ゲームプログラミング改

○評価要件

☑レイキャストでキャラクターを地面に立たせる
☑下り坂移動でガタガタしないようにする

○概要

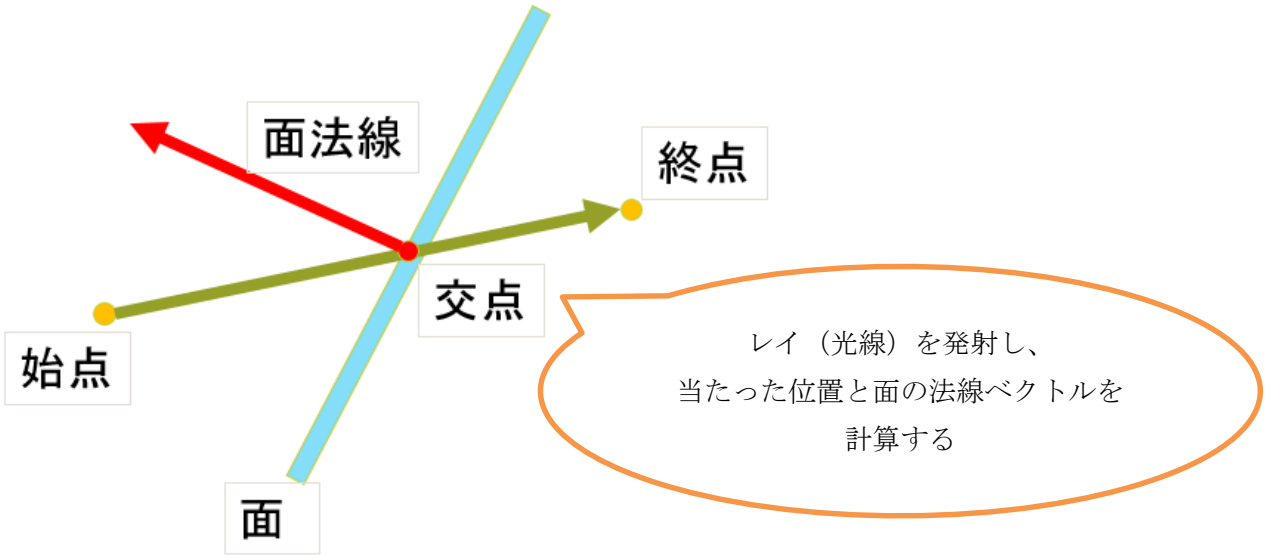
今回はレイキャストを実装します。
レイキャストとはレイ（光線）をキャストする（投げる）ことでレイと他の形状のオブジェクトとの衝突判定をする技術です。

一般的なキャラクターを操作するゲームでは地形に合わせてキャラクターを移動させます。
デコボコした地面やスロープのような坂道などでキャラクターが地形にめり込まないように位置を補正するときなどに使用します。

今回はレイと地形（ポリゴン）の衝突処理を実装してキャラクターを地形の高さに合わせて立たせる処理を実装しましょう。

○レイキャスト

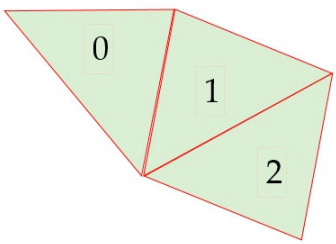
今回実装するレイキャストはレイ（光線）とポリゴン（平面）の衝突を算出します。
レイには始点と終点があり、始点から終点の間にポリゴンがある場合、交点を算出し、地形の向き（法線）情報を取り出します。

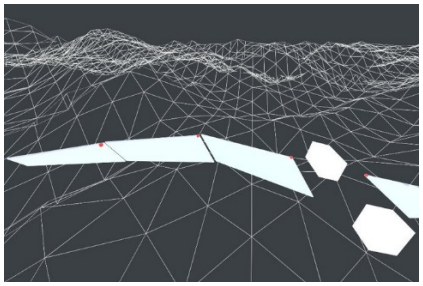


○ポリゴン

レイキャストはレイとポリゴンとの衝突処理です。
レイは始点と終点なので、2 つの 3D 座標データです。
ポリゴンデータとはどのようなものなのでしょうか。

3D モデルは大量の三角形が組み合わさって表示されています。
この三角形一つ一つとレイの衝突判定を計算していくわけです。





最近のゲームでは 3D モデルは数万～数十万のポリゴンデータで構成されているため、一つ一つ計算していると処理が重すぎて使い物にならないため、様々な工夫をして最適化処理をしています。
最適化の一つとして衝突判定用の 3D モデルデータを用意してポリゴン数を減らすこともします。

今回の課題では最適化はせず、単純にレイと全てのポリゴンとの衝突処理をやっていきます。
レイキャストの計算処理は難しく、実装が終わるまで道のりが長いので、先にキャラクターを地面に立たせるまでの流れを実装しましょう。

