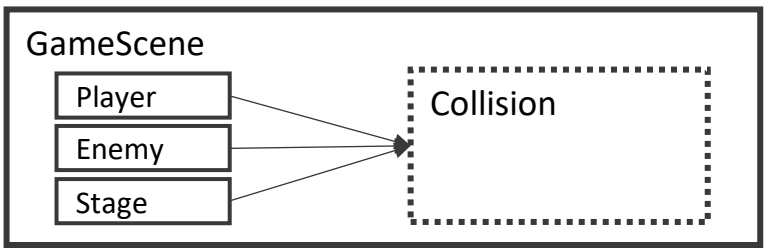
オブジェクト指向的な衝突設計

衝突の判定や管理は、ゲーム内容によって最適解が異なります。
これまでに授業で取り扱った衝突の管理方法も含めて、いくつか例に上げます。

① GameSceneでPlayerやStageの情報を集めて処理する

難易度★☆☆☆☆



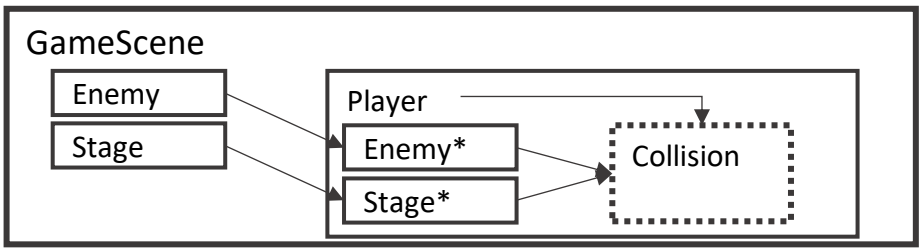
GameSceneは、ゲーム全体の進行を管理するための中心的なクラスであり、
Player・Stage・Enemyなど全てのオブジェクトを直接保持しているため、
衝突判定などの情報をまとめて扱いやすい。

押し戻しやダメージ処理も、各クラスへパラメータ設定用の関数を
作れば良いので、規模が小さければ、プログラムが組みやすい。

メリット　：　構造がシンプルで、少人数開発やプロトタイプに向いている
デメリット　：　GameSceneが肥大化し、修正のたびにScene全体を
変更するケースが出てきてしまう。
処理を分散させるオブジェクト指向からも少し反する。
⇒処理の再利用性や保守性、作業分担がやりづらい

② Playerなどの各オブジェクトに衝突判定対象のオブジェクトを渡す

難易度★★☆☆☆



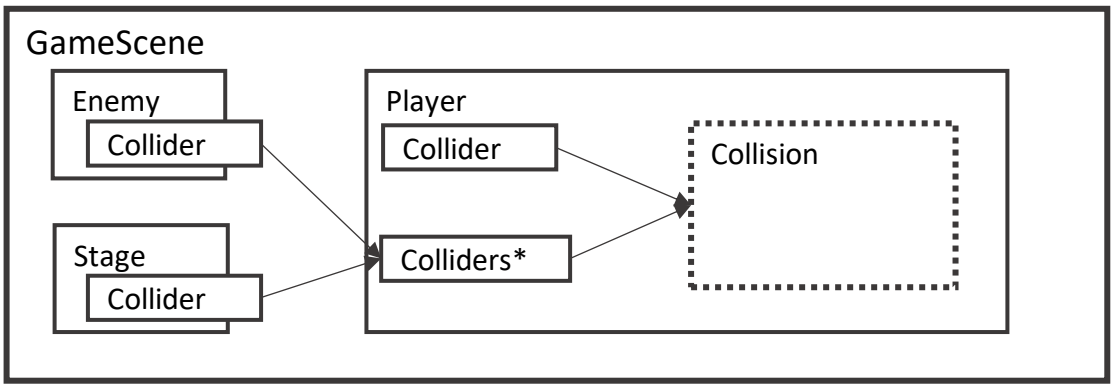
GameScene内での衝突処理は無くなり、オブジェクトごとに処理が分散されるようになります。衝突対象となるオブジェクトを渡す際に、Stage、Enemy1、2… と量が増えるのが手間だと思いがちですが、それらのオブジェクトの基底クラスの配列を渡せば良いですので、事前のオブジェクト指向プログラミングがしっかりできていれば拡張もしやすくなります。

また、衝突後の押し戻しやダメージ、状態変更も、自身のクラスに対する処理になるため、とてもやりやすいです。

メリット　：　衝突後の処理が作りやすい
デメリット　：　各オブジェクトで同様の衝突処理を実装する必要があり、処理の重複が起きやすい
オブジェクトごとの依存関係が複雑になり、循環参照に注意が必要。

