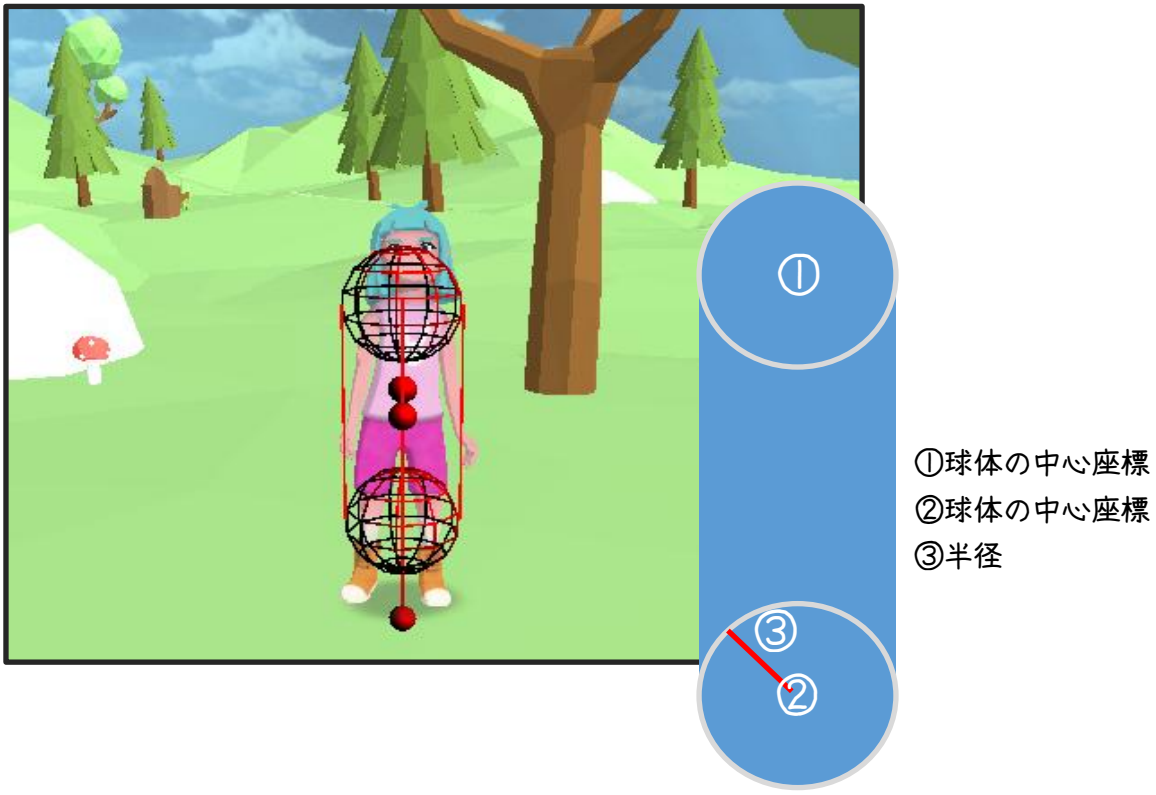
カプセルコライダで壁や木との衝突

地面との衝突判定だけではなく、
壁や木にも衝突判定を行うため、キャラクターで良く使用されるカプセル
コライダを作成していきます。

カプセルコライダのメリット・デメリット

メリット　：　球体と比較して、柔軟な形状を作ることができる。
例としては、縦長のカプセル。
キャラクターの形状に酷似しており、球体では表現できない。

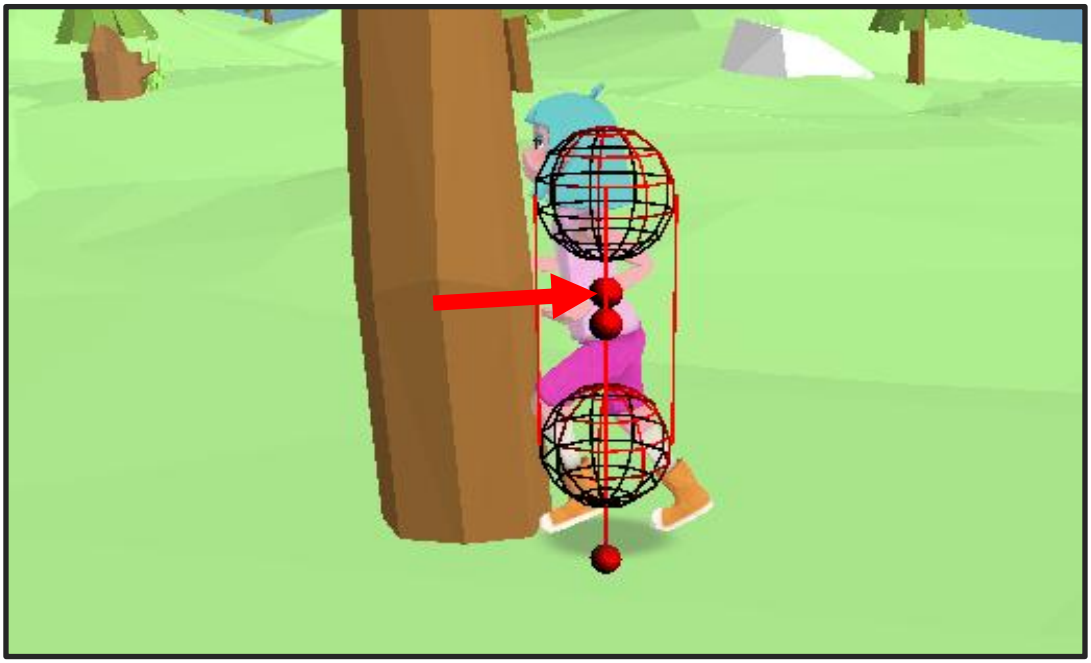
デメリット　：　球体よりは、複雑な衝突判定になる。
但し、モデルよりかは遥かに早い。



①球体の中心座標
②球体の中心座標
③半径

カプセルの形状は、
球体の中心座標２つと、半径の３つ要素で構成することができる。

カプセルとステージのモデルを衝突させて、押し戻す。



ColliderCapsule.h

```
#pragma once
#include <DxLib.h>
#include "ColliderBase.h"
class Transform;

class ColliderCapsule : public ColliderBase
{

public:

    // コンストラクタ
    ColliderCapsule(
        TAG tag, const Transform* follow,
        const VECTOR& localPosTop, const VECTOR& localPosDown, float radius);

    // デストラクタ
    ~ColliderCapsule(void);

    // 親Transformからの相対位置を取得
    const VECTOR& GetLocalPosTop(void) const;
```

```
    const VECTOR& GetLocalPosDown(void) const;

    // 親Transformからの相対位置をセット
    void SetLocalPosTop(const VECTOR& pos);
    void SetLocalPosDown(const VECTOR& pos);

    // ワールド座標を取得
    VECTOR GetPosTop(void) const;
    VECTOR GetPosDown(void) const;

    // 半径
    float GetRadius(void) const;
    void SetRadius(float radius);

    // 高さ
    float GetHeight(void) const;

    // カプセルの中心座標
    VECTOR GetCenter(void) const;
protected:

    // デバッグ用描画
    void DrawDebug(int color) override;

private:

    // 親Transformからの相対位置(上側)
    VECTOR localPosTop_;

    // 親Transformからの相対位置(下側)
    VECTOR localPosDown_;

    // 半径
    float radius_;

};
```