○キャラクターをステージに立たせるまでの骨組みを実装する

まず、レイと 3D モデルとの衝突判定関数を作りましょう。
この関数の中身はすごく長くなるので実装は後回しです。

Collision.h

```
---省略---
#include "Graphics/Model.h"

// ヒット結果
struct HitResult
{
    DirectX::XMFLOAT3  position = { 0, 0, 0 };// レイとポリゴンの交点
    DirectX::XMFLOAT3  normal = { 0, 0, 0 };  // 衝突したポリゴンの法線ベクトル
    float              distance = 0.0f;       // レイの始点から交点までの距離
    int                materialIndex = -1;    // 衝突したポリゴンのマテリアル番号
};

// コリジョン
class Collision
{
public:
    ---省略---

    // レイとモデルの交差判定
    static bool IntersectRayVsModel(
        const DirectX::XMFLOAT3& start,
        const DirectX::XMFLOAT3& end,
        const Model* model,
        HitResult& result
    );
};
```

レイキャストで取り出したい情報

Collision.cpp

```
// レイとモデルの交差判定
bool Collision::IntersectRayVsModel(
    const DirectX::XMFLOAT3& start,
    const DirectX::XMFLOAT3& end,
    const Model* model,
    HitResult& result)
{
    // 後で実装する
    return false;
}
```

この計算関数を使ってレイキャストを行います。
ステージモデルとの衝突を判定したいのでステージクラスにレイキャスト関数を追加します。

Stage.h

```
---省略---
#include "Collision.h"

// ステージ
class Stage
{
public:
    ---省略---

    // レイキャスト
    bool RayCast(const DirectX::XMFLOAT3& start, const DirectX::XMFLOAT3& end, HitResult& hit);

    ---省略---
};
```

Stage.cpp

```
---省略---

// レイキャスト
bool Stage::RayCast(const DirectX::XMFLOAT3& start, const DirectX::XMFLOAT3& end, HitResult& hit)
{
    return Collision::IntersectRayVsModel(start, end, model, hit);
}
```

これでステージに対してレイキャストを行うことで衝突判定ができるようになりました。
この関数をキャラクターの移動処理で使うことで地面に立たせることができます。

しかし、現時点ではキャラクタークラスから表示されているステージに対してアクセスする手段がないため、使うことができません。

今後、ステージマネージャーを作成し、どこからでもアクセスできるようにする予定ですが、今回は面倒なので、仮の実装としてステージをシングルトン風に改造します。

Stage.h

```
---省略---

// ステージ
class Stage
{
public:
    ---省略---
    // インスタンス取得
    static Stage& Instance();

    ---省略---
};
```

Stage.cpp

```
---省略---

static Stage* instance = nullptr;

// インスタンス取得
Stage& Stage::Instance()
{
    return *instance;
}

// コンストラクタ
Stage::Stage()
{
    instance = this;

    ---省略---
}
---省略---
```

これでどこからでもステージにアクセスできるようになったのでキャラクタークラスでレイキャスト処理を呼び出しましょう。

Character.h

```
---省略---

// キャラクター
class Character
{
    ---省略---
protected:
    ---省略---
    float                   stepOffset = 1.0f;
};
```

Character.cpp

```
---省略---
#include "Stage.h"

---省略---

// 垂直移動更新処理
void Character::UpdateVerticalMove(float elapsedTime)
{
    // 移動処理
    position.y += velocity.y * elapsedTime;

    // 地面判定
    if (position.y < 0.0f)
    {
        position.y = 0.0f;
```

地面判定をレイキャストで行うため、
今までの処理はすべて削除し、
垂直移動や地面判定処理を修正する

```
        // 着地した
        if (!isGround)
        {
            OnLanding();
        }
        isGround = true;
        velocity.y = 0.0f;
    }
    else
    {
        isGround = false;
    }

    // 垂直方向の移動量
    float my = velocity.y * elapsedTime;

    // 落下中
    if (my < 0.0f)
    {
        // レイの開始位置は足元より少し上
        DirectX::XMFLOAT3 start = { position.x, position.y + stepOffset, position.z };
        // レイの終点位置は移動後の位置
        DirectX::XMFLOAT3 end = { position.x, position.y + my, position.z };

        // レイキャストによる地面判定
        HitResult hit;
        if (Stage::Instance().RayCast(start, end, hit))
        {
            // 地面に接地している
            position.y = hit.position.y;

            // 着地した
            if (!isGround)
            {
                OnLanding();
            }
            isGround = true;
            velocity.y = 0.0f;
        }
        else
        {
            // 空中に浮いている
            position.y += my;
            isGround = false;
        }
    }
    // 上昇中
    else if (my > 0.0f)
    {
        position.y += my;
        isGround = false;
    }
}

---省略---
```

