

20 日で理解する3Dプログラミング「その 1: 導入」

■ 3D ゲームの仕組み

1) 3D ゲームを作る為には、3DCG の知識が必要不可欠となります。

パソコンなどで 3DCG を表現する為には「OpenGL」「DirectX」といった API を使用しますが、どちらも計算方式（レンダリング方式）は「**ラスタライズ方式**」と言います。これは、リアルタイムで高速に描画する為に、Zバッファという考え方で表現を行う方式となります。この方式は、高速性を優先する為に反面反射や屈折などの計算を簡易的に行う様になっています。

逆に、速度性を犠牲にして光の反射や屈折などに注力した方式を「**レイトレーシング方式**」と言います。ゲーム数学の授業でも取り組んでいると思いますが、いわゆるリアルな CG 表現を追求するものです。速度面で辛い部分があったのですが、最近ではハードウェアの性能向上により「**リアルタイムレイトレーシング**」が可能となったりしてきています。

とは言え、今回はDirectX(DxLib)を基準とするため「ラスタライズ方式」の 3DCG を扱っていきます。



2) 2D も 3D の仲間

3D のゲームイメージを考えると、「2D はどうなっているの？」と疑問に思う事もあるかもしれません。基本的には、2D も 3D 空間の一つと考えていきます。「3D 空間の中にある平面の板が配置しある」「描画面のスクリーンに張り付いているもの」という 2 種類の性質があるので、状況に合わせて合成していきます。

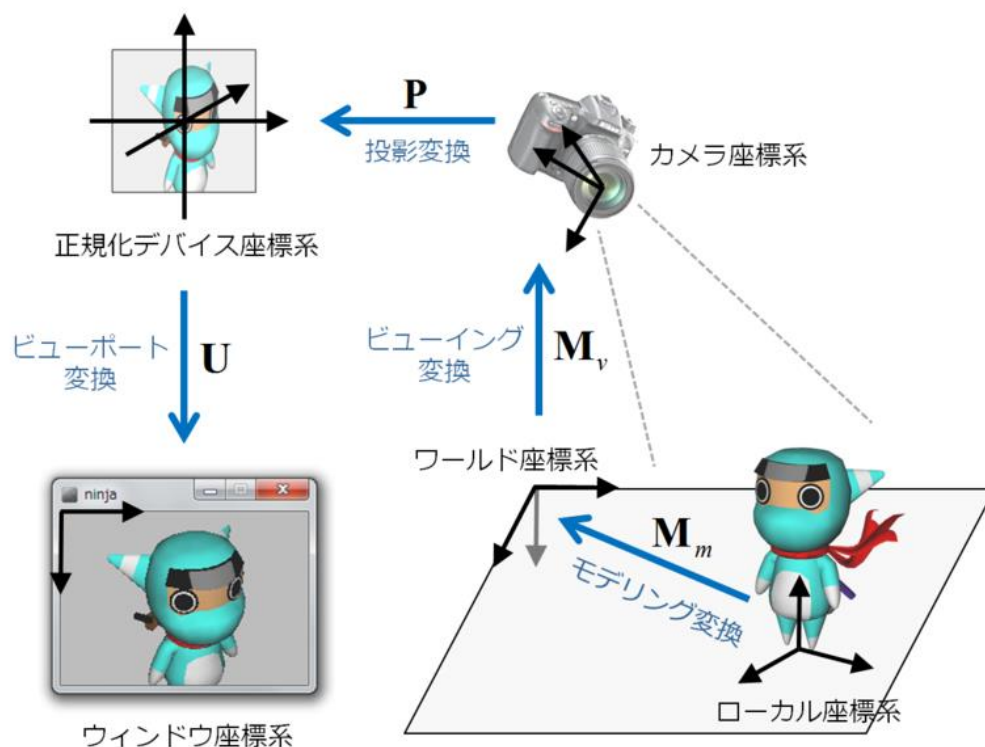
3) 3D ゲームはどうなっているのか？

3D 表現では、モデルデータを 3D 空間に配置しただけでは思い通りに画面に描画されません。3D 空間の「どこから」「どの方向」で見えるかなどを設定する事で画面に映る事になります。

