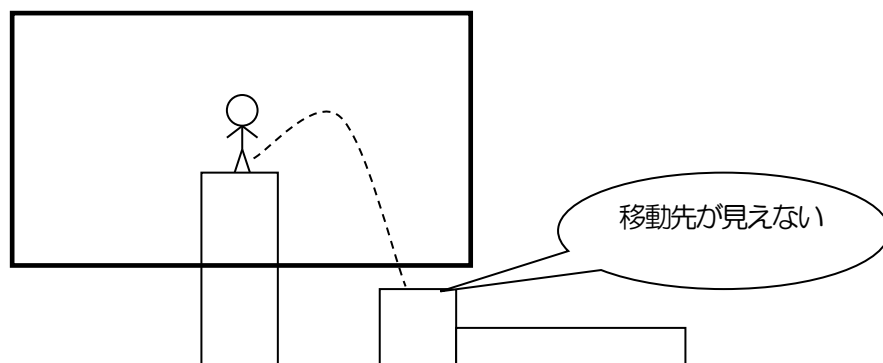


1. プレイヤを中心としないカメラ制御

1. 1. プレイヤをカメラの注視対象にする事の問題点

基本的にはプレイヤキャラクタを画面の中心に表示するというのは問題ありませんが、それで全ての状況に対応できるわけではありません。

例えば「高いところへ登る」「低いところへ飛び降りる」等の状況では、移動先が見えなくて不都合になる可能性もあります。



また、イベントシーンなどでプレイヤ以外のキャラクタを画面中央に表示する必要性が生じるかもしれません。

このような状況を考えるとスクロール、つまりカメラの制御はプレイヤとは独立したものである方が都合がよいのです。

ただし、基本的にはプレイヤを基準にする事に変わりはありません。

プレイヤをカメラのターゲットにするのではなく、プレイヤを追従する「何か」をカメラのターゲットにすると考えてください。

この「何か」は普段はプレイヤに追従していて、例えば「右スティックを操作したときだけ自由に動く」、イベントシーンではプレイヤ以外でも「指定されたキャラクタに追従する」という制御が出来ればよいわけです。

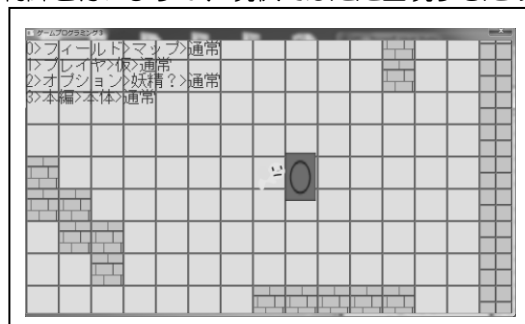
この追従する「何か」は、別に目に見えるものである必要はありません。

(もちろん、作成中は見えた方が解りやすいですが)

1. 2. プレイヤに追従する妖精?の追加

まずは、プレイヤに追従する妖精的なキャラクタを追加しましょう。

最終的にこれがカメラの制御を行います、現状ではただ出現するだけです。



まず、プロジェクトフォルダを用意しましょう。サーバーから1章初期状態のプロジェクトをコピーするか、GPG2の16章終了時点のプロジェクトを用意してください。

妖精をプレイヤーに追従させる為に、追尾対象のポインタを入れておくための変数 `target` を `BChara` クラスに追加します。

`BChara . h` 追加変数のみ抜粋

```
WP          target;
//省略しなかった場合 weak_ptr<BChara> target;
```

これは `weak_ptr` と呼ばれるもので、追尾対象が存在するか否かを安全に確認したうえでアクセスすることが出来る仕組みを提供してくれます。実際にはポインタではなくクラスです。以降のプログラムでこの変数がどう使われているか注目してください。

追加タスク (BChara 継承)

ファイル名:	<code>Task_Sprite.h</code> 及び <code>.cpp</code>
ファイルタイトル:	かわいい妖精
ネームスペース名:	<code>Sprite</code>
デフォルトグループ名:	"オブション"
デフォルトタスク名:	"妖精?"

追加変数 (Resource): `string` `imageName;`

追加変数 (Object): 無し

追加メソッド: 無し

`Task_Sprite . cpp` 追加・変更部分のみ抜粋

```
//-----
//リソースの初期化
bool Resource::Initialize()
{
    this->imageName = "SpriteImg";
    DG::Image_Create(this->imageName, "./data/image/妖精.png");
    return true;
}
//-----
//リソースの解放
bool Resource::Finalize()
{
    DG::Image_Erase(this->imageName);
    return true;
}
//-----
//「初期化」タスク生成時に1回だけ行う処理
bool Object::Initialize()
{

```

```

//スーパークラス初期化
__super::Initialize(defGroupName, defName, true);
//リソースクラス生成 or リソース共有
this->res = Resource::Create();

//★データ初期化
this->render2D_Priority[1] = 0.5f;

//★タスクの生成

return true;
}
//-----
//「更新」1フレーム毎に行う処理
void Object::UpDate()
{
    //ターゲットが存在するか調べてからアクセス
    if (auto tg = this->target.lock()) {
        //ターゲットへの相対座標を求める
        ML::Vec2 toVec = tg->pos - this->pos;
        //ターゲットに5%近づく
        this->pos += toVec * 0.05f;
    }
}
//-----
//「2D描画」1フレーム毎に行う処理
void Object::Render2D_AF()
{
    ML::Box2D draw(-16, -16, 32, 32);
    draw.Offset(this->pos);
    ML::Box2D src(0, 0, 32, 32);

    draw.Offset(-ge->camera2D.x, -ge->camera2D.y);
    DG::Image_Draw(this->res->imageName, draw, src, ML::Color(0.5f, 1, 1, 1));
}

```

UpDate メソッドの網掛け部に着目しましょう。

weak_ptr である target から追尾対象に直接アクセスすることはできず、lock()メソッドを介して有効なポインタ（実際は shared_ptr）を受け取ります。

このとき、対象が既に消滅しているなら NULL が帰ります。

それを判断基準とすることで、安全に対象にアクセスできるようになるわけです。

そうして得た追尾対象と自分の間の距離を5%だけ詰めることで追尾させています。

これにより妖精はプレイヤーの動きに緩急つける形で追従するようになります。

最後に、妖精の生成処理を追加して、追従することを確認しましょう。

Task_Game . cpp 初期化処理内 追加部分のみ抜粋

```
//妖精の生成
auto spr = Sprite::Object::Create(true);
spr->pos = p1->pos;
spr->target = p1;
```