レイキャストの処理が正しく実装されるまでの間、仮の処理を実装しておきましょう。

Collision.cpp

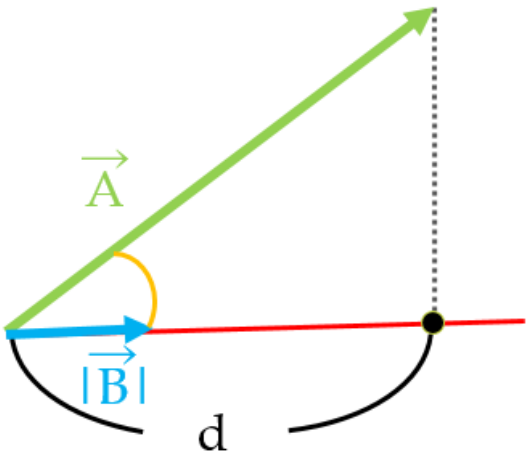
```
---省略---
// レイとモデルの交差判定
bool Collision::IntersectRayVsModel(
    const DirectX::XMFLOAT3& start,
    const DirectX::XMFLOAT3& end,
    const Model* model,
    HitResult& result)
{
    // 以前の処理が正しく動くように仮の実装
    if (end.y < 0.0f)
    {
        result.position.x = end.x;
        result.position.y = 0.0f;
        result.position.z = end.z;
        result.normal.x = 0.0f;
        result.normal.y = 1.0f;
        result.normal.z = 0.0f;
        return true;
    }
    return false;
}
```

ここまでで一度実行確認をしてみましょう。
以前までと同じようにキャラクターが移動操作できていれば OK です。

これで後はレイキャストの衝突処理を正しく実装出来ていればキャラクターが地形の高さに合わせて立つようになります。
レイキャストの衝突処理をするには三角関数を駆使して実装します。

○三角関数

三角関数とは簡単に言うと直角三角形から様々な情報を取得するための数学テクニックです。
具体的にどういうことができるか知っておきましょう。



- ２つのベクトルの内積を求めることで角度、または距離を算出できる。
- 両方のベクトルが単位ベクトルの場合、角度を求めることができる。
- 片方のベクトルが単位ベクトルの場合、距離を求めることができる。
  ベクトルＡと単位ベクトルＢの内積で距離ｄが算出できる

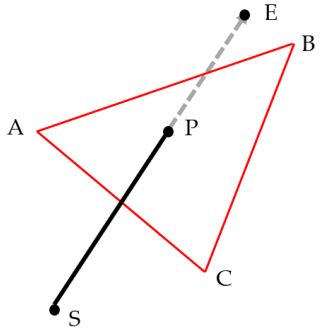
○レイとポリゴンの衝突処理

まずはレイとポリゴンの衝突を計算する関数を作成しましょう。
コリジョンクラスにレイとモデルの交差計算処理関数を作成します。
このあと、レイキャストの計算処理を実装していくわけですが、以下の手順で行います。

1. 三角形頂点の抽出
2. 三角形の三辺を算出
3. 三角形の法線を算出
4. 平面の裏表判定
5. レイと平面の交点を算出
6. 交点が三角形の内側にあるか判定
7. 衝突結果を保存

