PORTFOLIO

ASOポップカルチャー専門学校

ゲーム・CG・アニメ専攻科　ゲーム専攻

プログラマー

KAKITA　RYOSUKE

柿田凌介

HORRIFIC HOUSE

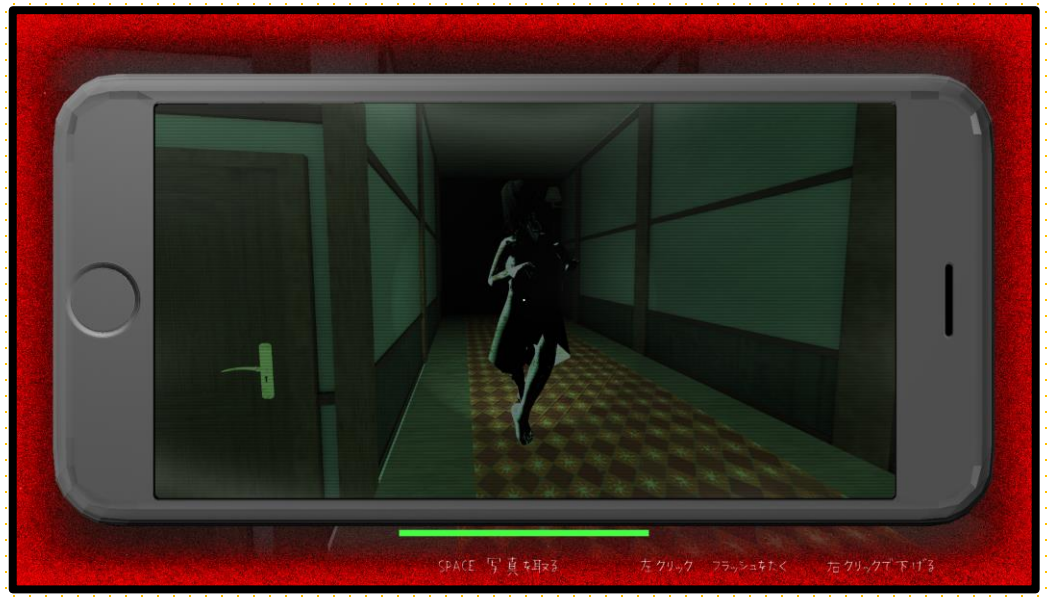
学内のコンテストに向けて制作し、学内コンテストでは1位を獲得することができたゲームです。
DXLibとUnityの両方を使用しました。アンドロイド端末をコントローラとして使用することができ、その場合はARも楽しむことができます。
Unityはアンドロイド端末用に使用しました。

脱出することを目的としたホラーゲームで、脱出に必要となるアイテムを集めて館から脱出するゲームです。カメラを通すと見えるオブジェクトを見つけたり謎解きをしたりして、ゲームを進めます。

・シェーダを使用してブルームや被写界深度を実装してゲームのリアリティを上げました。
・敵の経路探索としてダイクストラ法を使用し、効率的に最短経路を計算しています。

動画

URL:https://youtu.be/N_SyTb9fNVQ



対応機種：PC
使用言語：C++,C#(スクリプト)
ライブラリ：DXLib,Unity
制作期間：2022年9月~2023年2月
(約5ヶ月)
制作人数：5人
ジャンル：脱出ホラーゲーム

担当箇所

○ResultScene
ゲームクリア時の描画を作成しています。

○InventoryScene
背景のぼかしを実装しています。

○Player
移動やダッシュ等の基本的な動作を実装しています。

○Stamina
ダッシュの処理を実装しています。

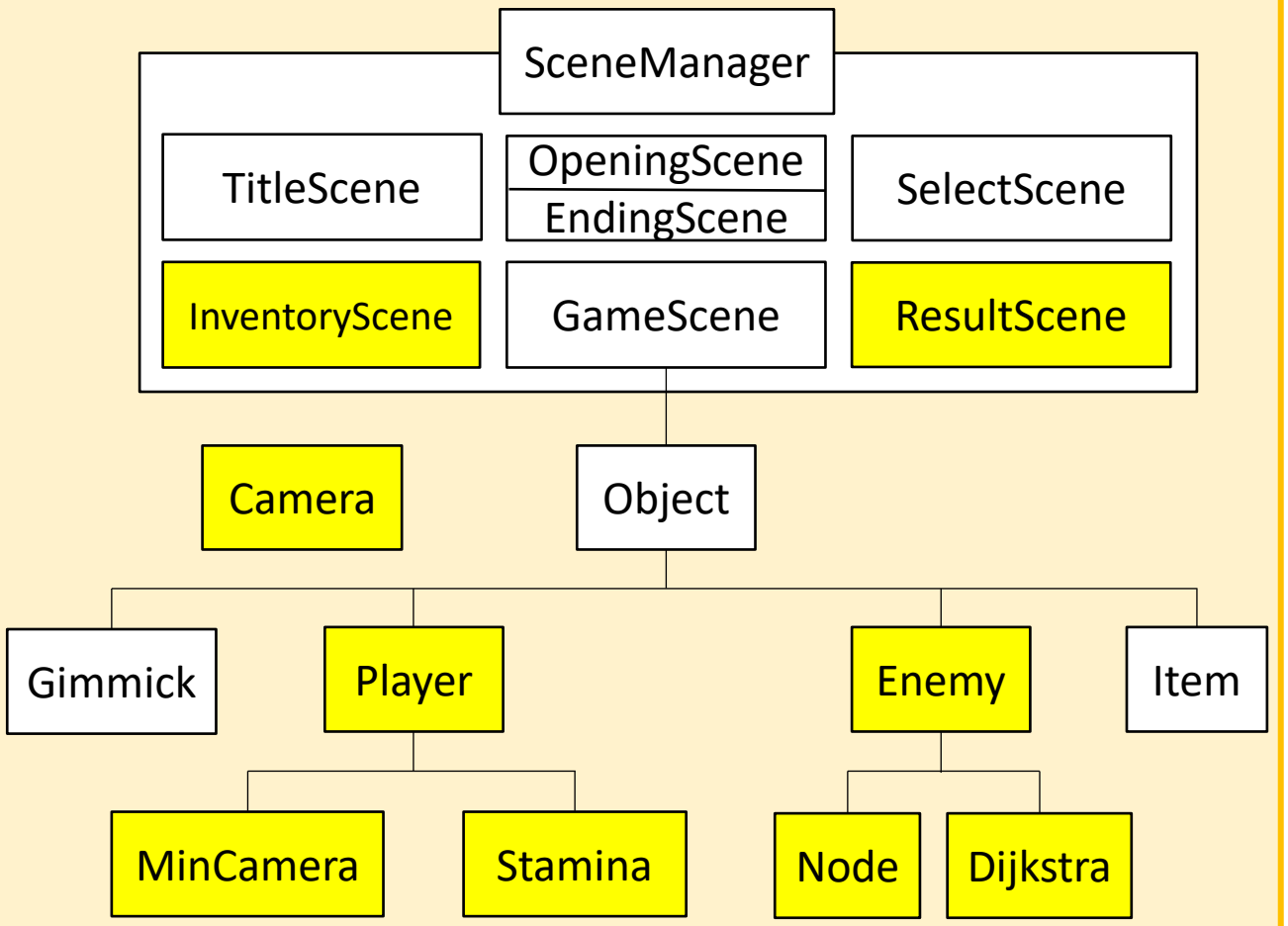
○Camera
カメラ回転やダッシュ時の画面揺れを実装しています。

○Enemy
移動や経路探索等、敵の処理を実装しています。

○MinCamera
カメラを構えたときの描画や処理を実装しています。

HLSL
○ポストエフェクト
・敵に追跡されているときのノイズ
・ゲーム内カメラの描画
・ガウシアンブラー

○ブルーム
○被写界深度



ポストエフェクト

ガウシアンブラー

インベントリの背景は、ゲーム画面をぼかしたものにしています。
中心からの重みを計算し、定数として渡してサンプリングをしています。
ダウンサンプリングを行ってブラーの強度を上げ、高速化にも繋げています。

重み計算

```
void InventoryScene::CalcBlurWeights(void)
{
    //重みの合計
    float totalWeight = 0.0f;

    //xは基準テクセルからの距離
    for (int x = 0; x < WEIGHTS_NUM; x++)
    {
        weights_[x] = expf(-0.5f * static_cast<float>(x * x) / BLUR_SIGMA);
        totalWeight += 2.0f * weights_[x];
    }

    //除算し、合計を1にする
    for (int i = 0; i < WEIGHTS_NUM; i++)
    {
        weights_[i] /= totalWeight;
    }
}
```

それぞれの画面に描画

```
//メインレンダリングターゲットに描画
SetDrawScreen(mainRenderTarget_);
ClsDrawScreen();

//背景
DrawPrimitive2D(totalBlurVerts_.data(), totalBlurVerts_.size(), DX_PRIMTYPE_TRIANGLESTRIP, backGroundHandle_, false);

//横ブラーをかける
SetDrawScreen(blurX_.renderTarget);
ClsDrawScreen();
blurX_.origShader->DrawPrimitive3D({ 0.0f,0.0f,0.0f }, blurX_.imageSize, mainRenderTarget_);

//縦ブラーをかける
SetDrawScreen(blurY_.renderTarget);
ClsDrawScreen();
blurY_.origShader->DrawPrimitive3D({ 0.0f,0.0f,0.0f }, blurY_.imageSize, blurX_.renderTarget);

//最終描画
SetDrawScreen(screenID_);
ClsDrawScreen();

//縦ブラーのものを描画
DrawPrimitive2D(totalBlurVerts_.data(), totalBlurVerts_.size(), DX_PRIMTYPE_TRIANGLESTRIP, blurY_.renderTarget, false);
```

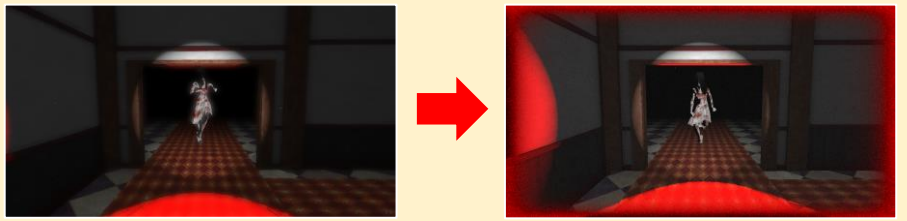
サンプリング

```
color  = weight1x * tex.Sample(sam, input.uv0.xy);
color += weight1y * tex.Sample(sam, input.uv1.xy);
color += weight1z * tex.Sample(sam, input.uv2.xy);
color += weight1w * tex.Sample(sam, input.uv3.xy);
color += weight2x * tex.Sample(sam, input.uv4.xy);
color += weight2y * tex.Sample(sam, input.uv5.xy);
color += weight2z * tex.Sample(sam, input.uv6.xy);
color += weight2w * tex.Sample(sam, input.uv7.xy);

color += weight1x * tex.Sample(sam, input.uv0.zw);
color += weight1y * tex.Sample(sam, input.uv1.zw);
color += weight1z * tex.Sample(sam, input.uv2.zw);
color += weight1w * tex.Sample(sam, input.uv3.zw);
color += weight2x * tex.Sample(sam, input.uv4.zw);
color += weight2y * tex.Sample(sam, input.uv5.zw);
color += weight2z * tex.Sample(sam, input.uv6.zw);
color += weight2w * tex.Sample(sam, input.uv7.zw);
```

