3D アクションゲームプログラミング

○評価要件

☑複数のステージオブジェクトに対し、レイキャストをできるようにする。
☑移動床の移動にあわせてキャラクターが移動するようにする。
☑移動床の回転にあわせてキャラクターも回転するようにする。

○概要

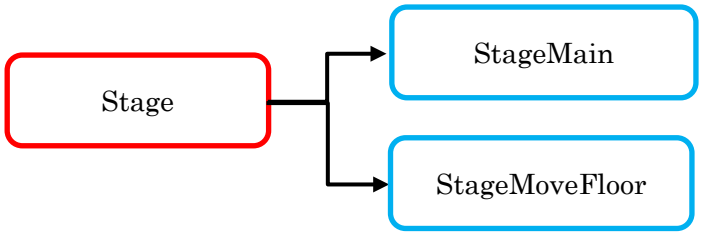
今回はステージを拡張し、移動する床を配置します。
今までは１つのステージに対してレイキャストを行うだけで良かったですが、今回は移動床もステージの一部としてレイキャストを行えるようにします。

また、移動床にキャラクターが乗った状態で床が移動した場合、床だけが移動してキャラクターを置いていってしまうため、床の移動にあわせてキャラクターも移動するような実装をしていきます。

○ステージのクラス設計を変更する

まずは移動床を作成しましょう。
現状ではステージが１つだけあり、そのステージに対してレイキャストを行っていました。
今回は現在表示しているステージに加えて移動床も配置することになり、移動床に対してもレイキャストをできるようにする必要があります。

現在、キャラクターは１つのステージに対してレイキャストを行い、地面や壁との衝突判定を行っています。
これをすべてのステージオブジェクトを対象としたレイキャストをできるようにステージオブジェクトを管理するステージマネージャークラスを作成しましょう。
ステージマネージャーは他のマネージャーと同じく、ステージ基底クラスを管理するようにするため、ステージクラスの設計を以下のように変更します。



Stage.cpp と Stage.h はファイル名を変更し、StageMain.cpp と StageMain.h にしましょう。
そして新たに Stage.h を作成し、下記プログラムコードを記述しましょう。

Stage.h

```
#pragma once

#include "Graphics/Shader.h"
#include "Collision.h"

// ステージ
class Stage
{
public:
    Stage() {}
    virtual ~Stage() {}

    // 更新処理
    virtual void Update(float elapsedTime) = 0;

    // 描画処理
    virtual void Render(ID3D11DeviceContext* dc, Shader* shader) = 0;

    // レイキャスト
    virtual bool RayCast(const DirectX::XMFLOAT3& start, const DirectX::XMFLOAT3& end,
                                                                HitResult& hit) = 0;
};
```

次は元々のステージクラスをメインステージクラスに変更します。

StageMain.h

```
#pragma once

#include "Graphics/Shader.h"
#include "Graphics/Model.h"
#include "Collision.h"
#include "Stage.h"

// メインステージ
class Stage
class StageMain : public Stage
{
public:
    Stage();
    StageMain ();
    ~Stage();
    ~StageMain () override;

    // インスタンス取得
    static Stage& Instance();

    // 更新処理
    void Update(float elapsedTime);
    void Update(float elapsedTime) override;

    // 描画処理
    void Render(ID3D11DeviceContext* dc, Shader* shader);
    void Render(ID3D11DeviceContext* dc, Shader* shader) override;

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

private:
    Model*      model = nullptr;
};
```

StageMain.cpp

```
#include "Stage.h"
#include "StageMain.h"

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()
StageMain::StageMain()
{
    instance = this;

    // ステージモデルを読み込み
    model = new Model("Data/Model/ExampleStage/ExampleStage.mdl");
}

// デストラクタ
Stage::~Stage()
StageMain::~StageMain()
{
    // ステージモデルを破棄
    delete model;
}

// 更新処理
void Stage::Update(float elapsedTime)
void StageMain::Update(float elapsedTime)
{
    // 今は特にやることはない
}

// 描画処理
void Stage::Render(ID3D11DeviceContext* dc, Shader* shader)
void StageMain::Render(ID3D11DeviceContext* dc, Shader* shader)
{
    // シェーダーにモデルを描画してもらう
    shader->Draw(dc, model);
}

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

続いてステージマネージャーを作成します。

StageManager.cpp と StageManager.h を作成し、下記プログラムコードを記述しましょう。

StageManager.h

