カメラのプラスα

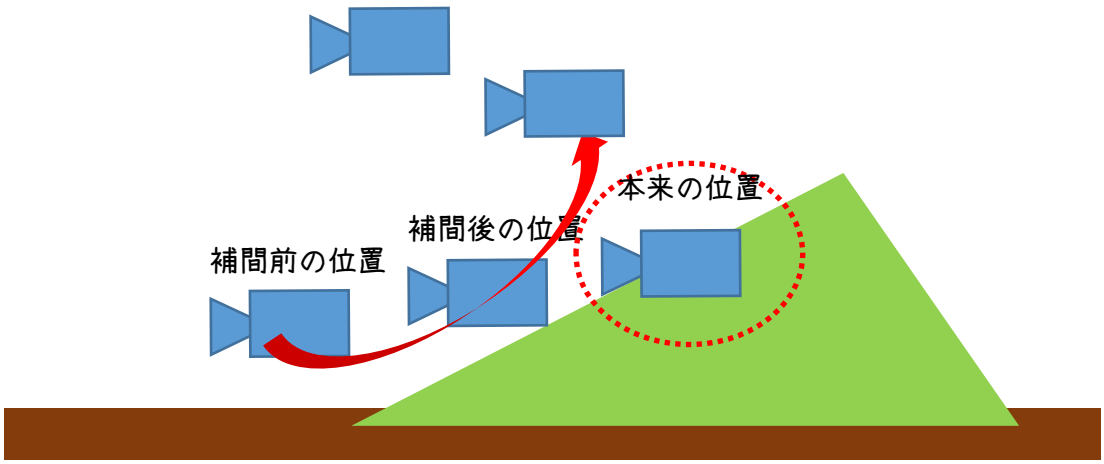
カメラは、ゲーム内の３Ｄ空間を投影し、
ユーザに見せるゲーム画面への影響度が最も高いため、
とても重要な機能になります。

カメラのアングルやカメラの移動方法についても、
まだまだ工夫ができますので、ゲーム性にもよりますが、
拘って作ってみても良いと思います。

① カメラ位置の線形補間

衝突判定を入れたことによって、カメラの位置が瞬間的に変わったり、
衝突する場所によっては、多少ガタガタとカメラが動いてしまいます。
急激なカメラ移動を防止するため、
カメラ位置を瞬間移動ではなく、補間移動にすることで、
滑らかにカメラが動くようになります。

また、衝突判定は、本来のカメラ位置で行い、
その位置に遅れて移動するため、ポリゴンの裏側がほぼ見えなくなる
というメリットもあります。



但し、カメラが遅れてヌルヌル動くので、ユーザが意図した画角にならず、
カメラの瞬間移動やガタガタと同じく、
不快に感じたり、酔いやすくなるユーザは多いと思います。

試してみましょう。

Camera.h

```
private:

    // カメラの補間移動率
    static constexpr float LERP_RATE_MOVE = 0.1f;

    // カメラの更新前位置
    VECTOR prePos_;
```

Camera.cpp

```
void Camera::SetBeforeDraw(void)
{

    // クリップ距離を設定する(SetDrawScreenでリセットされる)
    SetCameraNearFar(VIEW_NEAR, VIEW_FAR);

    // 更新前情報
    prePos_ = transform_.pos;

    ～ 省略 ～

}

void Camera::SetBeforeDrawFollow(void)
{

    ～ 省略 ～

    // カメラ位置の補間
    transform_.pos =
        AsoUtility::Lerp(prePos_, transform_.pos, LERP_RATE_MOVE);

}
```

【要件①】

画面が気持ち悪くなる、酔いやすくなる原因を分析して、対策すること。

アプローチのヒント

PTN_A　衝突していなくて、カメラも補間移動も無い時は、
カメラ酔いしなかったはず。その差を見つける。

PTN_B　補間移動全体が酔いの原因？特定の補間移動が原因？

PTN_C　衝突時は補間、衝突していない時は補間しないと
良いところだけ取れる？(実装制御難しめ)

