ポートフォリオ

ASOポップカルチャー専門学校　ゲーム学科
中村　天ノ助

目次

Crave Moisture

2年生の集大成として制作したゲームです。
「デビルメイクライ」や「エルデンリング」などの、
アクションゲームを参考に制作しています。

前回制作した「Re:Best Fight」で使用したAIを
さらに扱いやすく改良しました。

パリィやジャスト回避を実装しました。

HLSLを用いて、シャドウマッピングや
物理ベースレンダリングを実装しました。

AIやその他パラメータを外部データ化し、
編集しやすいように設計しています。

ゲーム制作でよく使用されるデザインパターンに
基づいた設計を意識して制作しました。

学内のゲームコンテストに提出しました。

プラットフォーム	PC
使用言語	C＋＋
使用ライブラリ	DxLib
制作期間・人数・時期	1人・ 3か月・2年生後期
ジャンル	3D対戦アクションゲーム

剣の軌跡

プレイヤーがアクションを行う際にある
剣の軌跡は、エフェクトではなく
画像を張り付けたポリゴンを自動生成
することで表現しています。
なるべく歪んだ軌跡になるように、
ポリゴンをジグザグに生成しています。

剣の軌跡

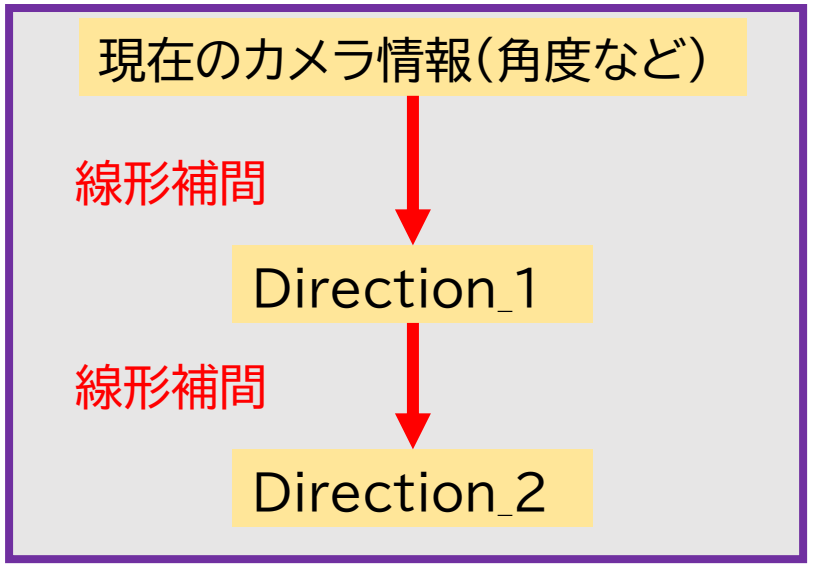
カメラ制御による演出

プレイヤーが攻撃を敵に当てた瞬間や
その他さまざまなアクションを行う際に、
カメラによる演出をしています。

アクションの迫力を強くするために
ほぼすべてのアクションに
取り入れています。

例：回避した際にカメラが斜めになる

カメラ制御データの外部ファイル化



柔軟なカメラ制御をするために
制御用のパラメータデータを
取得できる仕組みを実装しました。

現在のカメラ距離、
角度、視点からデータの
パラメータまで
それぞれ線形補間し、
それをデータ量分行うことで
カメラを複雑に制御する

プレイヤーとの距離

線形補間の速度

カメラの視点補正

カメラ角度

「特殊イベント」の実行時間

カメラシェイクなどの「特殊イベント」

```
"eventName": [ "CameraShake" ]
```

```
{
    "Direction_1" :{
        "distance" : [ 500.0 ],
        "speed": [ 2.0 ],
        "relativeTarget": [ 0.0, 0.0, 0.0 ],
        "cameraAngle": [ 0.0, 0.0, 0.0 ],
        "eventTime": [ 2.0 ],
        "eventName": [ "None" ]
    },
    "Direction_2": {
        "distance": [ 1000.0 ],
        "speed": [ 5.0 ],
        "relativeTarget": [ 0.0, 0.0, 0.0 ],
        "cameraAngle": [ 0.0, 0.0, 0.0 ],
        "eventTime": [ 45.0 ],
        "eventName": [ "None" ]
    },
```

```
839
840         // 値の初期化処理(タイマーが0の時)
841         if (!isEndLerp && time.GetTotalNowTime() <= LERP_MIN)
842         {
843             prevDis = GetDistanceTarget();
844             prevTarget = relativeTargetPos_;
845             prevDirUp = cameraUp_;
846
847             if (cameraControlIdx_ <= 0)
848             {
849                 backUpTarget = relativeTargetPos_;
850                 backUpDirUp = cameraUp_;
851             }
852         }
```

```
855
856         // 距離の線形補完処理
857         if (!IsCollision())
858         {
859             LerpDistance(prevDis, time.GetTotalNowTime());
860             LerpTargetPos(prevTarget, time.GetTotalNowTime());
861             LerpCameraUp(prevDirUp, time.GetTotalNowTime());
862
863
864
865         // 線形補完速度
866         float speed = GetRefelenceCameraContData().speed;
867         // 特殊イベントカメラが終了しているなら、元のカメラデータに戻す処理に切り替える
868         // タイマーが逆に減るようにしている
869         if (isEndEvent)speed *= REVERSE_VALUE;
870         time.UpdateTime(speed);
871
```

```
874         // 線形補完の完了処理(データの補完)
875         if (time.GetTotalNowTime() >= LERP_MAX)
876         {
877             isEndLerp = true;
878         }
879         // 線形補完の完了処理(元に戻す処理)
880         else if (time.GetTotalNowTime() <= LERP_MIN)
881         {
882             time.InitTotalNowTime();
883             isEndLerp = false;
884             isEndEvent = false;
885
886             // イベントデータの全消去
887             for (auto& data : cameraControlData_)
888             {
889                 data.reset();
890                 data = nullptr;
891             }
892             cameraControlData_.clear();
893             cameraControlIdx_ = 0;
894             isCanSetUpEventCamera_ = true;
895         }
```

線形補間が終了し、
セットした外部データ
も存在しない場合
特殊制御は終了

プレイヤーとの距離、視点、角度をそれぞれ線形補間

外部データの”eventName”に基づいた
固定のイベントを行う

```
902         if (!isEndEvent)
903         {
904             if (CameraDirectionEvent(GetRefelenceCameraContData().eventName))
905             {
906                 // イベントの終了処理
907                 isEndEvent = true;
908                 time.SetTotalNowTime(LERP_MAX);
909
910                 if (NextCameraControlData())
911                 {
912                     isEndLerp = false;
913                     isEndEvent = false;
914                     time.InitTotalNowTime();
915                 }
916                 else
917                 {
918                     prevDis = GetDefaultDistanceTarget();
919                     prevTarget = backUpTarget;
920                     prevDirUp = backUpDirUp;
921                     isCanSetUpEventCamera_ = false;
922                 }
923             }
924         }
925
```

