

PROGRAM EXPLANATION

プログラム説明資料

Action3D

自作フレームワークで動く 3D アクション

— 技術的な工夫点 —

制作期間 2026年6月 -(開発中)
制作形態 個人開発
開発環境 C++ / DirectX 11 / HLSL
版 2026年10月13日時点のソース

菊池 凌斗

Ryoto Kikuchi / HAL東京 ゲーム学科



開発情報 — 期間・体制・環境・制作意図

開発期間	2026年6月～(開発中)。約4.5ヶ月(2026年10月13日時点)
所要時間(概算)	約400時間
開発メンバー	1名(個人開発)
自身の担当範囲	企画・設計・調整・動作確認のすべて。コードの記述は主に AI(Claude Code・Codex)に依頼した(次ページに詳細)
制作意図	「最高に気持ちいいコンボアクション」を作りたいかった。そのため、キャンセル(動きの繋がり)・先行入力・ジャスト回避・ヒットストップなど、手触りに関わる部分を作り込んだ
動作環境	Windows 11 / Intel Core i7 / GeForce RTX 3050 Laptop GPU(VRAM 6GB、開発機で動作確認)。DirectX 11 対応の GPU が必要。ビルドは Visual Studio 2022(x64)
使用技術	C++ / DirectX 11 / HLSL。XAudio2・X3DAudio・Media Foundation。DirectXTex・cgltf・Dear ImGui・SDL3(ゲームパッド入力、静的リンク)・stb_truetype。既存のゲームエンジンは使用せず、フレームワークから自作

操作方法と、AI・外部素材の利用

— 操作方法

移動	W A S D / パッド: 左スティック
視点	マウス / パッド: 右スティック
弱攻撃	マウス左クリック / パッド: 左ボタン
強攻撃	マウス右クリック / パッド: 上ボタン
ジャンプ	Space / パッド: 下ボタン
ダッシュ・回避	Shift(押しっぱなしでダッシュ)、Ctrl(回避) / パッド: R2(押しっぱなしでダッシュ、押して回避)
ロックオン	マウスのホイールクリック / パッド: L2
技(4スロット)	E・Q・R・F / パッド: L1 を押しながら 上・左・右・下ボタン (スロットに入る技は装備で変わる)
触手スイング	C(押しっぱなし) / パッド: R1
触手で飛び移る	V / パッド: L1 + R1
ポーズ	Esc / パッド: Start

— AI の利用について

コード: 設計・仕様の決定・調整・動作確認は私が行い、コードの記述は主に **AI(Claude Code・Codex)** に依頼した。仕様を指示し、動作を確認したうえで採用している。入力まわり(input_manager・game_controller)は私が実装した。

アセット(3D モデル・テクスチャ): 画像生成 AI で元絵を作り、Tripo AI で 3D 化し、Mixamo でリグ・アニメーションを付け、Blender で調整した。

音声(BGM・SE): CC0(著作権放棄)の素材、または AI が生成した音源を使用している。

フォント: BIZ UDPGothic(SIL Open Font License)。

ライブラリ: 左のページの「使用技術」のとおり。ライセンス文は `source/licenses` に同梱している。

作品概要と、コードの構成

— 作品概要

DirectX 11 の API を直接使い、描画・物理・アセット管理・オーディオまでを自作したフレームワークと、その上で動く3Dアクションゲーム。

ゲーム側は、剣と「虚装」(体から伸びる影の武器)・各種の技で、敵の群れとボスと戦う。

— 開発環境

言語 / API	C++ / DirectX 11(HLSL)
ライブラリ	DirectXTex / cgltf / Dear ImGui / SDL3 / stb_truetype
音声	XAudio2 / X3DAudio / Media Foundation
期間・形態	2026/6 -(開発中)・個人開発
私の役割	設計・調整・動作確認(コードの記述は主に AI を利用)

— フレームワーク層 `src/engine/`

<code>engine/asset</code>	アセット管理・バイク(pak / DDS / PCM / MDL)
<code>engine/audio</code>	XAudio2・X3DAudio による再生と 3D 音響
<code>engine/component</code>	GameObject と Component の土台
<code>engine/core</code>	シーン・オブジェクト管理、時間
<code>engine/debug</code>	デバッグ描画・計測(Profiler)
<code>engine/feedback</code>	ヒットストップ・カメラ演出・画面効果
<code>engine/graphics</code>	DirectX 11 描画・glTF・ボーン物理・文字描画
<code>engine/input</code>	キーボード・マウス・ゲームパッド
<code>engine/physics</code>	衝突判定(形状・レイヤー・区画グリッド・三角形メッシュ)

— ゲーム層 `src/game/`

<code>game/actors</code>	プレイヤー(状態・入力・被弾)・敵・弾・斬撃
<code>game/weapon</code>	武器と技(剣・虚装・各スキル)
<code>game/effects</code>	爆発・虚装(触手)・パーティクル
<code>game/world</code>	街(ビル・道路)・地形・木・草・空
<code>game/ui</code>	HP バー・スコア・ウェーブ表示・ポーズ
<code>game/app</code>	ゲーム進行・チュートリアルほか

フレームワーク層は他のゲームでも使える部品、ゲーム層はこの作品専用のコード、という切り分け。次ページ以降は、それぞれの層で工夫した点を示す。

ゲームの中身 — 街を駆けて戦う 3D アクション

— ゲームの流れ

序章(最初の約10分): 部隊の一員として、剣だけで敵の波と戦う → 戦況が悪化し、避けられない一撃で瀕死になる → 覚醒して触手(虚装)が使えるようになる → 大型の敵を倒す → 触手の練習 → 最初の市民の救出 → 街へ。