```
#pragma once

#include <vector>
#include "Stage.h"

// ステージマネージャー
class StageManager
{
```

```
private:
    StageManager() {}
    ~StageManager() {}

public:
    // 唯一のインスタンス取得
    static StageManager& Instance()
    {
        static StageManager instance;
        return instance;
    }

    // 更新処理
    void Update(float elapsedTime);

    // 描画処理
    void Render(ID3D11DeviceContext* dc, Shader* shader);

    // ステージ登録
    void Register(Stage* stage);

    // ステージ全削除
    void Clear();

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

private:
    std::vector<Stage*>    stages;
};
```

StageManager.cpp

```
#include "StageManager.h"

// 更新処理
void StageManager::Update(float elapsedTime)
{
    for (Stage* stage : stages)
    {
        stage->Update(elapsedTime);
    }
}

// 描画処理
void StageManager::Render(ID3D11DeviceContext* context, Shader* shader)
{
    for (Stage*stage : stages)
    {
        stage->Render(context, shader);
    }
}

// ステージ登録
void StageManager::Register(Stage* stage)
```

```
{
    stages.emplace_back(stage);
}

// ステージ全削除
void StageManager::Clear()
{
    for (Stage* stage : stages)
    {
        delete stage;
    }
    stages.clear();
}

// レイキャスト
bool StageManager::RayCast(const DirectX::XMFLOAT3& start, const DirectX::XMFLOAT3& end,
                                                                    HitResult& hit)
{
    bool result = false;
    hit.distance = FLT_MAX;

    for (Stage* stage : stages)
    {
        HitResult tmp;
        if (stage->RayCast(start, end, tmp))
        {
            if (hit.distance > tmp.distance)
            {
                hit = tmp;
                result = true;
            }
        }
    }

    return result;
}
```

全てのステージオブジェクトに対して
レイキャストを行い、
衝突した交点が一番近い情報を取得する

ステージマネージャーが完成したので、各所を修正していきましょう。

SceneGame.h

```
#pragma once

#include "Stage.h"
---省略---

// ゲームシーン
class SceneGame : public Scene
{
public:
    ---省略---

private:
    Stage* stage = nullptr;
```

```
    ---省略---
};
```

SceneGame.cpp

```
---省略---
#include "StageManager.h"
#include "StageMain.h"

// 初期化
void SceneGame::Initialize()
{
    // ステージ初期化
    stage = new Stage();
    StageManager& stageManager = StageManager::Instance();
    StageMain* stageMain = new StageMain();
    stageManager.Register(stageMain);

    ---省略---
}

// 終了化
void SceneGame::Finalize()
{
    ---省略---

    // ステージ終了化
    if (stage != nullptr)
    {
        delete stage;
        stage = nullptr;
    }
    StageManager::Instance().Clear();
}

// 更新処理
void SceneGame::Update(float elapsedTime)
{
    ---省略---

    // ステージ更新処理
    stage->Update(elapsedTime);
    StageManager::Instance().Update(elapsedTime);

    ---省略---
}

// 描画処理
void SceneGame::Render()
{
    ---省略---

    // 3Dモデル描画
    {
        ---省略---
```

```
        // ステージ描画
        stage->Render(dc, shader);
        StageManager::Instance().Render(dc, shader);

        ---省略---
    }

    ---省略---
}
```

Character.cpp

```
---省略---
#include "Stage.h"
#include "StageManager.h"
---省略---

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

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

        // レイキャストによる地面判定
        ---省略---
        if (Stage::Instance().RayCast(start, end, hit))
        if (StageManager::Instance().RayCast(start, end, hit))
        {
            ---省略---
        }
        ---省略---
    }
    ---省略---
}