プレイヤーのアクションに応じた自動カメラ制御

外部データを使用したカメラ制御とは別に
プレイヤーのアクションに合わせて
カメラ角度を操作する仕組みを実装しています

例:プレイヤーの攻撃アクション時に
カメラを自動で敵の方角に向かせている

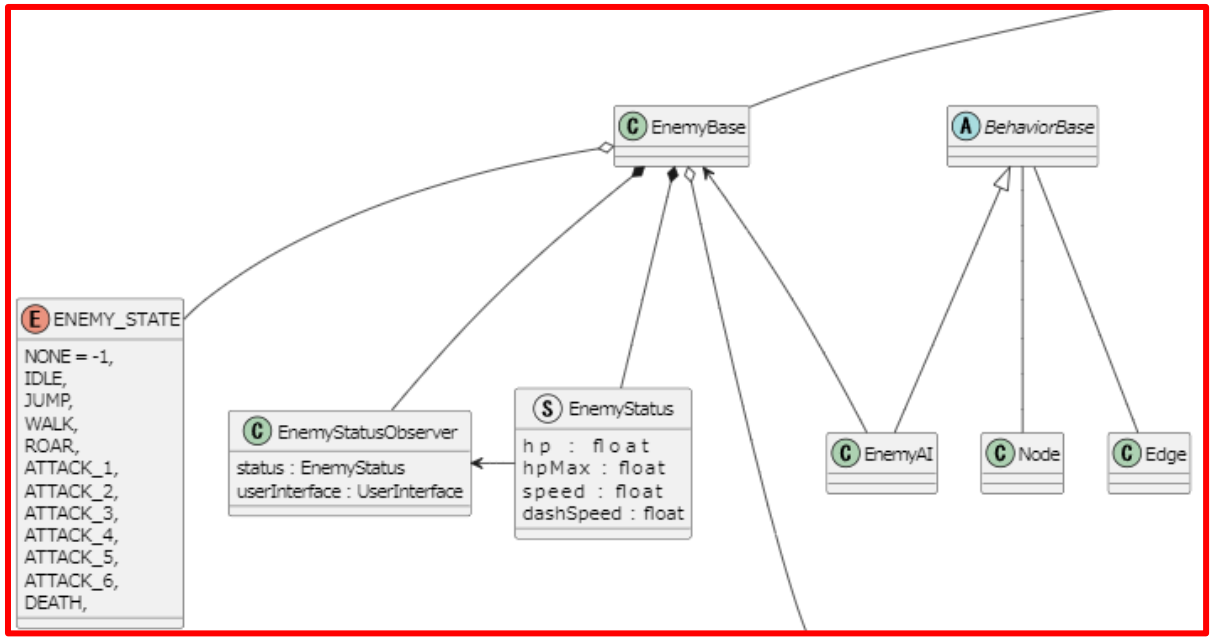
```
1810
1811            // カメラを強制的に相手の方向に向かせる(線形補間なので呼び出し続ける)
1812            SceneManager::GetInstance().GetCamera().lock()->
1813                CorecionRotationPos(shareOtherActorsData_->
1814                    GetShareActorsData(ShareActorsData::ACTOR_NAME::ENEMY).lock()->position);
1815        }
```

BehaviorBase

AIを学ぶ過程で、様々な構築方法
を学びました。
そんな中で、c++での構築のしやすい
「BehaviorBase」の考え方に
基づいたクラス構造で実装しました。

右図はエネミーのクラス構造です。
EnemyBaseを上位にし、
BehaviorBaseクラスを継承した
EnemyAIクラスを所持させることで
独立性の高いAIになっています。

エネミーAIのクラス図(一部抜粋)