その為、いくつかの座標系を持っており、それぞれの工程に沿って変換する事で必要な結果を得る事ができます。

主な座標系。

- ・モデルデータ単体の頂点の座標系…………… ローカル座標系
- ・モデルやカメラなどの空間の中での座標系…… ワールド座標系(モデリング変換)
- ・カメラの位置が原点座標となる座標系…………… ビュー座標系(ビューイング変換)
- ・視野、パースや投影面の範囲を設定する……… 射影変換行列(プロジェクション変換) ※クリップ空間 $-1 \sim +1$
- ・スクリーン上での座標系…………… スクリーン座標系(ビューポート変換 ※自動)



構成イメージ



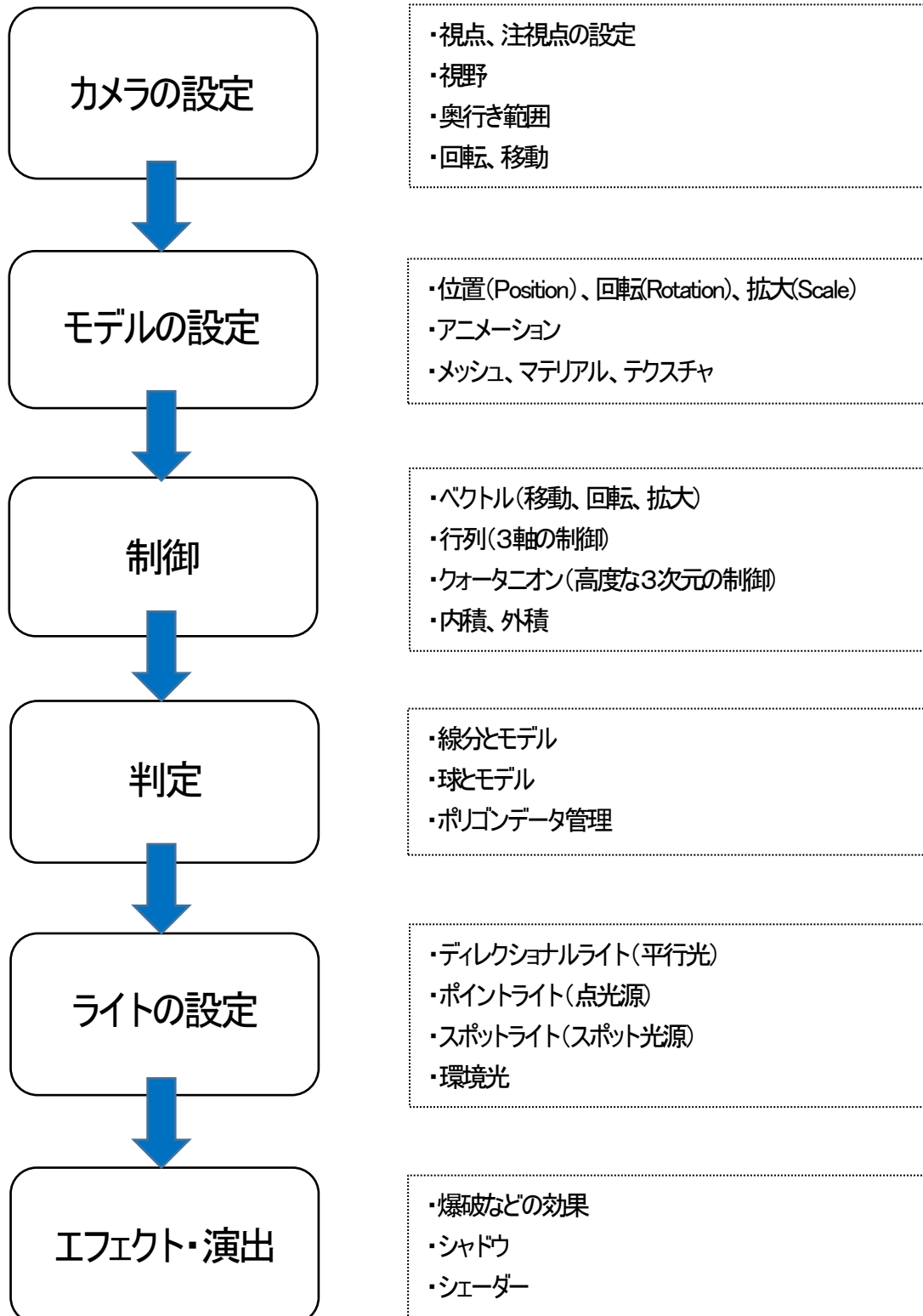
開発時の状態



ゲーム実行画面

■ 3D ゲームプログラミングの手順

- 1) 2D に比べ 3D での制御はやるべき事が多くなります。しかしながら、手順を踏む事で確実に実装できますので根気よく取り組んでいきましょう。下記に大まかな考え方ややるべき事を記載していますので、この順番で制御を行って行きたいと思います。

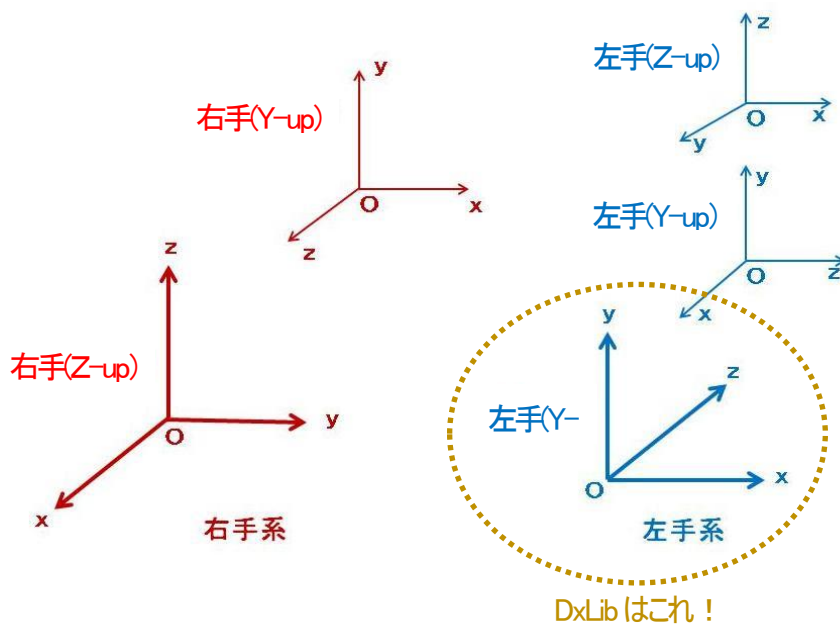
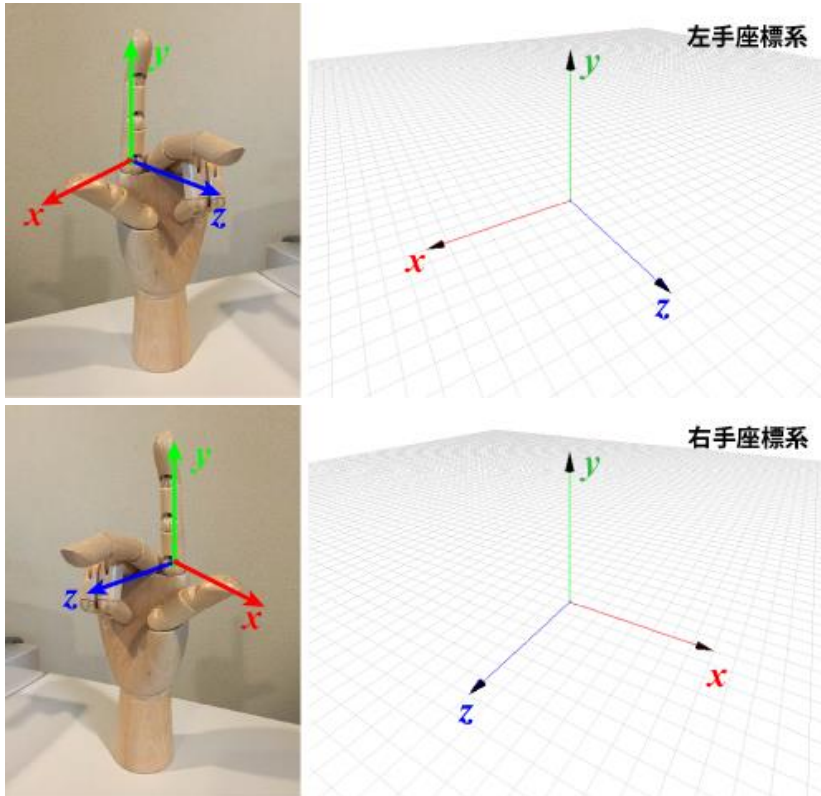