変数 p1 はプレイヤーキャラを示す変数です。

(当然このコードはプレイヤーの生成・登録処理より下に書きます)

1. 3. カメラのターゲットを妖精に変更する

プレイヤーの Update() 処理に描かれているカメラ制御（オフセット値の制御）を妖精に移し替えます。

Task_Player . cpp 対象部分のみ抜粋

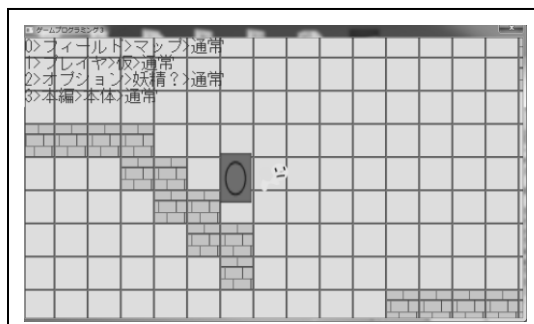
```
//-----
//「更新」1フレーム毎に行う処理
void Object::Update()
{
    以下省略・・・

    //足元接触判定
    this->hitFlag = this->CheckFoot();

    //カメラの位置を再調整
    {
        //自分を画面の何処に置くか（今回は画面中央）
        int px = ge->camera2D.w / 2;
        int py = ge->camera2D.h / 2;
        //自分を画面中央に置いた時のカメラの左上座標を求める
        int cpx = int(this->pos.x) - px;
        int cpy = int(this->pos.y) - py;
        //カメラの座標を更新
        ge->camera2D.x = cpx;
        ge->camera2D.y = cpy;
    }
    //マップ外を写さないようにする調整処理
    auto map = ge->GetTask_One_GN<Map2D::Object>("フィールド", "マップ");
    if (nullptr != map) {
        map->AjustCameraPos();
    }
}
```

この部分を妖精に移植する。
(プレイヤーからは消す)

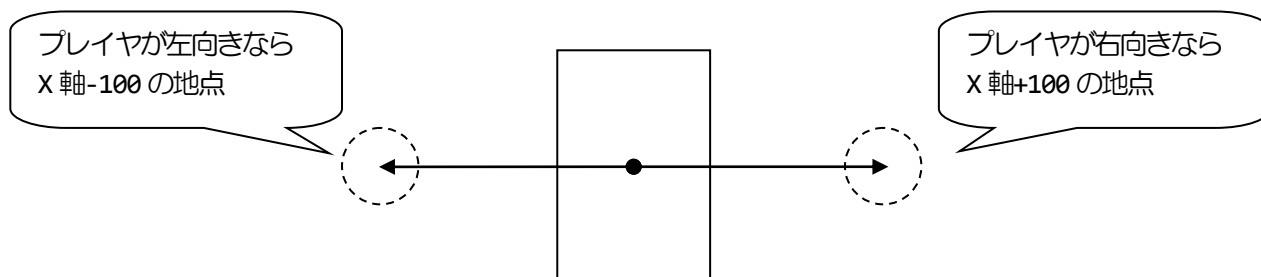
妖精タスクの.cpp ファイルに Task_Map2D.h をインクルードする事を忘れなければ、この部分のコピペで画面の中央に表示される対象がプレイヤーから妖精?にかわります。



妖精はプレイヤーに遅れてついてくるため、プレイヤーの動きにスピード感が増しますが、そのままでは進行方向の映像が見えづらくなるため、ゲームプレイには不都合が生じます。自分の作るゲームに合わせて調整が必要になるでしょう。

課題1-1

妖精が追従する地点をプレイヤーの座標ではなく、プレイヤーの正面 100 離れた位置になるように変更せよ。



2. キャラクタの制御（アクション系）

2. 1. 入力に応じるキャラクタ

キャラクタを制御する際に「右キーが入力されたらキャラクタを右方向に歩かせる」というような言い方をしますが、この考え方ではキャラクタをうまく制御できません。
制御できない理由は「キャラクタの状態」を無視している所にあります。
例えばキャラクタが落下中に方向キーを押してもキャラクタが歩くことは出来ません。
このようにキャラクタは「状態」によって入力に対応できない・別の対応をしなければならない場合が多々あります。

自然な形で動くキャラクタを作りたいなら、「思考」「状態」「状況」を条件に加える必要があるのです。

「思考」とは「〇〇しよう」と考える意識、プレイヤキャラクタにとっての「ユーザー入力」と考えましょう。

「状態」とは、立っているとか、座っているとか、歩いている等、キャラクタ状態です。

「状況」とは地面の有る無しや、狭い隙間に挟まっているとか、水中にいる等の環境です。

これらを加味して、キャラクタは「入力に応じて動こうとする」と考えてください。

2. 2. 状態を主体にした制御


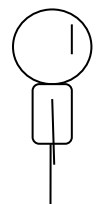

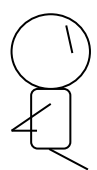
アクション系のキャラクタは「歩行」「停止」「ジャンプ」「攻撃」等様々な状態を持っています。
その状態を主体にしてプログラムを作り、キャラクタを制御しようというのが「状態を主体にした制御」です。

キーが押されたから動くのではなく、
今「停止」状態であるなら、方向キーが押されたら「歩行」状態に切り替える。

キーが離されたから止まるのではなく、
「歩行」状態の時方向キーが離されたら「停止」状態に切り替える。

ジャンプボタンが押されたから飛びあがるのではなく、
ジャンプ出来る状態（歩行や停止等）である時、ジャンプボタンが押されたら、「ジャンプ」状態に切り替える。
操作とは関係なく、状態が「ジャンプ」だから飛びあがると考えます。

処理の流れは、まず状態による分岐を経てから入力への対応という形になります。

状態毎の 入力への 対応	停止中 左右キー入力 で歩行に変更 ジャンプキー 入力で屈む	歩行中 左右キー解除 で停止に変更 ジャンプキー 入力で屈む	屈み中 全入力無効	
				上昇中 全入力無効 (左右キーで移動する 場合もある)
状態毎の 入力に関係 しない動作	停止中 動かない	歩行中 向いている方 に移動する。	屈み中 数フレーム後 上昇に変更	上昇中 上昇量を減衰させる。 0になったら落下に変更