ColliderCapsule.cpp

```
#include <DxLib.h>
#include "../Common/Transform.h"
```

```
#include "ColliderCapsule.h"

ColliderCapsule::ColliderCapsule(
    TAG tag, const Transform* follow,
    const VECTOR& localPosTop, const VECTOR& localPosDown, float radius)
    :
    ColliderBase(SHAPE::CAPSULE, tag, follow),
    localPosTop_(localPosTop),
    localPosDown_(localPosDown),
    radius_(radius)
{
}

ColliderCapsule::~ColliderCapsule(void)
{
}

const VECTOR& ColliderCapsule::GetLocalPosTop(void) const
{
    return localPosTop_;
}

const VECTOR& ColliderCapsule::GetLocalPosDown(void) const
{
    return localPosDown_;
}

void ColliderCapsule::SetLocalPosTop(const VECTOR& pos)
{
    localPosTop_ = pos;
}

void ColliderCapsule::SetLocalPosDown(const VECTOR& pos)
{
    localPosDown_ = pos;
}

VECTOR ColliderCapsule::GetPosTop(void) const
{
    return GetRotPos(localPosTop_);
}
```

```
VECTOR ColliderCapsule::GetPosDown(void) const
{
    return GetRotPos(localPosDown_);
}

float ColliderCapsule::GetRadius(void) const
{
    return radius_;
}

void ColliderCapsule::SetRadius(float radius)
{
    radius_ = radius;
}

float ColliderCapsule::GetHeight(void) const
{
    return localPosTop_.y;
}

VECTOR ColliderCapsule::GetCenter(void) const
{
    VECTOR top = GetPosTop();
    VECTOR down = GetPosDown();

    VECTOR diff = VSub(top, down);
    return VAdd(down, VScale(diff, 0.5f));
}

void ColliderCapsule::DrawDebug(int color)
{

    // 上の球体
    VECTOR pos1 = GetPosTop();
    DrawSphere3D(pos1, radius_, 5, color, color, false);

    // 下の球体
    VECTOR pos2 = GetPosDown();
    DrawSphere3D(pos2, radius_, 5, color, color, false);
```

```
    VECTOR dir;
    VECTOR s;
    VECTOR e;

    // 球体を繋ぐ線(X+)
    dir = follow_->GetRight();
    s = VAdd(pos1, VScale(dir, radius_));
    e = VAdd(pos2, VScale(dir, radius_));
    DrawLine3D(s, e, color);

    // 球体を繋ぐ線(X-)
    dir = follow_->GetLeft();
    s = VAdd(pos1, VScale(dir, radius_));
    e = VAdd(pos2, VScale(dir, radius_));
    DrawLine3D(s, e, color);

    // 球体を繋ぐ線(Z+)
    dir = follow_->GetForward();
    s = VAdd(pos1, VScale(dir, radius_));
    e = VAdd(pos2, VScale(dir, radius_));
    DrawLine3D(s, e, color);

    // 球体を繋ぐ線(Z-)
    dir = follow_->GetBack();
    s = VAdd(pos1, VScale(dir, radius_));
    e = VAdd(pos2, VScale(dir, radius_));
    DrawLine3D(s, e, color);

    // カプセルの中心
    DrawSphere3D(GetCenter(), 5.0f, 10, color, color, true);

}
```

Player.h

```
private:

    ～ 省略 ～

    // 衝突判定用カプセル上部球体
    static constexpr VECTOR COL_CAPSULE_TOP_LOCAL_POS = { 0.0f, 110.0f, 0.0f };
```

```
    // 衝突判定用カプセル下部球体
    static constexpr VECTOR COL_CAPSULE_DOWN_LOCAL_POS = { 0.0f, 30.0f, 0.0f };

    // 衝突判定用カプセル球体半径
    static constexpr float COL_CAPSULE_RADIUS = 20.0f;

    ～ 省略 ～
```

Player.cpp

```
void Player::InitCollider(void)
{

    ～ 省略 ～

    // 主に壁や木などの衝突で仕様するカプセルコライダ
    ColliderCapsule* colCapsule = new ColliderCapsule(
        ColliderBase::TAG::PLAYER, &transform_,
        COL_CAPSULE_TOP_LOCAL_POS, COL_CAPSULE_DOWN_LOCAL_POS,
        COL_CAPSULE_RADIUS);
    ownColliders_.emplace(static_cast<int>(COLLIDER_TYPE::CAPSULE), colCapsule);

}
```

CharactorBase.h

```
public:

    // 衝突判定種別
    enum class COLLIDER_TYPE
    {
        LINE,
        CAPSULE,
        MAX,
    };

    ～ 省略 ～

protected:
```

```
    // 最大落下速度
    static constexpr float MAX_FALL_SPEED = -30.0f;

    // 衝突時の押し戻し試行回数
    static constexpr int CNT_TRY_COLLISION = 20;

    // 衝突時の押し戻し量
    static constexpr float COLLISION_BACK_DIS = 1.0f;

    ～ 省略 ～

    void CollisionCapsule(void);
```