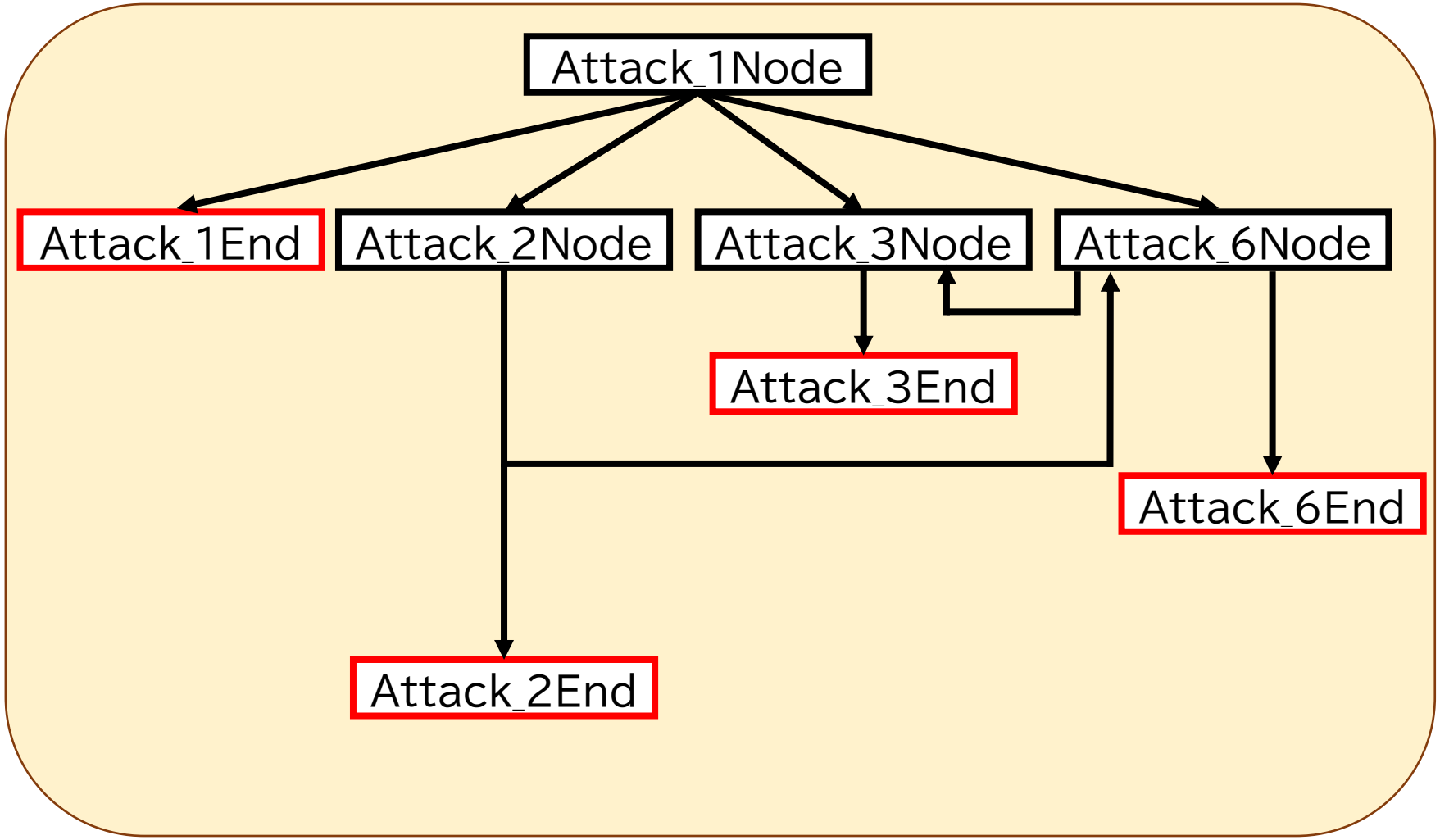
BehaviorTree

黒矢印で示した「エッジ」に
遷移条件があり、
条件を満たしたエッジが遷移先の
ノードへ遷移します。

これを「終端ノード」または
全ての遷移条件を満たさなかった
ノードまで繰り返し、
最終的に行動をします。

遷移する過程でノード同士で
ループすることがないように
気を付けて構築する必要が
ありますが視覚的にわかりやすい
AIの分岐をすることが
できています。

攻撃1からの分岐(一部省略)（赤が終端ノード）

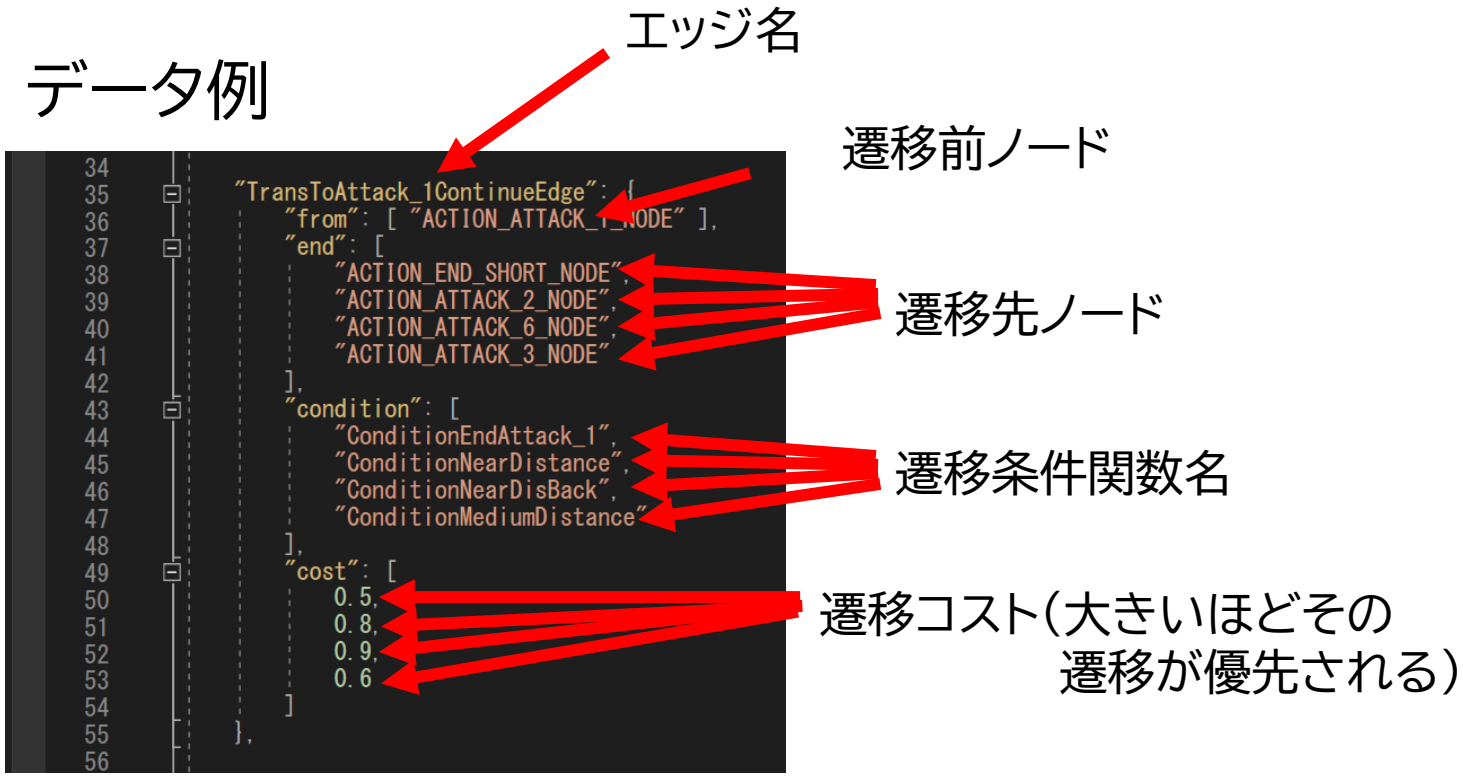


外部データ構造

AI構築に必要なエッジデータを
外部ファイル化しました。

JSONファイル形式で、
遷移先ノードと遷移条件関数を
それぞれ対応させ、
プログラム側で動的に保存し、
柔軟に構築できるように
しています。

データ例



外部データに基づいたノード探索処理

```
178     //探索完了したか
179     bool isSearch = false;
180     //遷移後のインデックス
181     int nextIndex = NONE_INDEX;
182
183     while (!isSearch)
184     {
185         nextIndex = TransitionNextNode(nowIndex_);
186
187         //次のノードに遷移しなかった
188         //ループの終了
189         if (nextIndex == NONE_INDEX)isSearch = true;
190
191         //次のインデックスに遷移
192         else
193         {
194             nowIndex_ = nextIndex;
195             //AIのアクション終了を取り消す
196             isEndAIAction_ = false;
197         }
198     }
```

現在のノードにつながっているすべてのエッジの遷移条件を判定する

```
143 int BehaviorBase::TransitionNextNode(int fromIdx)
144 {
145     int idx = NONE_INDEX;
146     float nowWeight = NONE_WEIGHT;
147
148     int endNodeSize = 0;
149
150     for (const auto& edge : edges_)
151     {
152         if (edge.second->GetFromIndex() != fromIdx)continue;
153
154         //接続されているすべてのノードを検索
155         endNodeSize = edge.second->GetEndNodeNum();
156         for (int ii = 0; ii < endNodeSize; ii++)
157         {
158             if (!edge.second->IsCondition(ii))continue;
159
160             //コストが大きいほど優先度が高い
161             //コストが同じなら、後に格納されている順
162             if (nowWeight <= edge.second->GetCost(ii))
163             {
164                 nowWeight = edge.second->GetCost(ii);
165                 idx = edge.second->GetEndIndex(ii);
```

遷移終了時に現在ノードに登録したアクションを行う

```
201 void BehaviorBase::NodeAction(void)
202 {
203     if (nowIndex_ == NONE_INDEX)return;
204
205     const auto& ite = nodes_.find(nowIndex_);
206     assert(ite != nodes_.end() || !"存在しないノードインデックスは指定できません");
207
208     //アクション
209     nodes_.at(nowIndex_)->NodeAction();
210 }
```