なし → 横ブラー → 縦ブラー → 最終描画

ゲーム内カメラの描画

追跡されている際のノイズ



被写界深度

被写界深度を実装しました。

通常のテクスチャとぼかしたテクスチャを用意し、補間することで被写界深度を表現しました。深度によって補間の度合いを変えています。

深度テクスチャ用の描画と通常の描画があり、同じものを二度描画することになるためマルチレンダーターゲットを使用しました。
これにより、一度の描画で二つのレンダーターゲットに描画をすることができました。

フォーカスが合っていない箇所はぼかしがかかっている

適用前

適用後

マルチレンダーターゲットで描画

```
struct PS_OUTPUT
{
    float4 color : SV_TARGET0;  //通常
    float4 depth : SV_TARGET1;  //深度
};
```

```
//通常描画と深度描画をするためマルチレンダーターゲット
SetRenderTargetToShader(0, screenRenderTarget_);
SetRenderTargetToShader(1, depthRenderTarget_);
ClsDrawScreen();

//カメラ処理(オブジェクト描画前に処理)
camera_->SetBeforeDraw(player_->GetPosition());
//ステージ
stage_->Draw(player_->GetminCameraFlag());
//敵
enemy_->Draw();

//描画先リセット
SetRenderTargetToShader(1, -1);
```

フォーカスの開始位置と終了位置から
ぼかし率を計算

```
//フォーカス位置からぼかし率を計算
if (depth < focusStartPos)
{
    fade = 1.0f - depth / focusStartPos;
}
else if (depth < focusEndPos)
{
    fade = 0.0f;
}
else
{
    fade = (depth - focusEndPos) / (1.0f - focusEndPos);
}
```

ぼかし率が高い場合は強めのブラー
がかかったテクスチャと合成

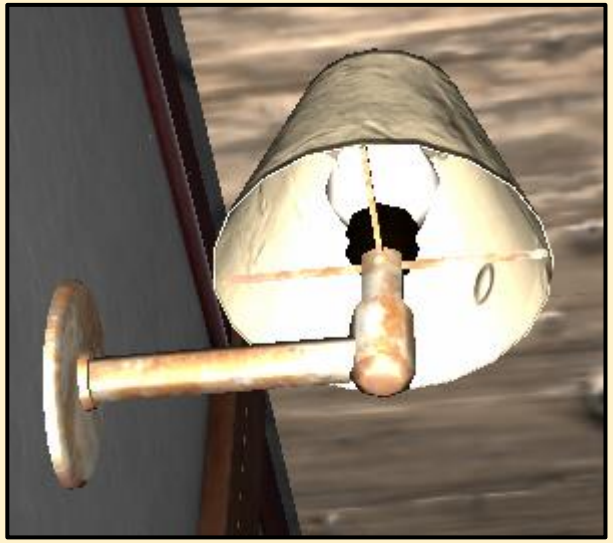
```
//ぼかし率から計算
if (fade < 0.5f)
{
    //低い場合は弱めのぼかした画像と合成
    color1 = screenTex.Sample(sam, input.uv);
    color2 = blurTex.Sample(sam, input.uv);
    blendRate = fade * 2.0f;
}
else
{
    //高い場合は強めのぼかした画像と合成
    color1 = blurTex.Sample(sam, input.uv);
    color2 = highBLurTex.Sample(sam, input.uv);
    blendRate = (fade - 0.5f) * 2.0f;
}
finalColor = lerp(color1, color2, blendRate);
```

ブルーム

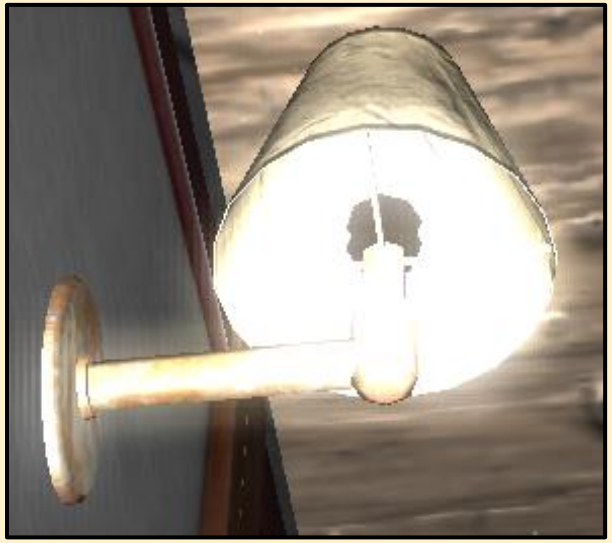
ブルームを実装しました。

レンダーターゲットにレンダリングし、そのテクスチャの輝度の高い部分を抽出してガウシアンブラーをかけています。
それを加算合成をすることで実装しました。

適用前



適用後



レンダーターゲットに描画

↓

輝度抽出

↓

輝度抽出したテクスチャにガウシアンブラーをかける

↓

加算合成

輝度抽出

```
PS_OUTPUT output;
float4 color = tex.Sample(sam, input.uv);

//輝度計算
float luminance = dot(color.xyz, float3(0.2125f, 0.7154f, 0.0721f));

//輝度が低いものは描画しない
clip(luminance - 0.95f);

output.color = color;

return output;
```

加算合成

```
//スクリーン描画
DrawGraph(0, 0, screenRenderTarget_, true);

//輝度を加算
SetDrawBlendMode(DX_BLENDMODE_ADD, 255);
DrawGraph(0, 0, bloomRenderTarget_, false);
SetDrawBlendMode(DX_BLENDMODE_NOBLEND, 255);
```

敵の処理

ダイクストラ法

敵の移動はダイクストラ法を使って描画しました。
構造体のノードを配置し、対応したノード同士を繋げています。
目的のノードを決め、現在のノードから最短の経路を計算して移動をさせました。

プレイヤーが視界に入った場合はノードを追加してプレイヤーを追跡するようにしました。

プレイヤーがキーアイテムを手に入れた際は、近くのノードを敵の目的地として設定し、簡単になりすぎないように工夫しました。

配置データや接続データはファイルに書き、XML形式で読み込んでいます。

経路探索処理

```
//ゴールから計算
goalNode_->cost = 0;
nowLevel->emplace_back(goalNode_);

float nodeCost = 0.0f;

while (nowLevel->size())
{
    for (const auto& level : *nowLevel)
    {
        for (auto& c : level->connectNode)
        {
            nodeCost = level->cost + c.cost;

            if (c.node->cost == -1 || nodeCost < c.node->cost)
            {
                //未探索ノードあるいは最短ルートを更新できる場合
                //経路コストとゴールへ向かうためのノードをセット
                c.node->cost = nodeCost;
                c.node->toGoal = level;
            }
            else
            {
                continue;
            }
            nextLevel->push_back(c.node);
        }
    }

    //リストを入れ替えて次の階層を探索する
    swapLevel = nowLevel;
    nowLevel = nextLevel;
    nextLevel = swapLevel;
    nextLevel->clear();
}
```

構造体

```
struct Node
{
    Node(Vector3 initPos);
    ~Node();

    //ノードのリセット
    void Reset(void);

    //特定ノードへの接続を削除
    void RemoveConnect(SharedNode node);

    //位置
    Vector3 pos;

    //接続ノード
    std::list<NodeConnection> connectNode;

    float cost = -1;        //探索コスト
    SharedNode toGoal;      //ゴールへの最短ルートに繋がるノード
};
```

ノードの配置と接続後

ノードの位置データ

```
<NodePos name="NodePos" version="1.0">
    <Pos1  name="Pos1"  posX="25402"  posY="0" posZ="-1185"  index="0" />
    <Pos2  name="Pos2"  posX="25402"  posY="0" posZ="-13675" index="1" />
    <Pos3  name="Pos3"  posX="-7471"  posY="0" posZ="-13675" index="2" />
    <Pos4  name="Pos4"  posX="-7471"  posY="0" posZ="-4549"  index="3" />
    <Pos5  name="Pos5"  posX="1367"   posY="0" posZ="-1185"  index="4" />
    <Pos6  name="Pos6"  posX="1367"   posY="0" posZ="-4457"  index="5" />
    <Pos7  name="Pos7"  posX="-12076" posY="0" posZ="-4549"  index="6" />
    <Pos8  name="Pos8"  posX="7200"  posY="0" posZ="-20500"  index="7" />
    <Pos9  name="Pos9"  posX="-7471"  posY="0" posZ="-25430"  index="8" />
    <Pos10 name="Pos10" posX="-13507" posY="0" posZ="-25430"  index="9" />
```

ノードの接続データ

```
<ConnectNode name="ConnectNode" version="1.0">
    <Connect1 name="Connect1" index1="0" index2="1"/>
    <Connect2 name="Connect2" index1="1" index2="13"/>
    <Connect3 name="Connect3" index1="2" index2="3"/>
    <Connect4 name="Connect4" index1="0" index2="4"/>
    <Connect5 name="Connect5" index1="3" index2="5"/>
    <Connect6 name="Connect6" index1="4" index2="5"/>
    <Connect7 name="Connect7" index1="3" index2="6"/>
    <Connect8 name="Connect8" index1="18" index2="7"/>
    <Connect9 name="Connect9" index1="2" index2="8"/>
    <Connect10 name="Connect10" index1="8" index2="9"/>
```

NOT SCRAP

学内のコンテストやゲームクリエイター甲子園に向けて制作したゲームです。学内のコンテストでは数十作品の中から上位に選ばれて決勝に進出することができました。

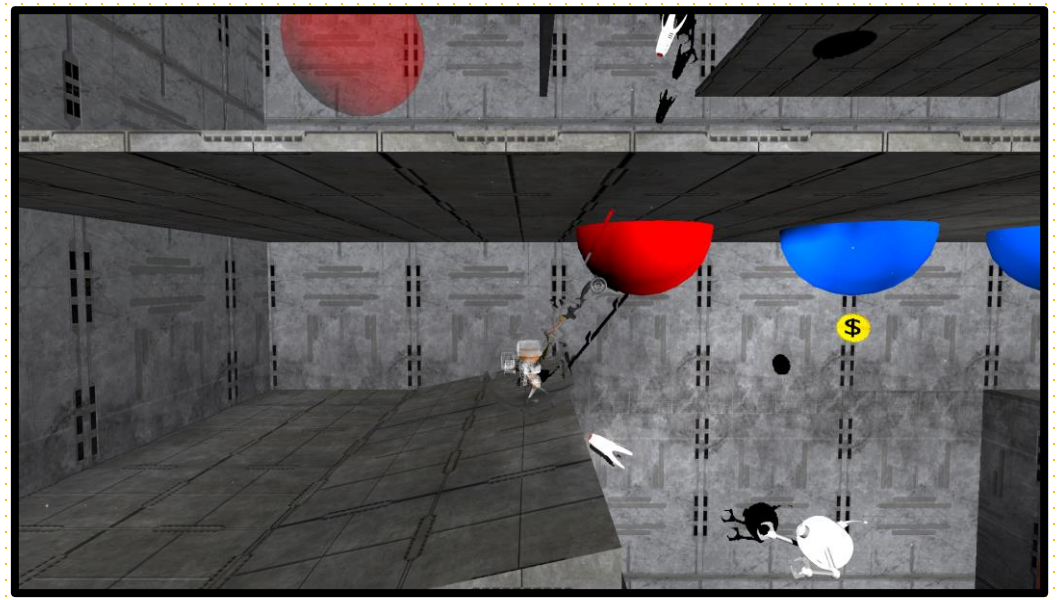
磁石をテーマに制作したゲームで、N極とS極を使い分けてゴールを目指すゲームです。強い磁石と弱い磁石があり、素早い移動やゆっくりとした移動ができます。