操作の案内は、実際にその操作をするまで次へ進まない(`PrologueTutorial`)。

街(オープンワールド): 序章のあとは、ビルが立ち並ぶ街を自由に動き回って戦う。

— 戦い方

- 剣の弱・強攻撃、ロックオン、ダッシュ、回避。回避の無敵の間に攻撃を受けるとジャスト回避になり、反撃の猶予とスローが入る。
- 触手(虚装)の技が6種(雷・一網打尽・急接近・串刺し・落雷・火球)。装備できるのは 4 枠。
- 触手のスイングと飛び移りで、ビルの間を移動できる。

— 街で起きること

- 敵の群れ: 交差点・公園・広場に群れが置かれ、近づいた所だけ出る(`EncounterDirector`)。群れは最初は気付いておらず、近づくか攻撃すると、仲間も一斉に気付く。離れると片付けられ、全滅させた所は時間が経つと戻る。
- 襲撃: 一定時間ごとに、離れた場所で敵の群れが市民を襲う(`IncidentDirector`)。敵を全滅させれば救出成功、市民が全員倒れれば失敗。同時に起きるのは 1 件だけ。

— 敵

- 役割は射撃・近接・迫撃の 3 種に、エリートとボス。
- 全員が同時に襲ってこないよう、同時に攻撃できる数に上限を決めている(最大 3 体。近接 2・射撃 2・迫撃 1)。

アセットの追加を「1行」で済ませる — X-Macro

課題

テクスチャやモデルを1つ足すたびに、enum・パスの配列・キャッシュの3か所を同じ順番で直す必要がある。順番がずれると、別のアセットが読み込まれる。

やったこと

定義を `asset_def.h` の1か所に集約した①。TEXTURE(ID, パス) を1行書くだけで、マクロを差し替えて展開し、enum②とパスの配列③を同じリストから作る(X-Macro)。

`AssetManager` は ID から取得するだけの窓口。一度作ったテクスチャはキャッシュし、起動時に `PreloadAll` でまとめて読む。

ポイント

- Release ではバイク済みの `.dds` を読む(デコードもミップ生成も無し)④。Debug は生の png / hdr を読む。
- 1行の追加で済むので、アセットの追加時にバグが入り込まない。

```
asset_def.h
15 #define TEXTURE_LIST \
16     TEXTURE(FIELD,      "assets/textures/environment/field.png") \
17     TEXTURE(FIELD_NORMAL, "assets/textures/environment/field_normal.png") \
18     :
32 #define TEXTURE(id, path) id,
33 enum class TextureID {
34     TEXTURE_LIST
35     MAX
36 };

asset_def.h L15-L17 / L32-L36

asset_manager.cpp
21 #define TEXTURE(id, path) path,
22 static const char* g_texturePaths[static_cast<int>(TextureID::MAX)] = {
23     TEXTURE_LIST
24 };
25 #undef TEXTURE
26 :

171 //--- Release: バイク済み .dds を読む (デコード・ミップ生成なし) ---
172 std::string path = origPath;
173 size_t dot = path.find_last_of('.');
174 if (dot != std::string::npos) path = path.substr(0, dot) + ".dds";
175

asset_manager.cpp L21-L25 / L171-L175
```

起動を速くする — 自作のアセットベイクと VFS

— 課題

起動のたびに、MP3・PNG のデコード、ミップ生成、GLB のパースが走り、起動が遅い。

— やったこと

ゲーム起動時、`data.pak` が無い・古いときだけベイクを実行し、1つの pak にまとめる。

- MP3 はPCM にデコードしてから格納する③。
- PNG / HDR はミップ付きの DDS にする④(非 HDR・非法線は BC7 圧縮)。
- GLB はベイク済みの .mdl にする。頂点・ボーン・アニメーションのキーを、GPU へコピーするだけの形へ事前変換しておく。テクスチャは BC7 / BC5 圧縮。

— ポイント

- 作り直すかは、pak のフォーマット版数①と、ソースの更新時刻②で判定する。ベイク処理を変えたら版数を上げるだけで、次回起動で自動的に作り直される。
- pak の先頭は、作っている間は `"PAK?"` にしておき⑤、書き終えてから `"PACK"` に戻す⑥。途中で落ちて壊れた pak を読ませない。

```
asset_baker.cpp
63     if (n != 1 || memcmp(h.magic, "PACK", 4) != 0 || h.version != PackFormat::VERSION) { ①
64         return true;
65     }
    :
77     if (entry.last_write_time() > pakTime) return true; ②
    :
591 memcpy(header.magic, "PAK?", 4); // 途中で落ちて壊れた pak を読ませない。完了時に "PACK" へ書き換⑤
    :
622     if (!DecodeMP3ToPCM(srcPath, fileData, is3d)) { ③
    :
636     if (!EncodeImageToDDS(srcPath, fileData)) { ④
    :
777 memcpy(header.magic, "PACK", 4); // ここまで来たら完成。有効なマジックにする ⑥
```

asset_baker.cpp L63-L65 / L77 / L591 / L622 / L636 / L777

速くなったかを数字で確かめる — 起動時間の計測

— 課題

「速くなった気がする」で終わらず、変更のたびに起動時間を計測して、効いたかどうかを数字で確かめたかった。

— やったこと

各処理に `ScopedTimer` を入れ、起動の内訳（アセット読み込み・音声・シーン初期化）を毎回ログに出す。変更の前後で、Release ビルドを同じ条件で2～3回測って平均した。

