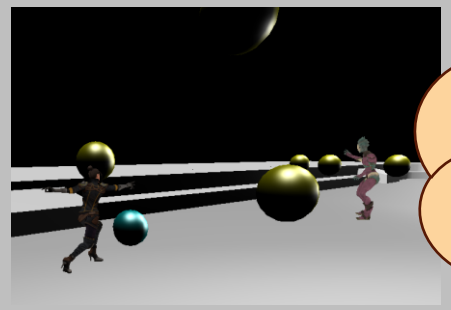
お米大好きなプログラマーの
ポートフォリオ

隙間時間に
Photoshopで作った
お米風プロフィール画像

学校：福岡情報IT・クリエイター専門学校
名前：河村 晃希（かわむら こうき）
好きな食べ物：ご米
PGスキル：C, C++, C#, HLSL
ゲーム制作環境：VisualStudio2022 C++ (DXライブラリ使用)

最新の現在製作中のゲーム



操作武器は
剣、斧、魔法の三種を実装
それぞれ3連撃コンボが可能

クォータービュー視点

TPS視点
with ターゲットロック

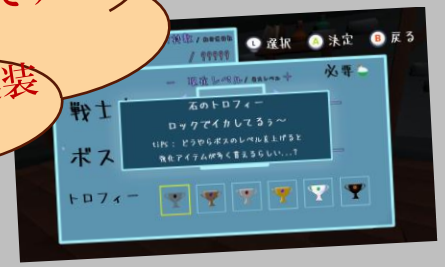
ハイアングル視点

```enum class State
{
    MOVE,
    ATTACK,
    DODGE,
    KNOCK,
    JUMP
};

class StateBase
{
public:
    StateBase(){}
    ~StateBase(){}

    State GetState() { return state; }

    /* 各状態移行の許可項目 */
    // ダメージをうけるか
    bool isOnDamage = false;
    // 動けるか
    bool isMove = false;
    // 動作方向を変えられるか
    bool isMoveDir = false;
    // ノックバックを受けるか
    bool isKnockBack = false;
    // 攻撃に移れるか
    bool isAttack = false;
    // 回避できるか
    bool isDodge = false;
    // ジャンプできるか
    bool isJump = false;

    State state = State::MOVE;

};```

参考にしたサイトでは、
「各ステートはキャラクター達の
　行動決定をするためのもの」
という意向のもとで実装
別の考え方もありそうだったため、
今後も研究予定

# 2024年11月~
# (2月頃完成予定)
# ハック＆スラッシュ
# 1マップバトルアクション
# TACTICAL POWER


現在制作中の作品。

キャラクターにはステートクラスを実装。

カメラはクォータービュー、TPS、
ハイアングルの３視点実装し、
状況に応じて使い分ける予定。
また、カメラのイージングやボスへの
ターゲット視点固定も可能。

また、カプセルの当たり判定の実装により、武器が敵にヒットしたときのような、詳細な当たり判定をとることが可能に。

今後は各キャラクターのパラメータ周りをcsvファイル等を用いながら設計して
ハック＆スラッシュの部分を
作り上げつつ、制作を進める予定。
```

