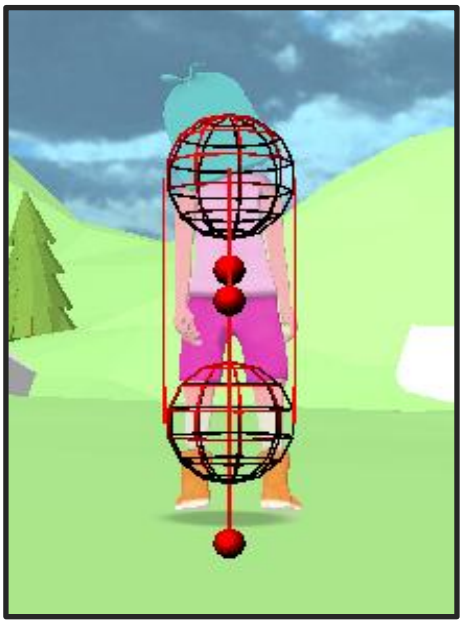
衝突判定の調整

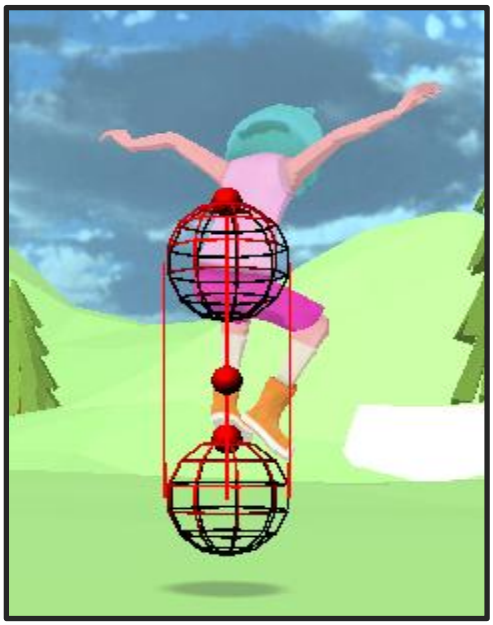
大枠の衝突処理はできましたが、細かいところが気になる方へ、
調整方法をいくつか解説します。

○ ジャンプ後、接地しない不具合解消

カプセルよりも、着地用の線分の長さが地面に向かって長くなるよう、
意図的に作成しています。
これは、カプセルよりも、線分の方を先に地面と衝突させて、
ジャンプ状態を取り消しているからです。
(先にカプセルが衝突すると、押し戻されて、線分が地面と衝突しない)

ところが、ジャンプモーション中に、
線分の位置調整を行っているため、カプセルの方が地面側に位置していますので、
ジャンプ中は、カプセルコライダの位置も調整する必要があります。





コード量も増えてきましたので、
新たに処理フローの中に衝突準備フェーズを加え、
その中で、コライダの位置調整を行っていきたいと思います。

CharactorBase.h

```
protected:

    // 衝突判定
    virtual void CollisionReserve(void) {}
    void Collision(void);
    void CollisionGravity(void);
    void CollisionCapsule(void);
```

CharactorBase.cpp

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

    ～ 省略 ～

    // 重力による移動量
    CalcGravityPow();

    // 衝突判定前準備
    CollisionReserve();

    // 衝突判定
    Collision();

    ～ 省略 ～

}
```

Player.h

```
private:

    // 衝突判定用カプセル上部球体(ジャンプ時)
    static constexpr VECTOR COL_CAPSULE_TOP_JUMP_LOCAL_POS =
     { 0.0f, 160.0f, 0.0f };

    // 衝突判定用カプセル下部球体(ジャンプ時)
    static constexpr VECTOR COL_CAPSULE_DOWN_JUMP_LOCAL_POS =
```

```
    { 0.0f, 80.0f, 0.0f };

    ～ 省略 ～

    // 衝突判定
    void CollisionReserve(void) override;
```

Player.cpp

```
void Player::UpdateProcess(void)
{

    // 移動操作
    ProcessMove();

    // ジャンプ処理
    ProcessJump();