シャドウマッピング

被写界深度を応用して、すべてのモデルに
影を落とすレンダリングを行いました。
HLSL言語を用いて独自のシェーダープログラムを
書きました。

本来有るべき場所に影がないという問題が発生し、
その解決に最も時間をかけました。

最終的に、光源とモデル頂点の法線とで内積をとり、
一定以上の大きさだった部分は影を落とさないように
することで解決しました。

あえて影を落とさないことで、影の形をきれいにする
ことはできましたが、それでも影の形が歪な部分が
あるので、これも解決できるように精進しようと
思っています。

物理ベースレンダリング

クックトランス計算モデルによる
物理ベースレンダリングに挑戦しました。

フレネル反射の公式、
幾何減衰(クック・トランスモデル)、
マイクロファセット法線分布関数
これらを使用して、SpecularColorを求めています。

フォトリアルなレンダリングを目標としていましたが、最終的に右図のようなグラフィックで妥協をしました。

次に制作する機会があれば、
もっとリアルなレンダリングをできるようになりたいと思っています。

画像だとわかりにくいですが、
カメラの位置で明るさが変化しています

物理ベースレンダリングの計算処理（一部抜粋）

```
//D_GGXの項
float D = D_GGX(halfVector, norm, PSInput.uv);

//F(フレネル反射項)
float F = Flesnel(viewDirNorm, halfVector);

//G(マイクロファセット)の項
float G = G_CookTorrance(lightDirNorm,
        viewDirNorm, halfVector, norm);

//スペキュラーおよびディフューズを計算
float specularRef = (D * F * G) / (4.0f * NdotV * NdotL + 0.000001f);
specularRef *= g_lightPower;

float3 diffuseRef = color.xyz * NdotL * g_lightPower;
diffuseRef += g_lightPower * g_lightColor;

//アンビエント
float3 ambientLight = diffuseRef * g_lightPower;
```

```
// フレネルの項
float Flesnel(float3 V, float3 H)
{
    float VdotH = saturate(dot(V, H));
    float F0 = saturate(fresnelRef_);
    float F = pow(VdotH, 5.0);
    F *= (1.0 - F0);
    F += F0;
    return F;
}
```

```
// G - 幾何減衰の項（クック トランスモデル）
float G_CookTorrance(float3 L, float3 V, float3 H, float3 N)
{
    float NdotH = saturate(dot(N, H));
    float NdotL = saturate(dot(N, L));
    float NdotV = saturate(dot(N, V));
    float VdotH = saturate(dot(V, H));

    float NH2 = 2.0 * NdotH;
    float g1 = (NH2 * NdotV) / VdotH;
    float g2 = (NH2 * NdotL) / VdotH;
    float G = min(1.0, min(g1, g2));
    return G;
}
```

```
//D(GGX)の項
float D_GGX(float3 H, float3 N, float2 uv)
{
    float NdotH = saturate(dot(H, N));
    float roughness = saturate(rough.Sample(diffuseMapSampler, uv).r);
    float alpha = roughness * roughness;
    float alpha2 = alpha * alpha;
    float t = ((NdotH * NdotH) * (alpha2 - 1.0) + 1.0);
    float PI = 3.1415926535897;
    return (1.0f - step(NdotH, 0)) * alpha2 / (PI * t * t);
}
```

Re:Best Fight

プラットフォーム	PC
使用言語	C＋＋
使用ライブラリ	DxLib
制作期間・人数・時期	1人・ 3か月・2年生前期
ジャンル	3D対戦アクションゲーム

初めて、3Dゲームの制作に挑戦しました。
プレイヤーは1人～2人でキャラクターを選び対戦し、先に相手のHPを0にしたほうが勝利となります。

HLSLを用いたシェーダープログラムを組んで
太陽光や影、ポイントライトを実装しました。

1人でも遊べるように、BehaviorベースのAIを実装し、CPUと対戦ができるようにしました。

攻撃を入力によって派生させられるようにコマンド機能を実装しました。

「ジャンプフォース」など、3種類以上の対戦ゲームを参考にし、それらに近いゲームにすることを目標に制作しました。

カメラ制御

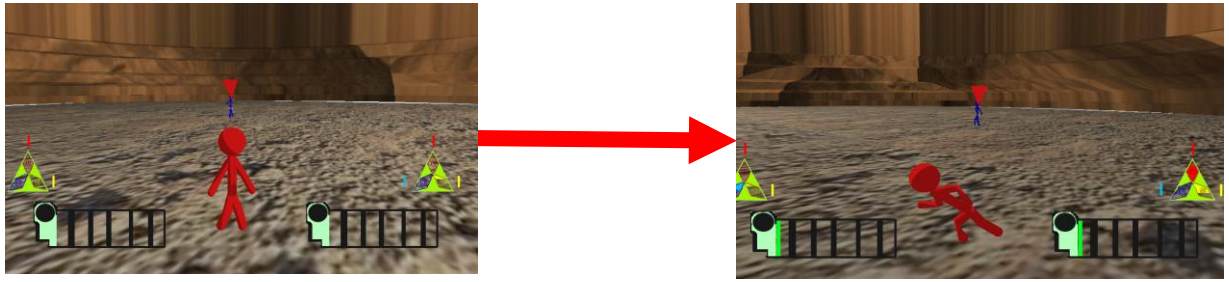
初めはプレイヤーにカメラが追従する形で実装していましたが、どうしても対戦相手がカメラ外に出てしまい、プレイしにくいという問題点がありました。

そこで、対戦相手を常にロックオンし続けるという方法で実装する形で落ち着きました。

対戦開始時にプレイヤーと対戦相手のベクトルを定数値で保存し、その後再びベクトルを使用してベクトル間の角度を求め、カメラを回転させる形で実装しました。

常にカメラがプレイヤーと対戦相手のベクトル上に位置することで、対戦相手をロックオンした状態になり両方をカメラ内に写すことができました。

プレイヤーが移動しても、対戦相手が見える



カメラの追従(一部抜粋)

```
322     //追従更新
323     void Camera::UpdateFollow()
324     {
325         //空の場合は処理しない
326         if (targetTrans_.lock() == nullptr)return;
327
328         Position3DF target = targetTrans_.lock()->pos;
329         target.y = 0.0f;
330         Position3DF lockon = lockonTrans_.lock()->pos;
331         lockon.y = 0.0f;
332
333         Dir3DF vec1 = MyUtility::AXIS_Z;
334         Dir3DF vec2 = VSub(lockon, target);
335
336         //ロックオン対象の方向を向く
337         Quaternion angle = std::move(Quaternion::FromToRotation(vec1, vec2));
338         //89.9598度以上だとz軸以外が反対方向になるので強制的に修正
339         if (Quaternion::Angle(def_rot_, angle) >= 89.9598f)
340         {
341             angle = angle.Mult(Quaternion::AngleAxis(MyUtility::Deg2RadD(180.0f), vec2));
342         }
343
344         if (lockonTrans_.lock() == nullptr)
345         {
346             targetPos_ = VAdd(targetTrans_.lock()->pos, IDEAL_RELATIVE_TARGET_POS);
347         }
348         //ロックオン対象がいない場合は追従対象の上らへんが視点
349         else
350         {
351             //ロックオン対象の上らへんが視点
352             targetPos_ = VAdd(lockonTrans_.lock()->pos, IDEAL_RELATIVE_TARGET_POS);
353             rot_ = std::move(angle);
354             rot_.Normalize();
355         }
356
357         //理想の座標
358         Position3DF localPos = VAdd(targetTrans_.lock()->pos, rot_.PosAxis(IDEAL_RELATIVE_POS));
359         //距離が一定以上に近い場合
360         if (VSize(vec2) < TARGET_LOCKON_DIS)
361         {
362             localPos = VAdd(targetTrans_.lock()->pos, rot_.PosAxis(IDEAL_RELATIVE_NEAR_POS_L));
363         }
364
365         //徐々に理想の座標に近づけていく
366         MyUtility::RelativeVector(speed_, localPos, pos_);
367     }
```