---省略---

// 水平移動更新処理
void Character::UpdateHorizontalMove(float elapsedTime)
{
    // 水平速力量計算
    ---省略---
    if (---省略---)
    {
        ---省略---

        // レイキャストによる壁判定
        ---省略---
        if (Stage::Instance().RayCast(start, end, hit))
        if (StageManager::Instance().RayCast(start, end, hit))
        {
```

```
            ---省略---

            // 壁ずり方向へレイキャスト
            ---省略---
            if (!Stage::Instance().RayCast(hit.position, collectPosition, hit2))
            if (!StageManager::Instance().RayCast(hit.position, collectPosition, hit2))
            {
                ---省略---
            }
            ---省略---
        }
        ---省略---
    }
}
```

ここまで実装出来たら実行確認をしてみましょう。
以前と同じようにステージが表示され、移動操作ができていれば OK です。

○移動床を作成する

ステージマネージャーを実装したことにより、複数のステージオブジェクトを扱えるようになりました。次は移動床を作成し、ステージマネージャーに登録しましょう。
今回の移動床はスタート地点とゴール地点を設定し、一定速度でスタートとゴールを行ったり来たりすることにします。
StageMoveFloor.cpp と StageMoveFloor.h を作成し、下記プログラムコードを記述しましょう。

StageMoveFloor.h

```
#pragma once

#include "Graphics/Model.h"
#include "Stage.h"

// 移動床ステージ
class StageMoveFloor : public Stage
{
public:
    StageMoveFloor();
    ~StageMoveFloor();

    // 更新処理
    void Update(float elapsedTime) override;

    // 描画処理
    void Render(ID3D11DeviceContext* dc, Shader* shader) override;

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

    // スタート位置設定
```

```
    void SetStartPoint(const DirectX::XMFLOAT3& start) { this->start = start; }

    // ゴール位置設定
    void SetGoalPoint(const DirectX::XMFLOAT3& goal) { this->goal = goal; }

    // トルク設定
    void SetTorque(const DirectX::XMFLOAT3& torque) { this->torque = torque; }

private:
    // 行列更新処理
    void UpdateTransform();

private:
    Model*                  model = nullptr;
    DirectX::XMFLOAT3       position = { 0, 0, 0 };
    DirectX::XMFLOAT3       angle = { 0, 0, 0 };
    DirectX::XMFLOAT3       scale = { 1, 1, 1 };
    DirectX::XMFLOAT4X4     transform = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 };

    DirectX::XMFLOAT3       torque = { 0, 0, 0 };
    DirectX::XMFLOAT3       start = { 0, 0, 0 };
    DirectX::XMFLOAT3       goal = { 0, 0, 0 };

    float                   moveSpeed = 2.0f;
    float                   moveRate = 0.0f;
};
```

StageMoveFloor.cpp

```
#include "StageMoveFloor.h"

// コンストラクタ
StageMoveFloor::StageMoveFloor()
{
    scale.x = scale.z = 3.0f;
    scale.y = 0.5f;

    // ステージモデルを読み込み
    model = new Model("Data/Model/Cube/Cube.mdl");
}

StageMoveFloor::~StageMoveFloor()
{
    // ステージモデルを破棄
    delete model;
}

// 更新処理
void StageMoveFloor::Update(float elapsedTime)
{
    // スタートからゴールまでの距離を算出する
    DirectX::XMVECTOR Start = DirectX::XMLoadFloat3(&start);
    DirectX::XMVECTOR Goal = DirectX::XMLoadFloat3(&goal);
    DirectX::XMVECTOR Vec = DirectX::XMVectorSubtract(Goal, Start);
    DirectX::XMVECTOR Length = DirectX::XMVector3Length(Vec);
```

```
    float length;
    DirectX::XMStoreFloat(&length, Length);