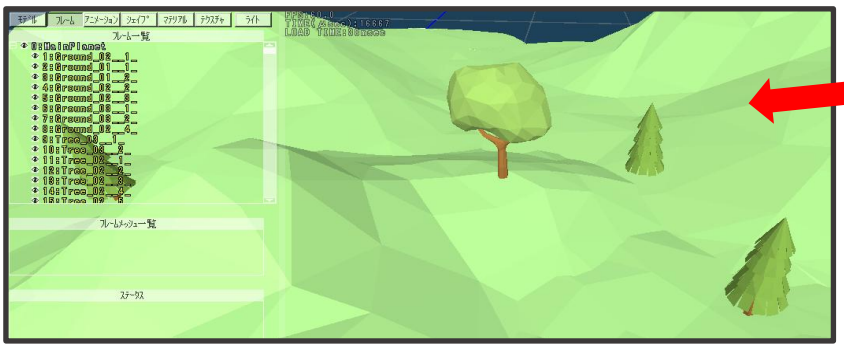
【目標①】

カメラ移動による、３Ｄ酔いを軽減する。

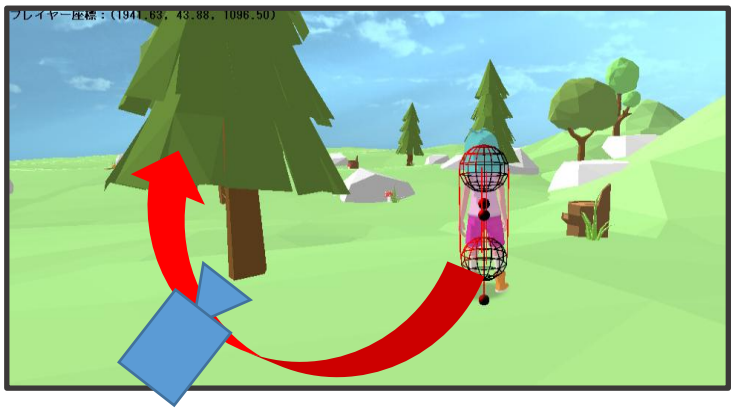
②　カメラと衝突判定を行うステージモデルのフレームを絞る

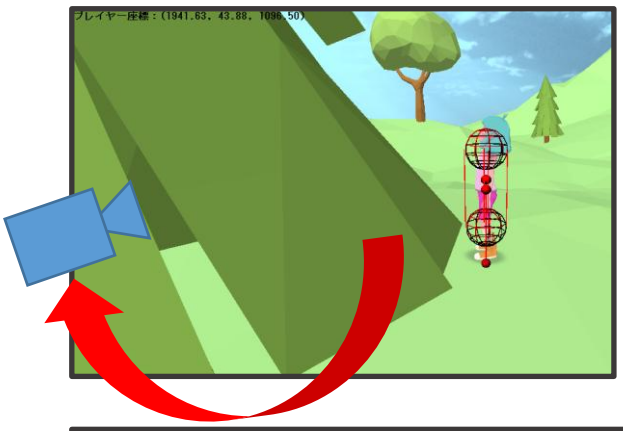
キャラクターの線分衝突と、カメラの球体衝突は、
きのこや草のフレームを除いたフレームと衝突するように組んでいます。

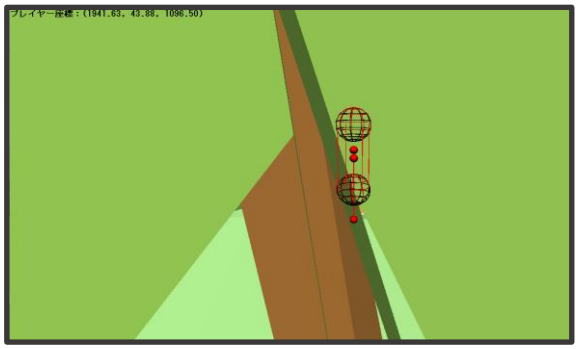
ゲーム性にもよりますが、
地面全体ではなく、木などのオブジェクトとカメラとの衝突を行わないことで、
不要なガタガタを抑えたり、オブジェクトを衝突回避ではなく、
半透明化するなど別の制御を行うこともあります。