特殊なカメラ制御

二人で対戦を行う際のカメラ制御について特に悩みました。

二人のうちの一人にずっと追従する形では、もう片方の人が使用するキャラクターが見えにくく、プレイしにくくなってしまっていました。

結果、二人の使うキャラクターと現在動作しているカメラとの距離をそれぞれとり、より近いほうにカメラが追従することで両方の人に追従できる形に落ち着きました。

理想ではPCを2台使用しての通信プレイだったので、次回からは実現できるように技術力を高めようと思っています。

どちらのキャラにカメラを追従させるか

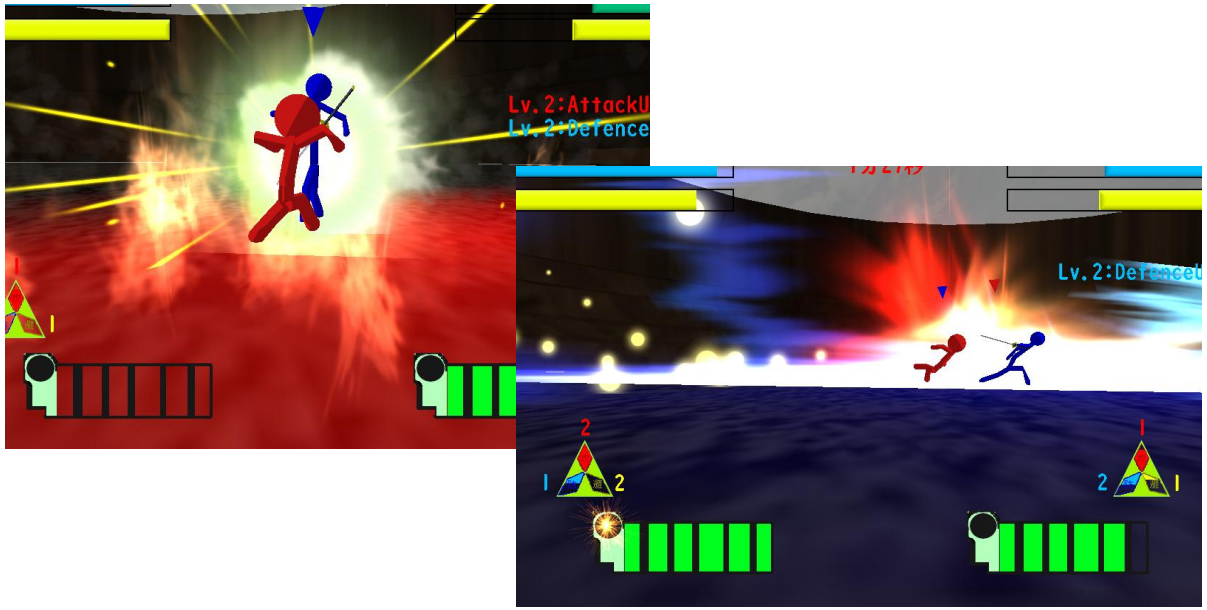
```
234     // プレイヤーと対戦相手で、それぞれのカメラとの距離を比較し、
235     // 近いほうを新たに追従対象とする
236     else
237     {
238         float PDis = VSquareSize(VSub(player_->transform_->pos, camera_->GetPos()));
239         float EDis = VSquareSize(VSub(enemy_->transform_->pos, camera_->GetPos()));
240
241         // どちらかには必ず追従
242         if (PDis <= EDis)
243         {
244             camera_->CompSetCameraInfo(player_->actorCam_->GetPos(),
245                 player_->actorCam_->GetTarget(), player_->actorCam_->GetRot());
246         }
247         else
248         {
249             camera_->CompSetCameraInfo(enemy_->actorCam_->GetPos(),
250                 enemy_->actorCam_->GetTarget(), enemy_->actorCam_->GetRot());
251         }
252
253         camera_->SetDefaultSpeed();
254     }
255 }
```

CompSetCameraInfo関数の第1,2引数で追従対象とロックオン対象を決めています。

```
241         // どちらかには必ず追従
242         if (PDis <= EDis)
243         {
244             camera_->CompSetCameraInfo(player_->actorCam_->GetPos(),
245                 player_->actorCam_->GetTarget(), player_->actorCam_->GetRot());
246         }
247         else
248         {
249             camera_->CompSetCameraInfo(enemy_->actorCam_->GetPos(),
250                 enemy_->actorCam_->GetTarget(), enemy_->actorCam_->GetRot());
251         }
```

必殺技や特定のアクションには
カメラ制御による疑似的な3Dアニメーション
を実装しています。

必殺技アニメーションの一例



カメラの角度や座標をプログラムで制御しています。(一部抜粋)

```
    //角度更新
    Quaternion rot;
    if (animController_->GetStepAnimTime() < ANIM_STEP_MIN && !CFlag)
    {
        rot = Quaternion::AngleAxis(MyUtility::Deg2RadD(ANGLE_ROT_1), MyUtility::AXIS_Y);
        angle = angle.Mult(rot);
        CFlag = true;
    }
    else if (animController_->GetStepAnimTime() >= ANIM_STEP_MIN &&
        animController_->GetStepAnimTime() < ANIM_STEP_MIDLE)
    {
        angle = angle.Mult(Quaternion::AngleAxis(MyUtility::Deg2RadD(-ANGLE_ROT_2), MyUtility::AXIS_Y));
    }
    else if (animController_->GetStepAnimTime() >= ANIM_STEP_MIDLE &&
        animController_->GetStepAnimTime() < ANIM_STEP_MIDLE_3 && CFlag)
    {
        CFlag = false;
        rot = Quaternion::AngleAxis(MyUtility::Deg2RadD(-ANGLE_ROT_1), MyUtility::AXIS_Y);
        angle = angle.Mult(rot);
    }
    else if (animController_->GetStepAnimTime() >= ANIM_STEP_MIDLE_3 && !CFlag)
    {
        CFlag = true;
        rot = Quaternion::AngleAxis(MyUtility::Deg2RadD(-ANGLE_ROT_1), MyUtility::AXIS_Y);
        angle = angle.Mult(rot);
    }
    else if (!animController_->IsPlayAnim())
    {
        CFlag = false;
    }

    camera_.lock()->SetRot(angle);
```

オリジナルゲーム

宇宙戦艦防衛戦

詳細

プラットフォーム　PC

使用言語　C＋＋

使用ライブラリ　DxLib

制作人数　1人

制作期間・時期　3か月・1年生前期

ジャンル　2Dシューディングゲーム

説明

初めて制作したオリジナルゲームです。
企画、実装まですべて担当しました。
スクリーンが1つ～3つに分かれており、
スクリーンを切り替えながら、敵を砲台で倒し、
制限時間まで戦艦を守り切ることでクリアです。
敵の種類に応じたスコアも実装しました。

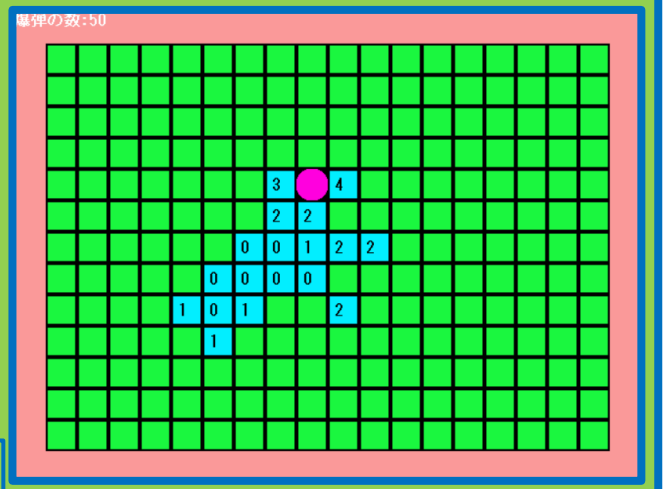
スコアは外部ファイルに記録しており、
ランキングとしてゲーム内に表示できます。

アイテムを8種類実装しており、
一定時間でランダムに3つ取得・使用できます。

使用言語・ライブラリなどすべて初めてで、
まずは完成させることを目標に制作しました。

MineSweeper

詳細



プラットフォーム　PC

使用言語　C＋＋

使用ライブラリ　DxLib

制作人数　1人

制作期間・時期　1日・1年生前期

ジャンル　MineSweeper

説明

プログラムの配列の計算を効率化する手段を学ぶために制作しました。
ゲームはGoogleで遊べるMineSweeperと同じ仕様です。

配列で格子状に表現しています。計算量を削減し、処理速度を上げるために、
配列の初期化時に先頭に地雷を個数分置き、
初期化終了時に、ランダムな二つの要素番号で配列をswapする処理を配列の要素数分行うアルゴリズムを実装しました。

