window::Instance(じーえるえふだぶりゅーいーだぶりゅー・うぃんどう・いんすたんす)というのは、「GLFWEW::Windowというラベルの付いたInstance関数」というように読むことができます。

**[補足]**  
C++言語のラベルを付ける機能は、正確には「名前空間(なまえくうかん)」と呼ばれます。英語では「namespace(ねーむすぺーす))」といいます。

## 型専用の関数(メンバ関数)

C++では型ごとに専用の関数を作ることができます。専用の関数を使うには、その型の変数名と関数名を「.(ドット)」でつなぎます。上記のプログラムでは「window.Initialize」の部分にあたります。この場合、windowが変数名でInitializeが関数名です。  
このような「型専用の関数」のことを「メンバ関数」と呼びます。メンバ関数は普通の関数と使い方が少し違いますが、それ以外の基本的な機能については普通の関数とまったく同じです。

**[補足]**  
名前空間やメンバ関数はC/C++言語の授業で「構造体」や「クラス」を学習するときに出てくると思います。今は「そういうものがあるんだな」という程度の認識で大丈夫です。

## テクスチャを操作する機能を準備する

ウィンドウの場合と同様、テクスチャを扱うにもさまざまな準備プログラムを書く必要があります。これも簡単に準備ができる関数を用意しました。  
この関数はTexture.h(てくすちゃ.h)というファイルで宣言されていますので、まずはこのファイルをインクルードします。Main.cppを開き、次のようにGLFWEW.hのインクルード文の下に、Texture.hのインクルード文を追加してください。

/\*\*  
 \* @file Main.cpp  
 \*/  
#include "GLFWEW.h"  
**+**#include "Texture.h"  
  
const char windowTitle[] = "OpenGL2D 2018"; // タイトルバーに表示される文章.  
const int windowWidth = 800; // ウィンドウの描画領域の幅.  
const int windowHeight = 600; // ウィンドウの描画領域の高さ.

続いて関数を書きます。  
main関数に次のプログラムを追加してください。

int main()  
 {  
 // アプリケーションの初期化.  
 GLFWEW::Window& window = GLFWEW::Window::Instance();  
 if (!window.Initialize(windowWidth, windowHeight, windowTitle)) {  
 return 1;  
 }  
**+** if (!Texture::Initialize()) {  
**+** return 1;  
**+** }  
**+**  
**+** Texture::Finalize();  
 return 0;  
 }

テクスチャ操作の準備を行う関数はTexture::Initialize(てくすちゃ・いにしゃらいず)です。  
この関数は特にパラメータを必要とせず、ただ実行するだけでテクスチャをあやつる準備が完了します。この関数も準備成功したらtrue、失敗したらfalseを返してきますから、falseの場合は終了コード1でプログラムを終了します。

Texture::Finalize(てくすちゃ・ふぁいならいず)関数はテクスチャ操作を終了させます。  
Texture::Initialize関数が成功した場合、プログラムを終了する前に実行しておく必要があります。  
詳しい理由は述べませんが、テクスチャ操作はちょっと特殊で、この関数を使って後始末する必要があるのです。

## 頂点を操作する機能を準備する

テクスチャの次は、頂点を操作する準備を行います。これも1から書くのは大変なので、簡単な関数を用意してあります。この関数はSprite.h(すぷらいと.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。Main.cppを開き、次のようにTexture.hのインクルード文の下に、Sprite.hのインクルード文を追加してください。それと、スプライトを描画するための変数も追加します。

/\*\*  
 \* @file Main.cpp  
 \*/  
#include "GLFWEW.h"  
#include "Texture.h"  
**+**#include "Sprite.h"  
  
const char windowTitle[] = "OpenGL2D 2018"; // タイトルバーに表示される文章.  
const int windowWidth = 800; // ウィンドウの描画領域の幅.  
const int windowHeight = 600; // ウィンドウの描画領域の高さ.  
  
