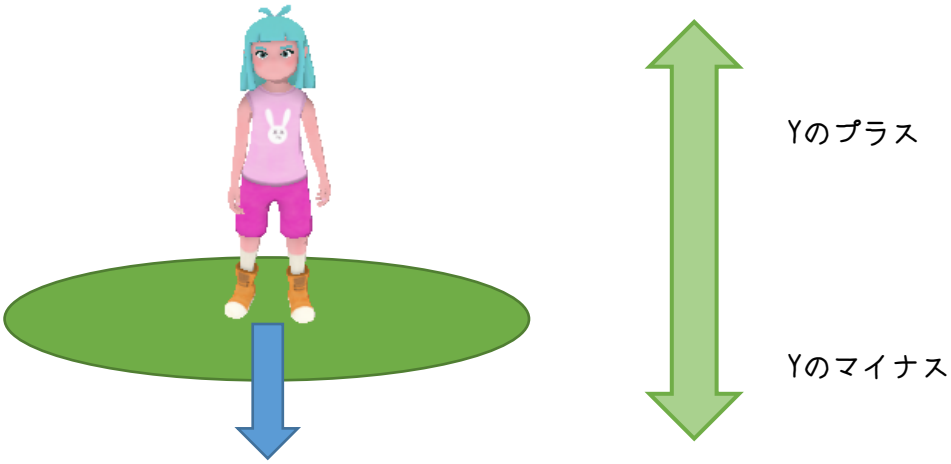
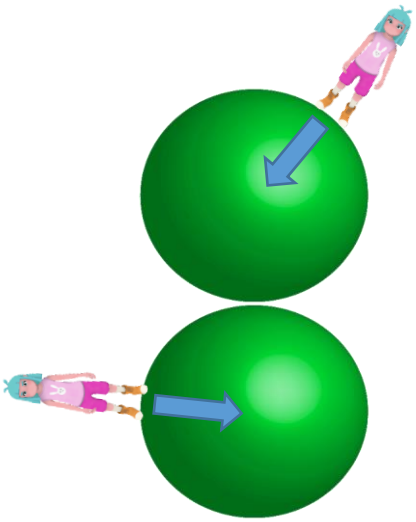
重力と衝突判定とジャンプ

キャラクターの移動と回転を行ってきましたが、
重力を実装していませんので、ずっと宙を浮いたままになっています。
そろそろ地面に足をつけて、フィールドを歩き回れるようにしていきます。



重力方向が変わらないゲームであれば、Yの値で高さを制御し、
重力方向は『 Yのマイナス値 』、ジャンプ方向は『 Yのプラス値 』で
実装することができます。

AsoGalaxyでは、前述のとおり、重力方向が変わるゲームです。



左の上図だと、{ -0.7, -0.7 } くらいが、
重力方向になり、

下図だと、{ 0.9, -0.1 } くらいが、
重力方向となります。

このようにAsoGalaxyは、固定化された重力方向を使用することができません。
重力マネージャーで、重力方向を計算して導き出し、
そのベクトル(方向)を使用していく必要があります。

GravityManager

```
// 重力方向を取得
VECTOR GetDirGravity(void);

// 重力方向の反対方向を取得
VECTOR GetDirUpGravity(void);

// 重力の強さを取得
float GetPower(void);
```

プレイヤーがいる惑星のタイプによって、重力方向や重力の強さが決まる。

その処理を、
GravityManagerで行う。

これらは、また後になって具体的に実装していきます。
今は、取り合えず重力と衝突判定を実装したいので、{ 0.0, -1.0, 0.0 }
方向に重力が発生するイメージで進めていきます。

地面との衝突判定

プレイヤーの座標(足元)を中心として、
ちょっと上から、ちょっと下あたりに、
線分を張って、モデルと線分の衝突判定を
行うと良いでしょう。

理由は、衝突するポリゴンが１つに限定され、
衝突後のプレイヤー座標の押し戻し処理が
安定するからです。

使用関数
　MV1CollCheck_Line

第１引数	モデルのハンドルID
第２引数	線分の始点座標
第３引数	線分の終点座標

Stage.cpp

```
void Stage::ChangeStage(NAME type)
{

    ～ 省略 ～

    // ステージの当たり判定をプレイヤーに設定
    player_.ClearCollider();
    player_.AddCollider(activePlanet_.lock()->GetTransform().collider);

    // 重力制御に惑星を渡す
    GravityManager::GetInstance().ChangeActivePlanet(activePlanet_);

    step_ = TIME_STAGE_CHANGE;

}
```