— わかったこと

- 音声のデコードが、起動の中でも重かった（約740ms）。
- ミップマップを起動時に作ると、逆に遅くなった。生成をバイクへ移すと、元の水準まで戻った。

Release ビルド / Total Startup Time の平均 / 実測(計測データ.xlsx)

① AssetManager を導入

導入前	1,827 ms
導入後	1,239 ms

② 音声を PCM にバイク(音声の読み込み時間)

バイク前	740 ms
バイク後	45 ms

同じ変更での起動時間全体

バイク前	1,866 ms
バイク後	1,309 ms

③ ミップマップ(起動時に生成 → バイクへ移す)

生成で +37% → バイクで -32%

生成前	2,000 ms
起動時に生成	2,738 ms
バイクへ移した後	1,865 ms

④ バイクの最適化(モデルの事前変換ほか)

最適化前	1,862 ms
最適化後	1,247 ms

ミップマップ生成を GPU へ逃がす

課題

ミップマップは CPU(DirectXTex の `GenerateMipMaps`) で作ると遅い。Debug ビルドでは、地面の 2880×2880 の 1 枚だけで約 1.2 秒かかり、起動全体では約 12.7 秒に膨らんだ。

やったこと

最上段の画像だけを GPU へ送り、残りのミップは GPU に作らせる(`GenerateMips`)ようにした。そのフォーマットが自動生成に対応するか先に調べ①、`RENDER_TARGET` と `GENERATE_MIPS` 付きでテクスチャを作る②。実行は 1 行③。

対応していないフォーマットだけ、これまでの CPU 版へ戻す④。

ポイント

- 呼び出し側は変えずに、取得の中で切り替えている。
- 計測は Debug ビルドのみ(CPU は 1 回、GPU は 2 回)。Release はバイク済みの DDS を読むため、この経路は主に開発中の待ち時間を縮める用途。

Debug ビルドで、生の png を読んだ時の実測(計測データ.xlsx)。各群の CPU を 100% とした長さ

地面テクスチャ(2880×2880)のミップ生成

CPU(1回)	1,237.1 ms
GPU(2回平均)	12.2 ms
	-99%

テクスチャ全体の読み込み

CPU(1回)	9,550 ms
GPU(2回平均)	1,068 ms
	-89%

起動時間の全体

CPU(1回)	12,700 ms
GPU(2回平均)	3,445 ms
	-73%

asset_manager.cpp

```
122 if (FAILED(device->CheckFormatSupport(top.format, &support))) return nullptr;
123 if (!(support & D3D11_FORMAT_SUPPORT_MIP_AUTOGEN)) return nullptr; ①
...
135 td.BindFlags = D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_RENDER_TARGET;
136 td.MiscFlags = D3D11_RESOURCE_MISC_GENERATE_MIPS; ②
...
149 context->GenerateMips(srv); ③
...
247 srv = CreateTextureWithGpuMips(*cur->GetImage(0, 0, 0), name);
248 if (srv) {
249     g_textureCache[idx] = srv;
250     return srv;
251 }
...
253 DirectX::ScratchImage mipChain; ④
254 {
255     ScopedTimer t(TexStep(name, "GenerateMipMaps (CPU)", meta).c_str());
```

asset_manager.cpp L122-L123 / L135-L136 / L149 / L247-L251 / L253-L255

衝突のルールをゲームから切り離す — レイヤーと関数表

— 課題

衝突判定のルールを、敵・弾・剣・地形などのゲームのロジックから切り離したい。「この敵はあの弾と当たらない」を、各クラスの中の if で書くと散らかる。

— やったこと

Unity の Physics Layer Matrix と同じ発想にした。GameObject は自分のレイヤーを1つ持つだけ。衝突するかどうかは、PhysicsWorld が持つレイヤー×レイヤーの2次元 bool 配列で決める②③。

初期値は「全部当たる」にして、例外だけを書く。設定は対称になるよう、片方を書けば両方に効く。

形の組み合わせ(球・箱・OBB・カプセル)は、4×4の関数表で振り分ける①。

— ポイント

- 動かないレイヤー(地形・草木)は動かさず、動くほうだけを押戻す。
- OBB×OBB は使う場面がないため未実装 (DispNotImpl)。必要になった時に足す方針。

```
physics_world.cpp

560 static constexpr int kTypeCount = 4; // Sphere, Box, OBB, Capsule
561 IntersectFn s_Table[kTypeCount][kTypeCount] = {
562     //          b=Sphere,    b=Box,      b=OBB,      b=Capsule
563     /* a=Sphere */ { DispSS,   DispSB,    DispSO,    DispSC },
564     /* a=Box      */ { DispBS,   DispBB,    DispBO,    DispBC },
565     /* a=OBB     */ { DispOS,   DispOB,    DispNotImpl, DispOC },
566     /* a=Capsule */ { DispCS,   DispCB,    DispCO,    DispCC },
567 };
568
569 for (int i = 0; i < LAYER_COUNT; ++i)
570     for (int j = 0; j < LAYER_COUNT; ++j)
571         m_LayerMatrix[i][j] = true;
572
573 SetLayerCollision(CollisionLayer::PlayerAttack, CollisionLayer::Player, false); // Player本体には当たらない
574 SetLayerCollision(CollisionLayer::PlayerAttack, CollisionLayer::PlayerAttack, false); // 複数剣同士も無効
575 SetLayerCollision(CollisionLayer::PlayerAttack, CollisionLayer::Terrain, false); // 地形は貫通
576
577 void PhysicsWorld::SetLayerCollision(CollisionLayer a, CollisionLayer b, bool enabled) {
578     int ai = static_cast<int>(a);
579     int bi = static_cast<int>(b);
580     m_LayerMatrix[ai][bi] = enabled;
581     m_LayerMatrix[bi][ai] = enabled; // 対称
582 }
```