**+**SpriteRenderer renderer; // スプライトを描画するオブジェクト.  
**+**  
 /\*\*  
 \* プログラムのエントリーポイント.  
 \*/

SpriteRenderer(すぷらいと・れんだらー)は1章で少しだけ説明した「スプライト」を画面に表示するための型です。英単語のrender(れんだー)の意味のひとつに「表示する」というものがあり、そこに「～するもの」という意味の「-er」を付けたのがrenderer(れんだらー)という英単語です。つまり、SpriteRendererとは「スプライトを描画するもの」という意味になるんですね。

続いて関数を書きます。  
main関数に次のプログラムを追加してください。

int main()  
 {  
 // アプリケーションの初期化.  
 GLFWEW::Window& window = GLFWEW::Window::Instance();  
 if (!window.Initialize(windowWidth, windowHeight, windowTitle)) {  
 return 1;  
 }  
 if (!Texture::Initialize()) {  
 return 1;  
 }  
**+** if (!renderer.Initialize(1024)) {  
**+** return 1;  
**+** }  
 Texture::Finalize();  
 return 0;  
 }

SpriteRendererもInitialize(いにしゃらいず)関数によって初期化できます。メンバ関数の場合、変数の型が異なると、名前が同じでも異なる関数が実行されることに注意しましょう。なぜならメンバ関数は「型専用の関数」だからです。GLFWEW::Window用のInitialize関数と、SpriteRenderer用のInitialize関数があるということです。  
この関数のパラメータは「一度に表示できるスプライトの数」です。とりあえず1024個あれば十分でしょう。より多くのスプライトを表示する必要が生じたときは、この数字を大きくしてください。

これで初期化は完了です。

# ゲームループ

## ゲームループの意義

現代の多くのゲームでは、ゲーム内のさまざまな事象がリアルタイムに変化していきます。  
アクションゲームだけでなく、ターン制のシミュレーションゲームだろうと、ノベルゲームだろうと、プレイヤーの操作がなくても画面内のなんらかの要素がアニメーションし続けたり、音楽が流れ続けたりしていることでしょう。scanfのような関数でプレイヤーの入力を待つ方法では、そういったことは実現できません。

ゲームループはこの問題を解決します。  
基本的なアイデアは、プレイヤーの入力を処理はしても待ちはしない、ということです。

## 3つの機能

最も基本的なゲームループは次のような構造です。

while (true) {  
 processInput(); // プレイヤーの入力を処理する(が、待たない).  
 update(); // ゲームの状態を更新する.  
 render(); // ゲームの状態を描画する.  
}

3つの関数が順番に実行され、それが永遠にループしています。  
processInput(ぷろせす・いんぷっと)は前回の呼び出し以降に発生したプレイヤーの入力を処理します。もし入力がなければ何もしません。update(あっぷでーと)はゲームの状態を前回のループから経過した時間のぶんだけ進めます。最後にrender(れんだー)は更新されたゲームの状態を描画します。

まずこの3つの関数を宣言しましょう。  
renderer変数定義の下に、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.  
  
SpriteRenderer renderer; // スプライトを描画するオブジェクト.  
  
**+**void processInput(GLFWEW::WindowRef);  
**+**void update(GLFWEW::WindowRef);  
**+**void render(GLFWEW::WindowRef);  
**+**  
 /\*\*  
 \* プログラムのエントリーポイント.  
 \*/

先程の説明ではなんのパラメータもありませんでしたが、実際の関数はゲームを管理するウィンドウを引数として受け取るようにしています。というのも、GLFWEW::Window型はプレイヤーの入力や経過時間の管理、画面の制御といった重要な機能を持っているので、これなしではどの関数も大したことはできないからです。