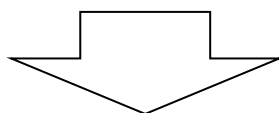
2. 3. キャラクタ制御の流れ

キャラクタの制御（基本的な行動に対する制御）は以下のように考えることができます。

状態（モーション）の決定

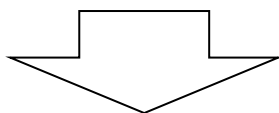
「キャラクタの現在の状態（モーション）」、「ユーザーの入力」、「周囲の状況」を見て、次のモーションを求めます。

ここでは、次のモーションを決定する事だけを目的にします。



状態（モーション）に対応した処理

現在のモーションに対応した処理を行います。歩行中であれば、移動速度を加速させる。落下中なら重力加速を行う、攻撃中なら経過時間に合わせて弾を発射するなど、現在のモーションに応じた処理を行います。



移動

前述の処理で決定した移動速度や上昇・落下量を元に予定移動量（ベクトル）を決定し、移動を行います。（この移動は、位置を変える事だけを目的とする）

敵や罠からダメージを受ける（モーションが変化する）処理はここには含めません。

これらの処理は、受ける側が主体になって行う処理ではなく、「与える側が主体になって行う処理」だからです。（自分から殴られているか確認するって変ですよね？）


```

};
Motion      motion;      // 現在の行動を示すフラグ
int          animCnt;     // アニメーションカウンタ
float        jumpPow;     // ジャンプ初速
float        maxFallSpeed; // 落下最大速度
float        gravity;     // フレーム単位の加算量
float        maxSpeed;    // 左右方向への移動の加算量
float        addSpeed;    // 左右方向への移動の加算量
float        decSpeed;    // 左右方向への移動の減衰量

//メンバ変数に最低限の初期化を行う
//★★メンバ変数を追加したら必ず初期化も追加する事★★
BChara()
    : pos(0, 0)
    , hitBase(0, 0, 0, 0)
    , moveVec(0, 0)
    , moveCnt(0)
    , angle_LR(Right)
    , motion(Stand)
    , jumpPow(0)
    , maxFallSpeed(0)
    , gravity(0)
    , maxSpeed(0)
    , addSpeed(0)
    , decSpeed(0)
{
}
virtual ~BChara(){}

//キャラクタ共通メソッド
//めり込まない移動処理
virtual void CheckMove(ML::Vec2& est_);
//足元接触判定
virtual bool CheckFoot();
//頭上接触判定
virtual bool CheckHead();
//モーションを更新（変更なしの場合 false）
bool UpdateMotion(Motion nm_);

// アニメーション情報構造体
struct DrawInfo {
    ML::Box2D    draw, src;
    ML::Color    color;
};
};

```

キャラクタの状態を表すフラグ及び各種パラメータが追加されています。
追加メソッドとして、頭上判定と、モーションの更新時に使用するメソッドが追加されています。

Task_Player . h Object クラス 追加メソッドのみ抜粋

```
//思考&状況判断(ステータス決定)
void Think();
//モーションに対応した処理
void Move();
//アニメーション制御
BChara::DrawInfo Anim();
```

2.3節で挙げた「モーションの決定」と「モーションに対応した処理」をそれぞれメソッドとして追加しています。

Task_Player . cpp 該当箇所のみ抜粋

```
//-----
//「初期化」タスク生成時に1回だけ行う処理
bool Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成 or リソース共有
    this->res = Resource::Create();

    //★データ初期化
    this->render2D_Priority[1] = 0.5f;
    this->hitBase = ML::Box2D(-16, -40, 32, 80);
    this->angle_LR = Right;
    this->controllerName = "P1";
    this->motion = Stand;            //キャラ初期状態
    this->maxSpeed = 5.0f;           //最大移動速度（横）
    this->addSpeed = 1.0f;           //歩行加速度（地面の影響である程度打ち消される）
    this->decSpeed = 0.5f;           //接地状態の時の速度減衰量（摩擦）
    this->maxFallSpeed = 10.0f;      //最大落下速度
    this->jumpPow = -10.0f;          //ジャンプ力（初速）
    this->gravity = ML::Gravity(32) * 5; //重力加速度&時間速度による加算量
    //★タスクの生成
    return true;
}

//-----
//「更新」1フレーム毎に行う処理
void Object::UpDate()
{
    this->moveCnt++;
    this->animCnt++;
    //思考・状況判断
    this->Think();
    //現モーションに対応した制御
    this->Move();
```

```
//めり込まない移動  
ML::Vec2 est = this->moveVec;  
this->CheckMove(est);  
}
```

先ほど挙げた、「モーションを決める処理」と「モーションに合わせて行う処理」を Update()メソッド内で呼び出しています。

Task_Player . cpp 該当箇所のみ抜粋

```
//-----
//思考&状況判断 モーション決定
void Object::Think()
{
    BChara::Motion nm = this->motion; //とりあえず今の状態を指定

    //思考（入力）や状況に応じてモーションを変更する事を目的としている。
    //モーションの変更以外の処理は行わない
    switch (nm) {
    case Stand: //立っている
        if (this->CheckFoot() == false) { nm = Fall; } //足元 障害 無し
        break;
    case Walk: //歩いている
        break;
    case Jump: //上昇中
        break;
    case Fall: //落下中
        break;
    case Attack: //攻撃中
        break;
    case TakeOff: //飛び立ち
        break;
    case Landing: //着地
        break;
    }
    //モーション更新
    this->UpdateMotion(nm);●-----
```

こちらはモーションを決定する処理、現時点で実装しているのは「立っている」時、足元に地面が無ければ、「落下中」に切り替える処理のみです。
だから実行した時、プレイヤーの状態は落下に切り替わり落ちていきますが、それ以降モーションが変化しません。 このメソッドは、あくまでも次のモーションを決める処理として扱います。
(今のモーションを継続するという選択も含めます)

```
//-----
//モーションを更新（変更なしの場合 false）
bool BChara::UpdateMotion(Motion nm_)
{
    if (nm_ == this->motion) {
        return false;
    }
    else {
        this->motion = nm_; //モーション変更
    }
}
```

```

    this->moveCnt = 0;    //行動カウンタクリア
    this->animCnt = 0;    //アニメーションカウンタのクリア
    return true;
}
}