動的に格子の数と地雷の数を決められるようにvector配列を使用して実装しています。

TypingGame

詳細

プラットフォーム　PC

使用言語　C＋＋

使用ライブラリ　DxLib

制作人数　1人

制作期間・時期　2日・1年生前期

ジャンル　TypingGame

説明

自分のタイピング速度を上げたいと思い制作しました。
内部でロードしてある文字列データをランダムに選択され、それをローマ字で打つ普通のタイピングゲームです。

UIを充実しており、打ち間違いがわかりやすいようにその文字の色だけ変えたり、確定ボタンを押した際の正解・不正解の演出を実装しました。

入力文字の比較処理で、計算効率を上げるために入力した瞬間だけ、入力回数に対応した文字と比較するようにしています。

Obvious/Strange

詳細

プラットフォーム　PC

使用言語　C＋＋

使用ライブラリ　DxLib

制作人数　2人

制作期間・時期　3か月・1年生後期

ジャンル　謎解き脱出ゲーム

説明

友人と一緒に二人チームで制作しました。
私はプログラムを担当しました。
ギミックのある部屋が4つあり、ギミックをすべて解くことで出口を開き、脱出できます。

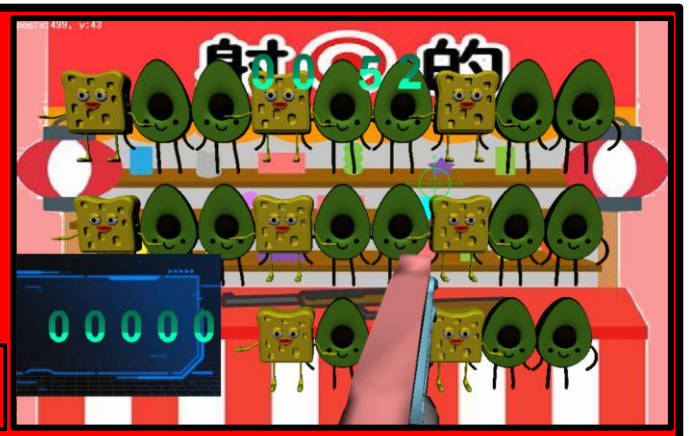
ギミックは磁石や化合など、「科学」を基にしています。磁石の極を切り替えたり、アイテムを化合し、別のアイテムを作り、使用する機能を実装しました。

ギミックに使用するアイテムは10種類を超えており、それらを管理するためのシステムを構築しました。

初めてのチーム制作で、コミュニケーションを積極的に取りながら認識の違いが起きないように政策を進めました。

お祭り射的ゲーム

詳細



プラットフォーム　PC

使用言語　C＋＋

使用ライブラリ　DxLib

制作人数　1人

制作期間・時期　1週間・2年生前期

ジャンル　3D射的ゲーム

説明

3Dゲームの当たり判定やベクトル操作を学ぶために、制作しました。
一定時間内に、配置されている景品をできるだけ多く撃つことで高いスコアを目指すゲームです。

射的の弾は発射時に、銃の向いている角度に合わせて、前方向のベクトルを計算しています。

当たり判定はDxLibライブラリの関数を使用していますが、当たり判定の必要な情報だけを取得するために、情報取得用の関数を自力で実装しました。

また、当たり判定は景品の3Dポリゴンと銃の発射時のベクトルで計算し、処理速度と精密性を向上させる工夫をしています。

CoockingGame

詳細

プラットフォーム　PC

使用言語　C＋＋

使用ライブラリ　DxLib

制作人数　4人

制作期間・時期　2日・2年生後期

ジャンル　闇鍋スコアアタック

説明

福岡学生ゲームジャムという組織のイベントに参加した際に、ランダムに決められたチームで、テーマ「ごちゃまぜ」で制作しました。私は、チームの中でリードプログラマーとして携わりました。

1フェーズで上空から食べ物がランダムで落下していき、それをマウスで拾い、猫の鍋に入れることでポイントを得ます。そして、2フェーズのルーレットの結果によってポイントを乗算し、スコアが決まります。

制作の中では企画に最も時間をかけ、チーム内でのゲームの形を統一することを意識しました。

2日間のゲームジャムということもあり、実装に限りがありました。いかに効率よくプログラムを組めるかを常に考えて作業し、完成させました。

授業作品

1年生

制作時期　　1年生前期

使用言語　　C++

使用ライブラリ　DxLib

1年生で最初に制作したゲームです。簡単な2Dのシューティングゲームになっています。
アニメーションや矩形の当たり判定など、ゲーム作りにおける基本的なことを学びました。
縦にスクロールする処理など、通常のプログラミングとは違う、ゲーム制作においてのプログラミングの考え方を学びました。