１本の作品として完成させたゲームたち

Quaternionクラスの実装
移動ベクトルを入れることで
平行移動もできるように実装

クリアで手に入るアイテムを用いてゲーム難易度を自分で調整可能
お遊び要素としてトロフィーも実装




```cpp
class Quaternion
{
public:
    Quaternion()
    {
        Qu.w = 1;
        Qu.x = 0;
        Qu.y = 0;
        Qu.z = 0;
        vec = VGet(0, 0, 0);
    };
    virtual ~Quaternion() {}

private:
    struct Q
    {
        float w;
        float x;
        float y;
        float z;


        Q operator * (const Q& _q) const
        {
            Q tempQ;

            /*クオータニオンの掛け算*/
            //公式通りです。
            tempQ.w = w * _q.w - x * _q.x - y * _q.y - z * _q.z;//実部
            tempQ.x = w * _q.x + x * _q.w + y * _q.z - z * _q.y;//虚部x
            tempQ.y = w * _q.y + y * _q.w + z * _q.x - x * _q.z;//虚部y
            tempQ.z = w * _q.z + z * _q.w + x * _q.y - y * _q.x;//虚部z

            return tempQ;
        }
    };

    Q Qu;
    VECTOR vec;

public:
    /// <summary>
    /// 移動量の設定
    /// </summary>
    /// <param name="_angle">1フレーム当たりの角度（ ラジアン値 ）</param>
    /// <param name="_axis">回転の軸ベクトル</param>
    /// <param name="_moveVec">平行移動ベクトル</param>
    void SetMove(float& _angle, VECTOR& _axis, VECTOR& _moveVec)
    {
        Qu.w = cos(_angle / 2.0f);//実部
        Qu.x = _axis.x * sin(_angle / 2.0f);
        Qu.y = _axis.y * sin(_angle / 2.0f);
        Qu.z = _axis.z * sin(_angle / 2.0f);
        vec = _moveVec;
    }

    /// <summary>
    /// クォータニオンによる実移動
    /// </summary>
    /// <param name="_rotPoint">回転する中心の座標</param>
    /// <param name="_pos">回転するものの座標</param>
    /// <returns>回転後の座標</returns>
    VECTOR Move(VECTOR& _rotPoint, VECTOR& _pos)
    {
        Q qPos, qInv;
        VECTOR vPos;

        //3次元座標をクオータニオンに変換
        qPos.w = 1.0f;
        qPos.x = _pos.x - _rotPoint.x;
        qPos.y = _pos.y - _rotPoint.y;
        qPos.z = _pos.z - _rotPoint.z;

        //回転クォータニオンのインバースの作成
        //逆クォータニオンを出すのは大変なので、
        //3次元だと同じ値になる共役クオータニオンで作成(虚部だけマイナス反転)
        qInv.w = Qu.w;
        qInv.x = -Qu.x;
        qInv.y = -Qu.y;
        qInv.z = -Qu.z;

        //回転後のクオータニオンの作成
        qPos = Qu * qPos * qInv;

        //3次元座標に戻す
        vPos.x = qPos.x + _rotPoint.x;
        vPos.y = qPos.y + _rotPoint.y;
        vPos.z = qPos.z + _rotPoint.z;

        // 回転側に移動
        vPos.x += vec.x;
        vPos.y += vec.y;
        vPos.z += vec.z;

        return vPos;
    }
};
```

2024年8月～10月
１マップ型3Dバトルアクション
ORDEAL MAZE

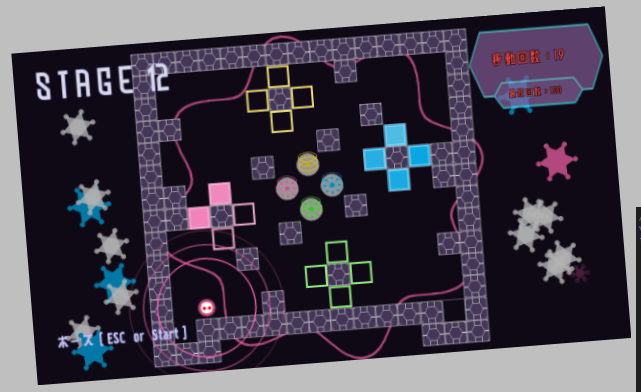
完成した１本のゲームとしては初の作品

1vs1でボスバトルを行い、
強化して自身で難易度を調整しつつ、
繰り返しバトルを行うゲーム。

この作品からカメラの回転には
Quaternionを採用。

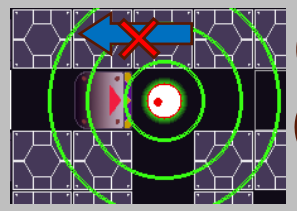
また、Effekseerというエフェクトの
制作ツール、ライブラリを使用して
より戦闘感を付与。

エフェクトやライトによるゲームの
重さの課題を抱え、作っていて自身
の勉強となった作品。

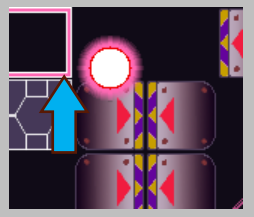


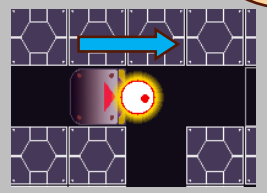

```
// セーブ
void File::Save()
{
    // ファイルの枠作成
    FILE* Writefp = nullptr;

    // == 0  でファイル存在
    if (fopen_s(&Writefp, "TimeData", "wb") == 0)
    {
        for (int i = 0; i < 5; i++)
        {
            if (m_timeRank[i] > 0)
            {
                fwrite(&m_timeRank[i], sizeof(int), 1, Writefp);
            }
        }

        fclose(Writefp);
    }
    else
    {
        assert(false);
        return;
    }
}

// ロード
void File::Load()
{
    FILE* Readfp = nullptr;
    // == 0  でファイル存在
    if (fopen_s(&Readfp, "TimeData", "rb") == 0)
    {
        // 配列個数分のintを入れる
        for (int i = 0; i < 5; i++)
        {
            fread(&m_timeRank[i], sizeof(int), 1, Readfp);
        }
        fclose(Readfp);
    }
    // ファイルが無い(初プレイ)時の初期データ作成
    else
    {
        CreateData();
        Save();
        Load();
        return;
    }
}
```