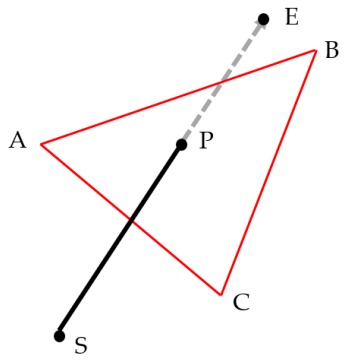
なかなか手順が多いですが１つ１つ説明していきます。

①三角形頂点の抽出



- レイキャストは始点 S から終点 E のレイと三角形 ABC の交点 P を算出していく。
- 三角形 ABC を構成する A、B、C の頂点座標を抽出する

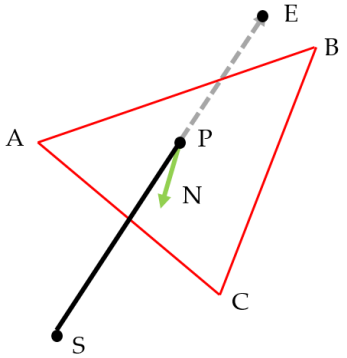
②三角形の三辺を算出



- 法線ベクトルの算出やレイと平面の交差判定に必要な
  三辺のベクトルを算出する

- AB = B – A
- BC = C – B
- CA = A – C

③三角形の法線を算出

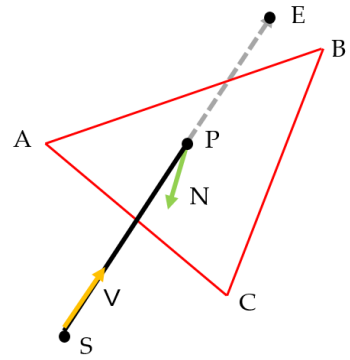


- ２つのベクトルの外積を算出することで
  ２つのベクトルに垂直なベクトルを取得する。

- N = AB × BC

数学では「×」記号で
外積(Cross)を表す

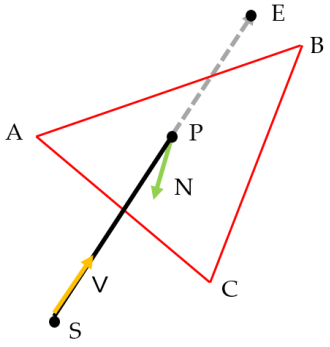
④平面の裏表判定



- ２つのベクトルの内積を算出し、符合を見ることで
  裏表判定ができる。
  内積値の符合がプラスの場合、裏面と判断できる。
  裏面の場合は衝突しないようにする。
- d = V ・ N

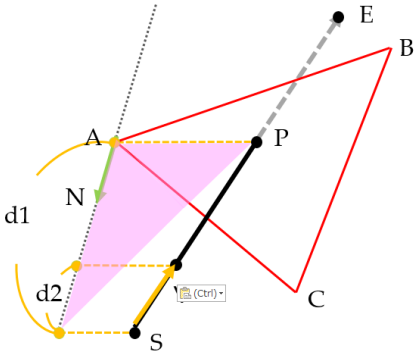
数学では「・」記号で
内積(Dot)を表す

④レイと平面の交点を算出①



- 図の情報から P の座標を求めることが目的。
- 考え方として点 S からベクトル V を x 倍したところが交点。
- ベクトルとは方向と長さ。
  この長さを x 倍することで交点が算出できる。
- つまり x を求めれば良い

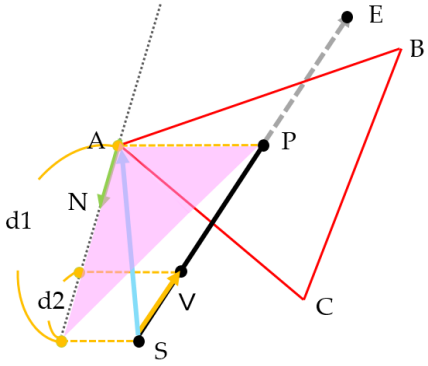
④レイと平面の交点を算出②



- 平面上の座標のわかる点を基準として
  点 S と点 V を法線 N に射影する。
- 射影前の線分と射影後の線分の長さの割合は同じになる。
  三角形の相似条件を満たしているため割合は同じである。