    // アニメーションごとの線分調整
    ⇒まとめて削除、新しい関数へ移動

}

void Player::CollisionReserve(void)
{

    // アニメーションごとの線分調整
    ⇒線分とカプセルの両方を記述しましょう

}
```

【要件①】

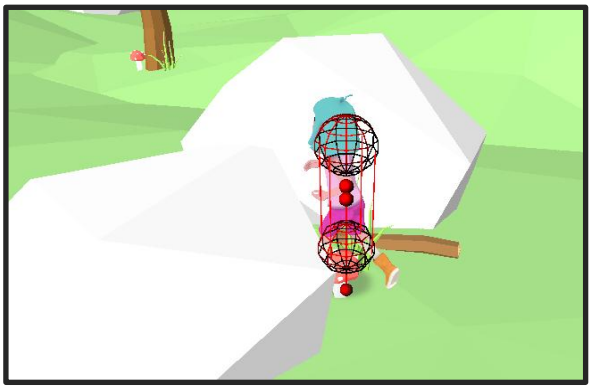
CollisionReserve関数を完成させること。

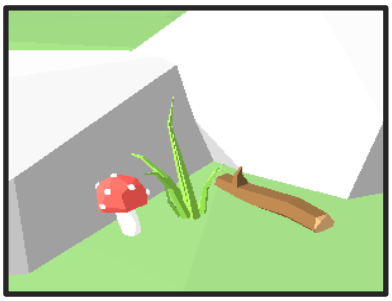
【目標①】

ジャンプ後、接地しない不具合解消されていること。

○ 細かい衝突物でガタガタする不具合解消

モデルとの衝突判定を行うと、
草や花などの細かいオブジェクト(ポリゴン)とも衝突判定を取るため、
そのオブジェクト付近では、複数のポリゴンによる押し戻しの影響で、
キャラクターやカメラがガタガタしてしまいます。





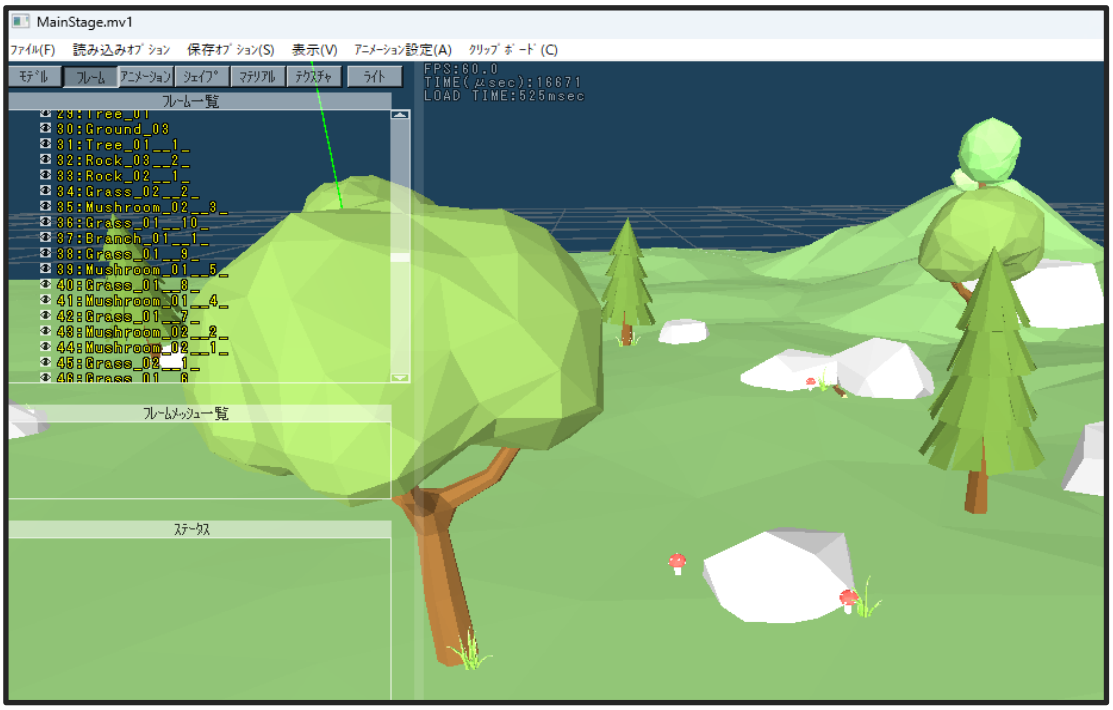
細かいオブジェクト
(ポリゴン)

通常は、描画用のポリゴンとは別に、
衝突判定用のローポリを作成し、衝突判定に使用することで、
処理負荷が下がり、衝突処理も安定します。

Unityで、Cubeを
組み合わせて
衝突判定用ポリゴンを
作ると良いです。

今回は、細かい衝突物は無視するという処理で、衝突処理を
安定させていきたいと思います。

DxModelViewerで、モデルのフレーム(モデルの中のグループ)を確認します。



きのこや地面の草は、
フレームに "Mushroom"や"Grass"という名前が付いているようです。

このように、綺麗にフレーム分けされているモデルの場合は、
フレームで分類分けを行い、一部のフレームを衝突判定から除外する
という機能をプログラムで作ることができます。

ColliderModel.h