宣言を終えたら、さっそくゲームループを追加しましょう。  
renderer.Initialize関数の成功判定をしているif文の下に、次のプログラムを追加してください。

if (!renderer.Initialize(1024)) {  
 return 1;  
 }  
**+**  
**+** // ゲームループ.  
**+** while (!window.ShouldClose()) {  
**+** processInput(window);  
**+** update(window);  
**+** render(window);  
**+** }  
  
 Texture::Finalize();  
 return 0;

ゲームループの説明ではwhile(true)としていましたが、それだとゲームを終了することができません。  
そこで、ループ条件にはGLFWEW::Window::ShouldClose関数を使っています。  
この関数は普段はfalseを返します。ただし、ウィンドウの「×」ボタンが押されたときなど、ゲームを終了する必要があると判断したらtrueを返します。

## processInput関数

続いて、入力を処理する関数を定義しましょう。  
main関数の閉じ波括弧の下に、次のプログラムを追加してください。

Texture::Finalize();  
 return 0;  
 }  
**+**  
**+**/\*\*  
**+**\* プレイヤーの入力を処理する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\*/  
**+**void processInput(GLFWEW::WindowRef window)  
**+**{  
**+** window.Update();  
**+**}

とりあえずは入力を受け取ったり経過時間を計算するために、GLFWEW::Window::Update関数を追加しています。今はそれだけです。

## update関数

次に、ゲーム状態を更新する関数を定義します。  
processInput関数の閉じ波括弧の下に、次のプログラムを追加してください。

void processInput(GLFWEW::WindowRef window)  
 {  
 window.Update();  
 }  
**+**  
**+**/\*\*  
**+**\* ゲームの状態を更新する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\*/  
**+**void update(GLFWEW::WindowRef window)  
**+**{  
**+**}

いまのところ、更新すべきものは何もありませんから関数の中身も空っぽです。

## render関数

最後にゲームの状態を描画する関数を定義しましょう。  
update関数の閉じ波括弧の下に、次のプログラムを追加してください。

void update(GLFWEW::WindowRef window)  
 {  
 }  
**+**  
**+**/\*\*  
**+**\* ゲームの状態を描画する.  
**+**\*  
**+**\* @param window ゲームを管理するウィンドウ.  
**+**\*/  
**+**void render(GLFWEW::WindowRef window)  
**+**{  
**+** window.SwapBuffers();  
**+**}

こちらもまだ何も描画するものがないのでGLFWEW::Window::SwapBuffers関数があるだけです。この関数が実行されると、描画した内容が画面に反映されます。

とりあえずここまで書けたらビルドして実行してください。  
真っ黒なウィンドウが2つ表示されたら成功です。

# 画像の表示

## スプライトを準備する

真っ黒なままではあまりに面白くありません。何か表示してみましょう。  
画像を表示するにはSprite(すぷらいと)という型を使います。  
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。

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.  
  
 SpriteRenderer renderer; // スプライトを描画するオブジェクト.  
**+**Sprite sprBackground; // 背景用スプライト.  
**+**Sprite sprPlayer; // 自機用スプライト.  
  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);

## 画像を設定する

次に、スプライトに画像を読み込みます。  
main関数のrenderer.Initialize関数の結果を判定するif文の閉じ波括弧の下に、次のプログラムを追加してください。

if (!renderer.Initialize(1024)) {  
 return 1;  
 }  
**+**  
**+**//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));  
  
 // ゲームループ.  
 while (!window.ShouldClose()) {

Sprite型の変数は、パラメータを与えて作成することができます。  
最初のパラメータは設定する画像ファイル名です。2つめのパラメータは表示位置です。3つめのパラメータは画像のどの部分を表示するかを指定します。また、全てのパラメータを設定する必要はありません。例えば画像ファイル名だけを設定した場合、画面中央に画像全体がそのまま表示されます。