地面。
フレーム名は
Ground

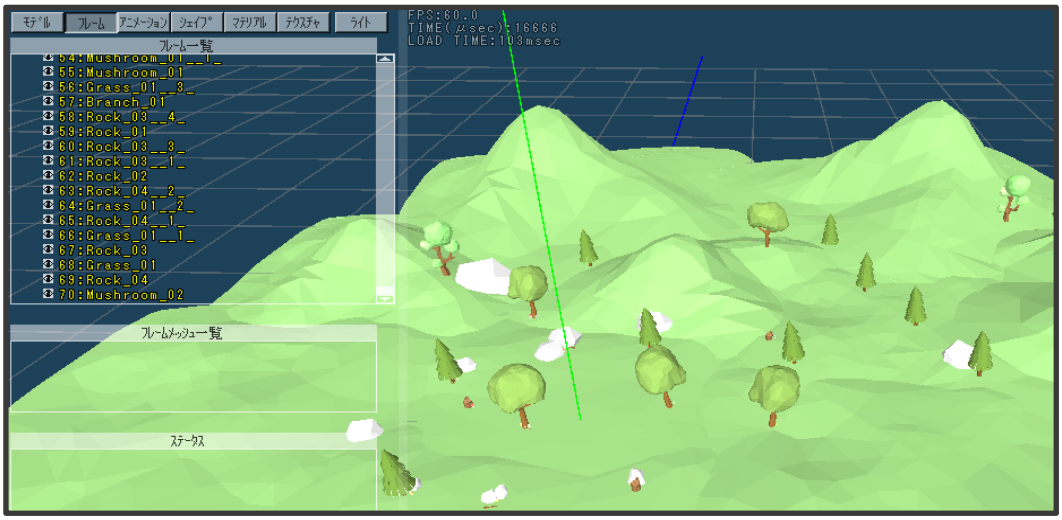

プレイヤー座標：(1941.63, 43.88, 1096.50)

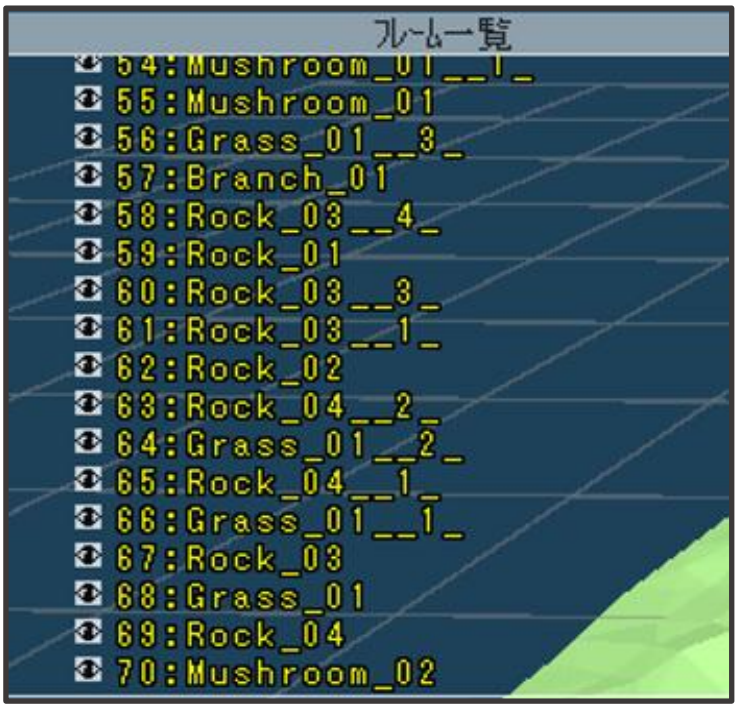

プレイヤー座標：(1941.63, 43.88, 1096.50)


プレイヤー座標：(1941.63, 43.88, 1096.50)

衝突しない。

地面以外のフレームを除外すれば、実現できますが、
フレームの数や種類が多く、除外するのが大変です。





ですので、"除外する" よりも、"許可する" 方が早そうです。

ColliderModelで既に実装してある、
"除外する" という対象物のことをブラックリストといい、
"許可する" という対象物のことをホワイトリストといいます。

ColliderModel.h

```
class Transform;

class ColliderModel : public ColliderBase
{

public:

    ～ 省略 ～

    // 指定された文字を含むフレームを衝突判定対象とする
    void AddTargetFrameIds(const std::string& name);

    // 衝突判定の対象するフレームをクリアする
    void ClearTargetFrame(void);

    // 対象フレーム判定
    bool IsTargetFrame(int frameIdx) const;

protected:

    ～ 省略 ～

    // 衝突判定の対象とするフレーム番号
    std::vector<int> targetFrameIds_;
```

Stage.h

```
private:

    ～ 省略 ～

    // 対象フレーム
    const std::vector<std::string> TARGET_FRAME_NAMES = {
        Ground,
    };
```

Stage.cpp

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

    ～ 省略 ～

    for (const std::string& name : TARGET_FRAME_NAMES)
    {
        colModel->AddTargetFrameIds(name);
    }
    ownColliders_.emplace(static_cast<int>(COLLIDER_TYPE::MODEL), colModel);

}
```

Camera.cpp

```
void Camera::Collision(void)
{

    ～ 省略 ～

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

            const auto& hit = hits.Dim[i];