・初の3Dを使用したゲーム制作で3Dの仕様に苦労しながらも全員でしっかりと話し合い、協力して制作を進めました。

・シェーダを使用してポストエフェクトを行いました。

動画

URL:https://youtu.be/RZ5a4qS3xY4



対応機種：PC
使用言語：C++
ライブラリ：DXLib
制作期間：2022年4月~8月(約5ヶ月)
制作人数：4人
ジャンル：横スクロールアクション

担当箇所

○MagStickクラス
プレイヤーの磁石の処理を実装しています。

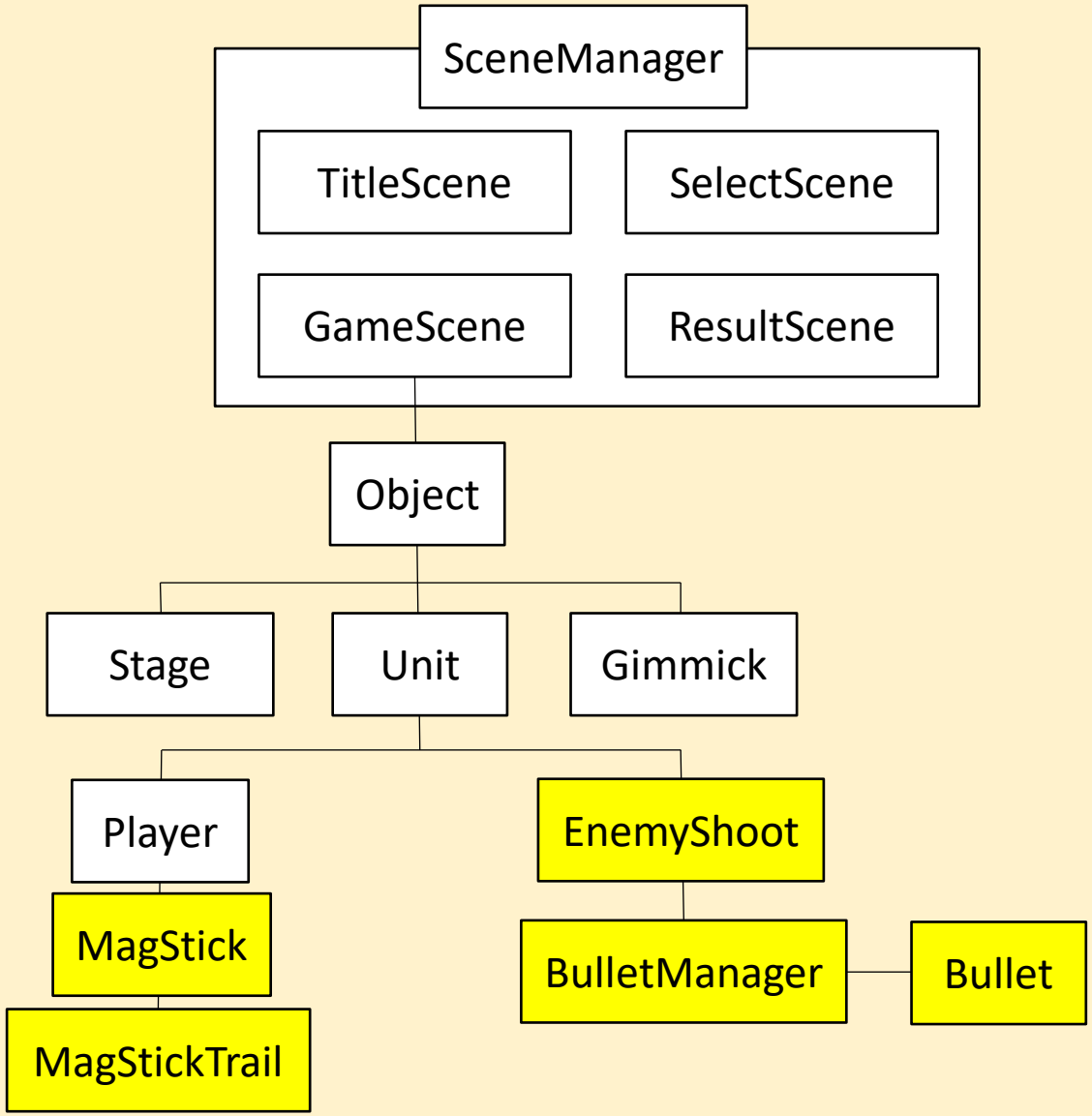
○MagStickTrailクラス
プレイヤーの磁石の回転時の軌跡を描画しています。

○EnemyShootクラス
弾を発射する敵の処理を実装しています。

○Bullet/BulletManagerクラス
敵が発射する弾の処理を実装しています。加えて弾の管理をするクラスも実装しています。

HLSL

○ポストエフェクト
・画面割れ、歪みを行っています。



敵の弾の処理

弾の管理

弾の移動や当たり判定を行うクラスと
そのクラスを管理するクラスに分けて
処理を行いました。
これにより、役割がはっきりしたので
コードの可読性も上がりました。

Bullet

↓

移動
当たり判定等

Bullet Manager

↓

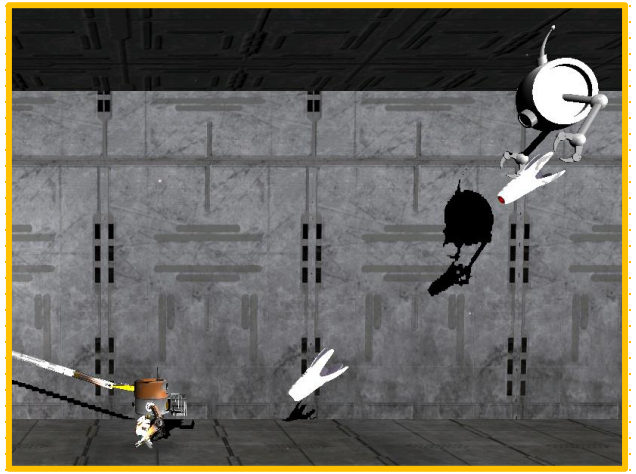
生成
範囲for文でまとめて処理

vectorで弾数を管理しました。
爆発した弾は破棄をせずに次以降の
ループで使い回すようにして、要素数
を増やしすぎないように意識しました。

要素数を増やしすぎないように使い回し

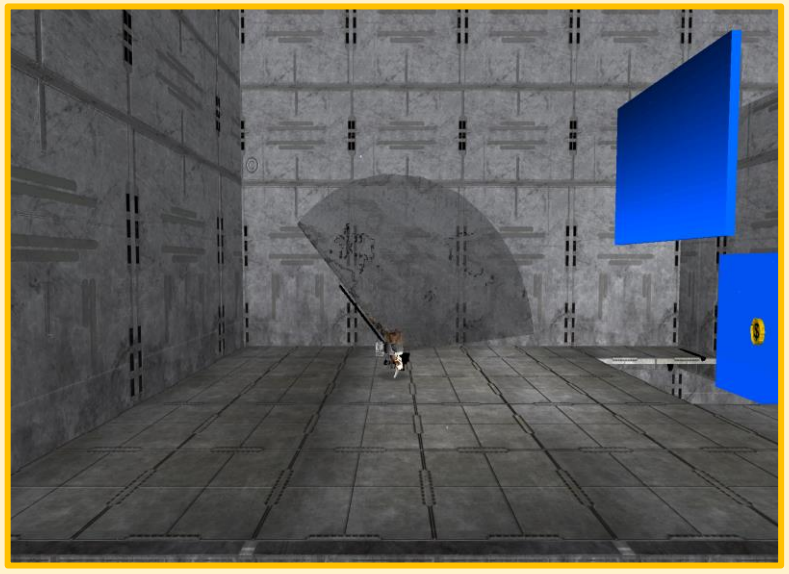
```
bool createFlag = false;
//要素のうち、生存していないものは使いまわす
for (auto& bullet : bullets_)
{
    if (!bullet->GetAliveFlag())
    {
        bullet->Init(direction, enemypos);
        bullet->SetAliveFlag(true);
        createFlag = true;
        break;
    }
}

//全て生存している場合は追加する
//上限を超えている場合は追加しない
if (!createFlag)
{
    //敵が何番目かで比較する値が変わる
    if (bulletNum_ < ENEMY_BULLET_MAX_NUM * (enemyNum_ + 1))
    {
        bullets_.emplace_back(std::make_unique<Bullet>(direction, enemypos, bulletNum_));
        bulletNum_++;
    }
}
```

発射している様子



磁石の軌跡

軌跡



頂点の位置管理

```
//プレイヤーの位置からマウス位置へのベクトルを格納
posList_.push_front(vecN);

//頂点数を超えている場合、古い頂点は削除
if (posList_.size() >= INDEX_MAX)
{
    posList_.pop_back();
}

//頂点情報更新
SetVertex(posList_);
```

頂点情報更新

```
auto itr = ePos.begin();

for (int i = 0; i < ePos.size(); i++)
{
    //根本
    trail_[i * 2].dif = diffuse;
    trail_[i * 2].spc = GetColorU8(255, 255, 255, 255);
    trail_[i * 2].pos.x = CENTER_POS_X;
    trail_[i * 2].pos.y = CENTER_POS_Y;
    trail_[i * 2].pos.z = 0.0f;
    trail_[i * 2].u = trail_[i * 2].pos.x / screenSize.x;
    trail_[i * 2].v = trail_[i * 2].pos.y / screenSize.y;
    trail_[i * 2].su = 0.0f;
    trail_[i * 2].sv = 0.0f;
    trail_[i * 2].rhw = 1.0f;

    //剣先
    trail_[i * 2 + 1].dif = diffuse;
    trail_[i * 2 + 1].spc = GetColorU8(255, 255, 255, 255);
    trail_[i * 2 + 1].pos.x = CENTER_POS_X + itr->x * ROD_LENGTH;
    trail_[i * 2 + 1].pos.y = CENTER_POS_Y + itr->y * ROD_LENGTH;
    trail_[i * 2 + 1].pos.z = 0.0f;
    trail_[i * 2 + 1].u = trail_[i * 2 + 1].pos.x / screenSize.x;
    trail_[i * 2 + 1].v = trail_[i * 2 + 1].pos.y / screenSize.y;
    trail_[i * 2 + 1].su = 0.0f;
    trail_[i * 2 + 1].sv = 0.0f;
    trail_[i * 2 + 1].rhw = 1.0f;
    itr++;

    diffuse.a -= minusAlpha;
}
```