    // スタートからゴールまでの間を一秒間で進む割合（0.0～1.0）を算出する
    float speed = moveSpeed * elapsedTime;
    float speedRate = speed / length;
    moveRate += speedRate;

    // ゴールに到達、またはスタートに戻った場合、移動方向を反転させる
    if (moveRate <= 0.0f || moveRate >= 1.0f)
    {
        moveSpeed = -moveSpeed;
    }

    // 線形補完で位置を算出する
    DirectX::XMVECTOR Position = DirectX::XMVectorLerp(Start, Goal, moveRate);
    DirectX::XMStoreFloat3(&position, Position);

    // 回転
    angle.x += torque.x * elapsedTime;
    angle.y += torque.y * elapsedTime;
    angle.z += torque.z * elapsedTime;

    // 行列更新
    UpdateTransform();

    // モデル行列更新
    model->UpdateTransform(transform);
}

// 描画処理
void StageMoveFloor::Render(ID3D11DeviceContext* dc, Shader* shader)
{
    shader->Draw(dc, model);
}

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

// 行列更新処理
void StageMoveFloor::UpdateTransform()
{
    DirectX::XMMATRIX S = DirectX::XMMatrixScaling(scale.x, scale.y, scale.z);
    DirectX::XMMATRIX R = DirectX::XMMatrixRotationRollPitchYaw(angle.x, angle.y, angle.z);
    DirectX::XMMATRIX T = DirectX::XMMatrixTranslation(position.x, position.y, position.z);
    DirectX::XMMATRIX W = S * R * T;
    DirectX::XMStoreFloat4x4(&transform, W);
}
```

移動床クラスができたのでシーンに配置しましょう。

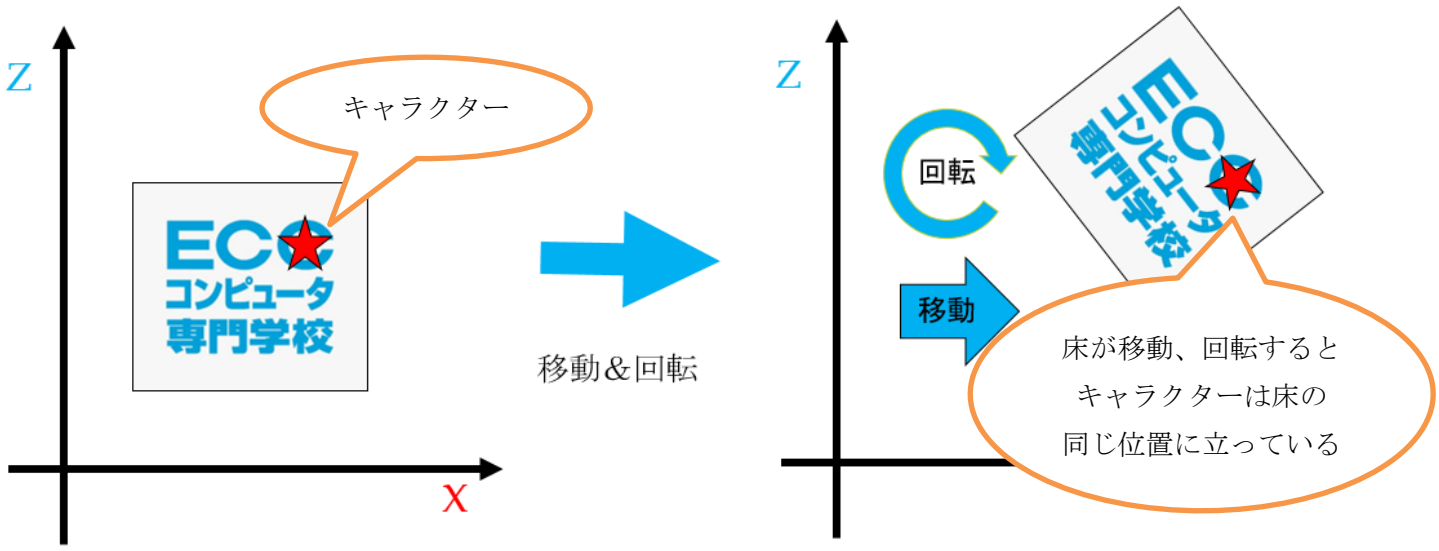
SceneGame.cpp