```
class ColliderModel : public ColliderBase
{

public:
```

```
    // コンストラクタ
    ColliderModel(TAG tag, const Transform* follow);

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

    // 指定された文字を含むフレームを衝突判定から除外
    void AddExcludeFrameIds(const std::string& name);

    // 衝突判定から除外するフレームをクリアする
    void ClearExcludeFrame(void);

    // 除外フレーム判定
    bool IsExcludeFrame(int frameIdx) const;

protected:

    // 衝突判定から除外するフレーム番号
    std::vector<int> excludeFrameIds_;

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

};
```

ColliderModel.cpp

```
void ColliderModel::AddExcludeFrameIds(const std::string& name)
{

    // フレーム数を取得
    int num = MV1GetFrameNum(follow_->modelId);
    for (int i = 0; i < num; i++)
    {
        // フレーム名称を取得
        std::string frameName = MV1GetFrameName(follow_->modelId, i);
        if (frameName.find(name) != std::string::npos)
        {
            // 除外フレームに追加
            ○○○
        }
```

```
    }

}

void ColliderModel::ClearExcludeFrame(void)
{
    excludeFrameIds_.clear();
}

bool ColliderModel::IsExcludeFrame(int frameIdx) const
{
    // 除外判定
    if (std::find(
        excludeFrameIds_.begin(),
        excludeFrameIds_.end(),
        frameIdx) != excludeFrameIds_.end())
    {
        // 除外に該当する
        return true;
    }

    return false;
}
```

Stage.h

```
private:

    // 除外フレーム名称
    const std::vector<std::string> EXCLUDE_FRAME_NAMES = {
        Mush, "Grass",
    };
```

Stage.cpp

```
void Stage::InitCollider(void)
{

    // DxLib側の衝突情報セットアップ
    MV1SetupCollInfo(transform_.modelId);
```

```
    // モデルのコライダ
    ColliderModel* colModel =
        new ColliderModel(ColliderBase::TAG::STAGE, &transform_);
    for (const std::string& name : EXCLUDE_FRAME_NAMES)
    {
        colModel->AddExcludeFrameIds(name);
    }
    ownColliders_.emplace(static_cast<int>(COLLIDER_TYPE::MODEL), colModel);

}
```

CharactorBase.cpp

```
void CharactorBase::CollisionGravity(void)
{

    ～ 省略 ～

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

        ～ 省略 ～

        // ステージモデル(地面)との衝突
        auto hit = MV1CollCheck_Line(
            colliderModel->GetFollow()->modelId, -1, s, e);

        // 除外フレームは無視する
        ○○○

        ～ 省略 ～

}

void CharactorBase::CollisionCapsule(void)
{

    ～ 省略 ～

        // 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];