physics_world.cpp L560-L566 / L609-L611 / L615-L617 / L631-L635

斜めの剣を判定する — OBB を「ローカル空間の AABB」に帰着

— 課題

剣の当たり判定は斜めに構える。AABB(軸に揃った箱)では、振りの向きに合わない余白が出て、当たっていないのに当たる。

— やったこと

向きを持てる OBB を入れた。相手の球やカプセルを、OBB のローカル座標へ移して①しまえば、OBB は「ただの AABB」になる。あとは、最近接点を箱の範囲へクランプするだけ②。

カプセルは、まず芯の線分上で箱の中心に近い点を取り③、箱へクランプし、もう一度線分へ射影する(2回の近似)。

結果は、ローカルの法線をワールドへ戻して返す。

— ポイント

- OBB×球 は「ローカルへ移して AABB×球 に帰着」させるだけなので、新しい判定式を作らずに済んだ。

```
physics_world.cpp

186 // OBBのローカル空間での球中心座標
187 float localX = d.x * ax.x + d.y * ax.y + d.z * ax.z;
188 float localY = d.x * ay.x + d.y * ay.y + d.z * ay.z;
189 float localZ = d.x * az.x + d.y * az.y + d.z * az.z;
190
191
192
193 float cx = std::max(-hs.x, std::min(localX, hs.x));
194 float cy = std::max(-hs.y, std::min(localY, hs.y));
195 float cz = std::max(-hs.z, std::min(localZ, hs.z));
196
197
198
359 Vector3 segPt = ClosestPointOnSegment(Vector3{ 0.0f, 0.0f, 0.0f }, a1, a2);
360 float cx = std::max(-hs.x, std::min(segPt.x, hs.x));
361 float cy = std::max(-hs.y, std::min(segPt.y, hs.y));
362 float cz = std::max(-hs.z, std::min(segPt.z, hs.z));
363 Vector3 boxPt = { cx, cy, cz };
364 segPt = ClosestPointOnSegment(boxPt, a1, a2);
```

physics_world.cpp L186-L189 / L193-L195 / L359-L364

ビルを「実際の面」で判定する — 三角形メッシュと BVH

— 課題

ビルを箱1つで囲むと、屋上の設備の上の空中や、段々の上の何もない所に足場ができてしまう。実際の面に立ち、壁に沿いたい。

— やったこと

ビルの当たり判定を、三角形だけを書き出した軽い形式（.colmesh、先頭 TCM1）にして持つ。読み込みでは大きさや有限値を検証し、描画と同じワールド行列で変換する。

三角形は BVH(包含箱の木)に入れる。最も長い軸で①中央値分割②し、葉は 8 枚以下。判定では、相手の形の箱と重なるノードだけをたどり③、残った三角形だけ計算する。球・カプセル・箱・OBB のどれも、最も深くめり込んだ面を返す。

屋上の角のように複数の面に触れる所では、押し出した位置から次の面を再評価する(最大 6 回)④。

— ポイント

- 描画 API に依存しない幾何クラス(TriangleMeshGeometry)にした。ゲームを起動せずに、総当たりの結果と比べるテストを書ける。

triangle_mesh_geometry.cpp

```
187     if(count<=TriangleMeshTuning::LEAF_TRIANGLES) return id;
188     const Vector3 ext=hi-lo; const int axis=ext.x>ext.y && ext.x>ext.z ? 0 : ext.y>ext.z ? 1 :
189     const uint32_t mid=first+count/2;
190     std::nth_element(m_Order.begin()+first,m_Order.begin()+mid,m_Order.begin()+first+count,
191     [&](uint32_t a,uint32_t b) {
192         const Triangle& ta=m_Triangles[a]; const Triangle& tb=m_Triangles[b];
193         return Axis(ta.a+ta.b+ta.c,axis)<Axis(tb.a+tb.b+tb.c,axis);
194     });
195     const int left=BuildNode(first,mid-first), right=BuildNode(mid,first+count-mid);
196     m_Nodes[id].left=left; m_Nodes[id].right=right; m_Nodes[id].count=0;
197
200 void TriangleMeshGeometry::Query(const Vector3& lo,const Vector3& hi,std::vector<uint32_t>& resul
201 if(m_Nodes.empty()) return;
202 std::vector<int> stack{0};
203 while(!stack.empty()) {
204     const Node& n=m_Nodes[stack.back()]; stack.pop_back();
205     if(!Overlap(lo,hi,n.min,n.max)) continue;
206     if(n.count) for(uint32_t i=n.first;i<n.first+n.count;++i) result.push_back(m_Order[i]);
207     else { stack.push_back(n.left); stack.push_back(n.right); }
208 }
```