■座標系について

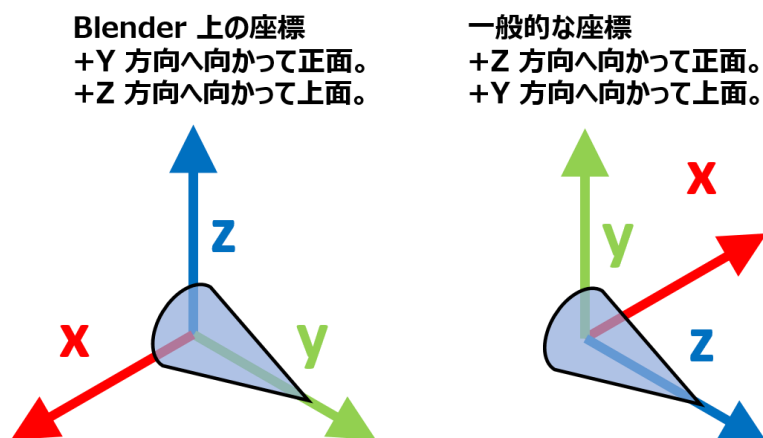
3DCGの座標系は「左手座標系」と「右手座標系」があります。親指がX軸、人差し指がY軸、中指がZ軸であり、軸の表現については、X軸が赤(R)、Y軸が緑(G)、Z軸が青(B)で表示されるのが一般的です。

ライブラリやエンジンでは下記のように設定されています。

- ・左手座標系…… DirectX(DxLib)、Unity、UnrealEngine
- ・右手座標系…… OpenGL、Maya、3dsMax、Blender



Blender では一般的な 3DCG と異なり、Z 軸を上方向とする 3 次元座標を利用しています。
 一般的には Y 軸を上方向で制御する方が多い為、その場合は座標系を一般的な 3 次元座標系に合わせて出力する
 為に、モデル出力時に上方向と正面方向を指定して出力する事を行います。(プログラム上でもできますが..)