## 頂点を書き込む

じゃんけんゲームの時とは違い、画像を読み込んだだけでは画面には何も表示されません。スプライトを表示するには、スプライトの頂点データをSpriteRendererに設定してあげる必要があります。

render関数に次のプログラムを追加してください。

void render(GLFWEW::WindowRef window)  
 {  
**+** renderer.BeginUpdate();  
**+** renderer.AddVertices(sprBackground);  
**+** renderer.AddVertices(sprPlayer);  
**+** renderer.EndUpdate();  
**+** renderer.Draw(glm::vec2(windowWidth, windowHeight));  
 window.SwapBuffers();  
 }

AddVertices(あど・ばーてぃしーず)というのが頂点データを設定する関数です。AddVertices関数は、BeginUpdate関数とEndUpdate関数の間でだけ使うことができます。  
OpenGLは頂点データを少しずつ描画するより、まとめて描画するほうがずっと得意です。そこで、BeginUpdate, AddVerticies, EndUpdateの3つの関数で描画したいスプライトの頂点データをまとめておき、最後にDraw関数を実行することで、設定した頂点データがまとめて描画されるという仕組みになっています。  
また、あとから設定した頂点データほど手前に表示されるので、自機スプライトを背景スプライトよりあとで設定しています。

書けたらビルドして実行してみてください。  
宇宙っぽい背景と白い自機が表示されたでしょうか。

# 自機の移動

## 入力を処理する

今度はプレイヤーの入力を処理して自機を移動させてみましょう。  
まずは自機の移動速度を示す変数を用意します。  
自機用スプライト変数の定義の下に、次のプログラムを追加してください。

SpriteRenderer renderer;  
 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.  
  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);

playerVelocity(ぷれいやー・べろしてぃ)はglm::vec3型の変数です。Velocity(べろしてぃ)は「速度」という意味の英単語です。  
glm::vec3はx, y, zの3つの成分を持つ型です(zは奥行き方向の成分です。今回は使いません)。

追加したら、今度はprocessInput関数に、次のプログラムを追加してください。

void processInput(GLFWEW::WindowRef window)  
 {  
 window.Update();  
**+**  
**+** // 自機の速度を設定する.  
**+** const GamePad gamepad = window.GetGamePad();  
**+** if (gamepad.buttons & GamePad::DPAD\_UP) {  
**+** playerVelocity.y = 1;  
**+** } else if (gamepad.buttons & GamePad::DPAD\_DOWN) {  
**+** playerVelocity.y = -1;  
**+** } else {  
**+** playerVelocity.y = 0;  
**+** }  
**+** if (gamepad.buttons & GamePad::DPAD\_RIGHT) {  
**+** playerVelocity.x = 1;  
**+** } else if (gamepad.buttons & GamePad::DPAD\_LEFT) {  
**+** playerVelocity.x = -1;  
**+** } else {  
**+** playerVelocity.x = 0;  
**+** }  
**+** if (playerVelocity.x || playerVelocity.y) {  
**+** playerVelocity \*= 400.0f;  
**+** }  
 }

GamePad型は、プレイヤーからの入力を値として持つことができます。つまりGamePad型の変数を調べれば、プレイヤーがどのキーを押したか分かるということです。押されたキーの状態はgamepad.buttonsという変数に設定されています。また、前回の入力判定で押されておらず、今回押されているキー、つまりキーの押し始めの状態はgamepad.buttonDownという変数に設定されています。今回は押されているかどうかが分かればいいので、gamepad.buttonsのほうを調べています。  
buttons変数の各ビットにはキーが対応付けられています。例えば右矢印キーは第4ビットに割り当てられていますから、右矢印キーが押されていれば第4ビットは1になり、押されていなければ0になります。GamePad型には各キーに対応するビットを示す変数が定義されています。例えば右矢印キーのビットを示す変数はGamePad::DPAD\_RIGHTです。この変数とbuttons変数で&演算すると、キーが押されているなら0以外、押されていなければ0となるため判定できるわけです。