triangle_mesh_geometry.cpp L187-L196 / L200-L208

physics_world.cpp

```
814 const bool meshContact = colA->GetType() == ColliderType::TriangleMesh
815 || colB->GetType() == ColliderType::TriangleMesh;
816 const int resolvePasses = meshContact ? TriangleMeshTuning::RESOLVE_PASSES : 1;
817 for (int pass = 0; pass < resolvePasses; ++pass) {
818     CollisionInfo infoA;
819     if (!Intersect(colA, colB, infoA)) break;
```

physics_world.cpp L814-L819

ブロードフェーズで絞り込む — 区画グリッドとナローフェーズ

— 起きたこと

それまでは全コライダーの組み合わせを調べる単純な作りで足りていた。ところが、ビルを三角形メッシュにすると 1 組あたりの判定が重くなり、全体が重くなり、30FPS まで落ちた。

— やったこと

判定をブロードフェーズ(候補の絞り込み)とナローフェーズ(正確な判定)に分け、メッシュの重い計算を当たりそうな組にだけ使う。

ブロードフェーズ

- 動かないもの同士(ビル・地面・木)は調べない。対象は動くものが絡む組だけ①。
- 動かないものは 32m の区画グリッドに登録しておき②、動くものの囲む箱(AABB)が重なる区画のものだけ取り出す③。
- さらに、囲む箱どうしが重なる組だけを残す④。

ナローフェーズ: 残った組だけ、形ごとの判定(メッシュなら BVH で面まで)を行い、押し戻す。

— ポイント

- 多くの区画にまたがる大きなものは、区画に入れず常に候補にする(取りこぼし防止)。動くもの同士は 1 回だけ調べる。

結果: 30FPS → 60FPS (開発機での確認。ms 単位の前後比較は未計測)

physics_world.cpp

```
666 constexpr float GRID_CELL = 32.0f;           // 区画の一辺 (m)。ビル1棟が1~4区画に収まる大きさ
667 constexpr int   GRID_LARGE_CELLS = 64;        // これより多くの区画にまたがるものは、区画に入れず常に調
        :
753     for (int ix = x0; ix <= x1; ++ix)
754         for (int iz = z0; iz <= z1; ++iz)
755             m_Grid[CellKey(ix, iz)].push_back(index);           ②
        :
798     if (m_Colliders[i] && !IsStaticLayer(m_Colliders[i]->GetLayer())) dynamicIndices.push_back(①i);
```

physics_world.cpp L666-L667 / L753-L755 / L798

physics_world.cpp

```
886     amn = { amn.x - BROAD_MARGIN, amn.y - BROAD_MARGIN, amn.z - BROAD_MARGIN };
887     amx = { amx.x + BROAD_MARGIN, amx.y + BROAD_MARGIN, amx.z + BROAD_MARGIN };
888     GatherStatics(amn, amx, nearby);           ③
889 } else {
890     for (int k = 0; k < (int)m_Statics.size(); ++k) nearby.push_back(k);
891 }
892 for (const int k : nearby) {
893     const StaticEntry& e = m_Statics[k];
894     if (!e.col) continue;
895     if (hasBounds && !Overlaps(amn, amx, e.min, e.max)) continue;           ④
896     resolvePair(colA, objA, e.col);
```

physics_world.cpp L886-L896

速い移動でもすり抜けない — 球の掃引と滑り

課題

触手のスイングや飛び移りは速い。1フレームの移動がビルや外周の壁の厚みを越えると、すり抜けてしまう。

やったこと

体(カプセル)を上下2つの球で近似し、進む先を掃引して調べる①。ぶつかったら面の手前で止め②、残りの移動と速度から面へ入る向きの成分を除いて③④、面に沿って滑らせる(最大 3 回)。上向きの面で止まったら着地として扱う。

`SweepSphere` は、太さぶん膨らませた箱へのレイで最初の当たりを求める。三角形メッシュには BVH の球の掃引を使う。始点がもう箱の中なら、一番浅い面の向きを見て、その面へさらに入る向きの時だけ距離 0 で当たりにする⑤。離れる・沿う向きは無視する。

ポイント

- 遅い移動(1フレーム 0.2m 未満)は掃引せず、物理の押し戻しに任せる。
- 動かないものは、線分の近くの区画だけを調べる(前ページのグリッドを再利用)。

player_motor.cpp

```
615 bool found = PhysicsWorld::Instance().SweepSphere(base + Vector3(0.0f, LOW_Y, 0.0f), dir, RAD1①,
616 if (PhysicsWorld::Instance().SweepSphere(base + Vector3(0.0f, HIGH_Y, 0.0f), dir, RADIUS, len, ma
    :
626 const float advance = (hit.distance - SKIN > 0.0f) ? hit.distance - SKIN : 0.0f; ②
627 total += dir * advance;
    :
635 const float vn = Vector3::dot(p.m_Motor.m_Velocity, n); ③
636 if (vn < 0.0f)
637     p.m_Motor.m_Velocity -= n * vn;
    :
641 const float rn = Vector3::dot(remaining, n); ④
642 if (rn < 0.0f)
643     remaining -= n * rn;
```

player_motor.cpp L615-L616 / L626-L627 / L635-L637 / L641-L643

physics_world.cpp

```
1193 const bool inside = origin.x > mn.x && origin.x < mx.x ⑤
1194                     && origin.y > mn.y && origin.y < mx.y
1195                     && origin.z > mn.z && origin.z < mx.z;
    :
1203     for (int i = 1; i < 6; ++i) if (pen[i] < pen[k]) k = i;
1204     if (Vector3::dot(d, nrm[k]) >= 0.0f) return; // 離れる・沿う向き
1205     t = 0.0f;
1206     n = nrm[k];
```

physics_world.cpp L1193-L1195 / L1203-L1206

ボスが天高く飛んだ — 自分自身との衝突

— 起きたこと

ボスの当たり判定を実寸に合わせて、1つの **GameObject** に2つのコライダー(下半身の Box と頭の Sphere)を持たせた。すると、ボスが天高く飛んでいった。