バイナリデータの
読み書きを行って、
回数記録の
セーブ＆ロードを実装



ギミックとして移動方向制限床を実装
プレイヤーの進行方向から通過可能かを判定





```
namespace
{
    // マップチップの情報
    constexpr int kChipWidth = 48;
    constexpr int kChipHeight = 48;

    // チップのマス数
    constexpr int kChipNumX = 19;
    constexpr int kChipNumY = 14;


    ////////////////////////
    // マップチップの配置情報 //
    ////////////////////////////////////////////////////
    //  0 : 空白                                      //
    //  1 : 壁                                        //
    //  2 : プレイヤー                                //
    //  3 : 色変更床 (30:赤 31:緑 32:青 33:黄 (34:白)) //
    //  4 : 着色床 (40:赤 41:緑 42:青 43:黄 (44:白))  //
    //  5 : 一方通行床 (50:上, 51:左, 52:下, 53:右)   //
    //  6 : ゴール                                    //
    ////////////////////////////////////////////////////
    constexpr int kStage[Game::kStageNum][kChipNumY][kChipNumX] =
    {
        // ステージ0(サンプルステージ)(本編に実装はしない)★★★★
        {
            {1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 },
            {1 ,30,0 ,0 ,0 ,51,0 ,0 ,40,1 ,41,0 ,0 ,53,0 ,0 ,0 ,31,1 },
            {1 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,1 },
            {1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 },
            {1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,1 },
            {1 ,43,0 ,0 ,0 ,50,0 ,0 ,1 ,0 ,1 ,0 ,0 ,50,0 ,0 ,0 ,42,1 },
            {1 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,2 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 },
            {1 ,41,0 ,0 ,0 ,52,0 ,0 ,1 ,0 ,1 ,0 ,0 ,52,0 ,0 ,0 ,40,1 },
            {1 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,1 },
            {1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 },
            {1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,6 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 },
            {1 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,1 },
            {1 ,32,0 ,0 ,0 ,51,0 ,0 ,42,1 ,43,0 ,0 ,53,0 ,0 ,0 ,33,1 },
            {1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 },
        },
        // ステージ1(色変え)☆
        {
            {0 ,0 ,0 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,40,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,1 ,0 ,1 ,1 ,1 ,1 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,1 ,0 ,1 ,1 ,1 ,1 ,1 ,0 ,1 ,1 ,1 ,1 ,1 ,0 ,0 ,0 },
            {0 ,0 ,0 ,1 ,32,0 ,0 ,0 ,0 ,40,0 ,0 ,0 ,0 ,42,1 ,0 ,0 ,0 },
            {0 ,0 ,0 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,0 ,1 ,1 ,1 ,1 ,1 ,0 ,0 ,0 },
            {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,30,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,6 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,2 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
            {0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 },
```

マップチップの配置は
namespace内にて数字で割り当てて管理、参照
当時はcsvファイルを使う
発想が無く…

2023年10月～2024年2月
1画面型パズルアクション2D
GLOW RULER

個人として初の作品。

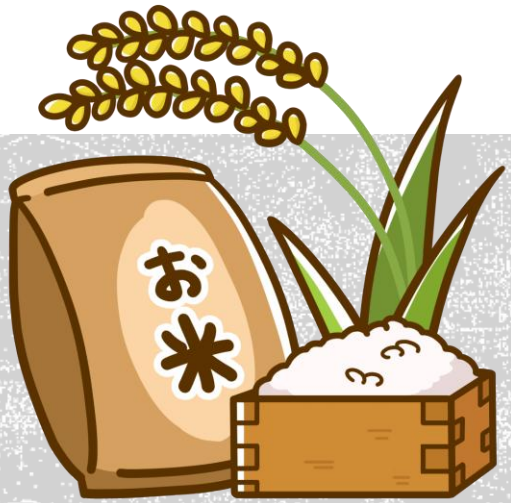
円状の装置で色を取得し、対応する色の床を光らせる。全て光らせたらゴールを目指し、その移動回数を測るゲーム。