- d1 の長さは d2 の長さの何倍(x)かわかれば
  P の位置を求められる

⑤レイと平面の交点を算出③



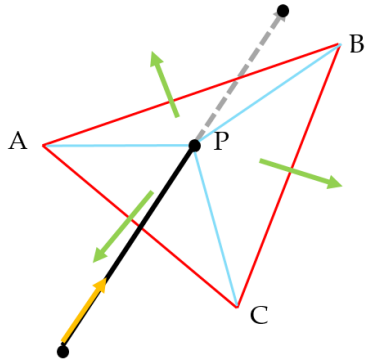
- d1 と d2 の長さは 2 つのベクトルの内積で求められる。

- d1 = SA ・ N
  d2 = V ・ N

d2 は裏表判定のときに
算出している

- x = d1 / d2
- P = S + V * x

⑥交点が三角形の内側にあるか判定

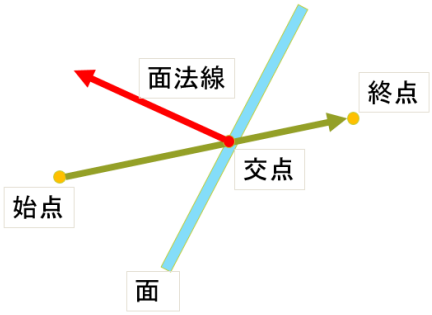


- 点 P が三角形の内側にあるか判定する。

- 三角形 ABP、BCP、CAP のそれぞれの外積を求め、
  それらと法線の内積を求める。

- すべての内積値の符合がプラスだったら
  点 P は三角形の内側にある。
- 考え方は面の裏表判定と同じ。

⑦衝突結果を保存



- 交点座標
- 面法線ベクトル
- 始点から交点までの距離

- 複数の三角形とのレイキャスト処理で
  保存するデータは交点までの距離が一番近い情報。

レイと三角形との衝突判定の処理の流れは以上となります。
この流れをプログラムとして実装していきましょう。

Collision.cpp

```
// レイとモデルの交差判定
bool Collision::IntersectRayVsModel(
    const DirectX::XMFLOAT3& start,
    const DirectX::XMFLOAT3& end,
    const Model* model,
    HitResult& result)
{
    DirectX::XMVECTOR WorldStart = DirectX::XMLoadFloat3(&start);
    DirectX::XMVECTOR WorldEnd = DirectX::XMLoadFloat3(&end);
    DirectX::XMVECTOR WorldRayVec = DirectX::XMVectorSubtract(WorldEnd, WorldStart);
    DirectX::XMVECTOR WorldRayLength = DirectX::XMVector3Length(WorldRayVec);

    // ワールド空間のレイの長さ
    DirectX::XMStoreFloat(&result.distance, WorldRayLength);

    bool hit = false;
    const ModelResource* resource = model->GetResource();
    for (const ModelResource::Mesh& mesh : resource->GetMeshes())
    {
        // メッシュノード取得
        const Model::Node& node = model->GetNodes().at(mesh.nodeIndex);

        // レイをワールド空間からローカル空間へ変換
        DirectX::XMMATRIX WorldTransform = DirectX::XMLoadFloat4x4(&node.worldTransform);
        DirectX::XMMATRIX InverseWorldTransform = DirectX::XMMatrixInverse(nullptr, WorldTransform);

        DirectX::XMVECTOR Start = DirectX::XMVector3TransformCoord(WorldStart, InverseWorldTransform);
        DirectX::XMVECTOR End = DirectX::XMVector3TransformCoord(WorldEnd, InverseWorldTransform);
        DirectX::XMVECTOR Vec = DirectX::XMVectorSubtract(End, Start);
        DirectX::XMVECTOR Dir = DirectX::XMVector3Normalize(Vec);
        DirectX::XMVECTOR Length = DirectX::XMVector3Length(Vec);

        // レイの長さ
        float neart;
        DirectX::XMStoreFloat(&neart, Length);

        // 三角形（面）との交差判定
        const std::vector<ModelResource::Vertex>& vertices = mesh.vertices;
        const std::vector<UINT> indices = mesh.indices;

        int materialIndex = -1;
        DirectX::XMVECTOR HitPosition;
        DirectX::XMVECTOR HitNormal;
        for (const ModelResource::Subset& subset : mesh.subsets)
        {
            for (UINT i = 0; i < subset.indexCount; i += 3)
            {
                UINT index = subset.startIndex + i;
```

このレイの空間変換に
ついては後で説明する

頂点バッファと
インデックスバッファから
三角形を抽出する

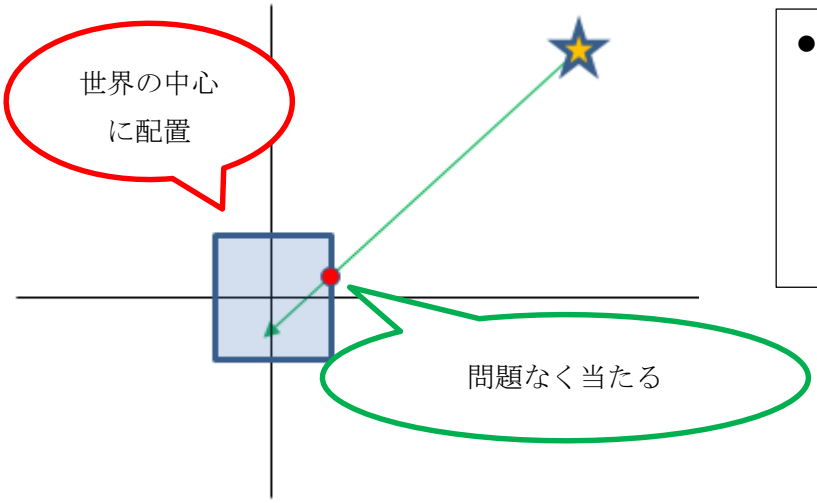
```
// 三角形の頂点を抽出
const ModelResource::Vertex& a = vertices.at(indices.at(index));
const ModelResource::Vertex& b = vertices.at(indices.at(index + 1));
const ModelResource::Vertex& c = vertices.at(indices.at(index + 2));

DirectX::XMVECTOR A = DirectX::XMLoadFloat3(&a.position);
DirectX::XMVECTOR B = DirectX::XMLoadFloat3(&b.position);
DirectX::XMVECTOR C = DirectX::XMLoadFloat3(&c.position);

// 三角形の三辺ベクトルを算出
DirectX::XMVECTOR AB =
DirectX::XMVECTOR BC =
DirectX::XMVECTOR CA =

// 三角形の法線ベクトルを算出
DirectX::XMVECTOR Normal =

// 内積の結果がプラスならば裏向き
DirectX::XMVECTOR Dot =

if (        ) continue;

// レイと平面の交点を算出
DirectX::XMVECTOR V =
DirectX::XMVECTOR T =
float t;
DirectX::XMStoreFloat(&t, T);
if (t < .0f || t > neart) continue;   // 交点までの距離が今までに計算した最近距離より
                                      // 大きい時はスキップ
DirectX::XMVECTOR Position =


// 交点が三角形の内側にあるか判定
// １つめ
DirectX::XMVECTOR V1     =
DirectX::XMVECTOR Cross1 =
DirectX::XMVECTOR Dot1   =

if (           ) continue;
// ２つめ
DirectX:: XMVECTOR V2    =
DirectX::XMVECTOR Cross2 =
DirectX::XMVECTOR Dot2   =

if (           ) continue;
// ３つめ
DirectX::XMVECTOR V3     =
DirectX::XMVECTOR Cross3 =
DirectX::XMVECTOR Dot3   =

if (           ) continue;

// 最近距離を更新
```

```
                neart = t;

                // 交点と法線を更新
                HitPosition = Position;
                HitNormal = Normal;
                materialIndex = subset.materialIndex;
            }
        }
        if (materialIndex >= 0)
        {
            // ローカル空間からワールド空間へ変換
            DirectX::XMVECTOR WorldPosition = DirectX::XMVector3TransformCoord(HitPosition,
                                                                    WorldTransform);
            DirectX::XMVECTOR WorldCrossVec = DirectX::XMVectorSubtract(WorldPosition, WorldStart);
            DirectX::XMVECTOR WorldCrossLength = DirectX::XMVector3Length(WorldCrossVec);
            float distance;
            DirectX::XMStoreFloat(&distance, WorldCrossLength);

            // ヒット情報保存
            if (result.distance > distance)
            {
                DirectX::XMVECTOR WorldNormal = DirectX::XMVector3TransformNormal(HitNormal,
                                                                        WorldTransform);

                result.distance = distance;
                result.materialIndex = materialIndex;
                DirectX::XMStoreFloat3(&result.position, WorldPosition);
                DirectX::XMStoreFloat3(&result.normal, DirectX::XMVector3Normalize(WorldNormal));
                hit = true;
            }
        }
    }

    return hit;
}
```