↑ ステージが変更されるとき、プレイヤーと衝突する惑星をリセットし、
新しいステージをのコライダをセットしている

Player.h

```
public:

    ～ 省略 ～

    // 衝突判定に用いられるコライダ制御
    void AddCollider(std::weak_ptr<Collider> collider);
    void ClearCollider(void);

private:

    // ジャンプ量
    VECTOR jumpPow_;

    // 衝突判定に用いられるコライダ
    std::vector<std::weak_ptr<Collider>> colliders_;

    // 衝突チェック
    VECTOR gravHitPosDown_;            ← 衝突用線分
```

```
    VECTOR gravHitPosUp_;               ← 衝突用線分

    ～ 省略 ～

    // 衝突判定
    void Collision(void);
    void CollisionGravity(void);

    // 移動量の計算
    void CalcGravityPow(void);
```

Player.cpp

```
Player::Player(void)
{

    ～ 省略 ～

    jumpPow_ = AsoUtility::VECTOR_ZERO;

    // 衝突チェック
    gravHitPosDown_ = AsoUtility::VECTOR_ZERO;
    gravHitPosUp_ = AsoUtility::VECTOR_ZERO;

}

void Player::AddCollider(std::weak_ptr<Collider> collider)
{
    colliders_.push_back(collider);
}

void Player::ClearCollider(void)
{
    colliders_.clear();
}

void Player::UpdatePlay(void)
{

    // 移動処理
    ProcessMove();
```

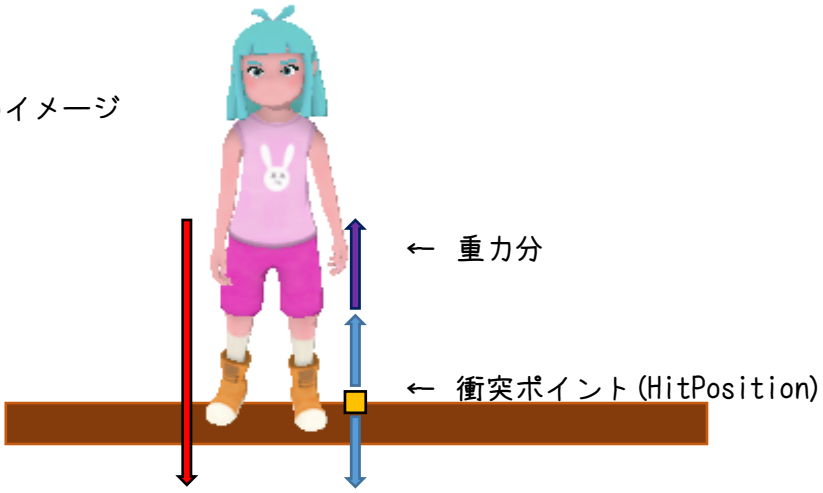
```
    // 移動方向に応じた回転
    Rotate();

    // 重力による移動量
    CalcGravityPow();

    // 衝突判定
    Collision();

    // 重力方向に沿って回転させる
    transform_.quaRot = grvMng_.GetTransform().quaRot;
    transform_.quaRot = transform_.quaRot.Mult(playerRotY_);

}

void Player::DrawDebug(void)
{

    ～ 省略 ～

    // 衝突
    DrawLine3D(gravHitPosUp_, gravHitPosDown_, 0x000000);

}

void Player::Collision(void)
{

    // 現在座標を起点に移動後座標を決める
    movedPos_ = VAdd(transform_.pos, movePow_);

    // 衝突(重力)
    CollisionGravity();

    // 移動
    transform_.pos = movedPos_;

}

void Player::CollisionGravity(void)
```

```
{

    // ジャンプ量を加算
    movedPos_ = VAdd(movedPos_, jumpPow_);

    // 重力方向
    VECTOR dirGravity = grvMng_.GetDirGravity();

    // 重力方向の反対
    VECTOR dirUpGravity = grvMng_.GetDirUpGravity();

    // 重力の強さ
    float gravityPow = grvMng_.GetPower();

    float checkPow = 10.0f;
    gravHitPosUp_ = VAdd(movedPos_, VScale(dirUpGravity, gravityPow));
    gravHitPosUp_ = VAdd(gravHitPosUp_, VScale(dirUpGravity, checkPow * 2.0f));
    gravHitPosDown_ = VAdd(movedPos_, VScale(dirGravity, checkPow));
    for (const auto c : colliders_)
    {

        // 地面との衝突
        auto hit = MV1CollCheck_Line(
            c.lock()->modelId_, -1, gravHitPosUp_, gravHitPosDown_);


        if (hit.HitFlag > 0)
        {

            // 地面と衝突している
            // 押し戻し処理とジャンプ力の打ち消しを実装しましょう

        }

    }

}

void Player::CalcGravityPow(void)
{
```