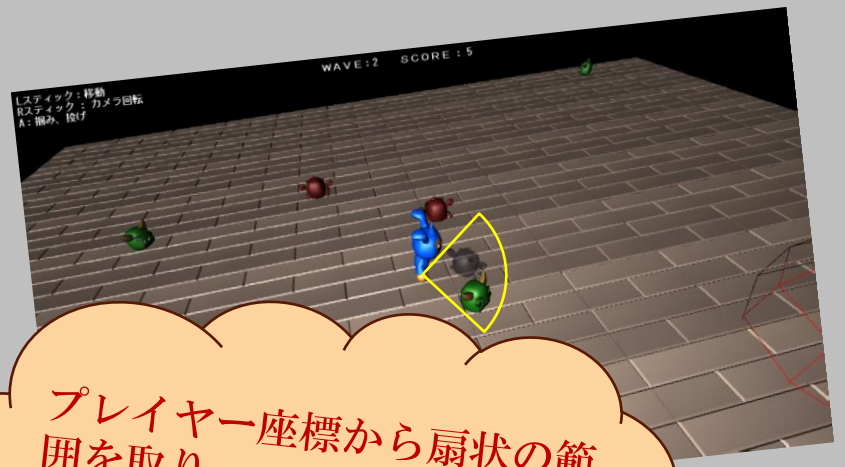
「失敗やゲームオーバーは作らない」
「繰り返し挑戦してもらう」
という考えを元に制作。