磁石を回転させた際に軌跡を出すようにしています。
位置情報は挿入と削除を毎フレーム行うためlistを使用し、頂点情報はvectorを使用して管理をしています。

頂点位置をpush_frontで追加し、頂点は徐々に透過させ、pop_backで古い頂点から削除するようにしています。

位置情報を追加、削除した後にそれぞれの頂点情報を更新しています。

ポストエフェクト

画面割れ

プレイヤーがやられた際は画面を割りの演出を入れています。

画面割れの法線マップを読み込み、uv座標に加算をすることで実装しています。

HLSL内

```
PS_OUTPUT output;
float2 normal = nor.Sample(sam, input.uv).xy;
normal = normal * 2 - 1;//-1~1

float2 crack = normal.xy * 0.3f;

output.color = tex.Sample(sam, input.uv + crack);

return output;
```

歪み

磁石の軌跡に歪みを描画しました。

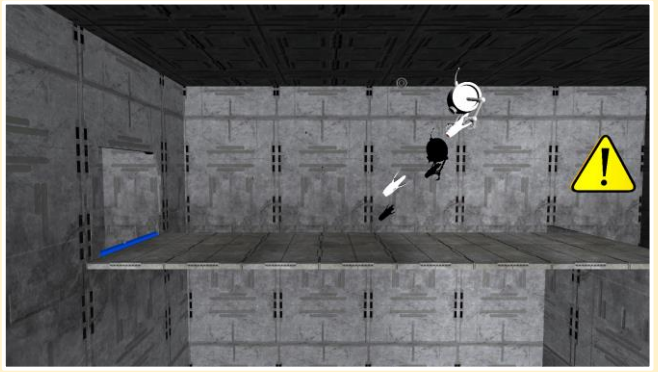
ノイズを読み込み、サンプリング時にuvを加算して歪ませています。

HLSL内

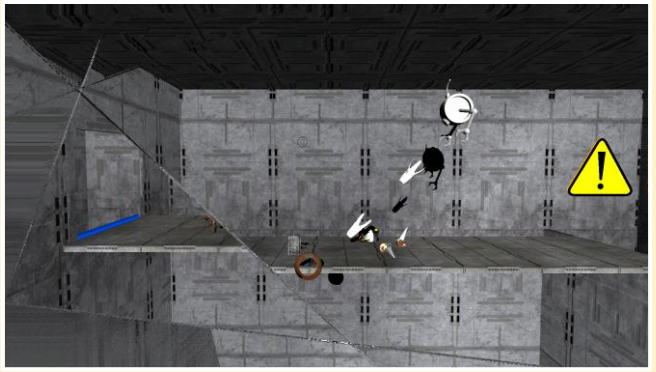
```
//uv値を回転
float2 uv = float2(
    input.uv.x + cos(angle) * 0.05f,
    input.uv.y + sin(angle) * 0.05f
    );

//ノイズをサンプリング
float4 p = noise.Sample(sam, uv);

p.xy = (p.xy * 2.0 - 1.0f) * 0.05f;

//背景をサンプリングして加算
output.color = screen.Sample(sam, input.uv + p.xy);

//徐々に透過
output.color.a = input.diffuse.a;
```

ポストエフェクトの有無の比較

なし



あり



なし

あり

Escape Universe Ship

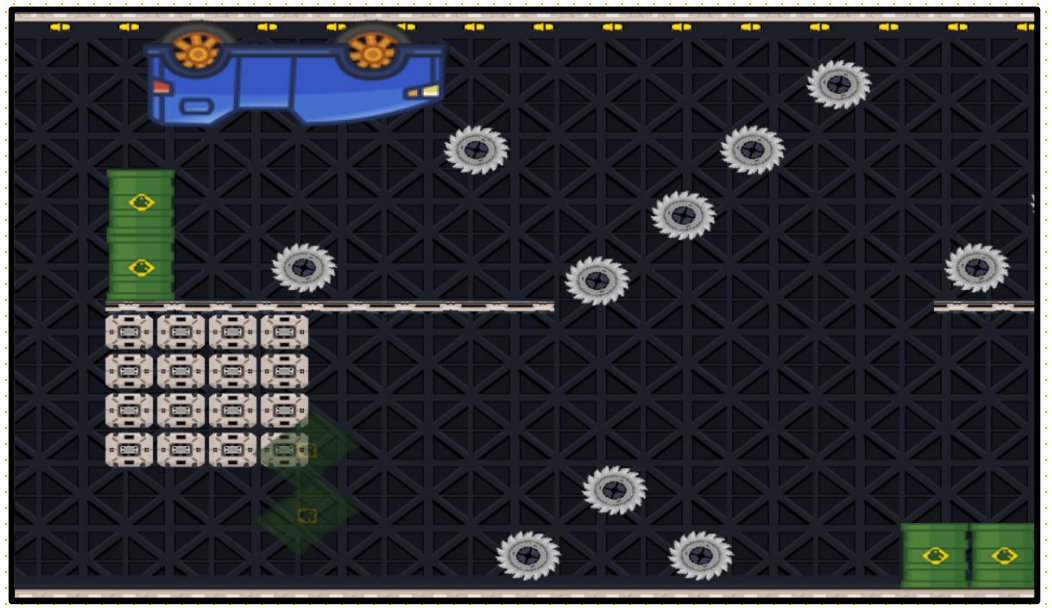
学内のコンテストに向けて制作したゲームです。個人で制作しました。

・宇宙船から脱出することを目的としたゲームです。自動で右に進むプレイヤーの大きさと重力を変更してゴールを目指します。ゲーム中は十字キーのみ使用するのでシンプルな操作でプレイをすることができます。

・ステージはツールで作成し、csv形式のデータで出力しています。そのデータをプログラム側で読み込んで描画しています。
・手触りの良さや爽快感にこだわって作成しました。

動画

URL:https://youtu.be/9ZVMKcIC9yg



対応機種：PC
使用言語：C++
ライブラリ：DXLib
制作期間：2021年9月~2022年2月
(約5ヶ月)
制作人数：1人
ジャンル：横スクロールアクション

ギミック

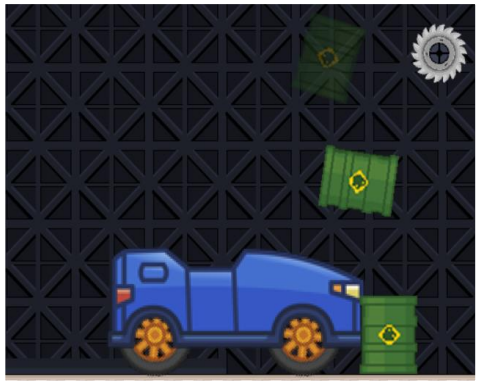
ドラム缶

ドラム缶を飛ばした際は、鉛直投げ上げを使用して上昇や落下を自然な動きにしました。
また、画像を点滅をさせてプレイには関係ないと思ってもらう工夫をしています。

処理

```
//座標更新
drum.pos.x += MOVE_POWER_X;
drum.pos.y += static_cast<float>((INIT_ACCEL) +
    ((1.0 / 2.0) * (drum.accel * (std::pow(delta, 2.0)))));

//回転量加算
drum.angle += ADD_ANGLE;

//加速量加算
drum.accel += 1.0;

//0.05秒ごとに点滅
if (drum.time >= BLINK_TIME)
{
    if (drum.alpha == ALPHA_FULL)
    {
        drum.alpha = ALPHA_HALF;
    }
    else
    {
        drum.alpha = ALPHA_FULL;
    }
```

描画



動画

当たり判定

ステージとの当たり判定はツールで作成し、プレイヤーと判定しています。
加えてめりこみ時の補正も同時に行っています。
ラムダ式にすることで、簡単に当たり判定をとることができました。

当たり判定関数

```
//線分と線分の当たり判定
bool CheckRay(Ray ray, Line line);

//線分と円の当たり判定
bool CheckCircle(Vector2 pos, Line line, float r, float& correct);

//線分と円の当たり判定
bool CheckSaw(Ray ray, Vector2 coll, float r);
```

```
//線分同士の衝突判定。車体と足場で使用
bool CheckCollisionRay(Ray ray,Collision coll,Vector2 offset);

//線分同士の左衝突判定
bool CheckCollisionRayL(Ray ray,Collision2 coll,Vector2 offset);

//円との上下衝突判定。タイヤと足場で使用
bool CheckCollisionCircleUpDown(Vector2 ray, Collision coll, Vector2 offset,float r,float& correct);

//円との上下左右衝突判定。車体とのこぎりで使用
bool CheckCollisionCircleAll(Ray ray, Vector2 coll, Vector2 offset,float r);

//円との上下衝突判定。タイヤと足場で使用
bool CheckCollisionCircleDownUp(Vector2 ray, Collision coll, Vector2 offset, float r, float& correct);

//円との上下衝突判定2。タイヤと足場で使用
bool CheckCollisionCircleUpDown(Vector2 ray, Collision2 coll, Vector2 offset, float r, float& correct);
```

ラムダ式

```
auto checkDrumCircleHitFlag = [&](Vector2 wheelpos) {
    Vector2 ray = wheelpos - offset;
    for (auto& coll : tmx.GetCollDrum())
    {
        if (raycast_.CheckCollisionCircleUpDown(ray, coll, offset, radius_, correct))
        {
            if (coll.second)
            {
                if (nowSize_ == PLAYER_SIZE::Big)
                {
                    coll.second = false;
                    return false;
                }
                return true;
            }
        }
    }
    return false;
};
```