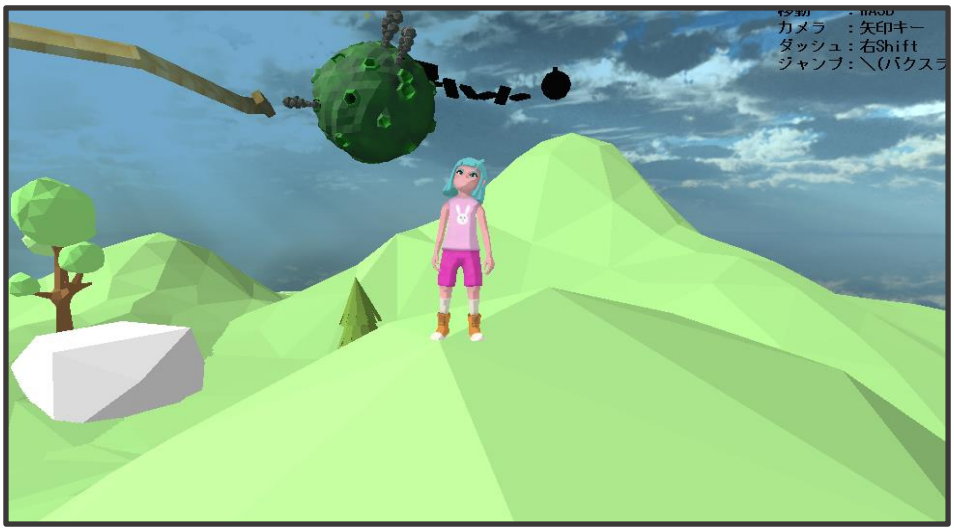
```
    // 重力方向
    VECTOR dirGravity = grvMng_.GetDirGravity();

    // 重力の強さ
    float gravityPow = grvMng_.GetPower();

    // 重力
    // 重力を作る
    // メンバ変数 jumpPow_ に重力計算を行う(加速度)

}
```

↑ 重力は加速度になりますので、計算に注意しながら、
プログラムを完成させてください

線分衝突と
押し戻しのイメージ



上手くいけば、地面に接地でき、山に登れるようになります。



ジャンプ処理

重力の反対方向が、ジャンプ方向になります。
これも重力マネージャーから方向を取得するように作ります。

Player.h

```
public:

    ～ 省略 ～

    // ジャンプ力
    static constexpr float POW_JUMP = 35.0f;

    // ジャンプ受付時間
    static constexpr float TIME_JUMP_IN = 0.5f;

    ～ 省略 ～

private:

    ～ 省略 ～

    // ジャンプ判定
    bool isJump_;

    // ジャンプの入力受付時間
    float stepJump_;

    // 丸影
    int imgShadow_;

    ～ 省略 ～

    void ProcessJump(void);

    // 着地モーション終了
    bool IsEndLanding(void);
```