```

モーションカウンタは、そのモーションになってからの経過時間を表す変数なので、モーション変更時には必ず0に戻す必要がある。

モーションフラグは直接書き換えるのではなく、このメソッドを介して書き換えます。

```

//-----
// モーションに対応した処理
//(モーションは変更しない)
void Object::Move()
{
    //重力加速
    switch (this->motion) {
    default:
        //上昇中もしくは足元に地面が無い
        if (this->moveVec.y < 0 ||
            this->CheckFoot() == false) {
            this->moveVec.y = 1.0f;//仮処理
        }
        //地面に接触している
        else {
            this->moveVec.y = 0.0f;
        }
        break;
        //重力加速を無効化する必要があるモーションは下に case を書く（現在対象無し）
    case Unnon:    break;
    }

    //移動速度減衰
    switch (this->motion) {
    default:
        //未実装
        break;
        //移動速度減衰を無効化する必要があるモーションは下に case を書く（現在対象無し）
    case Unnon:    break;
    }

//-----
//モーション毎に固有の処理
switch (this->motion) {
case Stand:    //立っている
    break;
case Walk:    //歩いている
    break;
case Fall:    //落下中

```

```

        break;
    case Jump:    //上昇中
        break;
    case Attack: //攻撃中
        break;
    }
}

```

モーションに応じて移動量を調節したり、弾を出したりする処理を行います。
今のところ、重力加速（それもまともに機能していない）以外の処理は実装されていません。
こちらのメソッドでは逆に、モーションの変更を禁じています。

アニメーションでは、現在のキャラクタの状態と経過時間に合わせて画像を選ぶ処理を行っています。
すでにいくつかのモーションにはアニメーション処理を組み込んであるので、モーションを変更するだけで画像が変わりますが、全てのモーションに対して設定してあるわけではないので注意して下さい。

```

//-----
//アニメーション制御
BChara::DrawInfo BChara::Anim()
{
    ML::Color dc(1, 1, 1, 1); //以下の文が教科書に入りきらないため
    BChara::DrawInfo imageTable[] = {
        //draw src
        { ML::Box2D(-16, -40, 32, 80), ML::Box2D(0, 0, 32, 80), dc}, //停止
        { ML::Box2D(-4, -40, 32, 80), ML::Box2D(32, 0, 32, 80), dc}, //歩行1
        { ML::Box2D(-20, -40, 48, 80), ML::Box2D(64, 0, 48, 80), dc}, //歩行2
        { ML::Box2D(-20, -40, 48, 80), ML::Box2D(112, 0, 48, 80), dc}, //歩行3
        { ML::Box2D(-24, -40, 48, 80), ML::Box2D(48, 80, 48, 80), dc}, //ジャンプ
        { ML::Box2D(-24, -40, 48, 80), ML::Box2D(96, 80, 48, 80), dc}, //落下
        { ML::Box2D(-24, -24, 48, 64), ML::Box2D(0, 80, 48, 64), dc}, //飛び立つ直前
        { ML::Box2D(-24, -24, 48, 64), ML::Box2D(144, 80, 48, 64), dc}, //着地
    };
    BChara::DrawInfo rtv;
    int work;
    switch (this->motion) {
    default:    rtv = imageTable[0]; break;
    // ジャンプ-----
    case Jump:  rtv = imageTable[4]; break;
    // 停止-----
    case Stand: rtv = imageTable[0]; break;
    // 歩行-----
    case Walk:
        work = this->animCnt / 8;
        work %= 3;
        rtv = imageTable[work + 1];
        break;
    // 落下-----
    case Fall:  rtv = imageTable[5]; break;
    }
}

```

```
}  
// 向きに応じて画像を左右反転する  
if (false == this->angle_LR) {  
    rtv.draw.x = -rtv.draw.x;  
    rtv.draw.w = -rtv.draw.w;  
    //// 表示位置の逆転  
    //rtv.draw.x = -(rtv.draw.x + rtv.draw.w);  
    //// 画像位置の逆転  
    //rtv.src.x = (rtv.src.x + rtv.src.w);  
    //rtv.src.w = -rtv.src.w;  
}  
return rtv;  
}
```

現状で細かい内容の理解は難しいので、キャラクタに動きを付けながら徐々に理解していきましょう。

2. 5. 着地の実装

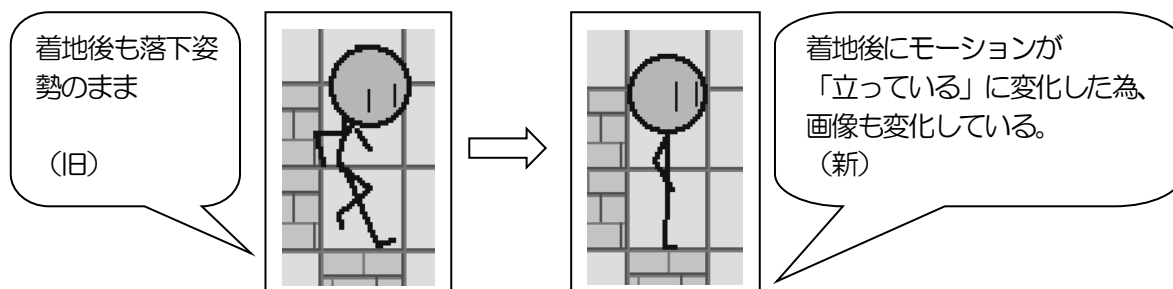
モーションが「落下中」のキャラクタが地面に接触した場合、着地をしたとみなし、状態を「立っている」に変更する処理を実装します。
これは、状況によりモーションを変更する処理なので、Think()メソッドに処理を加えます。

Task_Player . cpp 該当箇所のみ抜粋

```
//-----
//思考&状況判断 モーション決定
void Object::Think()
{
    BChara::Motion nm = this->motion; //とりあえず今の状態を指定

    //思考（入力）や状況に応じてモーションを変更する事を目的としている。
    //モーションの変更以外の処理は行わない
    switch (nm) {
    case Stand: //立っている
        if (this->CheckFoot() == false) { nm = Fall; } //足元 障害 無し
        break;
    case Walk: //歩いている
        break;
    case Jump: //上昇中
        break;
    case Fall: //落下中
        if (this->CheckFoot() == true) { nm = Stand; } //足元 障害 有り
        break;
    case Attack: //攻撃中
        break;
    case TakeOff: //飛び立ち
        break;
    case Landing: //着地
        break;
    }
    //モーション更新
    this->UpdateMotion(nm);
}
```