— 原因

衝突判定は、**GameObject** のペアではなくコライダーのペアで行っている。ボスの Box と Sphere は、設計上めり込んだ配置になるので、同じボスのコライダー同士が衝突と判定された。

押し戻しは「両方動く場合は半分ずつ」。同じオブジェクトに2回かかって互いに上書きし合い、位置が毎フレーム上へずれ続けた。

— 直したこと

ペアの先頭で、同じ **GameObject** 同士は飛ばす^①。1オブジェクト1コライダーを前提にしていた見落としだった。

— なぜ2つに分けたか

下から突進されるとボスが浮いた。単一のカプセルは下端が丸く、低い位置への接触で押し戻しが斜め上に向いたため。下半身を平らな Box にして解消した。

enemy.cpp

```
158 auto* body = AddComponent<BoxCollider>();
159 body->SetSize({ bodyHalfWidth * 2.0f, bodyHeight, bodyHalfWidth * 2.0f });
160 body->SetOffset({ 0.0f, bodyHeight * 0.5f, 0.0f });
161 body->SetTag(CollisionTag::Enemy);
162 body->SetLayer(CollisionLayer::Enemy);
163
164 auto* head = AddComponent<SphereCollider>();
165 head->SetRadius(headRadius);
166 head->SetOffset({ 0.0f, bodyHeight, 0.0f }); // 肩(Boxの天井)に乗せる
167 head->SetTag(CollisionTag::Enemy);
168 head->SetLayer(CollisionLayer::Enemy);
```

enemy.cpp L158-L168

physics_world.cpp

```
802 auto resolvePair = [&](Collider* colA, GameObject* objA, Collider* colB) {
803     GameObject* objB = colB->GetGameObject();
804     if (!objB || objB->IsDestroyed()) return;
805
806     // 複数のColliderを持つGameObject同士で、同じGameObject同士の衝突は無視する
807     if (objA == objB) return;
808
809     // Layerフィルタ
810     if (!CanCollide(colA, colB)) return;
```

physics_world.cpp L802-L810

動きの繋がりを State ごとの表にする — キャンセルと先行入力

— 課題

手触りは「どの動きから、どの動きへ、いつから移れるか」で決まる。if で散らすと、調整のたびに探し回ることになる。

— やったこと

プレイヤーの状態を **State** パターンにした。状態ごとに `Enter / Update / Exit` と、キャンセルの受付時間①を 1 クラスにまとめる。状態の変更は `Change` だけを通す②。

今の状態	ジャンプ	回避	攻撃	強攻撃
地上・空中	0.00	0.00	0.00	0.00
回避中	0.10	0.25	0.10	0.10
攻撃中(空振り)	0.28	0.28	0.28	0.28
被弾中	—	0.20	—	—
飛びつき中	—	0.15	—	—

移れるまでの秒数(各 State の `CancelRule()`)。— は移れない。

攻撃が命中すると、0.04秒後から繋がられる③。早く押された入力は0.26秒保持し④、受付が開いた瞬間に出す⑤。

```
player_state_machine.cpp
84     PlayerCancelRule CancelRule() const override {
85         return {Player::DODGE_CANCEL_EARLY, Player::DODGE_CANCEL_DODGE, Player::DODGE_CANCEL_EARLY};
86         Player::DODGE_CANCEL_EARLY};
87     }

291 void PlayerStateMachine::Change(Player &p, PlayerStateId id, PlayerStateEntry entry) {
292     const auto index = static_cast<size_t>(id);
293     if (index >= m_States.size() || !m_States[index])
294         return;
295     if (id != m_Id)
296         m_States[static_cast<size_t>(m_Id)]->Exit(p, id);
```

player_state_machine.cpp L84-L87 / L291-L296

```
player_action_controller.cpp
21 if (p.m_StateMachine.Id() == PlayerStateId::Attack && p.m_Combat.m_AttackSession.HasHit() &&
22     p.m_StateMachine.Elapsed() >= Player::HIT_CONFIRM_CANCEL_DELAY &&
23     (next == PlayerActionKind::Attack || next == PlayerActionKind::Heavy || next == PlayerActionKind::
24         return true;
25     }

47 if (acceptTime < 0.0f)
48     return false; // Player::NEVER
49 return p.m_StateMachine.Elapsed() >= acceptTime;

56 p.m_Actions.m_BufferTimer -= dt;
57 if (p.m_Actions.m_BufferTimer <= 0.0f) { // 古くなった入力は捨てる
58     p.m_Actions.m_BufferedAction = PlayerActionKind::None;

62 if (!p.CanCancelInto(p.m_Actions.m_BufferedAction))
63     return; // まだ受付時間外。次フレームで再挑戦
```