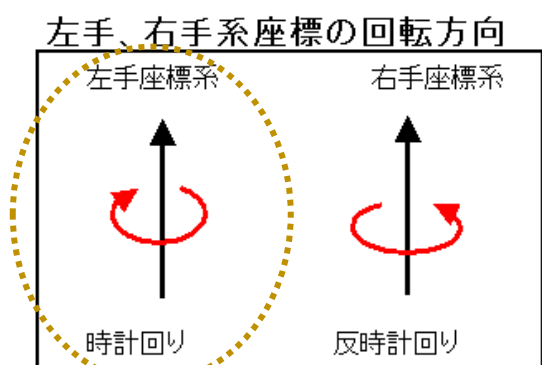


.fbx 出力時に軸を次のように調整すると同じ向きになる。

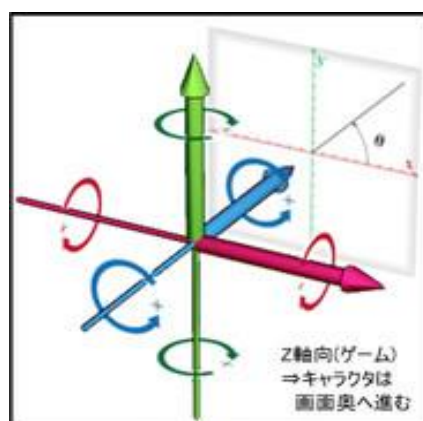
- Z Forward
- Y Up

Blender 座標系ではモデルの正面方向を +Y、上方向を +Z にします。出力時には、正面方向を +Z、上方向を +Y に設定します。この設定で出力すると、Blender とそれ以外の一般的なソフトウェア上で展開されるモデルの見た目が一致して、軸だけが変更されているかと思えます。

この座標系の違いで一番大きいのは回転方向で、左手系は時計回り、右手系は反時計回りに回転します。
 矢印の方向が+方向の回転で、逆が-方向の回転となります。



DxLib はこれ !



3Dの管理は(x, y, z)の3軸を基本としてベクトルで構成しますので、DxLibでは「VECTOR型」を使用します。

●DX ライブラリの **VECTOR** 構造体

```
typedef struct VECTOR {  
    float x, y, z;  
} VECTOR;
```

●XYZ 座標値から VECTOR 型の値を作成して返してくれる **VGet()** 関数

```
VECTOR VGet( float x, float y, float z );
```

例：VGet(1.0f, 2.0f, 3.0f) はベクトル(1.0f, 2.0f, 3.0f)を
VECTOR 型構造体の値として返してくれる。

「VECTOR 型」で制御する x、y、z のパラメーターの役割りは大きく2種類！

XYZ の**座標点**を表す

例) プレイヤーの座標をx、y、zの座標系で
(200, 50, 500)に設定する。

```
VECTOR pos1;  
pos1.x = 200.0f;  
pos1.y = 50.0f;  
pos1.z = 500.0f;
```

```
VECTOR pos2;  
pos2 = VGet(200.0f, 50.0f, 500.0f);
```

※pos1、pos2 は同じ意味合いです。
※VECTOR 型は**実数型**なので Float で定義

XYZ 方向の**ベクトル**を表す

例) x=10、y=-50、z=100の移動ベクトルを
変数に保存する。

```
VECTOR mov1;  
mov1.x = 10.0f;  
mov1.y = -50.0f;  
mov1.z = 100.0f;
```

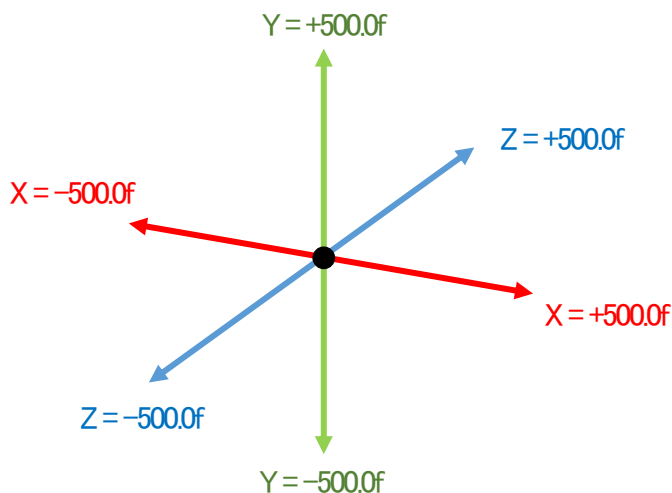
```
VECTOR mov2;  
mov2 = VGet(10.0f, -50.0f, 100.0f);
```