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

            // 対象フレーム以外は無視する
            if (!colliderModel->IsTargetFrame(hit.FrameIndex))
            {
                continue;
            }

    ～ 省略 ～
```

【要件②】

カメラの球体コライダは、地面(Ground)フレームとしか衝突しないようにすること。

- 既に実装されているColliderModelの除外リストを参考に、
  ColliderModel.cppに追加した関数を完成させること

  AddTargetFrameIds関数
  ClearTargetFrame関数
  IsTargetFrame関数

- キャラクターの線分コライダは、
  これまで通りの衝突条件として、挙動を変えないようにすること

【目標②】

木や岩とは、カメラは衝突せず、不要なガタガタを無くすこと。

【チャレンジ①】

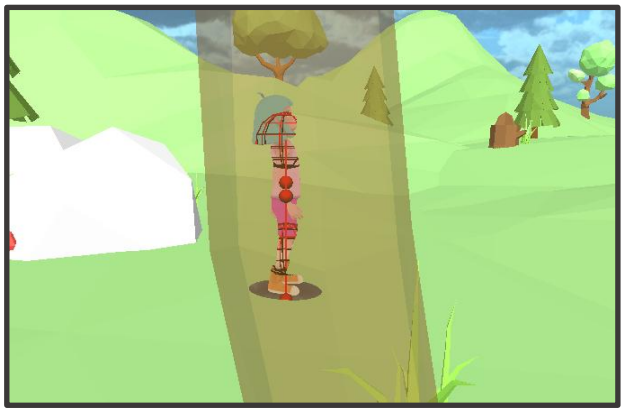
カメラの線分と衝突した木や岩を、半透明化にすること。

ヒント

- MVISetFrameOpacityRate関数を使用すれば、
  フレーム単位で半透明化を指定できます

- 半透明は、処理負荷が高く制御も難しいです。
  普通に描画してしまうと、半透明の先にキャラクターが描画されません

PlayerとStageの描画順を無理やり変えることで、
キャラクターが描画されるようになりますが、



このやり方だけだと、丸影を実装している方は、
丸影がおかしくなります。

正しい描画順は、下記のようになります。

不透明(opaque)
半透明(transparent)
※カメラに近い順にZソートが必要
UI
ポストエフェクト

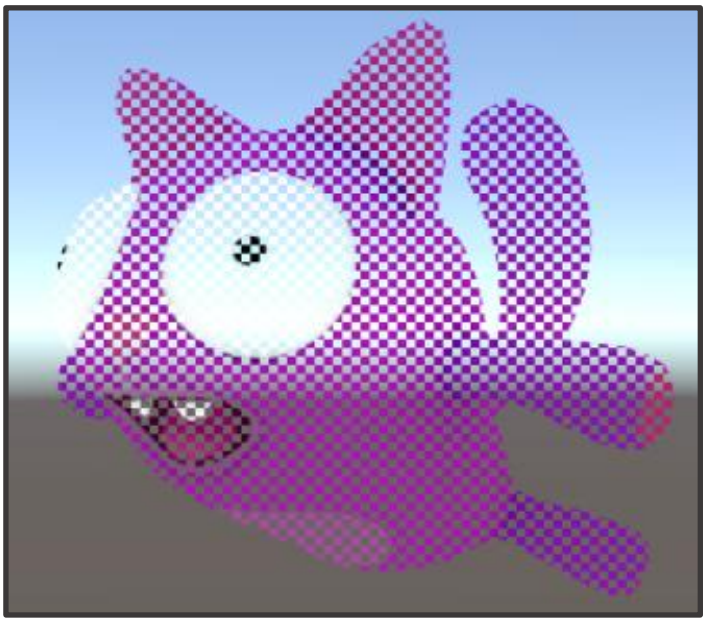
本格的に組むと大変ですが、
DrawOpaque
DrawTransparent
DrawUI
上記のようなDrawの処理フローをTemplateメソッドで作って上げて、
制御するくらいで今回のゲームは上手く動作すると思いますが、
半透明が連発されるゲームだと、
しっかりと描画システムを作り込まないといけません。

【チャレンジ②】

パッと半透明になると、それも気持ち悪いので、
イージングで半透明にするとより良いでしょう。

【スーパーエクストラチャレンジ①】

半透明は処理負荷が高く制御が難しいため、
オリジナルシェーダを使って、疑似半透明ディザリングをやってみよう。



ベイヤーディザ。

やらなくて良いです。