長いプログラムでしたが、これでレイと 3D モデルとの衝突判定ができます。
実装出来たら実行確認をしてみましょう。
キャラクターを移動操作してみて地面の高さに移動できていれば OK です。

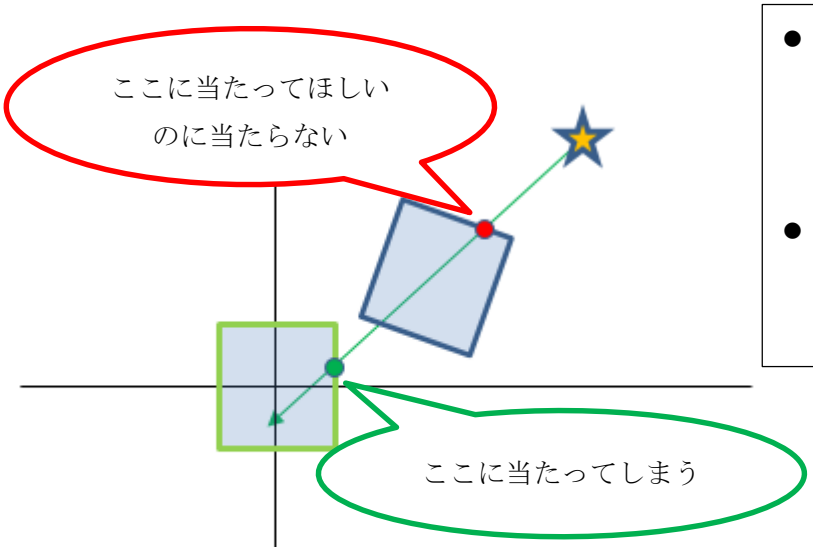
○レイの空間変換

今回の課題でレイの空間変換について説明していなかったので軽く解説します。
今回は敢えて考えさせることなく、答えを記載していましたが、今後の課題でも学習する予定です。