※mov1、mov2 は同じ意味合いです。
※VECTOR 型は**実数型**なので Float で定義

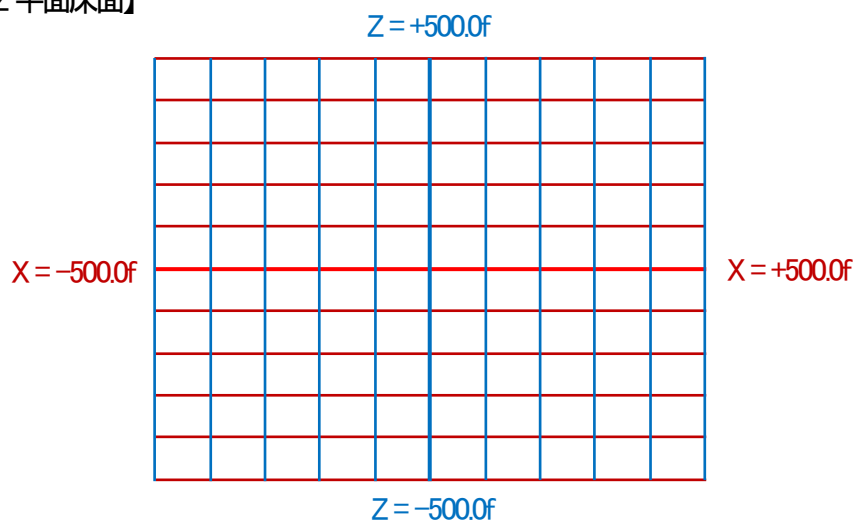
■フィールドの設定

3DCGは2Dと違って実行画面に対する固定の座標系を持っていません。画面への描画はカメラを通して行います。実際に思い通りに描画できているかの確認用に、原点を通るXYZの軸と床面に相当するXZ平面を作成します。

【XYZ 軸】



【XZ 平面床面】



【3D 空間に直線を引く関数】

```
int DrawLine3D( VECTOR Pos1, VECTOR Pos2, unsigned int Color ) ;
```

3 D空間に線分を描画する

VECTOR Pos1 : 線分の始点の座標

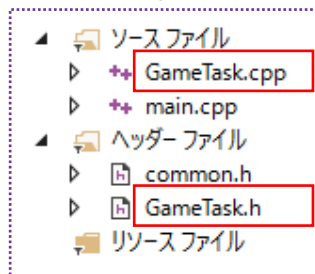
VECTOR Pos2 : 線分の終点の座標

unsigned int Color : 線分の色

■プロジェクトの設定

新規プロジェクトの作成を行い、3DPG のひな型をクラスを実装して作っていきましょう。
ゲームループに該当する部分を「GameTask クラス」としてファイルを追加していきます。