データファイルの読み込み、書き込みをこの時点で作成。この作品では主に移動回数の記録として使用。

背景や横の模様などは、アニメーション付きの画像などが見つからず、自力で描画。

１本のゲーム化とまではいかなかったが、機能だけ制作した簡易ゲームたち





プレイヤー座標から扇状の範囲を取り、
最接近キャラクターをつかむ対象として認識

行列クラスの実装
移動、回転、拡大縮小、
行列同士の計算まで可能

```
class Matrix
{
private:
    float matrix[4][4] = { 0.0f };

public:
    Matrix()
    {
        Unit();
    }

    // 移動するオブジェクトのVECTORを渡すことで移動後のVECTORを返す

    /// <summary>
    /// 移動、回転、拡大縮小の行列計算
    /// </summary>
    /// <param name="target">対象の座標</param>
    /// <param name="isMove">移動のフラグ(falseで移動のみ停止)</param>
    /// <returns></returns>
    VECTOR Move(VECTOR& target , bool isMove)
    {
        VECTOR response;
        response = VGet((matrix[0][0] * target.x + matrix[0][1] * target.y + matrix[0][2] * target.z + matrix[0][3] * isMove),     // X 座標
                        (matrix[1][0] * target.x + matrix[1][1] * target.y + matrix[1][2] * target.z + matrix[1][3] * isMove),     // Y 座標
                        (matrix[2][0] * target.x + matrix[2][1] * target.y + matrix[2][2] * target.z + matrix[2][3] * isMove));    // Z 座標

        return response;
    }

    // 拡大・縮小( 回転を設定値に設定 )
    void Scaling(float scale)
    {
        Matrix temp;
        temp.Unit();
        temp.matrix[0][0] = scale;
        temp.matrix[1][1] = scale;
        temp.matrix[2][2] = scale;

        Composition(temp);
    }

    // 移動
    void Shift(VECTOR& vec)
    {
        Matrix temp;
        temp.Unit();
        temp.matrix[0][3] = vec.x;
        temp.matrix[1][3] = vec.y;
        temp.matrix[2][3] = vec.z;

        Composition(temp);
    }

    // Y軸回転
    void RotateY(float& angle)
    {
        Matrix temp;
        temp.Unit();
        temp.matrix[0][0] = cos(angle);
        temp.matrix[0][2] = sin(angle);
        temp.matrix[2][0] = -sin(angle);
        temp.matrix[2][2] = cos(angle);

        Composition(temp);
    }

    // X軸回転
    void RotateX(float& angle)
    {
        Matrix temp;
        temp.Unit();
        temp.matrix[1][1] = cos(angle);
        temp.matrix[1][2] = sin(angle);
        temp.matrix[2][1] = -sin(angle);
        temp.matrix[2][2] = cos(angle);

        Composition(temp);
    }

    // 単位ベクトル化
    void Unit()
    {
        for (int line = 0; line < 4; line++)
        {
            for (int row = 0; row < 4; row++)
            {
                matrix[line][row] = 0;
            }
        }

        matrix[0][0] = 1.0f;
        matrix[1][1] = 1.0f;
        matrix[2][2] = 1.0f;
        matrix[3][3] = 1.0f;
    }

    // VECTOR化
    VECTOR GetMove() const
    {
        const VECTOR temp = VGet((matrix[0][3]),     // X 座標
                                 (matrix[1][3]),     // Y 座標
                                 (matrix[2][3]));    // Z 座標
        return temp;
    }

    // 行列合成
    void Composition(Matrix base)
    {
        Matrix temp;
        temp.matrix[0][0] = (matrix[0][0] * base.matrix[0][0]) + (matrix[0][1] * base.matrix[1][0]) + (matrix[0][2] * base.matrix[2][0]) + (matrix[0][3] * base.matrix[3][0]);
        temp.matrix[0][1] = (matrix[0][0] * base.matrix[0][1]) + (matrix[0][1] * base.matrix[1][1]) + (matrix[0][2] * base.matrix[2][1]) + (matrix[0][3] * base.matrix[3][1]);
        temp.matrix[0][2] = (matrix[0][0] * base.matrix[0][2]) + (matrix[0][1] * base.matrix[1][2]) + (matrix[0][2] * base.matrix[2][2]) + (matrix[0][3] * base.matrix[3][2]);
        temp.matrix[0][3] = (matrix[0][0] * base.matrix[0][3]) + (matrix[0][1] * base.matrix[1][3]) + (matrix[0][2] * base.matrix[2][3]) + (matrix[0][3] * base.matrix[3][3]);

        temp.matrix[1][0] = (matrix[1][0] * base.matrix[0][0]) + (matrix[1][1] * base.matrix[1][0]) + (matrix[1][2] * base.matrix[2][0]) + (matrix[1][3] * base.matrix[3][0]);
        temp.matrix[1][1] = (matrix[1][0] * base.matrix[0][1]) + (matrix[1][1] * base.matrix[1][1]) + (matrix[1][2] * base.matrix[2][1]) + (matrix[1][3] * base.matrix[3][1]);
        temp.matrix[1][2] = (matrix[1][0] * base.matrix[0][2]) + (matrix[1][1] * base.matrix[1][2]) + (matrix[1][2] * base.matrix[2][2]) + (matrix[1][3] * base.matrix[3][2]);
        temp.matrix[1][3] = (matrix[1][0] * base.matrix[0][3]) + (matrix[1][1] * base.matrix[1][3]) + (matrix[1][2] * base.matrix[2][3]) + (matrix[1][3] * base.matrix[3][3]);

        temp.matrix[2][0] = (matrix[2][0] * base.matrix[0][0]) + (matrix[2][1] * base.matrix[1][0]) + (matrix[2][2] * base.matrix[2][0]) + (matrix[2][3] * base.matrix[3][0]);
        temp.matrix[2][1] = (matrix[2][0] * base.matrix[0][1]) + (matrix[2][1] * base.matrix[1][1]) + (matrix[2][2] * base.matrix[2][1]) + (matrix[2][3] * base.matrix[3][1]);
        temp.matrix[2][2] = (matrix[2][0] * base.matrix[0][2]) + (matrix[2][1] * base.matrix[1][2]) + (matrix[2][2] * base.matrix[2][2]) + (matrix[2][3] * base.matrix[3][2]);
        temp.matrix[2][3] = (matrix[2][0] * base.matrix[0][3]) + (matrix[2][1] * base.matrix[1][3]) + (matrix[2][2] * base.matrix[2][3]) + (matrix[2][3] * base.matrix[3][3]);

        temp.matrix[3][0] = (matrix[3][0] * base.matrix[0][0]) + (matrix[3][1] * base.matrix[1][0]) + (matrix[3][2] * base.matrix[2][0]) + (matrix[3][3] * base.matrix[3][0]);
        temp.matrix[3][1] = (matrix[3][0] * base.matrix[0][1]) + (matrix[3][1] * base.matrix[1][1]) + (matrix[3][2] * base.matrix[2][1]) + (matrix[3][3] * base.matrix[3][1]);
        temp.matrix[3][2] = (matrix[3][0] * base.matrix[0][2]) + (matrix[3][1] * base.matrix[1][2]) + (matrix[3][2] * base.matrix[2][2]) + (matrix[3][3] * base.matrix[3][2]);
        temp.matrix[3][3] = (matrix[3][0] * base.matrix[0][3]) + (matrix[3][1] * base.matrix[1][3]) + (matrix[3][2] * base.matrix[2][3]) + (matrix[3][3] * base.matrix[3][3]);

        for (int line = 0; line < 4; line++)
        {
            for (int row = 0; row < 4; row++)
            {
                matrix[line][row] = temp.matrix[line][row];
            }
        }
    }
};
```