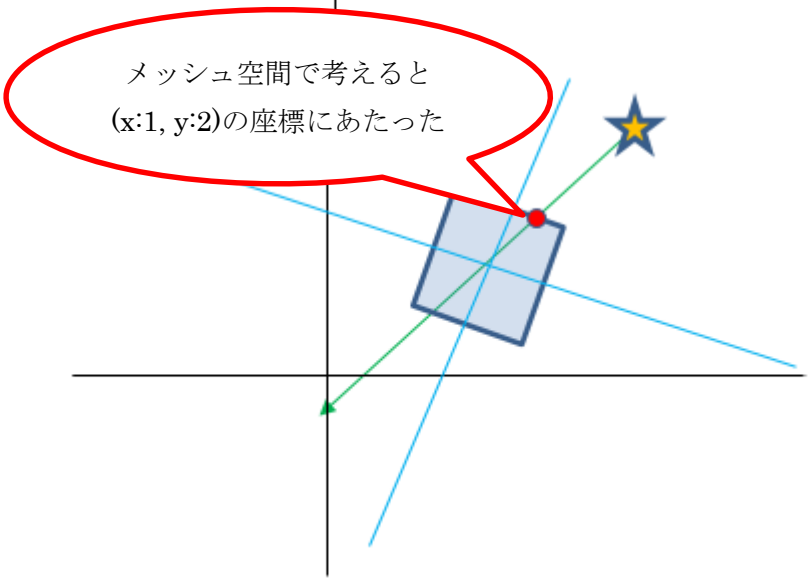


- メッシュが世界の中心（0,0）にある場合、メッシュの頂点座標はワールド座標である。

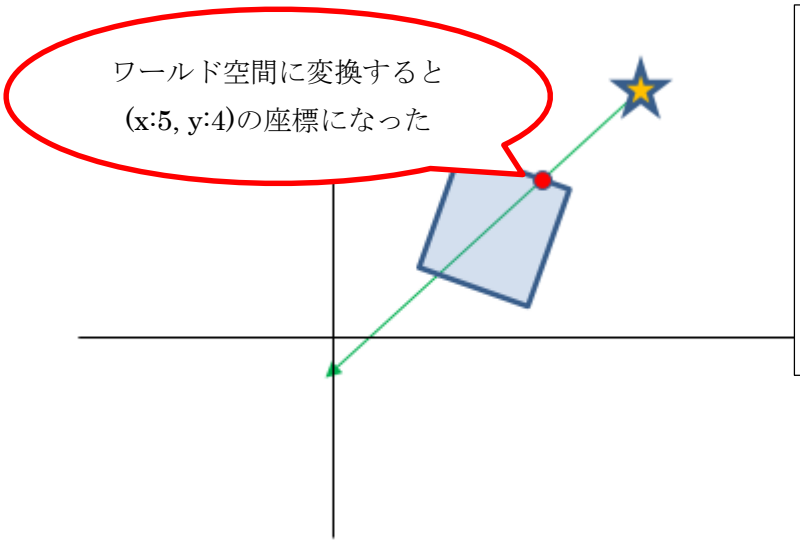
  この場合、レイの空間変換をしなくても問題なくレイキャストができる。



- メッシュが世界の中心から離れていたり、回転していたりすると、レイキャストがうまくいかない。
- これはメッシュの頂点座標がメッシュの位置を中心とした座標だからである。



- メッシュを世界の中心として考えたいので、レイをメッシュ空間に変換する
- レイをメッシュ空間に変換するには
  メッシュのワールド行列を逆行列化し、
  レイベクトルとメッシュ逆行列を乗算する

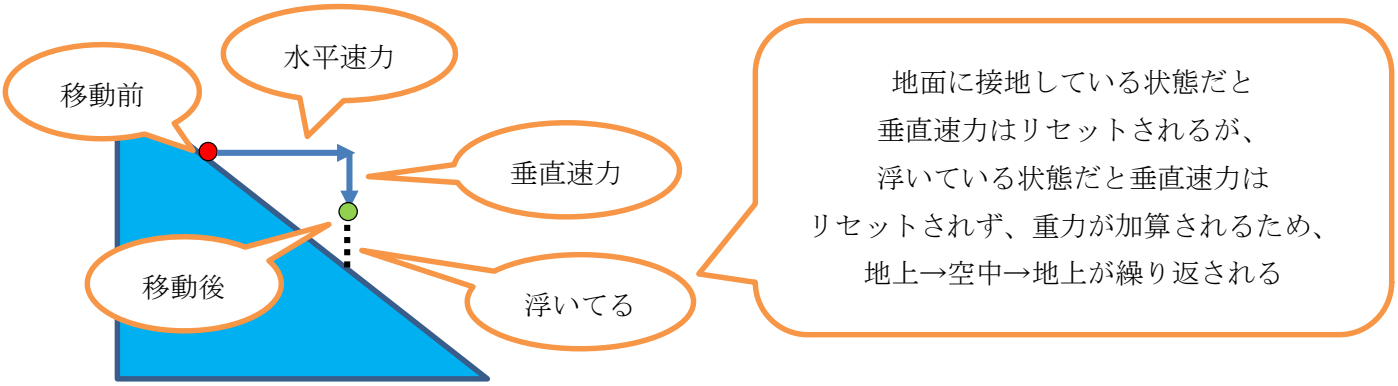


- 最終的の欲しい結果はワールド空間での座標なのでメッシュ空間の座標をワールド空間の座標へ変換する
- メッシュ空間の座標をワールド空間の座標に変換するにはメッシュ空間の座標とメッシュのワールド行列を乗算する