プロジェクト構成

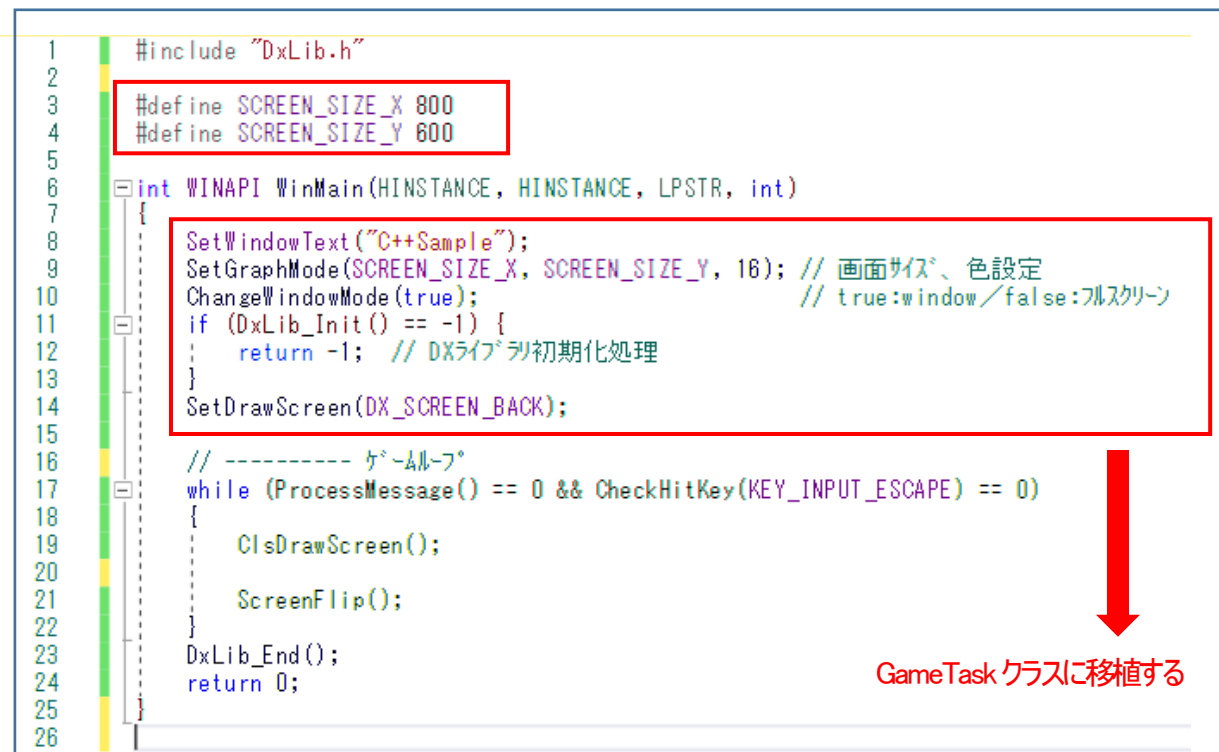


GameTask.h

```
#pragma once

class GameTask
{
public:
    GameTask();
    ~GameTask();
    int SystemInit(void); // 最初の初期化
    void GameMainLoop(void);
};
```

システム初期化の部分を GameTask::SystemInit()に移して、ゲームループオブジェクトで運用する。



WinMain関数のコード。1行目から26行目まで。3行目と4行目の#define SCREEN_SIZE_X 800と#define SCREEN_SIZE_Y 600は赤い枠で囲まれている。11行目から14行目のif (DxLib_Init() == -1) { return -1; }は赤い枠で囲まれている。17行目から22行目のwhile (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0) { ... }は赤い枠で囲まれている。23行目のDxLib_End();と24行目のreturn 0;は赤い枠で囲まれている。右側に赤い矢印があり、GameTask クラスに移植すると書かれている。

```
1 #include "DxLib.h"
2
3 #define SCREEN_SIZE_X 800
4 #define SCREEN_SIZE_Y 600
5
6 int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
7 {
8     SetWindowText("C++Sample");
9     SetGraphMode(SCREEN_SIZE_X, SCREEN_SIZE_Y, 16); // 画面サイズ、色設定
10    ChangeWindowMode(true); // true:window/false:フルスクリーン
11    if (DxLib_Init() == -1) {
12        return -1; // DXライブラリ初期化処理
13    }
14    SetDrawScreen(DX_SCREEN_BACK);
15
16    // ----- ゲームループ -----
17    while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0)
18    {
19        ClsDrawScreen();
20
21        ScreenFlip();
22    }
23    DxLib_End();
24    return 0;
25
26 }
```

