ゲーム概要

タイトル：ルーレットリガー
ジャンル：2Dコマンドバトル

プラットフォーム：PC
制作人数：1人
制作期間：3か月 （2023年5月～2023年7月）
開発環境：DXライブラリ
使用言語：C++、HLSL

ゲームルール

ルーレットを回すことで「ユニット」のコマンドを決定し、敵とバトルする2Dコマンドバトルです。

コマンドは、攻撃やバフ、ヒールやミスなど多彩であり、ルーレットによって勝敗が左右されます。

バトルに勝利すると、敵ユニットをデッキに加えることが出来、強いユニットを集めることが出来ます。

◀バトルシーン

▶リザルトシーン

アピールポイント①　『HLSLシェーダー』

◇ステータスUPシェーダー

◇シェーダー例：トランジション

```
float4 main(PS_INPUT PSInput) : SV_TARGET
{
    //UV座標を受け取る
    float2 uv = PSInput.TexCoords0;

    //UV座標とテクスチャを参照して、最適な色を取得する
    float4 srcCol =
        g_SrcTexture.Sample(g_SrcSampler, uv);
    float4 fadeCol =
        g_FadeTexture.Sample(g_SrcSampler, uv);

    //アルファ値
    float alpha = 1.0f - (fadeCol.r + fadeCol.g + fadeCol.b) / 3.0f;
    alpha -= 1.0;                   //[0.0f]～[-1.0f]の状態にする
    alpha += (g_pra * 2.0f);        //[0.0f]～[2.0f]の時間割合を加算する

    return float4(srcCol.rgb, srcCol.a * alpha);
}
```

今回の作品でシェーダーに挑戦しようと思い、いろんな表現をHLSLシェーダーを使って作成しました。

- トランジション
- ステータスUP、毒や麻痺の状態異常
- マスク処理
- UI　　etc…

シェーダー再生動画

URL：https://youtu.be/LcfMJX-ZY3M

アピールポイント①　『HLSLシェーダー』

◇Draw関数の呼び出し

```
    //描画
    ds.DrawExtendGraphToShader(
        pos, { DRAWING_SIZE ,DRAWING_SIZE }, unitImg_, nowPs_, buf);
```

◇Draw関数の中身

```
void DrawShader::DrawGraphToShader(
    const Vector2& pos, const int& handle, const PS_TYPE& type)
{
    //シェーダーの設定
    int ps = SearchPS(type);
    SetUsePixelShader(ps);
    //シェーダーにテクスチャを転送
    SetUseTextureToShader(0, handle);

    //サイズ
    int x, y;
    GetGraphSize(handle, &x, &y);

    //描画座標
    MakeSquereVertex(pos, { x,y });

    //描画
    SetDrawBlendMode(DX_BLENDMODE_ALPHA, 0);
    DrawPolygonIndexed2DToShader(vertex_, NUM_VERTEX, index_, NUM_POLYGON);
    SetDrawBlendMode(DX_BLENDMODE_NOBLEND, 0);
}
```

シェーダーでの描画を簡易的に行えるように、シェーダー描画クラスを設計し、Draw関数を呼び出すだけで、シェーダーを使った描画を行えるようにしました。

シェーダー描画の呼び出しは、シェーダーの種類と定数バッファを引数として渡して関数を呼び出すことで使用できます。通常描画やサイズ指定描画など使い分けできるように柔軟性を考えて作成しました。

アピールポイント②　『ユニットのデータベース管理』

◇ユニットデータ

```
//ユニットのデータ
struct UnitData
{
    //int num;                  //ユニット番号
    std::string name;           //名前
    int imgHandle;              //画像ハンドル
    int hp;                     //体力
    int attack;                 //攻撃力
    int speed;                  //素早さ
    std::vector<int> cmdNum;    //コマンド番号
};
```