Player.cpp

```
Player::Player(void)
{

    ～ 省略 ～

    isJump_ = false;
    imgShadow_ = -1;

}

void Player::Init(void)
{

    ～ 省略 ～

    // 丸影画像
    imgShadow_ = resMng_.Load(
        ResourceManager::SRC::PLAYER_SHADOW).handleId_;

    ～ 省略 ～

}

void Player::Draw(void)
{

    ～ 省略 ～

    // 丸影描画
    DrawShadow();

    ～ 省略 ～

}
```

```
void Player::UpdatePlay(void)
{

    // 移動処理
    ProcessMove();

    // 移動方向に応じた回転
    Rotate();

    // ジャンプ処理
    ProcessJump();

    ～ 省略 ～

}

void Player::DrawShadow(void)
{

    // 丸影の描画※

}

void Player::ProcessMove(void)
{

    ～ 省略 ～

    if (!AsoUtility::EqualsVZero(dir) && (isJump_ || IsEndLanding())) {

        // 移動処理
        speed_ = SPEED_MOVE;
        if (ins.IsNew(KEY_INPUT_RSHIFT))
        {
            speed_ = SPEED_RUN;
        }
        moveDir_ = dir;
        movePow_ = VScale(dir, speed_);
```

```
        // 回転処理
        SetGoalRotate(rotRad);

        if (!isJump_ && IsEndLanding())
        {
            // アニメーション
            if (ins.IsNew(KEY_INPUT_RSHIFT))
            {
                animationController_->Play((int)ANIM_TYPE::FAST_RUN);
            }
            else
            {
                animationController_->Play((int)ANIM_TYPE::RUN);
            }
        }

    }
    else
    {
        if (!isJump_ && IsEndLanding())
        {
            animationController_->Play((int)ANIM_TYPE::IDLE);
        }
    }

}

void Player::ProcessJump(void)
{

    bool isHit = CheckHitKey(KEY_INPUT_BACKSLASH);

    // ジャンプ
    if (isHit && (isJump_ || IsEndLanding()))
    {

        if (!isJump_)
        {
            // 制御無しジャンプ
            mAnimationController->Play((int)ANIM_TYPE::JUMP);
```

```
            // この後、いくつかのジャンプパターンを試します

        }

        isJump_ = true;

        // ジャンプの入力受付時間をへラス
        stepJump_ += scnMng_.GetDeltaTime();
        if (stepJump_ < TIME_JUMP_IN)
        {
            jumpPow_ = VScale(grvMng_.GetDirUpGravity(), POW_JUMP);
        }

    }

    // ボタンを離したらジャンプ力に加算しない
    if (!isHit)
    {
        stepJump_ = TIME_JUMP_IN;
    }

}

void Player::CollisionGravity(void)
{

    ～ 省略 ～

    for (const auto c : colliders_)
    {

        // 地面との衝突
        auto hit = MV1CollCheck_Line(
            c.lock()->modelId_, -1, gravHitPosUp_, gravHitPosDown_);

        // 最初は上の行のように実装して、木の上に登ってしまうことを確認する
        //if (hit.HitFlag > 0)
        if (hit.HitFlag > 0 && VDot(dirGravity, jumpPow_) > 0.9f)
        {

            // 衝突地点から、少し上に移動
```

```
            movedPos_ = VAdd(hit.HitPosition, VScale(dirUpGravity, 2.0f));

            // ジャンプリセット
            jumpPow_ = AsoUtility::VECTOR_ZERO;
            stepJump_ = 0.0f;

            if (isJump_)
            {
                // 着地モーション
                animationController_->Play(
                    (int)ANIM_TYPE::JUMP, false, 29.0f, 45.0f, false, true);
            }

            isJump_ = false;

        }

    }

}

void Player::CalcGravityPow(void)
{

    ～ 省略 ～

    // 内積
    float dot = VDot(dirGravity, jumpPow_);
    if (dot >= 0.0f)
    {
        // 重力方向と反対方向(マイナス)でなければ、ジャンプ力を無くす
        jumpPow_ = gravity;
    }

}

bool Player::IsEndLanding(void)
{

    bool ret = true;
```

```
    // アニメーションがジャンプではない
    if (animationController_->GetPlayType() != (int)ANIM_TYPE::JUMP)
    {
        return ret;
    }

    // アニメーションが終了しているか
    if (animationController_->IsEnd())
    {
        return ret;
    }

    return false;

}
```

ジャンプ実装の注意点

> ジャンプアニメーションは、
> 上昇、下降と最低２種類に分けて！

ジャンプの量が変わるということは、時間が変わるということです。
時間が変われば、アニメーションの時間も変わります。
速度を調整するという手もありますが、上昇・下降の時間比率が
一緒だとは限りません。
上昇中は、上昇アニメーション。
下降中は、下降アニメーションを再生しましょう。
（できれば、着地も分けて再生したい）

Mixamoサイトでは、上昇・下降を分けてくれている
アニメーションもありますが、それが無い場合もあります。
また、私達はプログラマーですので、３Ｄソフトを使って、
モーションデザインに手を出すことができません。

そういった場合は、繋がったアニメーションのうち、
自分の都合の良い部分を再生できるようなプログラムを作りましょう。

■ 制御無しジャンプ
```
animationController_->Play((int)ANIM_TYPE::JUMP);
```

■ ループしないジャンプ
```
animationController_->Play((int)ANIM_TYPE::JUMP, false);
```

■ 切り取りアニメーション
```
animationController_->Play(
                    (int)ANIM_TYPE::JUMP, false, 13.0f, 24.0f);
```