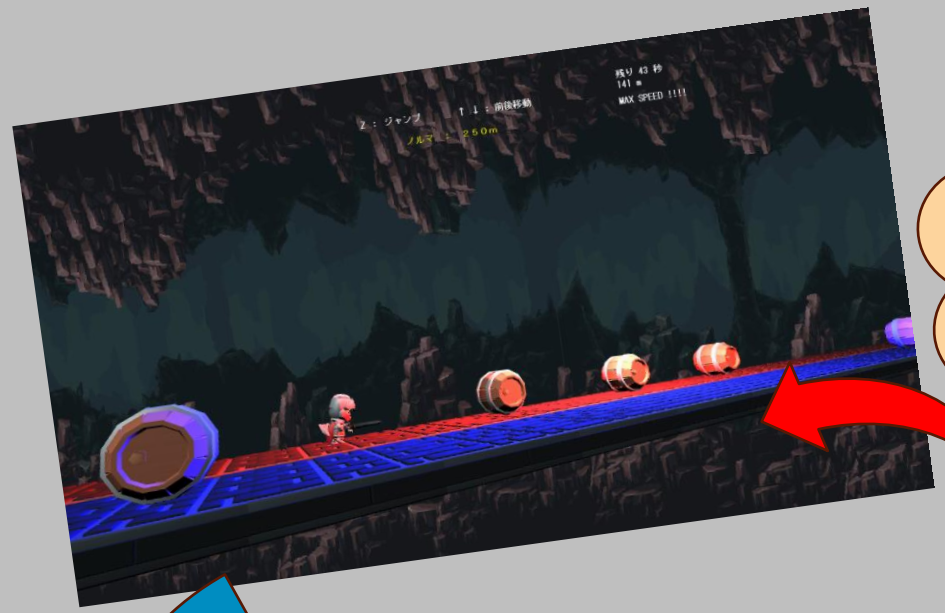
2024年5月~ 6月

仕分けゲーム

赤と緑のキャラクターを捕まえて
それぞれ対応した色の球状エリアに
投射していく簡易ゲーム。

このときに行列を用いたカメラ回転を実装。(後にQuaternionを使いだして、そちらに移行したけれども)

プレイヤーと仕分け対象との相互処理を特に意識して制作。



９０秒の間に
前後移動とジャンプを使い分け
より長い距離を走る

2024年4月

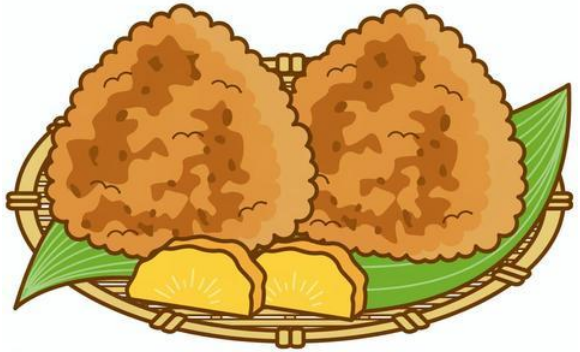
ジャンプ＆ランゲーム

初の３Dモデルを使用した
簡易的な時間制ランゲーム。

加速減速、ジャンプの挙動といった
プレイヤー周り動作処理を特に
意識しつつ、アニメーション管理の基
軸もここで作成。

前後の位置関係が分かりやすくなるよ
うにとライトをいじっている内に、
デュフューズ、スペキュラー、アンビ
エントなどのライティング知識をここ
でそれとなく学ぶ。

今後の目標



- これまでは実装を考える際にウェブサイトでの情報収集が主だったが、書籍などを通してより多く深くプログラミングパターンや技術を取り入れていきたい。

- 現状自分の作品に非同期処理が実装されておらず、毎プレイ時にロードで時間がかかることもあるため、非同期処理についても学び、プレイする側がよりストレスフリーに楽しめるような作品を目指して制作していく。

- キャラクター周りの動作処理だけではなく、シェーダーやレンダリングについてもさらに勉強を行い、柔軟な対応ができるように知識の幅を広げておく。