Water

シェーダを使用して水を作成しました。

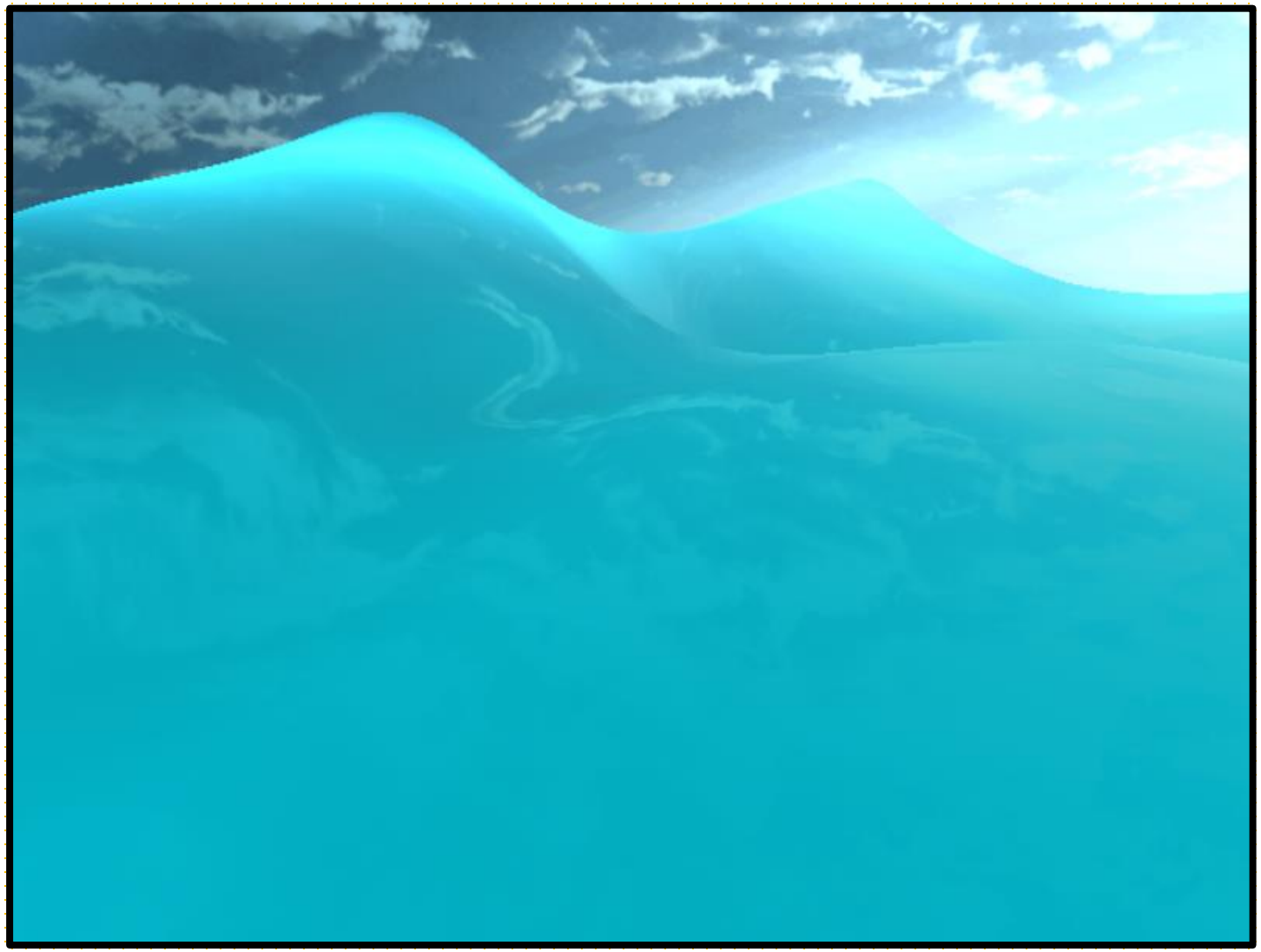
平面のモデルを用意し、そのモデルに対してシェーダを適用しています。

頂点シェーダでは波の動きを作り、
ピクセルシェーダでは水の見た目を作っています。

使 用 言 語 ：C++,HLSL
ラ イ ブ ラ リ ：Dxlib
制 作 期 間 ：約2週間

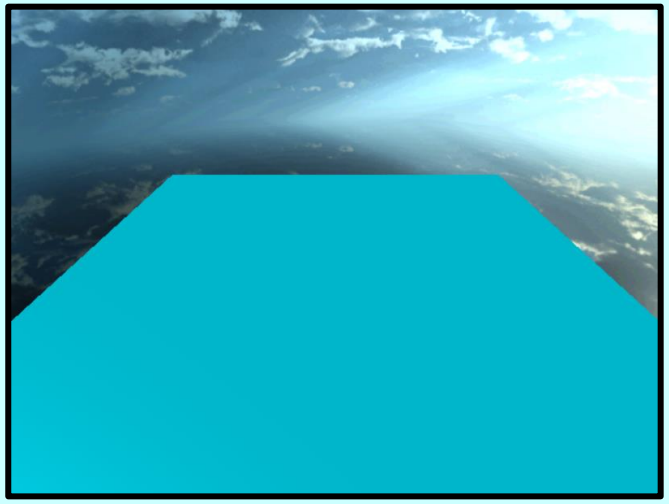
動画

URL:https://youtu.be/TzN72DeW4rg

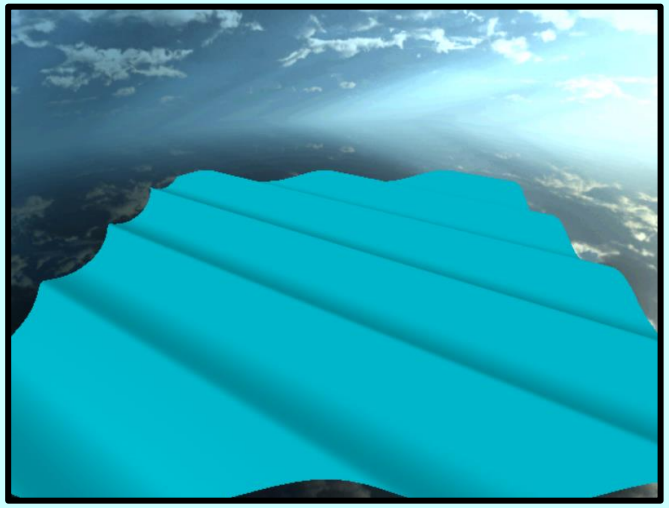


ゲルストナー波

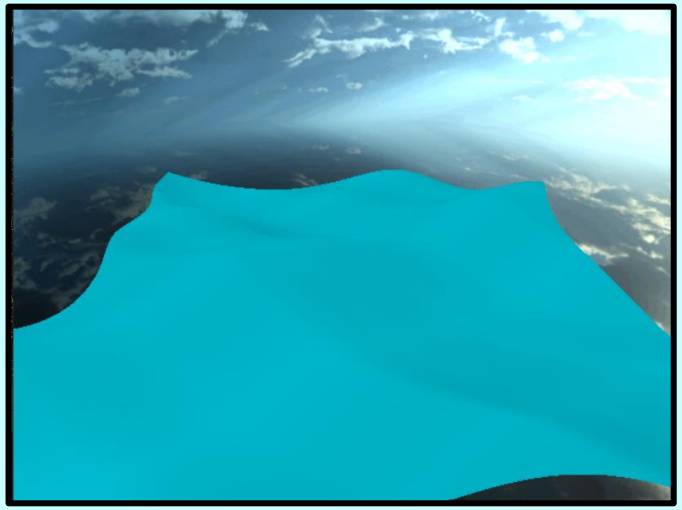
なし



1つ



4つ合成



波の動きとしてゲルストナー波を使用しました。いくつか作成し、合成することで波の動きを表現しました。
あわせて法線の計算も行い、ピクセルシェーダに渡しています。

計算部分は関数化しており、必要な引数を渡すと波が作られるようになっています。
引数は、主にローカル位置、経過時間、頂点同士の距離、高さ、方向を渡します。

関数内

```
//波の方向
float2 dir = float2(waveDir.x, waveDir.z);
dir = normalize(dir);
//振幅(波長に対して振幅は最大でも1/14)
float ampRate = waveLength / 14.0f;
//nπ/waveLength(値が大きいほど波も強くなる)
float PIperL = 4.0f * pi / waveLength;
//移動速度
float velocity = sqrt(gravity / PIperL) * t;
//頂点位置と波の方向の内積
float d = dot(dir, localPos.xz);
//シータ
float theta = PIperL * d + velocity;
float3 gPos = float3(0.0f, 0.0f, 0.0f);
//QとRはそれぞれ波の強さに関係
gPos.xz =  Q * ampRate * dir * cos(theta);
gPos.y =  R * ampRate * sin(theta);

//法線
float3 normal = float3(0.0f, 1.0f, 0.0f);
normal.xz  = (-dir * R * cos(theta)) /
    (7.0f / pi - Q * dir * dir * sin(theta));
normal = saturate(normal);

outPos.xyz += gPos.xyz;
norm += normalize(normal);
```

キューブマップ

HLSL側

textureCUBEとして定義

```
//動的キューブテクスチャ
textureCUBE dynamicCubeTex : register(t0);
```

反射ベクトルでサンプリング

```
//カメラに向かうベクトル
float3 viewVec = normalize(input.viewVec);

//キューブ用反射ベクトル
float3 cubeRef = reflect(-viewVec, input.normal);

//キューブカラー
float4 cubeColor = dynamicCubeTex.Sample(sam, cubeRef);
```

C++側

レンダーターゲットに描画

```
//環境を描画する面の数
for (int i = 0; i < CUBE_MAP_NUM; i++)
{
    //描画先をレンダーターゲットに設定
    SetRenderTargetToShader(0, dynamicCubeTexture, i);
    ClearDrawScreen();

    //カメラの画角は90度
    SetupCamera_Perspective(90.0f / 180.0f * DX_PI_F);

    //クリップ
    SetCameraNearFar(1.0f, 10000.0f);

    //カメラの設定
    SetCameraPositionAndTargetAndUpVec(WATER_POS, targetPos[i], cameraUp[i]);

    //モデル描画
    MV1DrawModel(skyDomeModel);
}
```

周囲のモデルを反射

水の反射を実装するために動的なキューブマップを作成しました。
計算する位置からカメラに向かうベクトルを計算し、法線との反射ベクトルを使いサンプリングを行っています。
C++側ではレンダーターゲットを用意し、6方向分の描画をしてからテクスチャとしてHLSL側に渡しています。

フレネル反射

反射の強度としてフレネル反射を実装しました。

これにより、視点から水平に近い場合は反射した景色が強く見え、垂直に近い場合は水の色が強く見えるようになりました。

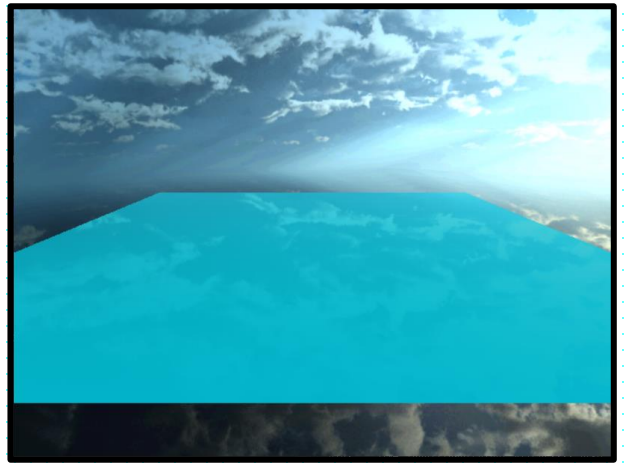
強度として計算

```
//フレネル(カメラへのベクトル,法線,屈折関係)
float fresnel = Fresnel(viewVec, input.normal, 1.000292f, 1.3334f);

//水の色
float3 waterColor = float3(0.0f, 0.73f, 0.81f);

//最終的な色
float3 finalColor = waterColor *
    (1.0f - fresnel) + cubeColor.rgb
    * fresnel;
```

関数内

```
float Fresnel(float3 toCamera,float3 normal,float n1, float n2)
{
    //屈折関係
    float A = n1 / n2;
    //入射角の cos
    float B = dot(toCamera, normal);
    float C = sqrt(1.0f - A * A * (1.0f - B * B));

    float r1 = ((A * B - C) / (A * B + C));
    float r2 = ((A * C - B) / (A * C + B));

    //反射の平均
    float ref = saturate(r1 * r1 + r2 * r2) / 2.0f;
    return saturate(ref);
}
```