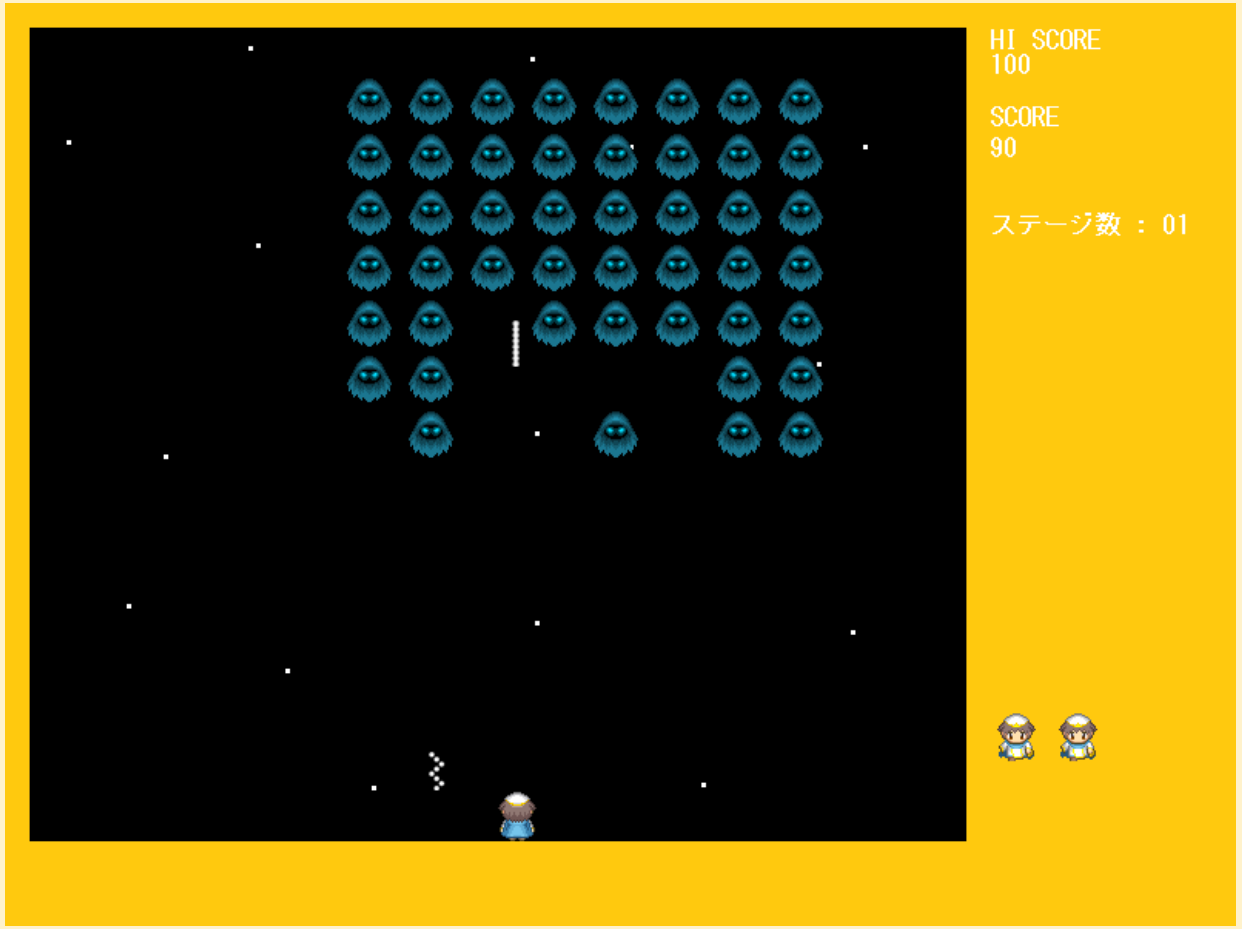
制作時期　　1年生前期

使用言語　　C++

使用ライブラリ　DxLib

エネミーの状態を管理するという課題でした。

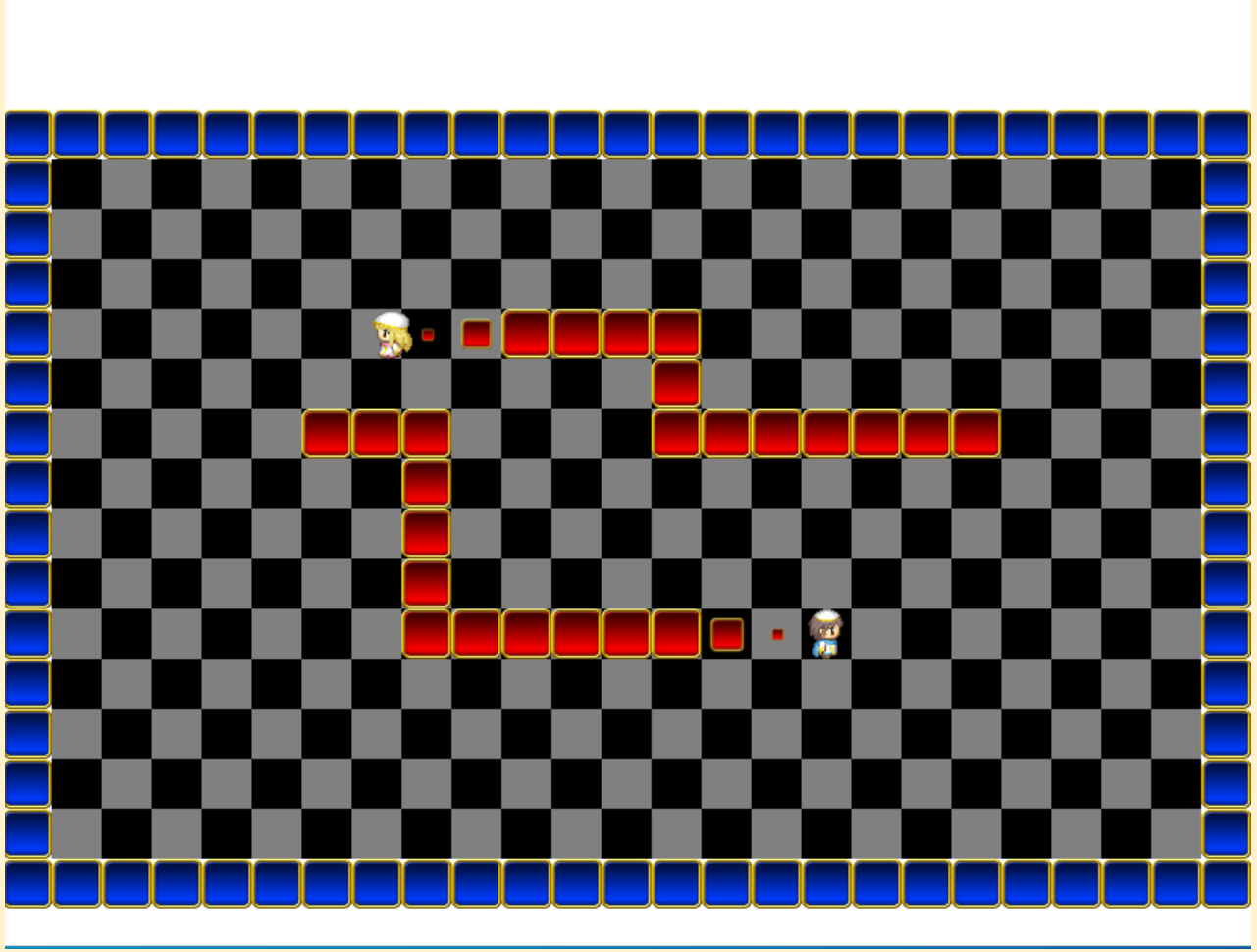
エネミーを個ではなく集団で制御をしています。
また、集団の中でもプレイヤーによって倒されているかで描画をするか決める使用を実装しています。
また、エネミーをすべて倒したら次のステージに進めるやハイスコアをゲームプレイ中だけ保存し、ゲームとして楽しめるようにしました。



制作時期	1年生前期
使用言語	C++
使用ライブラリ	DxLib

スネークゲームを制作しました。マップチップの配置を動的に行うことが課題でした。

プレイヤーは進行方向の情報をあらかじめ持ち、自動で進みます。1マスごとに方向転換できるように制御を行いました。またゲームの終了条件は、
「配置されたマップチップに衝突」、「壁に衝突」、
「キャラクター同士が衝突」と複数あるため、それらをまとめて管理するクラスを作成しました。



制作時期　　1年生前期

使用言語　　C++

使用ライブラリ　DxLib

簡単なRPG風のスコアアタックゲームです。

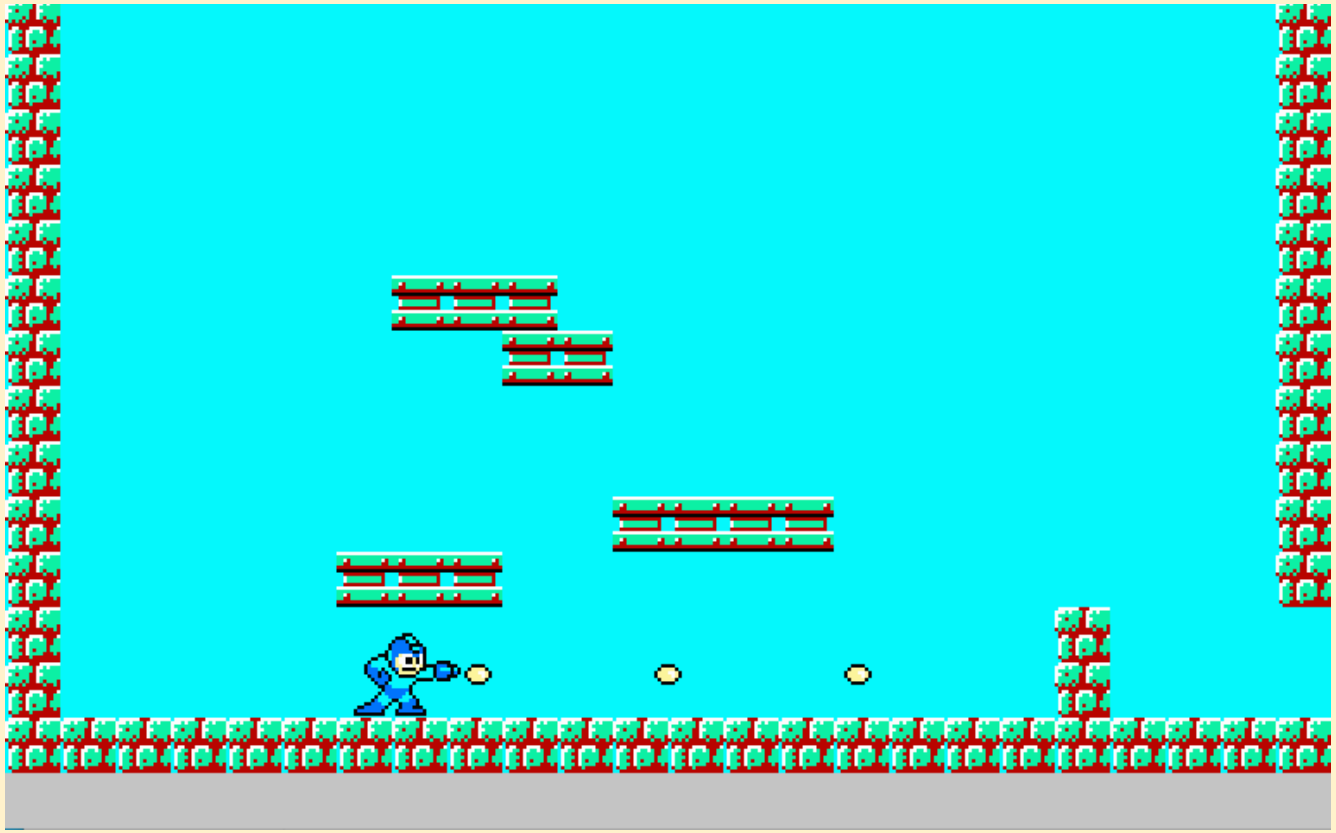
外部データから取り込んだ情報を元に、ステージを作成する課題でした。
CSV形式のデータからステージの広さに応じて動的に、プログラム内の配列に取り込み使用しました。
また、動きがそれぞれで異なる複数のエネミーを独立して動かしつつ、一元管理することにも挑戦しました。
左上にある階段に触れると、エネミーのいない
別のステージに移動することもできます。