            // 除外フレームは無視する
            ○○○

            ～ 省略 ～

        }
}
```

【要件②】

除外フレームの追加処理、除外フレームの無視処理を完成させること。

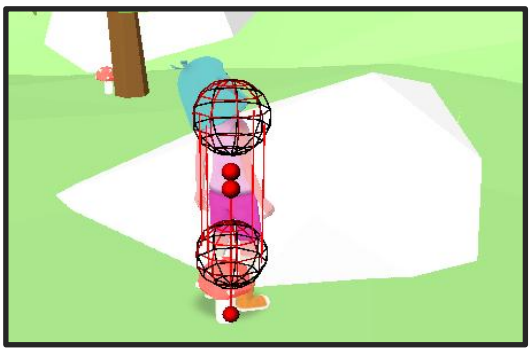
【目標②】

きのこや、草には衝突しないこと。

○ 地面のすり抜け不具合解消

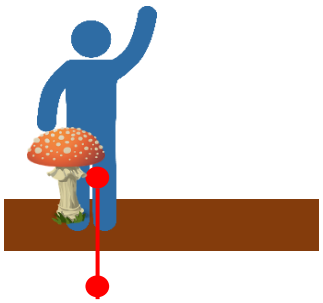
きのこや草を無視してしまう弊害で、
きのこや草との地面衝突で接地判定ができていた位置で、
地面の突き抜けやガタガタが発生するようになります。

ガタガタポイント



プログラム的な要因と致しましては、
DxLibのMV1CollCheck_Line関数の仕様で、線分とモデルの衝突判定結果が
1つのポリゴンしか返ってこないことにあります。

```
// ステージモデル(地面)との衝突
auto hit = MV1CollCheck_Line(colliderModel->GetFollow()->modelId, -1, s, e);
```



この場合、きのこポリゴンとの衝突結果のみが
返ってきます。
土との衝突判定結果は返ってきません。
リファレンスに明記されていませんが、
おそらく、始点座標に最も近い衝突ポリゴンを、
検出していると思われるが確実ではない。

ということで、対策としては、MV1CollCheck_Line関数を使用せずに、
MV1CollCheck_LineDim関数を使用して、
線分と衝突した全てのポリゴンを取得して、
きのこや草との衝突は省き、地面との衝突ポリゴンを見つけれるようにします。

線分に近いポリゴンを、距離でソートして、それを衝突位置として
採用するのが一般的ですが、地面との衝突に特化した処理になりますので、
単純に、最もYの値が大きい衝突地点に押し戻すやり方でもできます。

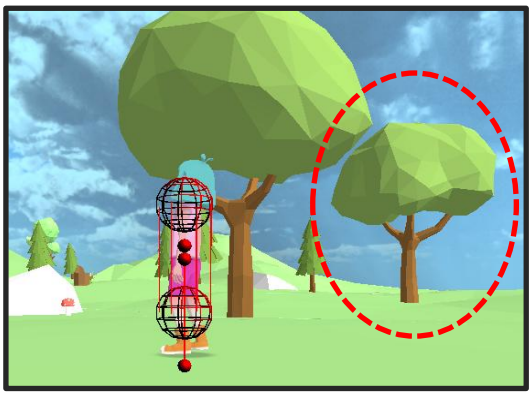
CharactorBase.cpp

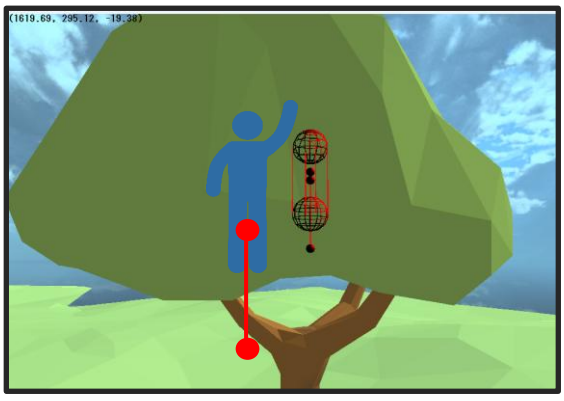
```
void CharactorBase::CollisionGravity(void)
{

    // 線分コライダ
    int lineType = static_cast<int>(COLLIDER_TYPE::LINE);

    // 線分コライダが無ければ処理を抜ける
    if (ownColliders_.count(lineType) == 0) return;

    // 線分コライダ情報
    ColliderLine* colliderLine_ =
        dynamic_cast<ColliderLine*>(ownColliders_.at(lineType));
    if (colliderLine_ == nullptr) return;
```