CharactorBase.cpp

```
void CharactorBase::Collision(void)
{

    // 移動処理
    transform_.pos = VAdd(transform_.pos, movePow_);

    // 衝突(カプセル)
    CollisionCapsule();

    // ジャンプ量を加算
    transform_.pos = VAdd(transform_.pos, jumpPow_);

    // 衝突(重力)
    CollisionGravity();

}

void CharactorBase::CollisionCapsule(void)
{

    // カプセルコライダ
    int capsuleType = static_cast<int>(COLLIDER_TYPE::CAPSULE);

    // カプセルコライダが無ければ処理を抜ける
    if (ownColliders_.count(capsuleType) == 0) return;
```

```
// カプセルコライダ情報
ColliderCapsule* colliderCapsule =
    dynamic_cast<ColliderCapsule*>(ownColliders_.at(capsuleType));
if (colliderCapsule == nullptr) return;

// 登録されている衝突物を全てチェック
for (const auto& hitCol : hitColliders_)
{

    // モデル以外は処理を飛ばす
    if (hitCol->GetShape() != ColliderBase::SHAPE::MODEL) continue;

    // 派生クラスへキャスト
    const ColliderModel* colliderModel =
        dynamic_cast<const ColliderModel*>(hitCol);

    if (colliderModel == nullptr) continue;

    auto hits = MV1CollCheck_Capsule(
        colliderModel->GetFollow()->modelId, -1,
        colliderCapsule->GetPosTop(), colliderCapsule->GetPosDown(),
        colliderCapsule->GetRadius());

    // 衝突した複数のポリゴンと衝突回避するまで、
    // プレイヤーの位置を移動させる
    for (int i = 0; i < hits.HitNum; i++)
    {

        auto hit = hits.Dim[i];

        // 地面と異なり、衝突回避位置が不明なため、何度か移動させる
        // この時、移動させる方向は、移動前座標に向いた方向であったり、
        // 衝突したポリゴンの法線方向だったりする
        for (int tryCnt = 0; tryCnt < CNT_TRY_COLLISION; tryCnt++)
        {

            // 再度、モデル全体と衝突検出するには、効率が悪過ぎるので、
            // 最初の衝突判定で検出した衝突ポリゴン1枚と衝突判定を取る
            int pHit = HitCheck_Capsule_Triangle(
                colliderCapsule->GetPosTop(), colliderCapsule->GetPosDown(),
                colliderCapsule->GetRadius(),
```

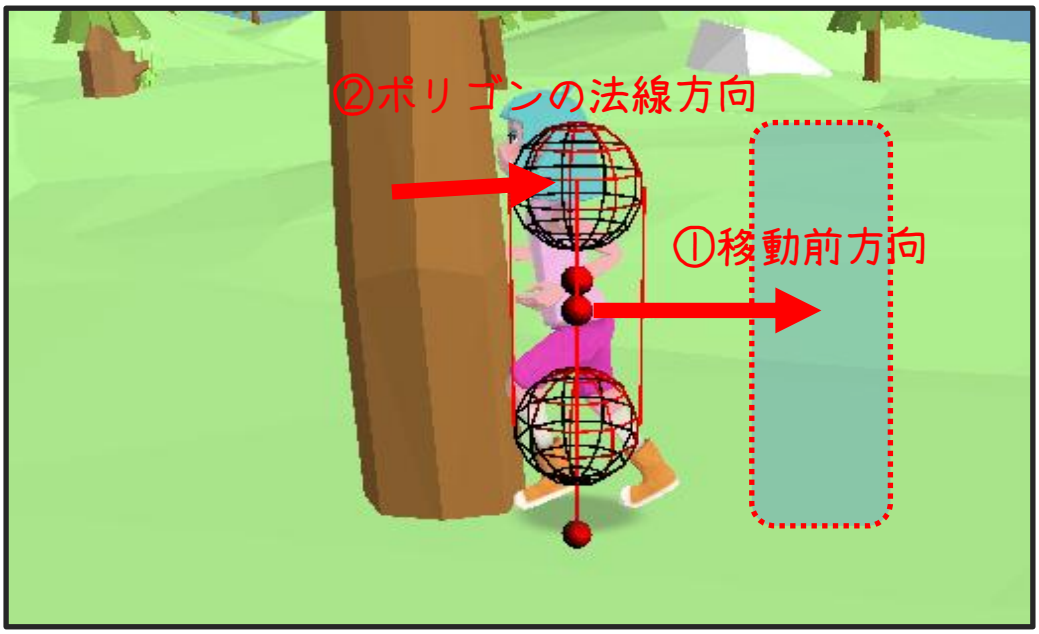
```
                hit.Position[0], hit.Position[1], hit.Position[2]);

            if (pHit)
            {
                // 法線の方向にちょっとだけ移動させる
                transform_.pos =
                    VAdd(transform_.pos,
                        VScale(hit.Normal, COLLISION_BACK_DIS));
                continue;
            }

            break;

        }

    }

    // 検出した地面ポリゴン情報の後始末
    MV1CollResultPolyDimTerminate(hits);

  }

}
```