GameTask クラスに移植する

「GameTask.cpp」の「SystemInit()」に、DxLib 関係の初期化処理をまるごと移動させます。

GameTask.cpp

```
#include "DxLib.h"
#include "GameTask.h"

#define SCREEN_SIZE_X 800
#define SCREEN_SIZE_Y 600

int GameTask::SystemInit()
{
    SetWindowText("C++Sample");
    SetGraphMode(SCREEN_SIZE_X, SCREEN_SIZE_Y, 16); // 画面サイズ、色設定
    ChangeWindowMode(true); // true:window/false:フルスクリーン
    if (DxLib_Init() == -1) {
        return -1; // DXライブラリ初期化処理
    }
    SetDrawScreen(DX_SCREEN_BACK);

    return 0;
}
```

この状態でコンパイルをするとエラーがでます

```
1  #include "DxLib.h"
2
3  int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
4  {
5      // ----- ゲームループ -----
6      while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0)
7      {
8          ClsDrawScreen();
9
10         ScreenFlip();
11     }
12     DxLib_End();
13     return 0;
14 }
15
```

例外がスローされました

0x00B5EF53 で例外がスローされました (GameTemple++.exe 内):
0xC0000005: 場所 0x00000014 の読み取り中にアクセス違反が発生しました

理由は分かりますか？

これは、まだ DxLib の初期化が実行されていない状態で DxLib の命令を使用しているからです。
では、実際に GameTask オブジェクトを生成してシステム初期化が実行される様にしてみます。

main()から GameTask をインスタンスして実際に使用する仕組みを構築します。

main.cpp

```
#include "DxLib.h"
#include "common.h"
#include "GameTask.h"

// ----- メイン処理 ※WinMain の警告の消し方
//int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
// 関数のパラメーターと戻り値に注釈を付けて警告を回避する
int WINAPI WinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance,
                  _In_ LPSTR lpCmdLine, _In_ int nShowCmd)
{
    GameTask* gameTask; // GameTaskを動的に確保 ①
    gameTask = new GameTask();

    // ----- ゲームループ
    while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) != 1)
    {
        ClsDrawScreen();

        gameTask->GameMainLoop(); // GameMainLoop() 関数を実行 ②

        ScreenFlip();
    }

    delete gameTask; // 領域を解放
    gameTask = nullptr; // ポインタを初期化 ③

    DxLib_End(); // DxLibの後始末
    return 0; // アプリの終了
}
```

GameTask のポインター「gameTask」を作成し GameTask オブジェクトを new して作成します。オブジェクトの生成の事をインスタンスと言います。

インスタンスのメンバ関数(クラスメソッド)へのアクセスは、gameTask->で行います。

※今回は動的確保していますが、静的確保でも OK。余裕のある方はシングルトンもアリです。

画面サイズなど、他のクラスでも使用する場合がある情報は「common.h」に定義して読み込みを行ってみましょう。

common.h

```
#pragma once

#define WINDOW_NAME "3DPG_Test"
#define SCREEN_SIZE_X 1024
#define SCREEN_SIZE_Y 600
```

#defineのマクロを constexpr に変換することも可。

```
constexpr auto WINDOW_NAME = "3DPG_Test";
constexpr auto SCREEN_SIZE_X = 1024;
constexpr auto SCREEN_SIZE_Y = 600;
```

constexpr 指定子は「constant expression (定数式)」の略語でコンパイル時に値が決定する定数となります。
他にも宣言された関数やコンストラクタは、コンパイル時と実行時に呼び出す事ができるなどの機能があります。

GameTask.cpp

```
#include "DxLib.h"
#include "common.h"
#include "GameTask.h"

GameTask::GameTask()
{
    SystemInit(); // 最初の初期化
}

GameTask::~GameTask()
{
}

int GameTask::SystemInit()
{
    ChangeWindowMode(true); // ウィンドウモード
    SetWindowText(WINDOW_NAME);
    SetGraphMode(SCREEN_SIZE_X, SCREEN_SIZE_Y, 16);
    if (DxLib_Init() == -1) return -1; // 初期化と裏画面化
    SetDrawScreen(DX_SCREEN_BACK);

    return 0;
}

void GameTask::GameMainLoop()
{
    DrawString(0, 0, "MAIN", 0xffff00);
}
```