モーション「落下中」に対する処理として、着地判定とモーションの変更を組み込みました。
実行結果を確認すると、着地後の姿勢（画像）が変化しているのが確認できます。



2. 6. 重力加速の実装

落下速度が等速のままなので、重力加速を実装します。

こちらは、モーションに対する処理なので、Move()メソッドが対象です。

Task_Player . cpp 該当箇所のみ抜粋

```
//-----  
// モーションに対応した処理  
//(モーションは変更しない)  
void Object::Move()  
{  
    //重力加速  
    switch (this->motion) {  
    default:  
        //上昇中もしくは足元に地面が無い  
        if (this->moveVec.y < 0 ||  
            this->CheckFoot() == false) {  
            this->moveVec.y=min( this->moveVec.y + this->gravity,  
                                this->maxFallSpeed);  
        }  
        //地面に接触している  
        else {  
            this->moveVec.y = 0.0f;  
        }  
        break;  
        //重力加速を無効化する必要があるモーションは下に case を書く (現在対象無し)  
    case Unnon:    break;  
    }  
    以下省略・・・
```

「現在の落下速度」にフレーム単位の加速度を加算しています。

また、落下速度が最大速を超えて加速しない様にもしています。

実行確認して、落下速度の加速を確認しましょう。

2. 7. ジャンプの実装

「立っている」状態でジャンプボタンを押されたとき、「ジャンプ」する処理を実装します。
 こちらは、「モーションの変更」と「モーションに応じた処理」の双方に処理を追加します。

Task_Player . cpp Think()メソッド内 該当箇所のみ抜粋

```
case Stand: //立っている
    if (in.B1.down) { nm = Jump; }
    if (this->CheckFoot() == false) { nm = Fall; } //足元 障害 無し
    break;
```

ジャンプと落下が同時に発現する時、優先度が高いのは「落下」だと考えられます。

(ジャンプボタンが押され、足元に地面が無い瞬間)

優先度の高い方が残るように、処理の順番を意識しましょう。

Task_Player . cpp Move()メソッド内 該当箇所のみ抜粋

```
case Jump: //上昇中
    if (this->moveCnt == 0) {
        this->moveVec.y = this->jumpPow; //初速設定
    }
    break;
```

この状態でジャンプすると、たしかにジャンプします。

しかし、ジャンプから落下状態への切り替えが行われない為、上昇の姿勢のまま落下します。

着地後も上昇姿勢のまま（モーション上はジャンプのまま）で変化しないため、それ以降動けません。

もう一つ処理を追加して、「上昇中」→「落下中」→「立っている」という流れを実現しましょう。

Task_Player . cpp Think()メソッド内 該当箇所のみ抜粋

```
case Fall: //落下中
    if (this->CheckFoot() == true) { nm = Stand; }
    break;
```

「上昇中」に移動ベクトルY軸が+に転じた場合、「落下中」に切り替える処理を追加しました。

2. 8. 歩行の実装

「立っている」状態で左右キーを押されたとき、「歩行中」に切り替える処理を実装します。
更に、「歩行中」から「立っている」に切り替える処理も実装します。

Task_Player . cpp Think()メソッド内 該当箇所のみ抜粋

```
case Stand: //立っている
    if (in.LStick.L.on) { nm = Walk; }
    if (in.LStick.R.on) { nm = Walk; }
    if (in.B1.down) { nm = Jump; }
    if (this->CheckFoot() == false) { nm = Fall; } //足元 障害 無し
    break;
case Walk: //歩いている
    if (in.LStick.L.off && in.LStick.R.off) { nm = Stand; }
    break;
```

方向キーの入力で「立っている」と「歩行中」が切り替わるのは確認できましたか？
確かに切り替わるのは確認できたでしょうが、その場から動かない・向きが変わらないという問題が残っています。

「歩行中」のモーションに対する処理として、横方向の移動量を更新する処理を Move()メソッドに実装しましょう。

Task_Player . cpp Move()メソッド内 該当箇所のみ抜粋

```
case Walk: //歩いている
    if (in.LStick.L.on) {
        this->angle_LR = Left;
        this->moveVec.x = -this->maxSpeed;
    }
    if (in.LStick.R.on) {
        this->angle_LR = Right;
        this->moveVec.x = this->maxSpeed;
    }
    break;
```

「歩行中」というモーションの時は、左右入力に対して「向きを変える」・「移動ベクトルを設定する」という応じ方をしていると考えて下さい。

いちいち2つのメソッドに処理を分けて書くのは面倒だと感じるかもしれませんが、ルールを守っておく方が後で得なはずですよ。

これでキャラクタは入力した方向に動くようになりましたが、また新たな問題が出てきました。
方向キーを離して「立っている」状態に戻っても、キャラクターが移動し続けます。
移動ベクトルのX軸方向に減衰を加えて停止させましょう。

Task_Player . cpp Move()メソッド内 該当箇所のみ抜粋

```

//移動速度減衰
switch (this->motion) {
default:
    if (this->moveVec.x < 0) {
        this->moveVec.x = min(this->moveVec.x + this->decSpeed, 0);
    }
    else {
        this->moveVec.x = max(this->moveVec.x - this->decSpeed, 0);
    }
    break;
    //移動速度減衰を無効化する必要があるモーションは下に case を書く（現在対象無し）
case Unnon:    break;
}

```

移動ベクトルのX軸が0でないとき、0に近づけていく処理を実装しました。
イメージ的には、地面との摩擦で減速していると考えてもらえばよいでしょう。

ここまでの内容で、キャラクタがいろいろと動くようになりました。
しかし、動きが付いたことでいくつかおかしいところが見えてきていると思います。
これを直すことを課題とします。

課題2-1

歩行中でもジャンプ出来るように変更せよ。（横に跳べなくてよい）

課題2-2

歩行中に落下した場合、歩行の姿勢のままで落下する。
また、このときジャンプボタンを押すと、空中でジャンプを行ってしまう。
この問題を解消するために、歩行中の処理に、足元判定を加え、「落下中」に切り替える処理を追加せよ。

2. 9. ジャンプ・落下中も横方向移動に対応する

ジャンプ・落下中も横方向移動が出来るようにしましょう。
この時、キャラクタの向きは変えない事にします。(マリオ風?)