制作時期　　1年生前期

使用言語　　C++

使用ライブラリ　DxLib

初めてのアクションゲームの制作をしました。
1年生前期の集大成で、今までに習ってきたことを
全て活用して制作しました。
プレイヤーに対する重力や体の部位に合わせた矩形
の当たり判定を実装しています。具体的には、
プレイヤーの足のつま先がブロックに接していると
つま先立ちができるようにしています。
撃っている銃弾や種類の異なるブロックなどを
なるべく細かくクラス分けし、オブジェクト指向を
意識しながら制作しています。



2年生

制作時期　　2年生前期

使用言語　　C++

使用ライブラリ　DxLib

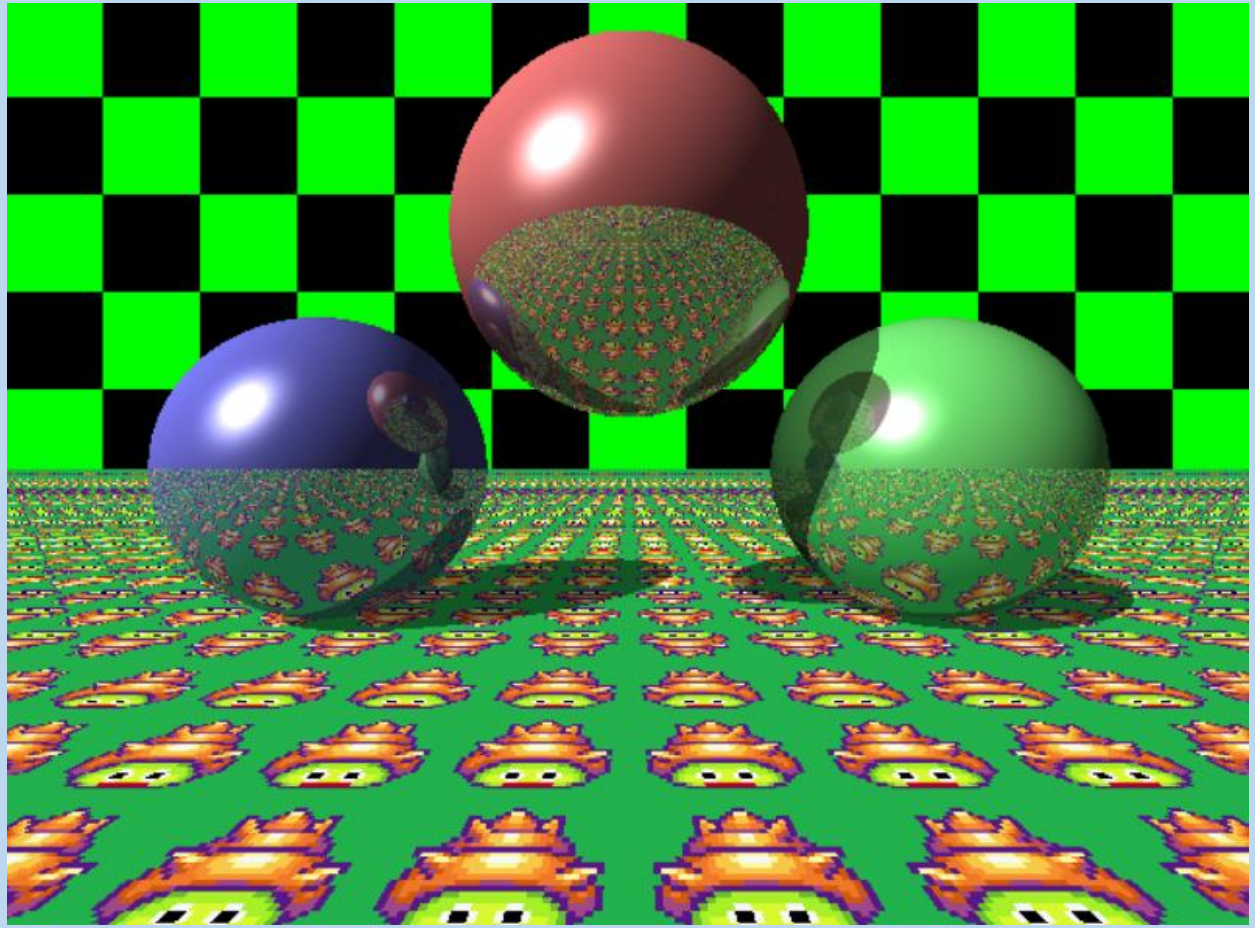
2年生の初期に、3Dのゲーム制作の授業を受けました。DxLibライブラリにおける、3D画面上のカメラやオブジェクトの基本的な扱い方を学びました。
プレイヤーの地面に対する当たり判定を正確に行うために、ベクトルと3Dオブジェクトとの当たり判定を使用するなど、見た目の違和感をなるべく減らすことを意識しました。また、カメラをプレイヤーに追従させる機能も実装しています。プレイヤーが動くと同時にカメラを同じ方向に動かし、またキー入力でカメラのアングルを変えることもできるようにしました。

制作時期　　2年生前期

使用言語　　C++

使用ライブラリ　DxLib

前回に制作した3Dゲームをさらに改良して、HLSL言語を用いた様々なレンダリングを実装しました。
今まではDxLibライブラリが自動で行ってくれていたのですが、それらをあえて無くし、独自のシェーディングを行っています。
メタリックやリムライト、水面の揺らぎを独自でシェーディングしています。またステージ全体の陰影処理を実装しました。
被写界深度など、授業だけでは理解が難しかったものもあり、オリジナルゲームの制作をしながら理解を深めていきました。



制作時期	2年生後期
使用言語	C++
使用ライブラリ	DxLib

HLSL言語によるシェーダープログラムを使用せずに、レイトレーシングをc++側で実装するという授業でした。3D上の情報を元に2D画面で交差判定を取り、球を描画しています。球の陰影やスペキュラーはもちろん、床の鏡面反射もc++側で行っています。

数学の知識が多く必要で、交差判定の数学的な理屈を理解することにとても苦労をしました。

この授業以降では、より深い理解で3Dモデルのレンダリングを行うことができるようになりました。

ありがとうございました