```
---省略---
#include "StageMoveFloor.h"

// 初期化
void SceneGame::Initialize()
{
    // ステージ初期化
    ---省略---
    StageMoveFloor* stageMoveFloor = new StageMoveFloor();
    stageMoveFloor->SetStartPoint(DirectX::XMFLOAT3(0, 1, 3));
    stageMoveFloor->SetGoalPoint(DirectX::XMFLOAT3(10, 2, 3));
    stageMoveFloor->SetTorque(DirectX::XMFLOAT3(0, 1.0f, 0));
    stageManager.Register(stageMoveFloor);

    ---省略---
}
```

移動床の配置ができたら実行確認をしてみましょう。
移動床が表示され、移動床がスタート地点からゴール地点までの往復を移動していれば OK です。
この後、移動床にジャンプして乗ってみて下さい。
移動床に上に立つことはできるが、移動床の移動によってキャラクターが置いていかれるはずです。

○移動床に乗れるようにする

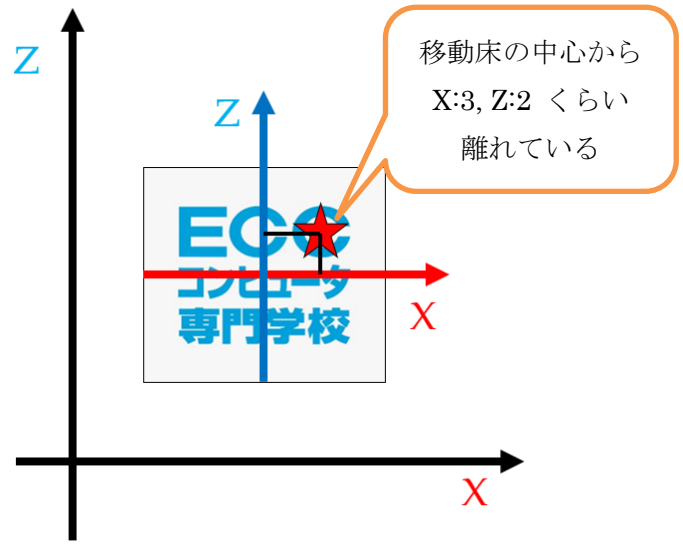
キャラクターが移動床に乗っていた時、移動床が移動したとき、キャラクターも移動、回転する。



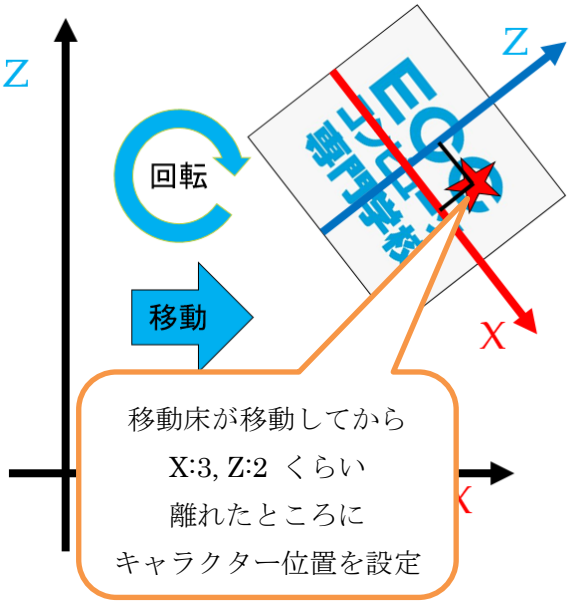
これを実現させるには逆行列を使います。
以前、レイキャストの実装の際、レイの空間変換について解説しました。
原理はあれと同じです。

上図だと移動床「ECC」の C の位置にキャラクターが立っています。
この移動床が移動したとき、キャラクターを移動させるというよりは移動床が移動しても C の位置に立っていれば良いわけです。

つまり、キャラクターの位置は移動床を中心に相対位置を計算し、移動床が移動した後にキャラクターの位置を移動床からの計算した相対位置に設定すれば良いのです。



- キャラクターの位置を移動床の位置を中心として考える。
- 移動床の行列を逆行列化する。
- 逆行列化した行列とキャラクターのワールド座標と乗算する。
- その結果、移動床を中心と考えたキャラクターの位置が算出される。



- 移動床を移動、回転する。
- 移動、回転後の移動床のワールド行列と移動床を中心と考えたキャラクターの位置を乗算する。
- その結果、移動した移動床に追従したキャラクターのワールド位置が算出される。

考え方としては以上です。
これをキャラクターがステージにレイキャストした結果として返すようにします。
つまり、移動床クラスのレイキャスト関数では単純にレイとポリゴンの交点を返すのではなく、移動床の移動量が反映した結果を返すようにするということです。

具体的にどうするのかというと、まずレイキャストを移動前の行列に行います。

1. レイキャスト関数でレイを移動前の行列を逆行列化し、レイの空間変換を行います。
2. レイとモデルの衝突判定を行い、移動床を中心とした空間での交点を算出します。
3. 衝突した交点と移動後の行列の乗算を行い、ワールド空間での交点を算出します。

この際、レイとモデルの衝突判定の際に渡すモデルは UpdateTransform()でワールド行列を原点にした状態で渡す必要があります。
空間変換に慣れていないとかなりややこしいですがプログラムを実装してみましょう。

StageMoveFloor.h