```
// 線分の始点と終点を取得
VECTOR s = colliderLine_->GetPosStart();
VECTOR e = colliderLine_->GetPosEnd();

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

    // ステージ以外は処理を飛ばす
    if (hitCol->GetTag() != ColliderBase::TAG::STAGE) continue;

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

    if (colliderModel == nullptr) continue;

    // ステージモデル(地面)との衝突
    auto hits = MV1CollCheck_LineDim(
        colliderModel->GetFollow()->modelId, -1, s, e);

    for (int i = 0; i < hits.HitNum; i++)
    {

        auto hit = hits.Dim[i];

        // 除外フレームは無視する
        if (colliderModel->IsExcludeFrame(hit.FrameIndex))
        {
            continue;
        }

        // 衝突地点から、少し上に移動
        if (transform_.pos.y < hit.HitPosition.y)
        {
            // 衝突物より、下側にいる場合のみ、位置を修正する
            transform_.pos =
                VAdd(hit.HitPosition, VScale(AsoUtility::DIR_U, 2.0f));
        }
```

```
            // ジャンプ判定
            isJump_ = false;

        }

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

    }

    if (!isJump_)
    {
        // ジャンプリセット
        jumpPow_ = AsoUtility::VECTOR_ZERO;

        // ジャンプの入力受付時間をリセット
        stepJump_ = 0.0f;
    }

}
```

○ **木のめり込み不具合解消**





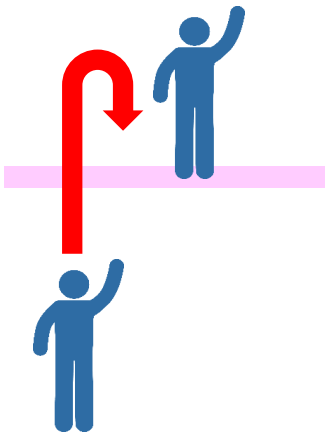
頭の上に衝突物がある場合、ジャンプ力(移動量)が大きいため、
カプセルの押し戻し処理では、押し戻し量が不足して、
線分との衝突判定が反応してしまうケースがあります。

処理順的には、カプセルが先ですが、押し戻し量が足りない。。。

```
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();

}
```

移動量が大きい時に、モデルを突き抜けないような確実な衝突処理は、
やはり移動前座標と移動後座標の間を線分やカプセルで衝突判定を
取るやり方ですが、別のギミックでも使えそうな対策を今回は行います。

マリオなどのアクションゲームで、時々、ジャンプしたら乗れる板が
出てくると思います。



上昇ジャンプでは衝突しませんが、
下降中は、衝突判定が行われ、
着地できるという挙動です。

ジャンプモーションの３～４段でも上昇か、下降かのジャンプ判定は
大切になってきます。

重力加速度の制限も行いましたが、ジャンプ量のYの値が、
プラスかマイナスかで、その判定を行うことができますので、
そちらでも良いのですが、ベクトルの計算をしっかり行いたい方は、
内積を使用して、上昇か、下降か判定してみましょう。

```
// 落下中しか判定しない
if (!(VDot(AsoUtility::DIR_D, jumpPow_) > 0.9f))
{
    continue;
}
```

上記のコードは、ジャンプ量が、上方向にかなり近ければ、上昇中、
そうでなければ下降中、という意味になります。
これをCollisionGravity関数に追加することで、
上昇中は、地面との衝突判定が発生せず、カプセルの押し戻しに猶予が
生まれるため、めり込みしにくくなります。

また、先ほどのマリオのジャンプの乗れる板を実装したければ、
カプセルコライダも、線分コライダも、この処理を追加することによって、
実現することができます。

○ 完全ではありませんが

こういった形で、ゲームのステージ形状や、移動量、ジャンプ量、
ゲームの仕様によって、最適な衝突判定は異なり、やりたいことを実現するために
色々な対策を行っていく必要があります。

トライ＆エラーを繰り返していけば、その分、衝突も安定してくると
思いますが、想像以上に検証コストがかかりますので、
制作の過程で、ある程度見切りを付けて、優先度の高いゲームのメイン機能を
先に実装するようにしましょう。