③ 衝突判定、形状を管理するクラスを作り、その情報だけ各オブジェクトに渡す

難易度★★★★☆



Unity同様に球体コライダ、カプセルコライダのような、
コンポーネントクラスを作り、衝突判定自体はどのクラス同士で行います。
各オブジェクトの基底クラスActor全体を渡すわけではありませんので、
カプセル化も考慮され、比較的安全な設計になりやすいです。

また、各オブジェクトは「Collider」コンポーネントを持つ構造にすることで、
衝突判定の処理をオブジェクト本体から分離できます。
これにより、Actorは「何をするか」、Colliderは「どう当たるか」というように
処理を明確に分けられ、再利用性や安全性が向上します。

こういった設計を「コンポーネント志向」といいます。

メリット　：　衝突判定処理がColliderクラスで完結するため、
　　コードの責務分離が明確になる。
デメリット　：　設計段階での理解コストが高い。
　　(抽象クラス、ダウンキャスト、参照関係など)
　　工夫しないと、衝突後の処理が組みづらい。

④ Colliderコンポーネントを専用マネージャーで管理し、衝突処理を完全にオブジェクトから切り離す

難易度★★★★☆

③を実施した上で、各Colliderの登録や衝突判定の更新処理を
CollisionManagerで行う。
10種類以上の衝突種別や大量のオブジェクトの種類がある場合は、
Colliderの登録や接続処理が多くなるので、
この処理自体も分離するため、管理クラスを作る。

上記のように、段階分けすることができますが、
無理に難易度の高い手法を選ばず、自身の理解度に合わせて、
ゲームを完成させることを目標に制作を行いましょう。

とはいえ、
③以降を独学で習得するのは、なかなか難しいですし、
CollisionManagerを作ることが良いわけではなく、
③のオブジェクト設計、コンポーネント設計がしっかりできるかが大切で、
それが習得できれば、他の攻撃や移動にも応用が効きますので、
少しずつ、③の設計・実装を進めていきたいと思います。

①Colliderクラスの準備

ColliderBase.h

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

class ColliderBase
{

public:

    // 形状
    enum class SHAPE
    {
        NONE,
        LINE,
        SPHERE,
        CAPSULE,
        MODEL,
    };

    // 衝突種別
    enum class TAG
    {
        STAGE,
        PLAYER,
    };

    // コンストラクタ
    ColliderBase(SHAPE shape, TAG tag, const Transform* follow);

    // デストラクタ
    virtual ~ColliderBase(void);

    // 描画
    void Draw(void);
```

```
    // 追従先の取得
    const Transform* GetFollow(void) const { return follow_; };

    // 追従先の再設定
    void SetFollow(Transform* follow);

    // 形状
    SHAPE GetShape(void) const { return shape_; }

    // 衝突種別
    TAG GetTag(void) const { return tag_; }

protected:

    // デバッグ表示の色
    static constexpr int COLOR_VALID = 0xff0000;
    static constexpr int COLOR_INVALID = 0xaaaaaa;

    // 形状
    SHAPE shape_;

    // 衝突種別
    TAG tag_;

    // 追従先
    const Transform* follow_;

    // 有効フラグ
    bool isValid_;

    // ローカル座標をワールド座標に変換
    VECTOR GetRotPos(const VECTOR& localPos) const;

    // デバッグ用描画
    virtual void DrawDebug(int color) = 0;

};
```

ColliderBase.cpp

```
#include "../Common/Transform.h"
#include "ColliderBase.h"

ColliderBase::ColliderBase(SHAPE shape, TAG tag, const Transform* follow)
    :
    shape_(shape),
    tag_(tag),
    follow_(follow),
    isValid_(true)
{
}

ColliderBase::~ColliderBase(void)
{
}

void ColliderBase::Draw(void)
{

    int color = COLOR_INVALID;

    if (isValid_)
    {
        color = COLOR_VALID;
    }

    DrawDebug(color);

}

void ColliderBase::SetFollow(Transform* follow)
{
    follow_ = follow;
}

VECTOR ColliderBase::GetRotPos(const VECTOR& localPos) const
{
    // 追従相手の回転に合わせて指定ローカル座標を回転し、
```