◇ユニットデータXML

全てのユニットに番号を振り、データベース化することでユニットデータを管理しやすくしました。

XMLからユニットデータを読み込み、そのデータをマップ配列に

＜キー番号、データ＞

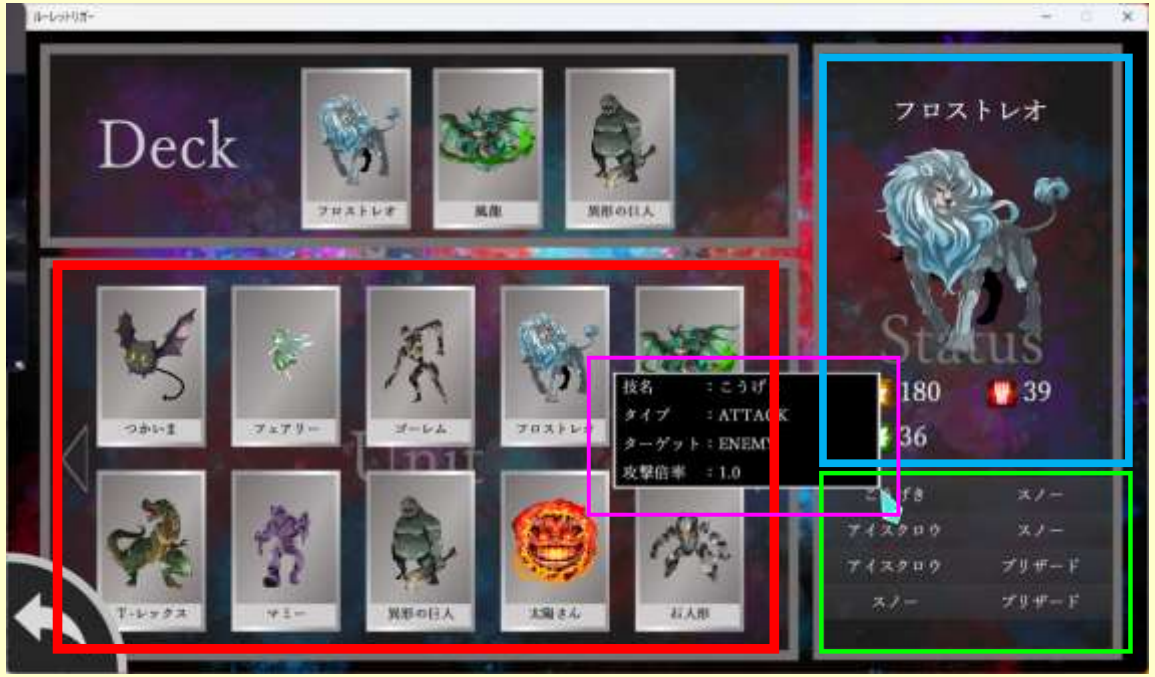
という形でデータを管理しています。

◇データ管理配列

```
// ユニットデータ（番号をキー値とする）
std::unordered_map<int,UnitData> unitDataMap_;
```

アピールポイント②　『ユニットのデータベース管理』

▼デッキ編集画面



ユニットの種類も100を超えるほど多く、データは個別で頻繁に使います。そのためデータの取り扱いの簡略化は必要不可欠でした

ユニットのデータを多く使用しているデッキ編集画面でも、これにより比較的楽に管理することが出来ています。

デッキ編集画面一覧

①ユニットプール
②ユニットステータス
③コマンド（ユニットの技）
④コマンド詳細

アピールポイント③　『ユニット死亡時の演出』

▼ユニット死亡時画面

死亡時の演出を特にこだわりました。

画面揺れ、マスク処理、色相反転を入れること

で、目を引くような演出作りを行いました。

```
//背景描画
ds.DrawGraphToShader({ 0.0 }, backImg_, PsType::REVERSE_TEXTURE, COLOR_F{});
//ユニット描画
ds.DrawExtendGraphToShader(
    UNIT_IMG_SHOW_POS, UNIT_IMG_SHOW_SIZE,
    unitImg_, PsType::DEATH_UNIT, COLOR_F{});

//描画先指定
SetDrawScreen(deathStagingScreen_);
ClearDrawScreen();

//マスク描画
ds.DrawGraphAndSubToShader(
    { 0.0 }, { mSizeX_, Application::SCREEN_SIZE_Y },
    maskScreen_, maskImg_, PsType::MASK, COLOR_F{});
//フレーム描画
ds.DrawExtendGraphToShader(
    { 0.0 }, { mSizeX_, Application::SCREEN_SIZE_Y },
    frameImg_, PsType::REVERSE_TEXTURE, COLOR_F{});

//描画先指定
SetDrawScreen(DX_SCREEN_BACK);
```

スクリーン別に描画を行っている▶