player_action_controller.cpp L21-L25 / L47-L49 / L56-L58 / L62-L63

回避にリスクと見返りを — ジャスト回避

— 課題

回避の間ずっと無敵だと、連打すれば永久に無敵になる。回避にリスクも見返りもなくなってしまう。

— やったこと

無敵は、回避の出だしの0.35秒だけにした①（回避全体は0.5秒）。無敵かどうかは回避の経過時間を見る②。その間に当たったら「ジャスト回避」とする③。

成功すると、反撃の猶予が1.2秒できて与ダメージが3倍（通常は1）になる。同時に、時間が0.45秒だけ0.25倍になり④、画面の白フラッシュと専用の演出を出す。

— ポイント

- 判定を先読みしない。TakeDamage に届いた時点で「本当に当たっていた」ことが保証されているので、その瞬間に無敵時間中かを見るだけでよい。
- 避けた直後は 0.6 秒の無敵を足す。同じ攻撃の続き（連射・残った判定）を食らわず、ジャスト回避が連続で何度も出ることもない。
- スローは、ヒットストップと同じ仕組み（時間の進み方を変える）を使い回している。入力の実時間なので、自分だけ速く動ける感覚になる。

player_tuning.h

```
69 static constexpr float DODGE_IFRAME = 0.35f; // 回避の無敵時間（回避開始から） ①
72 static constexpr float SLOWMO_TIME = 0.45f; // スロー演出の長さ
73 static constexpr float SLOWMO_SCALE = 0.25f; // スロー中の時間の進み方
76 static constexpr float PERFECT_DODGE_IFRAME = 0.6f;
```

player_tuning.h L69 / L72-L73 / L76

player.h

```
201 bool IsDodgeInvincible() const {
202     return m_StateMachine.Id() == PlayerState::Dodge && m_StateMachine.Elapsed() <= DODGE_IFRAME; ②;
203 }
```

player.h L201-L203

player_vitals.cpp

```
38 if (!unavoidable && p.IsDodgeInvincible()) { ③
39     p.OnPerfectDodge();
40     return;
41 }
42 // 避けた直後に同じ攻撃の続きを食らわないよう、しばらく無敵にする。
43 // 無敵中の攻撃は TakeDamage の入口で弾かれるので、ジャスト回避が連続で何度も出ることもない
44 if (p.m_Vitals.m_InvincibleTimer < Player::PERFECT_DODGE_IFRAME)
45     p.m_Vitals.m_InvincibleTimer = Player::PERFECT_DODGE_IFRAME;
46 // 世界だけ遅くする。プレイヤーの入力は実時間なので「自分だけ速い」感覚になる
47 HitStop::Trigger(Player::SLOWMO_TIME, Player::SLOWMO_SCALE); ④
```

player_vitals.cpp L38-L41 / L114-L120

技を足しても本体を触らない — ISkill と SkillFactory

課題

技(雷・引き寄せ・飛びつき・串刺し・火球など)が増えるほど、発動条件や更新の順序が `Player` の中に混ざり込み、1つ足すたびに全体を触ることになる。

やったこと

技ごとに `ISkill` を実装する(Strategy)。Cast / Update / Cancel / IsActive とクールタイムを共通の入口にした①。生成は `SkillFactory` の 1 か所に集約し②、呼び出し側は具体クラスを知らない。

`SkillController` が、入力のスロットを装備中の技の ID に解決して振り分け③、更新の順序④・中断・クールタイム・発動できない条件(覚醒中・串刺し後のロック)を一括で扱う。

ポイント

- 技どうしが共有するのは、時計と串刺し後のロックだけ (`SkillEnvironment`)。進行状態は各技が持つ。
- 触手の描画は、各技が `AppendTendrils` で足したものをまとめて、1 回だけ虚装の描画側へ渡す。

arc_skill.h

```
29 virtual void Cast(Player *player) = 0;
30 virtual void Update(float, Player *, bool) {
    ...
36 virtual bool IsActive() const {
37     return false;
```

①

arc_skill.h L29-L30 / L36-L37

skill_factory.cpp

```
18 std::unique_ptr<ISkill> SkillFactory::Create(SkillId id, SkillEnvironment &env) {
19     switch (id) {
20     case SkillId::Lightning:
21         return std::make_unique<LightningSkill>(env);
22     case SkillId::Gather:
```

②

skill_factory.cpp L18-L22

skill_controller.cpp

```
26 ISkill *SkillController::Slot(int slot) const {
27     return slot >= 0 && slot < SLOT_COUNT ? Find(PlayerLoadout::Skills()[slot]) : nullptr;
28 }
38 if (player->GetHP() <= 0 || player->IsAwakening())
39     return;
40 if (id != SkillId::Rush && m_Environment.lock > 0.0f)
41     return;
75 // 元の順序を維持する。火球の離し判定→引き寄せ→急接近→串刺し。
76 for (SkillId id : {SkillId::Fireball, SkillId::Gather, SkillId::Rush, SkillId::Impale})
77     Find(id)->Update(dt, player, id == SkillId::Fireball && fireHeld);
```

④

skill_controller.cpp L26-L28 / L38-L41 / L75-L77

予告と発生を同じ時計で動かす — 攻撃の時間軸

課題

敵の攻撃は「予告 → 発生 → 追撃」の順に進む。予告の表示と、実際に当たる瞬間の時刻がずれると、ジャスト回避が運になる。しかも、処理落ちで 1 フレームが長くなると、発生の瞬間を飛び越えてしまう。