遠くにある程反射率が高い



ノイズ

また、fbmノイズを作成し、法線を計算することで水に立体感を少し出すことも行いました。

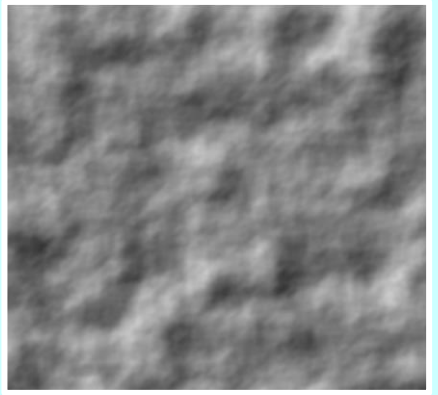
作成したパーリンノイズを一定の割合で合成しています。法線を加算し、ライティングを行っています。

fbm

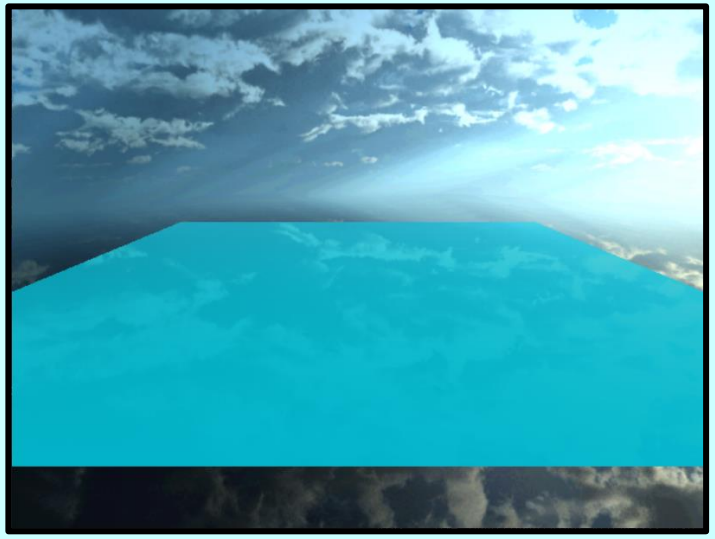
関数内

```
//fbmノイズ
float4 fbm(float2 uv)
{
    float f = 0;
    float2 q = uv;
    f += 0.5f * perlin(q, 0).x;
    q = q * 2.01f;
    f += 0.25f * perlin(q, 0).x;
    q = q * 2.02f;
    f += 0.125f * perlin(q, 0).x;
    q = q * 2.03f;
    f += 0.0625f * perlin(q, 0).x;
    q = q * 2.01f;
    f += 0.5f;
    return float4(f, f, f, 1);
}
```

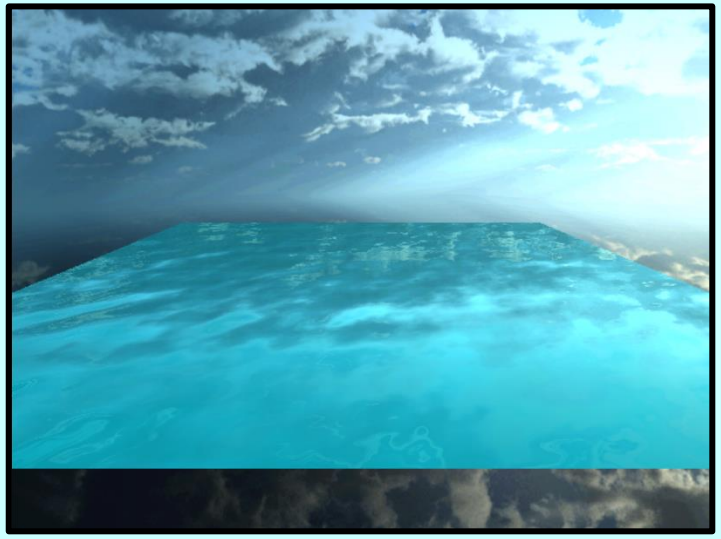
出力



適用前



適用後

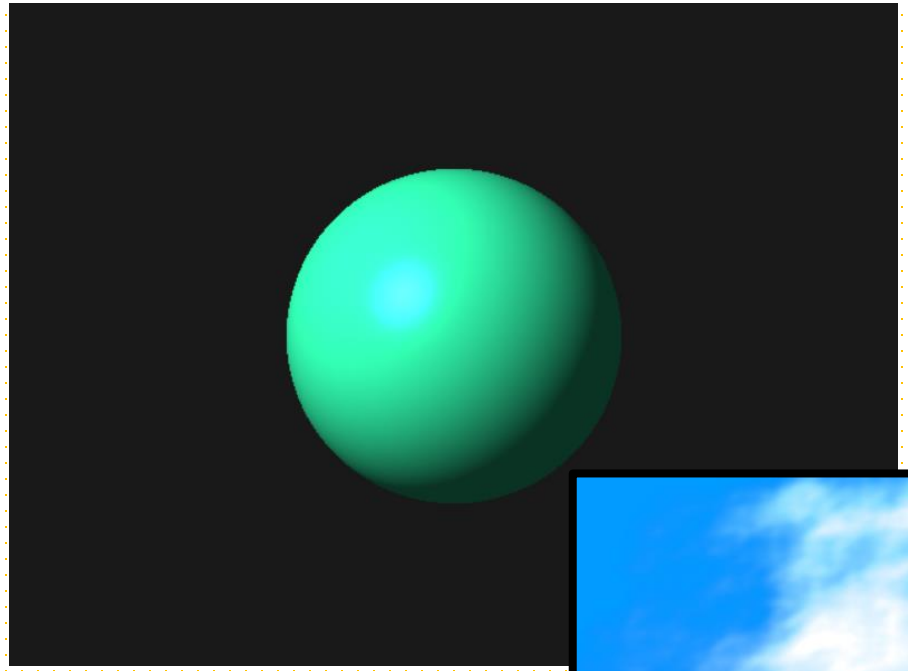


Ray Marching

レイマーチングを使用してモデリングを行いました。

ポリゴンを使わずにピクセルシェーダのみでモデリングを行っています。
少しずつレイを飛ばし、オブジェクトと接触している場合に色を付けています。

使 用 言 語 ：C++,HLSL
ラ イ ブ ラ リ ：Dxlib
制 作 期 間 ：約1週間



球体

球体のモデリングを行いました。レイを飛ばし、オブジェクトと接触している場合に色を付けています。軽くライティングを行っています。

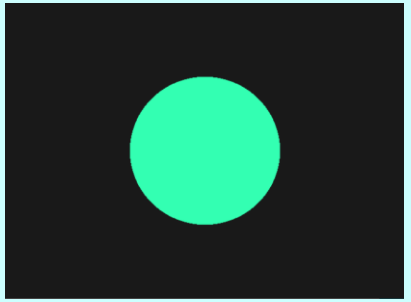
距離関数

```
float CalcSphere(float3 pos, float radius)
{
    return length(pos) - radius;
}
```

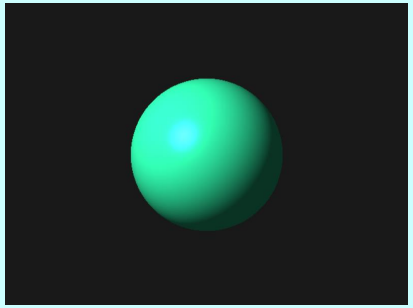
処理

```
for (int i = 0; i < 99; i++)
{
    //オブジェクトとの距離
    float distance = CalcSphere(cameraPos, radius);

    //接触しているか判定
    if (distance < 0.0001f)
    {
        //ディフューズ
        float3 norm = CalcNormal(cameraPos, radius);
        float t = dot(norm, -lightDir);
        if (t < 0.0f)
        {
            t = 0.0f;
        }
        float diffuseColor = lightColor * t;

        //スペキュラ
        float3 refVec = reflect(lightDir, norm);
        t = dot(refVec, -rayDir);
        if (t < 0.0f)
        {
            t = 0.0f;
        }
        t = pow(t, 20.0f);
        float3 specColor = lightColor * t;

        color = sphereColor * (diffuseColor + specColor + ambientColor);
        break;
    }
    cameraPos += rayDir * distance;
}
```

モデリング



ライティング後



雲

雲のモデリングを行いました。fbmノイズを合成してモデリングを行っています。加えて、衝突した距離に応じて透過率の計算も行っています。

距離関数

```
float CalcDensity(float3 pos,float2 uv)
{
    return 2.0f - (length(pos) + saturate(fbm(uv * 8 + float2(0.4f, 0.0f) * cnt) + 0.5f));
}
```

処理

```
//吸収率
float absorption = 20.0f;

for (int i = 0; i < 64; i++)
{
    //密度
    float density = CalcDensity(cameraPos,input.uv);

    if (density > 0.0001f)
    {
        //透過率
        float tmp = density / loopNum;
        trans *= 1.0f - (tmp * absorption);
        //color = float4(1, 1, 1, 1);
        if (trans <= 0.01f)
        {
            break;
        }

        //色計算
        //不透明度
        float opacity = 50.0f;
        float coeff = opacity * tmp * trans;
        float cloudColor = float4(1, 1, 1, 1);
        float4 diffuse = cloudColor * coeff;

        color += diffuse;
    }

    cameraPos += rayDir * stepZ;
}
```