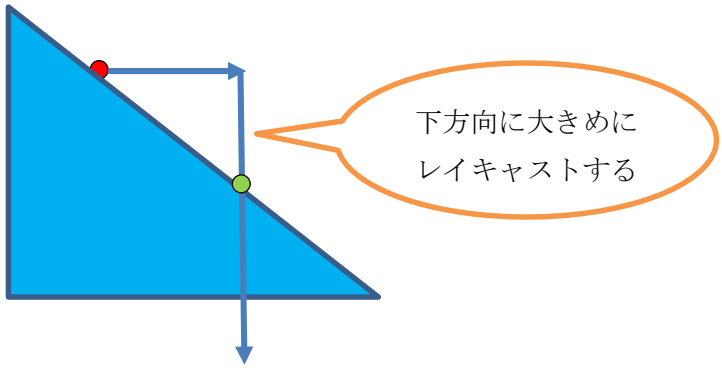
このように逆行列をうまく利用し、どこを中心に考えるかが重要になってきます。
今後も逆行列を使った空間変換は出てくるので慣れておきましょう。

〇下り坂移動でのガタガタを直す

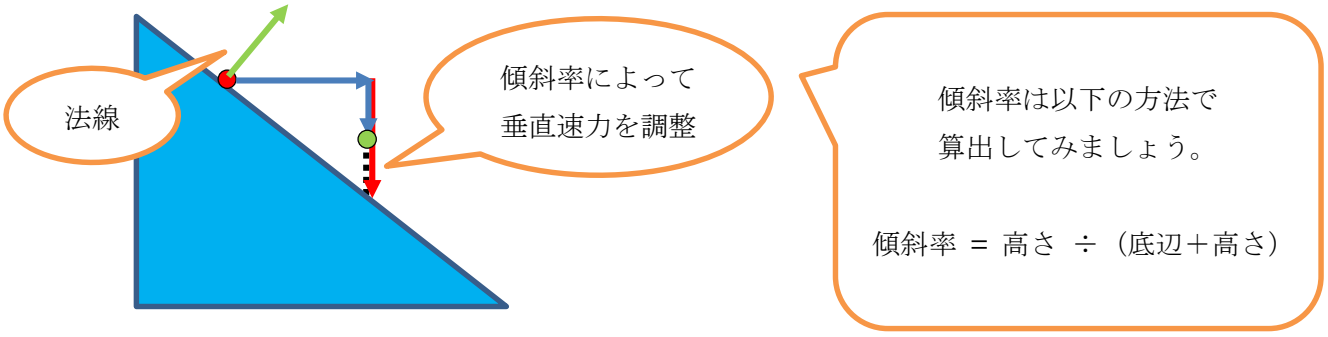
地形の高さに合わせて移動できるようになりましたが、現状の処理では下り坂を移動するとガタガタしてしまう場合があります。
これは坂の傾斜が高いと移動後に一瞬だけ浮いてしまうためです。
地上→空中→地上→空中を繰り返しているためにガタガタします。



修正方法としてはいくつかあります。
ジャンプ操作などをしないゲームの場合は下方向に対して大きめにレイキャストすれば良いです。



ジャンプなどするゲームの場合は傾斜の法線ベクトルを見て垂直速力を調整してあげましょう。
接地している状態で移動した場合は法線ベクトルから傾斜率を求めて調整しましょう。



Character.h

```
---省略---

// キャラクター
class Character
{
protected:
    ---省略---
    float                   slopeRate = 1.0f;
};
```

Character.cpp

```
---省略---

// 垂直移動更新処理
void Character::UpdateVerticalMove(float elapsedTime)
{
    ---省略---

    slopeRate = 0.0f;

    // 落下中
    if (my < 0.0f)
    {
        ---省略---

        // レイキャストによる地面判定
        HitResult hit;
        if (Stage::Instance().RayCast(start, end, hit))
        {
            ---省略---
            // 傾斜率の計算
            

            ---省略---
        }
```

```
        ---省略---
    }
    ---省略---
}

// 水平速力更新処理
void Character::UpdateHorizontalVelocity(float elapsedFrame)
{
    ---省略---

    // XZ平面の速力を加速する
    if (length <= maxMoveSpeed)
    {
        // 移動ベクトルがゼロベクトルでないなら加速する
        ---省略---
        if (moveVecLength > 0.0f)
        {
            ---省略---

            // 下り坂でガタガタしないようにする
            
        }
    }
    ---省略---
}
```

実装出来たら実行確認してみましょう。
下り坂をガタガタせず移動できたら OK です。

お疲れ様でした。