```
    // 基準座標に加えることでワールド座標へ変換
    VECTOR localRotPos = follow_->quaRot.PosAxis(localPos);
    return VAdd(follow_->pos, localRotPos);
}
```

ColliderLine.h

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

class ColliderLine : public ColliderBase
{

public:

    // コンストラクタ
    ColliderLine(
        TAG tag, const Transform* follow,
        const VECTOR& localPosStart, const VECTOR& localPosEnd);

    // デストラクタ
    ~ColliderLine(void) override;

    // ローカル座標での設定
    void SetLocalPosStart(const VECTOR& pos);
    void SetLocalPosEnd(const VECTOR& pos);

    // ローカル座標の取得
    const VECTOR& GetLocalPosStart(void) const;
    const VECTOR& GetLocalPosEnd(void) const;

    // ワールド座標の取得
    VECTOR GetPosStart(void) const;
    VECTOR GetPosEnd(void) const;

protected:

    // デバッグ用描画
```

```
    void DrawDebug(int color) override;

private:

    // デバッグ表示の球体半径
    static constexpr float RADIUS = 5.0f;

    // デバッグ表示の球体ポリゴン分割数
    static constexpr int DIV_NUM = 6;

    // 線分の開始座標(ローカル)
    VECTOR localPosStart_;

    // 線分の終了座標(ローカル)
    VECTOR localPosEnd_;

};
```

ColliderLine.cpp

```
#include "../Common/Transform.h"
#include "ColliderLine.h"

ColliderLine::ColliderLine(
    TAG tag, const Transform* follow,
    const VECTOR& localPosStart, const VECTOR& localPosEnd)
    :
    ColliderBase(SHAPE::LINE, tag, follow),
    localPosStart_(localPosStart),
    localPosEnd_(localPosEnd)
{
}

ColliderLine::~ColliderLine(void)
{
}

void ColliderLine::SetLocalPosStart(const VECTOR& pos)
{
    localPosStart_ = pos;
```

```
}

void ColliderLine::SetLocalPosEnd(const VECTOR& pos)
{
    localPosEnd_ = pos;
}

const VECTOR& ColliderLine::GetLocalPosStart(void) const
{
    return localPosStart_;
}

const VECTOR& ColliderLine::GetLocalPosEnd(void) const
{
    return localPosEnd_;
}

VECTOR ColliderLine::GetPosStart(void) const
{
    return GetRotPos(localPosStart_);
}

VECTOR ColliderLine::GetPosEnd(void) const
{
    return GetRotPos(localPosEnd_);
}

void ColliderLine::DrawDebug(int color)
{

    VECTOR s = GetPosStart();
    VECTOR e = GetPosEnd();

    // 線分を描画
    DrawLine3D(s, e, color);

    // 始点・終点を球体で補助表示
    DrawSphere3D(s, RADIUS, DIV_NUM, color, color, true);
    DrawSphere3D(e, RADIUS, DIV_NUM, color, color, true);

}
```

解説

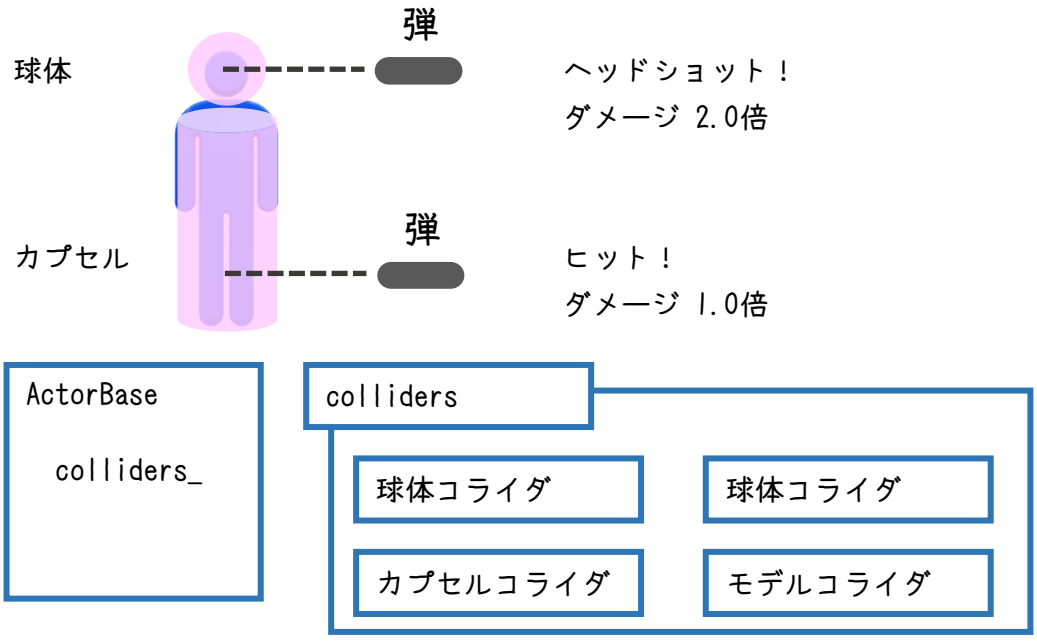
○ Colliderとは
線分、球体、カプセルなどの基本的な形状を構築するための
座標や球体の半径などの情報を管理するクラス。
PlayerやEnemyにくっつけて、衝突判定を行う際に使用します。