難しい話ではないので、ここまでの流れを理解しているなら自力で答えに辿りつける人もいるでしょう。
ソースコードは次のページにありますが、少し考えてみて下さい。

Task_Player . cpp Move()メソッド内 該当箇所のみ抜粋

```

    case Fall:    //落下中
        if (in.LStick.L.on) {
            this->moveVec.x = -this->maxSpeed;
        }
        if (in.LStick.R.on) {
            this->moveVec.x = this->maxSpeed;
        }
        break;

    case Jump:    //上昇中
        if (this->moveCnt == 0) {
            this->moveVec.y = this->jumpPow; //初速設定
        }
        if (in.LStick.L.on) {
            this->moveVec.x = -this->maxSpeed;
        }
        if (in.LStick.R.on) {
            this->moveVec.x = this->maxSpeed;
        }
        break;

```

モーションに応じた処理として、落下とジャンプ双方のモーションで左右キーの入力に合わせて歩行時と同様に移動ベクトルを与えるだけです。(向きは変えない)
これでジャンプアクションが自在に出来るようになりました。

課題2-3

- ジャンプ中に天井に衝突した場合、即落下に切り替える処理を実装せよ。
- * 上昇力を0にする事で、結果落下に切り替わると考える。
 - * 頭上の判定は BChara のメソッドとして実装してある

2. 10. 飛び立ちを追加する

現実の世界で考えれば、しゃがんで反動を付けなければ高くは跳べないはずなので、ジャンプする際にしゃがみの姿勢を挟んでからジャンプするようにしてみましょう。
ただし、これがゲームにとって良い結果を招くか否かは正直微妙です。
しゃがみが入る分だけタイミングがずれたり、テンポが悪くなったりするので、あくまで勉強の一環として捉えて下さい。

まず、飛び立ち (TakeOff) に対応したアニメーションの設定を行ってください。

Task_Player . cpp Anim()メソッド内 該当箇所のみ抜粋

```
// 飛び立つ直前-----
case TakeOff: rtv = imageTable[6]; break;
```

配布状態で、予めしゃがみ姿勢の矩形情報は設定済みなので、case 文を追加するだけです。
これで、プレイヤーの状態が TakeOff になればしゃがみ画像に代わる事が確認できるでしょう。

ジャンプボタンを押した時、モーションを「上昇中」に切り替える部分を「飛び立ち」に変更して下さい。(この時点ではジャンプが出来なくなります)

Task_Player . cpp Think()メソッド内 該当箇所のみ抜粋

```
if (in.B1.down) { nm = TakeOff; }
```

*現時点で2か所あります。

ジャンプボタンを押して「飛び立ち」(しゃがみ) 状態になる事が確認出来たら、30 フレーム後にジャンプに代わる処理を加えます。
当然、まともなアクションゲームとしては長すぎるしゃがみ時間ですが、動作確認の為に時間とを考えてください。

Task_Player . cpp Think()メソッド内 該当箇所のみ抜粋

```
case TakeOff: //飛び立ち
    if (this->moveCnt >= 30) {nm = Jump; }
    break;
```


2. 11. 着地硬直を追加する

着地後しばらく動けなくなる「着地硬直」を追加します。

これもやはり、飛び立ちと同じようにゲームのテンポを悪くする可能性はありますが、影響は飛び立ちよりも小さいといえるでしょう。

こちらでも動作確認のため、硬直時間を30フレームとします。

Task_Player . cpp Think()メソッド内 該当箇所のみ抜粋

```
case Jump:    //上昇中
    if (this->moveVec.y >= 0) { nm = Fall; }
    break;
case Fall:    //落下中
    if (this->CheckFoot() == true) { nm = Landing; }
    break;
case TakeOff: //飛び立ち
    if (this->moveCnt >= 30) {nm = Jump; }
    if (this->CheckFoot() == false) { nm = Fall; }//足元 障害 無し
    break;
case Landing: //着地
    if (this->moveCnt >= 30) { nm = Stand; }
    break;
```

着地硬直中の画像も用意してあります。

自分でアニメーション処理にプログラムを追加し、画像が表示されるようにしてください。

2. 12. 硬直中の落下を追加する

飛び立ちや着地の際にもキャラクタは横方向に移動しています。移動量は減衰しては行きますが、数ドットの移動は発生します。このとき、場合によっては足場から落下することがあります。特に飛び立ちの時は、そのあとジャンプで上昇してしまうこともあり、動きに違和感が大きく出ます。

これを実際に動作確認したのち（飛び立ちからの落下だけでよい）、以下のコードを追加して、飛び立ちと着地中の落下を実装してください。

Task_Player . cpp Think()メソッド内 追加内容のみ抜粋

```
if (this->CheckFoot() == false) { nm = Fall; }//足元 障害 無し
```

先ほどと同様に動作確認して、落下に切り替わる事を確認したら、飛び立ちと着地の硬直時間を3フレームに変更してください。

（確認が難しいときは、速度の減衰量（this->decSpeed）を小さくして、摩擦を弱めてみましょう）

課題2-SP1

ジャンプボタンを押している時間でジャンプ力が変わるように作り替えよ。

- まず、UpdateMotion()メソッドを以下のように変更する。(必要な追加変数は自分で宣言を加える)

```
//-----
//モーションを更新(変更なしの場合 false)
bool BChara::UpdateMotion(Motion nm_)
{
    if (nm_ == this->motion) {
        return false;
    }
    else {
        this->preMotion = this->motion;
        this->preMoveCnt = this->moveCnt;
        this->motion = nm_; //モーション変更
        this->moveCnt = 0; //行動カウンタクリア
        this->animCnt = 0; //アニメーションカウンタのクリア
        return true;
    }
}
```

- *この変更により、1つ前のモーションがなんであったか、またそのモーションが何フレーム目で終了したかを知ることが出来るようになる。
- 「跳び立ち」状態から「ジャンプ」に移る条件は、30フレーム経過するかジャンプボタンを離れた時とする。
- ジャンプ力の増加量は、「飛び立ち」状態のフレーム数の10分の1とする。
(28フレームであれば、ジャンプの初速に2.8fを上乗せする)

課題2-SP2

二段ジャンプを実装せよ。(SP1のジャンプ力変化の内容は引き継がない)

- 2つ目のジャンプと落下のモーション(Jump2 Fall2)を追加する。
- 2回目のジャンプの初速は、通常のジャンプの8割とする。
- 2回目のジャンプ発動の条件は、ジャンプ&落下中にもう一度ジャンプボタンが押された時とする。
- 3段以降のジャンプはしてはならない。