解説

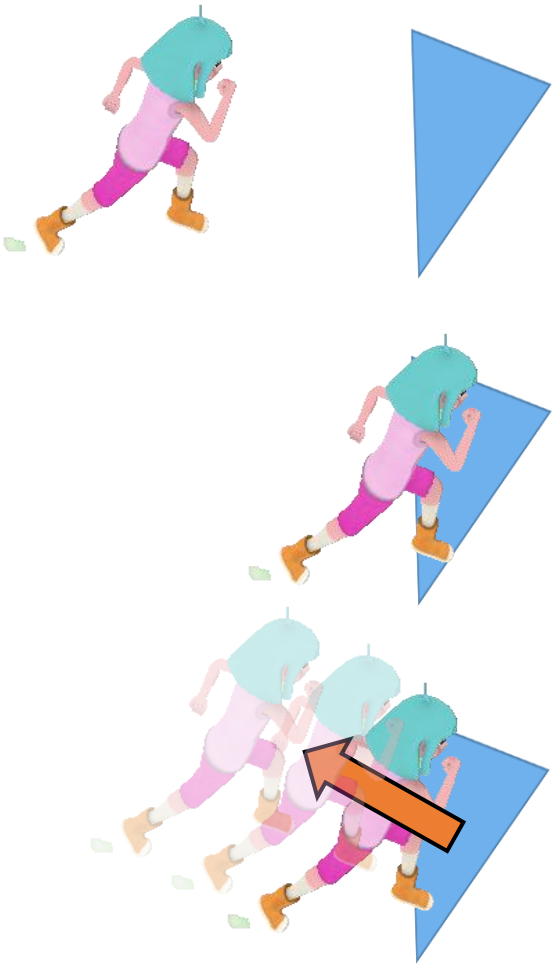
○ 押し戻し処理の詳細(CollisionCapsule関数)

DxLibのMV1CollCheck_Capsule関数で、
衝突したか否か、衝突座標を取得することができますが、
どれくらい押し戻しを行って、衝突しない位置にキャラクターを
動かすかは自分達で決めないといけません。

一般的には、衝突したモデルポリゴンの法線方向か、
移動前に戻す方向にキャラクターを移動させ、押し戻します。



②ポリゴンの法線方向



移動前方向だと、衝突していない地点へと戻り易い反面、
移動量が無くなってしまうため、壁を伝いながら移動するのが
やりづらいですが、法線方向だと、壁伝い移動がやりやすいです。

今回は、ポリゴンの法線方向に戻します。

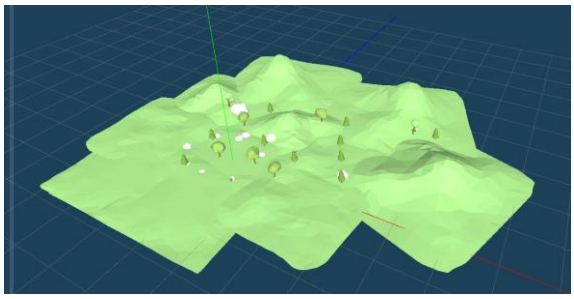
衝突したポリゴンの法線方向は、
DxLibの衝突構造体の中にある Normal 変数から取得することができます。

○ 衝突判定の効率化

押し戻す方向は、上記の説明で理解できたかと思いますが、
〝押し戻す量〟が、まだわかりません。

最もシンプルな方法は、
できるだけ細かい単位で移動させ、再度衝突していないか
チェックする方法です。
衝突しなくなるまで、細かい移動を繰り返します。

ただ、毎回、MV1CollCheck_Capsule関数を使用してしまうと、
モデル全体と何度も衝突判定をしないといけなくなり、処理負荷が
高くなります。




```
auto hits = MV1CollCheck_Capsule(
    colliderModel->GetFollow()->modelId, -1,
    colliderCapsule->GetPosTop(), colliderCapsule->GetPosDown(),
    colliderCapsule->GetRadius());
```

このコードだと、モデル内の全フレームの全ポリゴンと衝突判定を
行なおうとするため、17,000ポリゴンの処理が必要になります。

一方、下記のコードは、1枚のポリゴンとの衝突判定を行うため、

```
int pHit = HitCheck_Capsule_Triangle(
    colliderCapsule->GetPosTop(), colliderCapsule->GetPosDown(),
    colliderCapsule->GetRadius(),
    hit.Position[0], hit.Position[1], hit.Position[2]);
```

単純に前者の17,000分の1の処理速度で実行できます。

モデル全てではなく、フレームを限定したり、
ポリゴンを限定できないか、常に考えるようにしましょう。

○ オブジェクト指向的な衝突処理は、できている？

Playerクラスにかなりの量の衝突処理をベタ書きしていますので、
あまりできていないです。
追々、対応していきます。