■ 無理やりアニメーション
```
animationController_->Play(
                    (int)ANIM_TYPE::JUMP, true, 13.0f, 25.0f);
animationController_->SetEndLoop(23.0f, 25.0f, 5.0f);
```

→ 13～25フレームを再生して、上昇アニメーション風にする
そして、上記の再生が終わったら、
23～25フレームをスピード5で再生し、下降モーション風にする

CollisionGravity関数にて、ジャンプ中に接地条件を満たしたら、

■ 着地アニメーション
```
animationController_->Play(
          (int)ANIM_TYPE::JUMP, false, 29.0f, 45.0f, false, true);
```

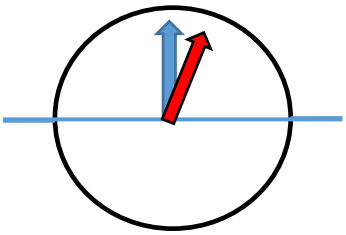
29～45フレームを再生して、着地アニメーション風にしている。
ループしない、JUMPステートが重なるので、強制アニメーションONに。

内積の特徴

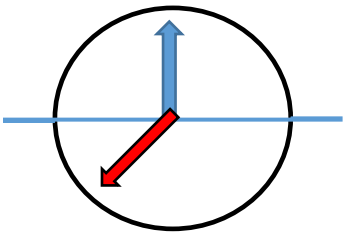
AsoGalaxyでの上昇方向、下降方向の判断は、
Yのプラスマイナスではなくて、内積で判断する！

内積の性質として、２つのベクトルが同じ方向を向いていたらプラス、
片方が反対の方向を向いていたら、マイナスになります。

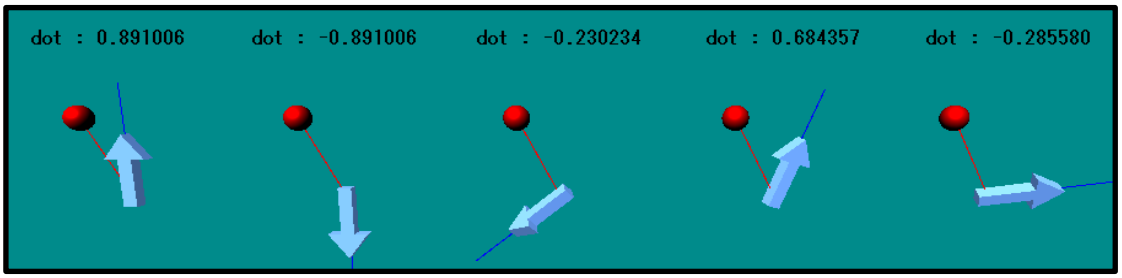
VDot(青ベクトル、赤ベクトル)