課題2-SP3

横方向の移動速度を徐々に加速するように作り替えよ。

- 現在歩行中などの処理では、「this->moveVec.x = this->maxSpeed;」というように、移動速度変数に対し、いきなり最高速を設定している為、徐々に加速する状態にはなっていない。加速量用の変数はメンバ変数「this->addSpeed」として用意してあるのでこれを利用し、徐々に加速していくように変更せよ。

加速量が1.0f、減衰量が0.5fで設定されているので、1フレーム当たりの加速量は0.5fとなる。最高速が5.0fなので、歩行開始から10フレーム後に最高速に達する。最高速を超えないように制限を加えることも忘れないように。

3. 敵キャラクタを実装

3. 1. プレイヤと敵の違い

プレイヤキャラクタは、ゲームプレイヤ(あなた)が画面を見て状況を判断しボタンを介して操作します。敵キャラクタはこの部分をプログラムで肩代わりするわけです。これを「AI」と呼んでいるわけですが、本来「AI」はもっと上等なモノの事を指します。「行動ルーチン」と呼ぶ方が妥当でしょう。

「行動ルーチン」が選んだ動きをコントローラのボタン ON/OFF に置き換える事が出来るのであれば、人が操作をするのと同じ仕組みで NPC を操作できるようになります。(そこまではやりませんか)

3. 2. 敵キャラクタの実装

プレイヤと同じ方法で敵キャラクタを実装します。とりあえず、動かない敵キャラクタをステージ上に出現させるところまで実装しましょう。(大枠はプレイヤと同じ)

追加タスク (BChara 継承)

ファイル名: Task_Enemy00.h 及び .cpp
 ファイルタイトル: 敵 00 (スライム)
 ネームスペース名: Enemy00
 デフォルトグループ名: "敵"
 デフォルトタスク名: "NoName"

追加変数 (Resource): string imageName;

追加変数 (Object): 無し

追加メソッド: void Think(); //思考&状況判断(ステータス決定)
 void Move(); //モーションに対応した処理
 BChara::DrawInfo Anim(); //アニメーション制御

cpp 追加・変更部分のみ抜粋

```
//-----
//リソースの初期化
bool Resource::Initialize()
{
    this->imageName = "Enemy00Img";
    DG::Image_Create(this->imageName, "./data/image/Enemy00.png");
    return true;
}
//-----
//リソースの解放
bool Resource::Finalize()
{

```

```

        DG::Image_Erase(this->imageName);
        return true;
    }
    //-----
    //「初期化」タスク生成時に1回だけ行う処理
    bool Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
        //リソースクラス生成 or リソース共有
        this->res = Resource::Create();

        //★データ初期化
        this->render2D_Priority[1] = 0.6f;
        this->hitBase = ML::Box2D(-28, -22, 56, 45);
        this->angle_LR = Left;
        this->motion = Stand;
        this->maxSpeed = 2.0f;           //最大移動速度（横）
        this->addSpeed = 0.7f;          //歩行加速度（地面の影響である程度打ち消される）
        this->decSpeed = 0.5f;          //接地状態の時の速度減衰量（摩擦）
        this->maxFallSpeed = 10.0f;     //最大落下速度
        this->jumpPow = -6.0f;           //ジャンプ力（初速）
        this->gravity = ML::Gravity(32) * 5; //重力加速度&時間速度による加算量
        //★タスクの生成
        return true;
    }
    //-----
    //「更新」1フレーム毎に行う処理
    void Object::Update()
    {
        this->moveCnt++;
        this->animCnt++;
        //思考・状況判断
        this->Think();
        //現モーションに対応した制御
        this->Move();
        //めり込まない移動
        ML::Vec2 est = this->moveVec;
        this->CheckMove(est);
    }
    //-----
    //「2D描画」1フレーム毎に行う処理
    void Object::Render2D_AF()
    {
        BChara::DrawInfo di = this->Anim();
        di.draw.Offset(this->pos);
        //スクロール対応
        di.draw.Offset(-ge->camera2D.x, -ge->camera2D.y);
        DG::Image_Draw(this->res->imageName, di.draw, di.src);
    }

```

```

}
//-----
//思考&状況判断 モーション決定
void Object::Think()
{
    BChara::Motion nm = this->motion; //とりあえず今の状態を指定

    //思考（入力）や状況に応じてモーションを変更する事を目的としている。
    //モーションの変更以外の処理は行わない
    switch (nm) {
    case Stand: //立っている
        break;
    case Walk: //歩いている
        break;
    case Jump: //上昇中
        break;
    case Fall: //落下中
        break;
    case Attack: //攻撃中
        break;
    case TakeOff: //飛び立ち
        break;
    case Landing: //着地
        break;
    }
    //モーション更新
    this->UpdateMotion(nm);
}
//-----
// モーションに対応した処理
//(モーションは変更しない)
void Object::Move()
{
    //重力加速
    switch (this->motion) {
    default:
        //上昇中もしくは足元に地面が無い
        if (this->moveVec.y < 0 ||
            this->CheckFoot() == false) {
            this->moveVec.y =
                min(this->moveVec.y + this->gravity, this->maxFallSpeed);
        }
        //地面に接触している
        else {
            this->moveVec.y = 0.0f;
        }
        break;
        //重力加速を無効化する必要があるモーションは下に case を書く（現在対象無し）
    case Unnon: break;
    }
}

```

```

}

//移動速度減衰
switch (this->motion) {
default:
    if (this->moveVec.x < 0) {
        this->moveVec.x = min(this->moveVec.x + this->decSpeed, 0);
    }
    else {
        this->moveVec.x = max(this->moveVec.x - this->decSpeed, 0);
    }
    break;
    //移動速度減衰を無効化する必要があるモーションは下に case を書く (現在対象無し)
case Unnon:    break;
}
//-----
//モーション毎に固有の処理
switch (this->motion) {
case Stand:    //立っている
    break;
case Walk:     //歩いている
    break;
case Fall:     //落下中
    break;
case Jump:     //上昇中
    break;
case Attack:   //攻撃中
    break;
}
}
//-----
//アニメーション制御
BChara::DrawInfo  Object::Anim()
{
    ML::Color  dc(1, 1, 1, 1);
    BChara::DrawInfo  imageTable[] = {
        //draw          src
        { ML::Box2D(-32, -24, 64, 48), ML::Box2D( 0,  0, 64, 48), dc},    //停止
        { ML::Box2D(-32, -32, 64, 64), ML::Box2D(128, 48, 64, 64), dc},    //落下
    };
    BChara::DrawInfo  rtv;
    int  work;
    switch (this->motion) {
    default:      rtv = imageTable[0];  break;
        // ジャンプ-----
    case Jump:    rtv = imageTable[0];  break;
        // 停止-----
    case Stand:   rtv = imageTable[0];  break;
        // 歩行-----