なお、プログラム中の400.0fという数値が実際のプレイヤーの移動速度になります。この数値は「1秒間に移動できるドット数」と考えてください。

次に、update関数において速度を位置に加えます。  
update関数に次のプログラムを追加してください。

void update(GLFWEW::WindowRef window)  
 {  
**+** const float deltaTime = window.DeltaTime(); // 前回の更新からの経過時間(秒).  
**+**  
**+** // 自機の移動.  
**+** if (playerVelocity.x || playerVelocity.y) {  
**+** glm::vec3 newPos = sprPlayer.Position() + playerVelocity \* deltaTime;  
**+** sprPlayer.Position(newPos);  
**+** }  
**+** sprPlayer.Update(deltaTime);  
 }

プログラムとしては「自機の移動速度(playerVelocity)が0でない場合、速度\*経過時間を自機の座標に加算する」という動きになります。つまり等速直線運動ですね。

書けたらビルドして実行してください。  
矢印キーで画面中央の白い自機を操作できたら成功です。

## 斜め移動の速度

先程のプログラムでは上下と左右の移動速度を別々に設定しているため、斜めに移動した時は上下と左右の移動速度が組み合わさって速度が上がっていました。しかし、現実の物体では斜めに移動しようと速度は一定のはずです。これを何とかするには、移動速度をxとyからなるベクトルと考え、方向によらずベクトルの長さが1になるようにすれば良さそうです。ベクトルの長さを1にするにはglm::normalize(じーえるえむ・のーまらいず)関数を使います。  
というわけで、processInput関数の速度を掛けている部分を次のように変更してください。

} else {  
 playerVelocity.x = 0;  
 }  
 if (playerVelocity.x || playerVelocity.y) {  
**-** playerVelocity \*= 400.0f;  
**+** playerVelocity = glm::normalize(playerVelocity) \* 400.0f;  
 }  
 }

書けたらビルドして実行してみてください。  
斜め移動の速度が変更前より遅くなっていれば成功です。

## 移動範囲を制限する

現在は自機の移動範囲に制限がありません。そのため、簡単に画面外に移動できてしまいます。  
しかし，自機は基本的に画面内にいてほしいものです。  
そこで、自機の移動範囲を制限することにします。  
update関数の自機を移動させるプログラムの途中に、次のプログラムを追加してください。

// 自機の移動.  
 if (playerVelocity.x || playerVelocity.y) {  
 glm::vec3 newPos = sprPlayer.Position() + playerVelocity \* deltaTime;  
**+** // 自機の移動範囲を画面内に制限する.  
**+** const Rect playerRect = sprPlayer.Rectangle();  
**+** if (newPos.x < -0.5f \* (windowWidth - playerRect.size.x)) {  
**+** newPos.x = -0.5f \* (windowWidth - playerRect.size.x);  
**+** } else if (newPos.x > 0.5f \* (windowWidth - playerRect.size.x)) {  
**+** newPos.x = 0.5f \* (windowWidth - playerRect.size.x);  
**+** }  
**+** if (newPos.y < -0.5f \* (windowHeight - playerRect.size.y)) {  
**+** newPos.y = -0.5f \* (windowHeight - playerRect.size.y);  
**+** } else if (newPos.y > 0.5f \* (windowHeight - playerRect.size.y)) {  
**+** newPos.y = 0.5f \* (windowHeight - playerRect.size.y);  
**+** }  
 sprPlayer.Position(newPos);  
 }  
 sprPlayer.Update(deltaTime);

中心が原点なので、左右の範囲は「-ウィンドウの幅/2～ウィンドウの幅/2」で上下の範囲は「-ウィンドウの高さ/2～ウィンドウの高さ/2」になりそうですが、ちょっと待ってください。自機にも大きさがあります。  
自機が少しも画面外に出ないようにするには、範囲から自機の大きさ(Rectangle関数で取得)/2を引けばいいのです。

プログラムが書けたらビルドして実行してください。  
そして、自機が上下左右のどの端に行っても、画面外に移動しないことを確認してください。