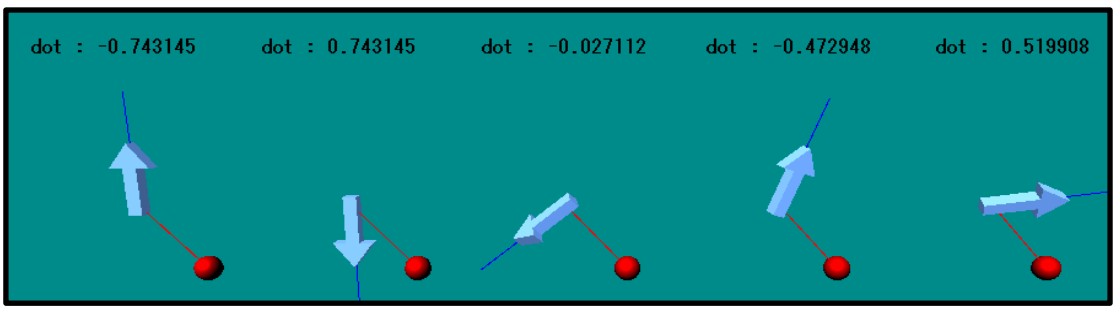


プラスになる。



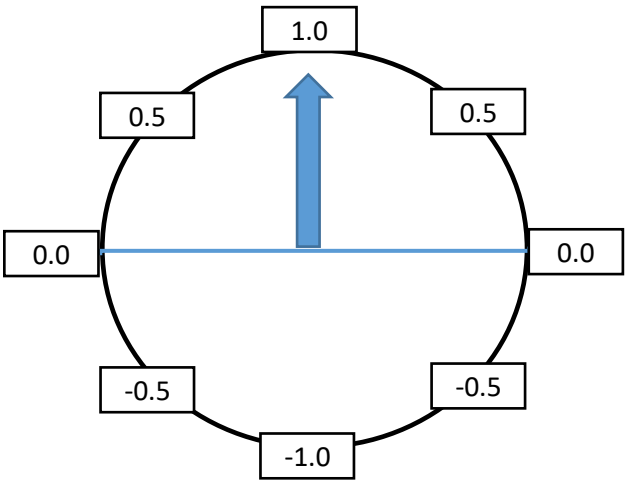
マイナスになる。





更に、渡すベクトルが単位ベクトルの場合、
-1.0 ～ +1.0 の値になる。

単位ベクトル同士の内積



ということは、

VDot(重力ベクトル、ジャンプの移動ベクトル)
VDot(dirGravity、jumpPow_)

この内積結果が、0.0以上であれば、下降中、という判断になります。
逆にマイナスであれば、上昇中です。

なぜこの解説を行ったかといいますと、

CalcGravityPow関数にて、とあるセーフ処理を実装しています。

```
// 内積
float dot = VDot(dirGravity, jumpPow_);
if (dot >= 0.0f)
{
    // 重力方向と反対方向(マイナス)でなければ、ジャンプ力を無くす
    jumpPow_ = gravity;
}
```

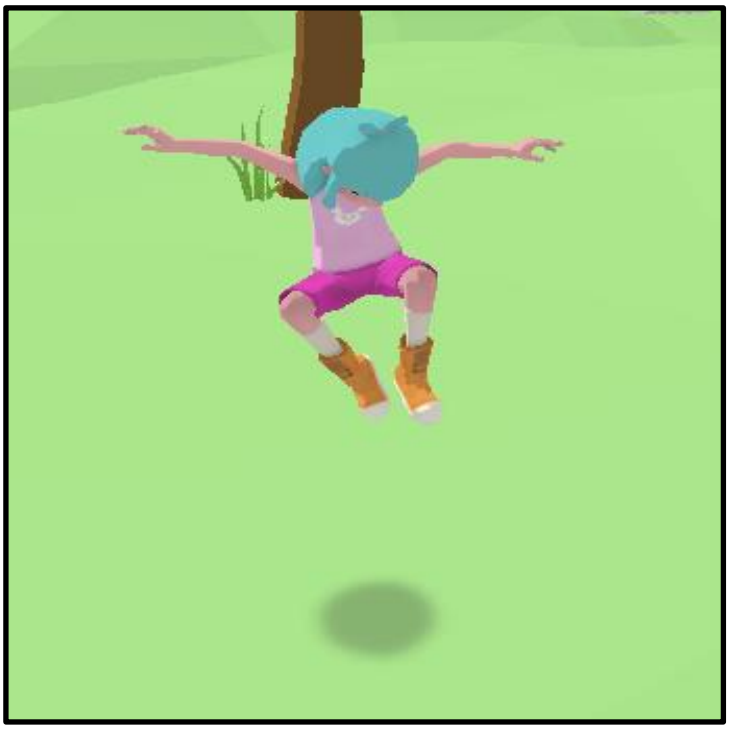
重力方向と、ジャンプ力の内積を取って、
キャラクターが上昇中なのか、下降中なのかを判断して、
下降中なら、重力を加速度にせず固定にしています。

地面との衝突チェックで使用している
線分が短いため、加速度で値が大きくなった
jumpPow_ だと移動量が大き過ぎて、
地面を突き抜けて、落下してしまうからです。

線分を長くすると、また別のバランスが崩れますので、
下降中は、重力の値が上がり過ぎないように
制御しています。

※移動前と移動後で線分チェックを行えば、
　モデルの突き抜け自体は解消できます

丸影の実装



ジャンプの実装と対になるのが、影の実装です。
影があるのとないのでは、キャラクターの位置把握に雲泥の差が出ます。
ユーザにとっても、開発者がデバッグするにしても、影はあった方が良いです。

モデル形状に合わせた影の実装も、
簡易的な影の実装も、DxLibの公式サイトにて解説されていますので、
そちらを参考に実装していきましょう。

■ ３Ｄのサンプルプログラム

https://dxlib.xsrv.jp/program/dxprogram_3D.html

■ 丸影の実装

https://dxlib.xsrv.jp/program/dxprogram_3DAction_CollObj.html

Player_ShadowRender関数

取り合えずコピペしてきて、今回のプロジェクト用に
実装を合わせていきましょう。

丸影の画像については、

ResourceManager::SRC::PLAYER_SHADOW

リソース管理で準備していますので、そちらを使用ください。