モデリング

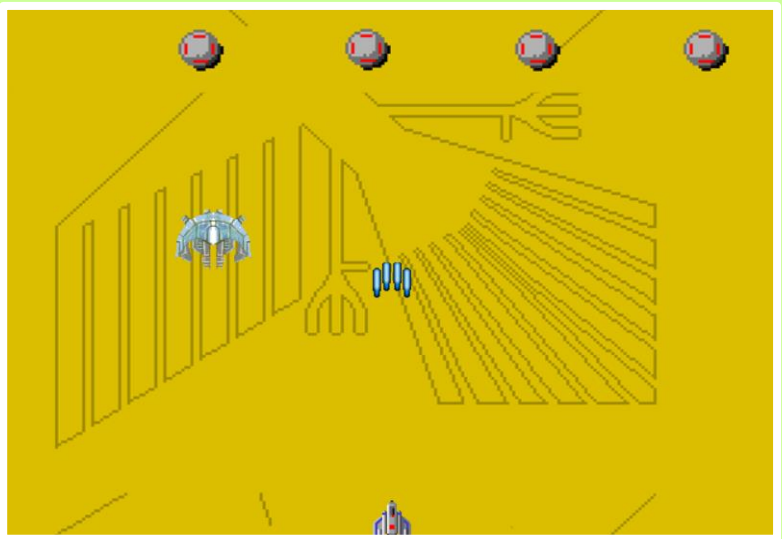
透過率適用後

授業作品

一年次

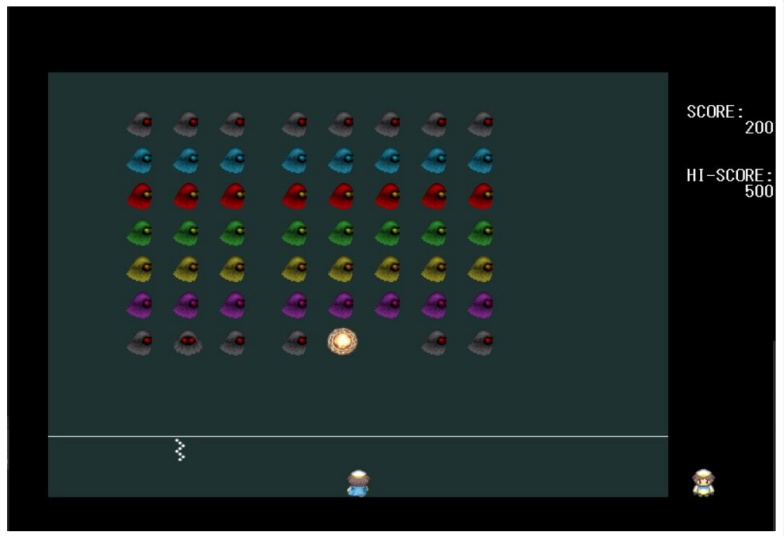
シューティング

使　用　言　語　：C
使用ライブラリ：DXLib
制　作　時　期　：1年前期



入学して授業で初めて制作したシューティングゲームです。
変数の宣言やif文の使い方等、基礎的なことについて学びました。

インベーダー



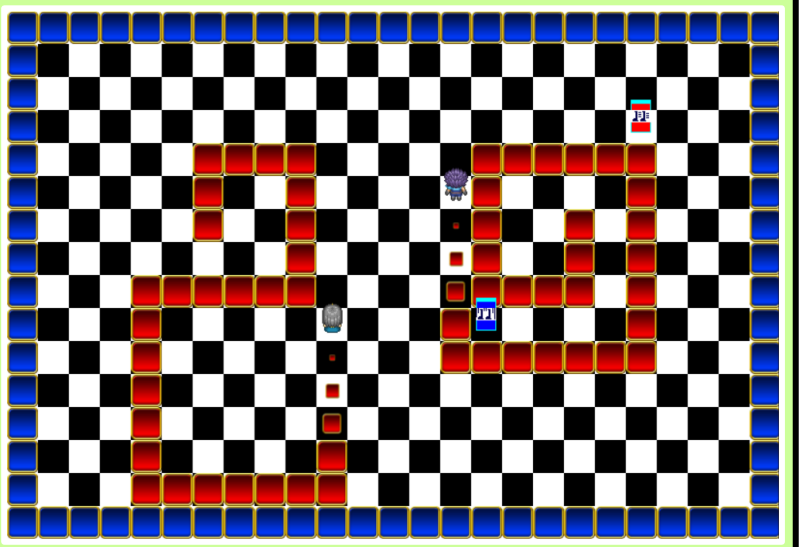
使　用　言　語　：C
使用ライブラリ：DXLib
制　作　時　期　：1年前期

配列やfor文について学びました。配列で敵の動きを制御させています。
シーンごとに描画や処理を変更するようにして、ゲームとしての流れを確立させました。

一年次

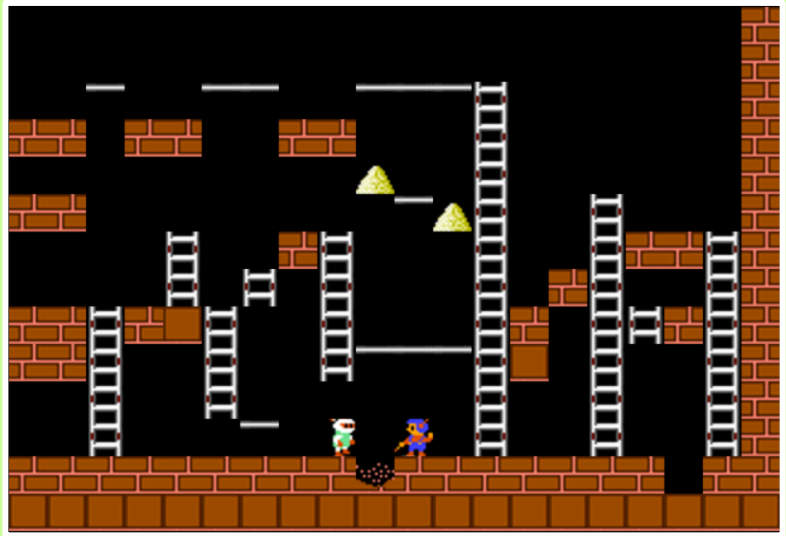
スネークゲーム

使 用 言 語 ： C++
使用ライブラリ： DXLib
制 作 時 期 ： 1年前期



C++について学び始めた作品です。シーンやオブジェクトごとにクラス分けをして実装しました。
追加の実装として移動速度が速くなるアイテムや遅くなるアイテムを実装しました。

ロードランナー



使 用 言 語 ： C++
使用ライブラリ： DXLib
制 作 時 期 ： 1年後期

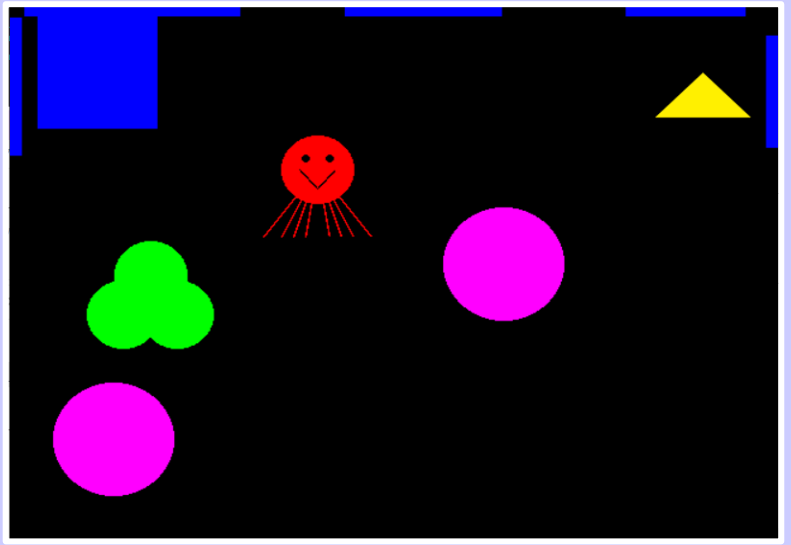
ステージのエディタを実装し、保存したり読み込んだりしました。敵の思考ルーチンを実装しました。ステージはマップチップ化をしています。

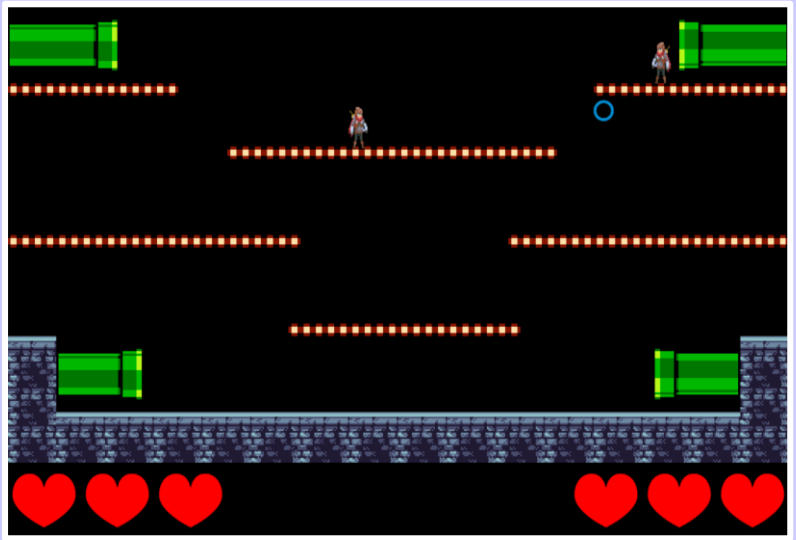
二年次

オブジェクト指向復習

使　用　言　語　：C++
使用ライブラリ：DXLib
制　作　時　期　：2年前期

オブジェクト指向の復習を行いました。
基底クラスから派生したクラスを作成して、それぞれの動きを実装しています。





格闘ゲーム

使　用　言　語　：C++
使用ライブラリ：DXLib
制　作　時　期　：2年前期

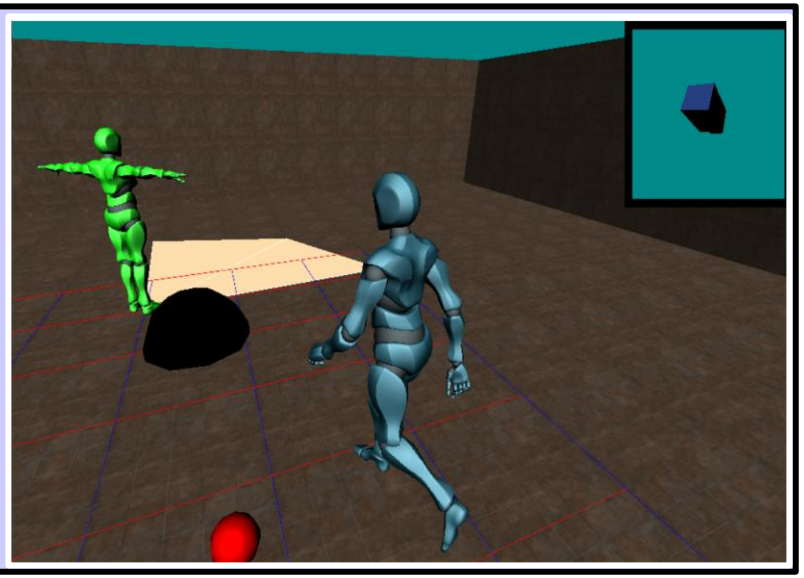
ツールで作成したステージを出力し、データを読み込んで描画しました。リングバッファを使用してコマンド技を撃つことを学びました。特にデータの読み込みは他の作品でも活かしています。

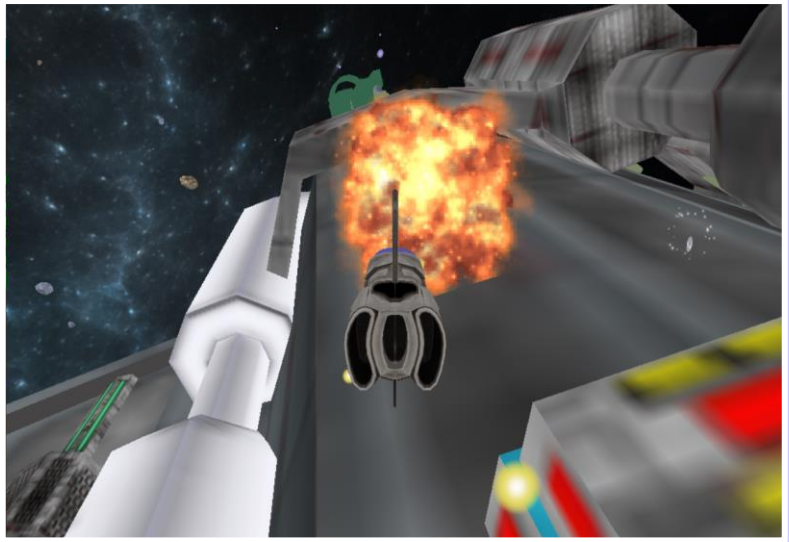
二年次

3D基礎

使　用　言　語　：C++
使用ライブラリ：DXLib
制　作　時　期　：2年後期

初めて3Dゲームの挙動を学んだ作品です。cosとsinを使った移動やカメラ回転等を実装しました。2Dと違って回転や移動するだけでも難しいと感じました。





3Dシューティング

使　用　言　語　：C++
使用ライブラリ：DXLib
制　作　時　期　：2年後期

3Dで制作したシューティングゲームです。オブジェクトとの当たり判定や向いている方向への弾の発射、オブジェクトと同期して配置すること等を行いました。

二年次

3Dアクション

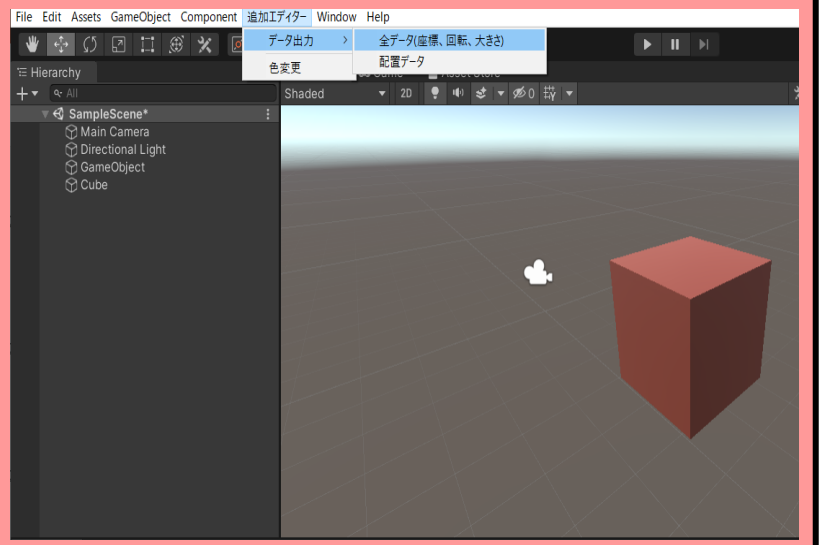
使　用　言　語　：C++
使用ライブラリ：DXLib
制　作　時　期　：2年後期