```
---省略---

// 移動床ステージ
class StageMoveFloor : public Stage
{
    ---省略---

private:
    ---省略---
    DirectX::XMFLOAT4X4    oldTransform = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 };
};
```

StageMoveFloor.cpp

```
---省略---

// 更新処理
void StageMoveFloor::Update(float elapsedTime)
{
    // 前回の情報を保存
    oldTransform = transform;

    ---省略---

    // モデル行列更新
    model->UpdateTransform(DirectX::XMLoadFloat4x4(&transform));
    // レイキャスト用にモデル空間行列にするため単位行列を渡す
    const DirectX::XMFLOAT4X4 transformIdentity = { 1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1 };
    model->UpdateTransform(transformIdentity);
}

// 描画処理
void StageMoveFloor::Render(ID3D11DeviceContext* dc, Shader* shader)
{
    // 表示用のためワールド行列に更新する
    model->UpdateTransform(transform);

    ---省略---
}
```

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

    // 前回のワールド行列と逆行列を求める
    DirectX::XMMATRIX WorldTransform = DirectX::XMLoadFloat4x4(&oldTransform);
    DirectX::XMMATRIX InverseWorldTransform = DirectX::XMMatrixInverse(nullptr, WorldTransform);

    // 前回のローカル空間でのレイに変換
    DirectX::XMVECTOR WorldStart = DirectX::XMLoadFloat3(&start);
    DirectX::XMVECTOR WorldEnd = DirectX::XMLoadFloat3(&end);
    DirectX::XMVECTOR LocalStart = DirectX::XMVector3TransformCoord(WorldStart,
                                                            InverseWorldTransform);
    DirectX::XMVECTOR LocalEnd = DirectX::XMVector3TransformCoord(WorldEnd,
                                                            InverseWorldTransform);

    // ローカル空間でのレイとの交点を求める
    DirectX::XMFLOAT3 localStart, localEnd;
    DirectX::XMStoreFloat3(&localStart, LocalStart);
    DirectX::XMStoreFloat3(&localEnd, LocalEnd);

    HitResult localHit;
    if (Collision::IntersectRayVsModel(localStart, localEnd, model, localHit))
    {
        // 前回のローカル空間から今回のワールド空間へ変換
        // 前回から今回にかけて変更された内容が乗っているオブジェクトに反映される。
        WorldTransform = DirectX::XMLoadFloat4x4(&transform);
        DirectX::XMVECTOR LocalPosition = DirectX::XMLoadFloat3(&localHit.position);
        DirectX::XMVECTOR WorldPosition = DirectX::XMVector3TransformCoord(LocalPosition,
                                                                WorldTransform);
        DirectX::XMVECTOR LocalNormal = DirectX::XMLoadFloat3(&localHit.normal);
        DirectX::XMVECTOR WorldNormal = DirectX::XMVector3TransformNormal(LocalNormal,
                                                                WorldTransform);
        DirectX::XMVECTOR Vec = DirectX::XMVectorSubtract(WorldPosition, WorldStart);
        DirectX::XMVECTOR Dist = DirectX::XMVector3Length(Vec);
        DirectX::XMStoreFloat3(&hit.position, WorldPosition);
        DirectX::XMStoreFloat3(&hit.normal, WorldNormal);
        DirectX::XMStoreFloat(&hit.distance, Dist);
        hit.materialIndex = localHit.materialIndex;
        return true;
    }
    return false;
}

---省略---
```