```

```

    case Walk:   rtv = imageTable[0]; break;
                // 落下-----
    case Fall:   rtv = imageTable[1]; break;
    }
    // 向きに応じて画像を左右反転する
    if (false == this->angle_LR) {
        rtv.draw.x = -rtv.draw.x;
        rtv.draw.w = -rtv.draw.w;
    }
    return rtv;
}

```

プレイヤーキャラクタに実装した部分が抜けていたり、パラメータの設定値が異なる点を除けば、ほとんどプレイヤーのコピーです。
このタスクを生成して動作確認しましょう。

Task_Game . cpp Object::Create()追加・変更部分のみ抜粋

```

//敵の生成
for (int c = 0; c < 6; ++c) {
    auto ene = Enemy00::Object::Create(true);
    ene->pos.x = 500 + c * 100;
    ene->pos.y = 80;
}

```

これでスタート地点の右側に、6体のスライムが出現します。(スライムはその場から動きません)

3. 3. 落下・着地の実装

スライムに対して落下と着地の判定を実装します。
落下中のと着地後（立っている）で姿勢（画像）が変わる事を確認しておきましょう。

Task_Enemy00 . cpp Think()追加・変更部分のみ抜粋

```

case Stand: //立っている
    if (this->CheckFoot() == false) { nm = Fall; }
    break;
case Walk: //歩いている
    if (this->CheckFoot() == false) { nm = Fall; }
    break;
case Jump: //上昇中
    if (this->moveVec.y >= 0) { nm = Fall; }
    break;
case Fall: //落下中
    if (this->CheckFoot() == true) { nm = Stand; }
    break;
case Attack: //攻撃中
    break;
case TakeOff: //飛び立ち
    if (this->CheckFoot() == true) { nm = Stand; }
    break;

```

```
case Landing: //着地
    if (this->CheckFoot() == false) { nm = Fall; }
    break;
```

3. 4. 移動の実装

スライムが「立っている」状態にあるとき、「歩行中」に切り替えて現在の向きに進む処理を追加します。プレイヤーであればユーザーの入力により変化する部分ですが、スライムは独自の考えで動きます。（と言うほど考えてはいませんが）

Task_Enemy00 . cpp Move()追加・変更部分のみ抜粋

```
//-----
//モーション毎に固有の処理
switch (this->motion) {
case Stand: //立っている
    break;
case Walk: //歩いている
    if (this->angle_LR == Left) {
        this->moveVec.x =
            max(-this->maxSpeed, this->moveVec.x - this->addSpeed);
    }
    if (this->angle_LR == Right) {
        this->moveVec.x =
            min(+this->maxSpeed, this->moveVec.x + this->addSpeed);
    }
    break;
case Fall: //落下中
    break;
case Jump: //上昇中
    break;
case Attack: //攻撃中
    break;
}
```

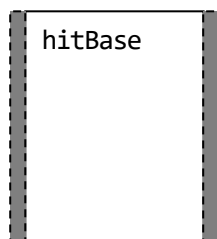
歩行状態であれば、徐々に加速しながら移動するようになります。
（加速時間が短いので、おそらく加速は感じ取れません）

課題3-1

スライムに歩行中壁と接触したら向きを変える能力を追加せよ。

- BChara クラスのメソッドに、正面方向1ドットの範囲であたり判定を行う機能を実装せよ。

```
//正面接触判定（サイドビューゲーム専用）
virtual bool CheckFront_LR();
```



キャラクタの向きに合わせて、どちらかの矩形を作る。
hitBase を元にして、横幅が1ドットの矩形を作る。

あとは、CheckFoot()と同じ

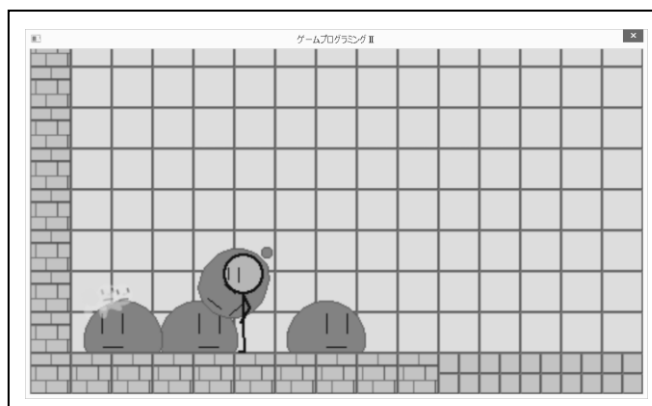
- 以下のコードを追加して、向きを変えさせる。（状態フラグ Turn は自分で追加する）

Task_Energy00 . cpp Think()追加・変更部分のみ抜粋

```
case Walk: //歩いている
    if (this->CheckFront_LR() == true) { nm = Turn; } //壁に衝突
    if (this->CheckFoot() == false) { nm = Fall; }
    break;
case Turn:
    if (this->moveCnt >= 5) { nm = Stand; }
    break;
```

Task_Energy00 . cpp Move()追加・変更部分のみ抜粋

```
case Turn:
    if (this->moveCnt == 3) {
        if (this->angle_LR == Left) { this->angle_LR = Right; }
        else{ this->angle_LR = Left; }
    }
    break;
```



壁に触れたスライムが方向転換するようになります。

（方向転換には5フレーム要して、向き自体は3フレーム目に変更する）

課題3-2

スライムが落下中（ジャンプ中）も横軸移動を行うように変更せよ。

＊歩行時と同じ速度で向きに合わせて移動すればよい。

課題3-SP1

壁にぶつかるだけでなく移動先に床が無い時も方向転換をする色違いのスライム Enemy01 を実装せよ。

- ・色違いの画像は自分で用意する。
- ・床がない事を確認するメソッドは以下の仕様で実装する事。（BChara に追加）

```
//正面足元チェック(サイドビューゲーム専用)  
virtual bool CheckFrontFoot_LR();
```



キャラクタの向きに合わせて、どちらかの矩形を作る。
hitBase を元にして、縦・横幅が1ドットの矩形を作る。

あとは、CheckFront_LR()と同じ