3Dのアクションゲームについて学んだ作品です。ジャンプや傾斜による滑り落ち、エフェクトの追加等を行いました。他にもアニメーションの制御も行いました。

三年次

エディタ拡張

使　用　言　語　：C#
使用ライブラリ：Unity
制　作　時　期　：3年前期



Unityのエディタ拡張をした作品です。Unityのオブジェクトの配置や回転情報等をデータで出力しています。3Dゲームのステージの作成に使用しています。

Unreal Engine4

使　用　言　語　：ブループリント
使用ライブラリ：Unreal Engine4
制　作　時　期　：3年前期

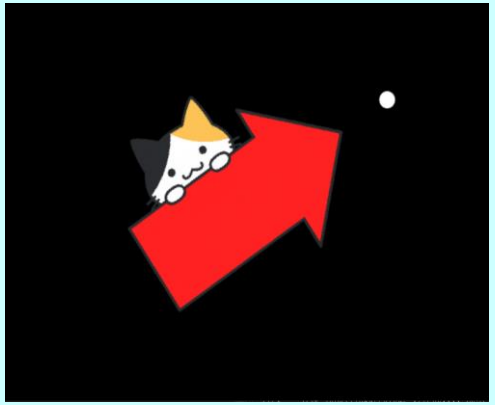
初めてアンリアルエンジンを使用して制作した簡単なゲームです。プレイヤーの基本的な挙動や弾の発射、敵の移動等、基礎的なことを学びました。エフェクトの描画も行いました。

数学

数学

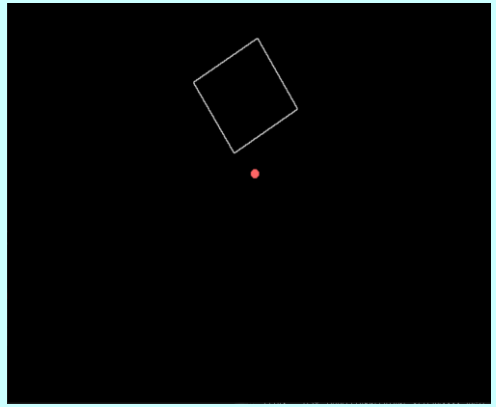
atan2

atan2について学びました。
これにより、オブジェクトを思った方向に向かせられるようになりました。



回転

矩形をマウスポインタの位置を中心に回転する処理を実装しました。

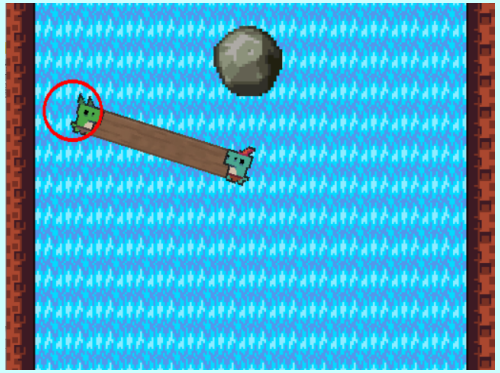


弾幕ゲーム

弾幕シューティングゲームを実装しました。
数式を利用してさまざまな弾幕を発射してくるようになっています。

滝登り

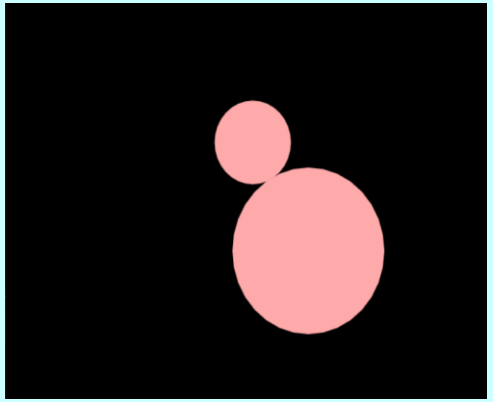
上から落下してくる岩を避けながら上に進んで行くゲームです。
回転を利用して上に進むようになっています。



数学

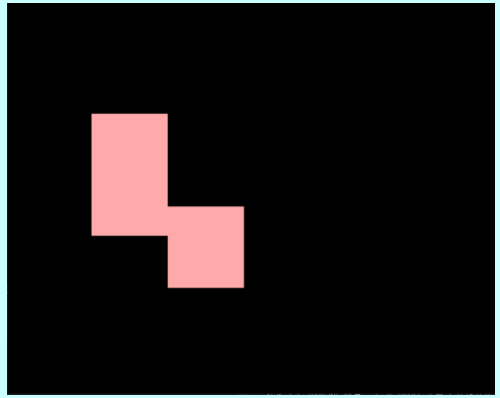
円同士の判定

円同士の当たり判定を実装しました。加えて押し出しの処理があるので、めりこまないようになっています。



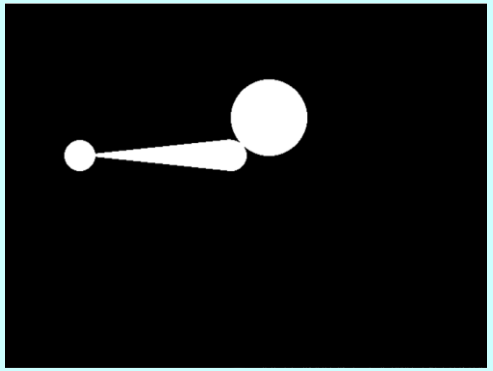
矩形同士の判定

矩形同士の当たり判定を実装しました。衝突している方向に移動すると矩形を押せるようにしています。



カプセル判定

カプセルと円の当たり判定を実装しました。

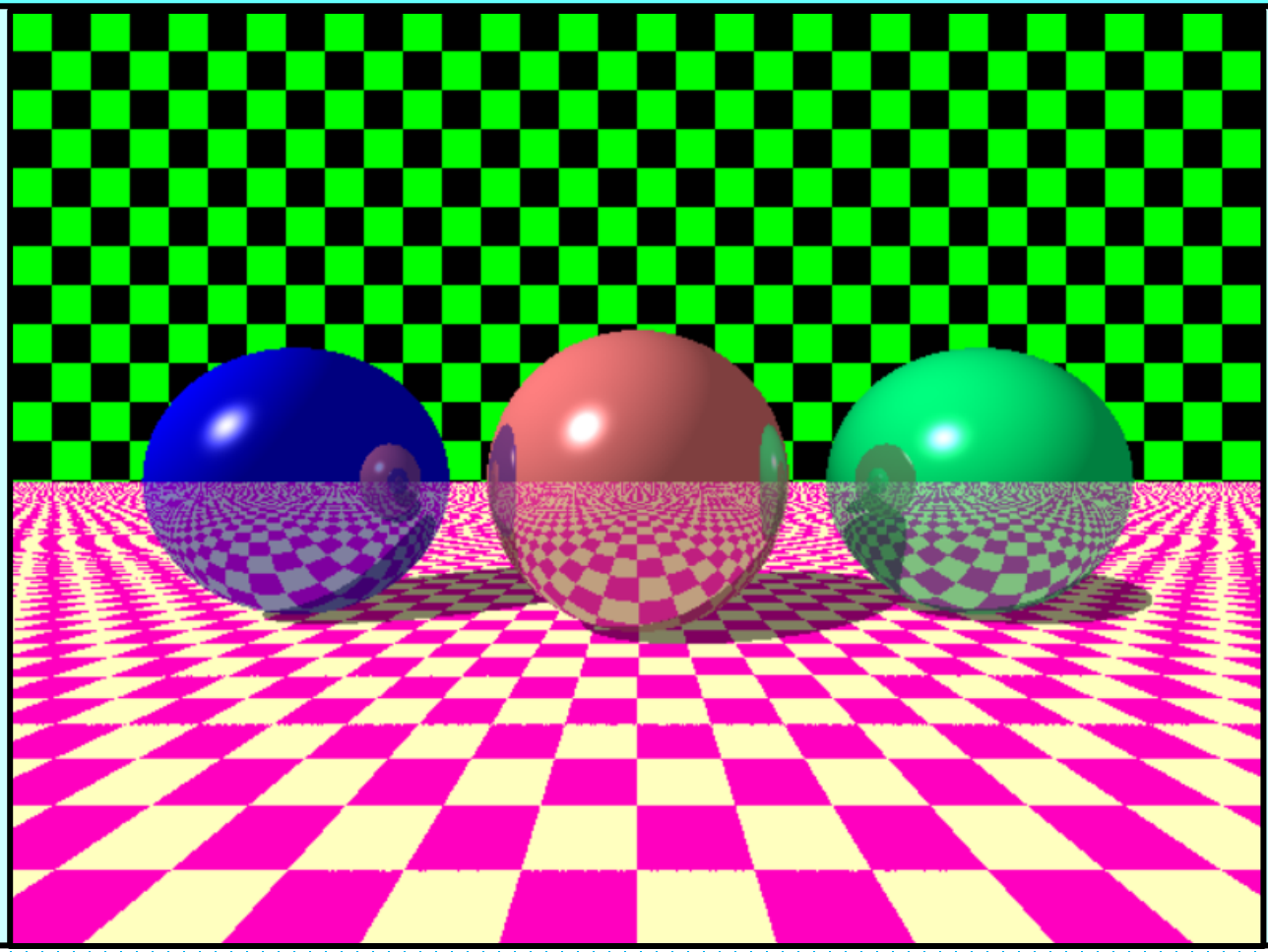


数学

レイトレーシング

古典的レイトレーシングを実装しました。床や球体を描画しています。ディフューズやスペキュラ、アンビエントを実装しました。球体に影を落としてもいます。反射も実装しています。球体を複数配置し、お互いに反射をするようにしています。

追加の課題としてオブジェクトの複数配置やOpenMPを使用した高速化を行いました。



THANK YOU FOR WATCHING