○ SHAPE
基底クラス側で、それぞれの形状を判別するための情報です。

○ TAG
衝突を行うか、行わないかの判定を行うための種別です。
PLAYERは、STAGEと衝突判定を行い、押し戻し処理を行います。
今後、PLAYER_WEAPONを追加するとした場合、
PLAYER_WEAPONは、ENEMYとは衝突判定を行いますが、
STAGEとは衝突判定を行わないなどの判別を行います。

②Colliderクラスの管理

これらのColliderは、Playerだけではなく、EnemyやWeaponなど、
全てのActorに必要になるケースが多く、1つのActorに対して、
複数の衝突物を持つ場合もあるため、
ActorBaseに配列型のColliderを持てるようにします。



ActorBase.h

```
class ActorBase
{

public:

    ～ 省略 ～

    // 自身の衝突情報取得
    const std::map<int, ColliderBase*>& GetOwnColliders(void) const
    { return ownColliders_; }

    // 特定の自身の衝突情報取得
    const ColliderBase* GetOwnCollider(int key) const;

protected:

    ～ 省略 ～

    // 自身の衝突情報
    std::map<int, ColliderBase*> ownColliders_;

    // リソースロード
    virtual void InitLoad(void) = 0;
```

ActorBase.cpp

```
void ActorBase::Draw(void)
{

    if (transform_.modelId != -1)
    {
        MV1DrawModel(transform_.modelId);
    }

#ifdef _DEBUG

    // 所有しているコライダの描画
    for (const auto& own : ownColliders_)
```

```
    {
        own.second->Draw();
    }

#endif // _DEBUG

}

void ActorBase::Release(void)
{

    transform_.Release();

    // 自身のコライダ解放
    for (auto& own : ownColliders_)
    {
        delete own.second;
    }

}
```

③実際に線分コライダをPlayerに作る

ここからは、実際にプレイヤーと地面との衝突判定を行う、
線分を作っていきます。

CharactorBase.h

```
class CharactorBase : public ActorBase
{

public:

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

CharactorBase.cpp

```
void CharactorBase::Update(void)
{

    ～ 省略 ～

    // 重力による移動量
    // 一時的に重力無効化
    //CalcGravityPow();

    ～ 省略 ～

}
```

Player.h

```
private:

    ～ 省略 ～

    // 衝突判定用線分開始
    static constexpr VECTOR COL_LINE_START_LOCAL_POS = { 0.0f, 80.0f, 0.0f };

    // 衝突判定用線分終了
    static constexpr VECTOR COL_LINE_END_LOCAL_POS = { 0.0f, -10.0f, 0.0f };
```

Player.cpp

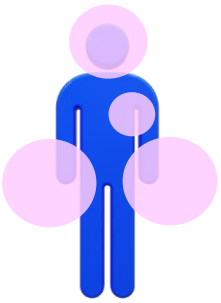
```
void Player::InitCollider(void)
{

    // 主に地面との衝突で仕様する線分コライダ
    ColliderLine* colLine = new ColliderLine(
        ColliderBase::TAG::PLAYER, &transform_,
        COL_LINE_START_LOCAL_POS, COL_LINE_END_LOCAL_POS);
    ownColliders_.emplace(static_cast<int>(COLLIDER_TYPE::LINE), colLine);

}
```

解説

○ 衝突種別のまとめ

SHAPE　　… 衝突の形状。線分、球体、カプセルなど。
TAG　　… PLAYER、STAGE、ENEMYなど。衝突有無の判断に使用。

上記2つの組み合わせは、重複することがある。



…　球体コライダ。PLAYERタグ。

…　球体コライダ。PLAYERタグ。

…　球体コライダ。PLAYER_HITタグ。両手に。

そのため、一意に判断できるようにするための種別を、
各Actorの派生クラスごとに設けるようにする。

COLLIDER_TYPE　… 自身のコライダを一意に判断できるようにするための種別。コライダ配列のキーとして登録する。

```
// 自身の衝突情報
std::map<int, ColliderBase*> ownColliders_;
```

上手くいけば、線分コライダが画面に描画されるようになります。