Character.cpp

```
---省略---

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

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

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

実装ができたら実行確認をしてみましょう。
移動床に乗ってみて移動床の移動にあわせてキャラクターが移動していれば OK です。
ただし、回転が反映されていません。

○回転の反映

今のレイキャストの仕様ではレイとポリゴンの交点の位置と法線ベクトルしか返していないため、回転値は反映できません。
真面目に回転の反映を行うならば移動床の回転差分を回転行列やクォータニオンなどで返すように拡張するべきですが、今回は簡易的に行います。

この移動床はオイラー角で回転制御を行っており、キャラクターもオイラー角で回転制御を行っているため、単純なオイラー角の差分を返すことにします。

Collision.h

```
---省略---

// ヒット結果
struct HitResult
{
    ---省略---
    DirectX::XMFLOAT3  rotation = { 0, 0, 0 };    // 回転量
};

---省略---
```

StageMoveFloor.h

```
---省略---

// 移動床ステージ
class StageMoveFloor : public Stage
{
    ---省略---
private:
    ---省略---
    DirectX::XMFLOAT3      oldAngle = { 0, 0, 0 };
};
```

StageMoveFloor.cpp

```
// 更新処理
void StageMoveFloor::Update(float elapsedTime)
{
    // 前回の情報を保存
    oldTransform = transform;
    oldAngle = angle;

    // スタートからゴールまでの距離を算出する
    ---省略---
}


---省略---
// レイキャスト
bool StageMoveFloor::RayCast(const DirectX::XMFLOAT3& start, const DirectX::XMFLOAT3& end,
HitResult& hit)
{
    // 前回のワールド行列と逆行列を求める
    ---省略---

    // 前回のローカル空間でのレイに変換
    ---省略---

    // ローカル空間でのレイとの交点を求める
    ---省略---

    HitResult localHit;
    if (Collision::IntersectRayVsModel(localStart, localEnd, model, localHit))
    {
        // 前回のローカル空間から今回のワールド空間へ変換
        // 前回から今回にかけて変更された内容が乗っているオブジェクトに反映される。
        ---省略---

        // 回転差分を算出
        hit.rotation.x = angle.x - oldAngle.x;
        hit.rotation.y = angle.y - oldAngle.y;
        hit.rotation.z = angle.z - oldAngle.z;
        return true;
    }
    return false;
}
```

Character.cpp

```
---省略---

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

    // 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 (StageManager::Instance().RayCast(start, end, hit))
        {
            // 地面に接地している
            ---省略---

            // 回転
            angle.y += hit.rotation.y;

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

実装ができたら実行確認をしてみましょう。
移動床に乗り、移動床の回転に合わせてキャラクターも回転していれば OK です。

お疲れ様でした。