やったこと

攻撃を時間軸のデータにした (`EnemyAttackPlan`)。予告の長さ・発生する時刻・全体の長さを、技の種類ごとに 1 か所で決める^①。進行は `EnemyAttackClock` が持つ。

`Advance(dt)` は、この更新で跨いだ発生をビットで返し、同じ発生は 1 回だけにする^②。dt が大きくても取りこぼさず、二重にも出ない。予告の進み具合 (`WarningProgress`) も同じ時計から出すので、見た目と判定がずれない。

突進は、有効な区間の秒数だけを積分して動かし^③、動いた区間を球で掃引して当たりを調べる。通り抜けや高低差による空振りを防ぐ。

ポイント

- 画面や物理に依存しない純粋なヘッダなので、ゲームを起動せずにテストできる (大きな dt・中断・順序など)。

enemy_attack_clock.h

```
36 case EnemyMove::BossDash: {
37     const bool boss=move == EnemyMove::BossDash;
38     :
43     p.events={ p.startup, p.startup+p.dashDuration,
44               p.startup+p.dashDuration+(boss ? BOSS_FOLLOWUP_DELAY : DASH_FOLLOWUP_DELAY) };
45     p.eventCount=3; p.duration=p.events[2]+PUNCH_ACTIVE; break;
46 }
```

enemy_attack_clock.h L36-L37 / L43-L46

enemy_attack_clock.h

```
66 unsigned Advance(float dt) {
67     if (m_Canceled) return 0;
68     m_Previous=m_Age; m_Age+=(std::max)(0.0f, dt);
69     unsigned due=0;
70     for (int i=0; i<m_Plan.eventCount; ++i) {
71         const unsigned bit=1u << i;
72         if (!(m_Fired & bit) && m_Age >= m_Plan.events[i]) { due|=bit; m_Fired|=bit; }
73     }
74     return due;
75 }
```

enemy_attack_clock.h L66-L75

enemy_tactics.cpp

```
188 const float slice=m_AttackClock.DashSeconds();
189 if (slice > 0.0f) {
```

enemy_tactics.cpp L188-L189

髪とコートを揺らす — 骨の名前で動く Verlet の鎖

— 課題

キャラの髪・コート・リボンが動かないと、アニメーションが固く見える。かといって、ボーンごとに手でアニメを付けるのは現実的でない。

— やったこと

モデルの骨のうち、名前の接頭辞(`Hair_`・`Coat_`・`Skirt_`・`Ribbon_` など)に合うものを、粒子の鎖として物理で動かす^①。重力・抵抗・風・元の形へ戻る力の強さは、接頭辞ごとの表にした。モデル側で骨に名前を付けるだけで、コードは変えない。

積分は **Verlet**^②。長さや当たりで位置を直接直しても速度が破綻しにくく、可変フレームでも安定する。1 フレームは最大 1/120 秒ずつに分割し、処理落ちで暴れないよう 1/20 秒で打ち切る^③。

拘束は 3 回繰り返す。布のつながり → 体(カプセル)との当たり → 元の形から離れすぎない → 骨の長さの順。骨の長さは根元から順に子だけを動かすので、根元の動きが遅れて先へ伝わり、しなりになる^④。

— ポイント

- 結果は骨の回転として書き戻す。描画側はふつうのスキニングのまま。

bone_physics.cpp

```
31 constexpr float SUBSTEP = 1.0f / 120.0f; // 1ステップの最大時間(これより長いフレームは分割) ③
32 constexpr int MAX_SUBSTEPS = 4;
33 constexpr float MAX_DT = 1.0f / 20.0f; // 処理落ちで暴れないよう、これ以上は進めない
34 constexpr int ITERATIONS = 3; // 拘束(長さ・布・当たり)を解く回数
...
41 { "Hair_Bangs_", 1.0f, 3.2f, 12.0f, 3.0f, 0.5f, 0.025f, 0.25f }, ①
42 // ポニーテール: 根元は頭に付いて回り、先は重力と慣性で振れる
43 { "Hair_", 1.0f, 2.4f, 8.0f, 0.8f, 1.2f, 0.05f, 0.6f },
```

bone_physics.cpp L31-L34 / L41-L43

bone_physics.cpp

```
515 const XMVECTOR vel = XMVectorScale(XMVectorSubtract(pos, L(ch.prev[k])), keep);
516 ch.prev[k] = ch.pos[k];
517 XMVECTOR next = XMVectorAdd(XMVectorAdd(pos, vel), XMVectorScale(accel, h * h)); ②
...
585 for (size_t k = 1; k < n; ++k) {
586     const XMVECTOR parent = L(ch.pos[k - 1]);
587     const XMVECTOR d = XMVectorSubtract(L(ch.pos[k]), parent);
588     const float len = XMVectorGetX(XMVector3Length(d));
589     if (len < 1e-6f) {
590         ch.pos[k] = ch.target[k];
591         continue;
592     }
593     ch.pos[k] = S(XMVectorAdd(parent, XMVectorScale(d, ch.len[k] / len))); ④
```

bone_physics.cpp L515-L517 / L585-L593

複数リグの GLB から、動くリグを選ぶ

課題

Mixamo などで作ったモデルの GLB には、リグ(骨格)が複数入っていることがある。先頭のリグが「バックアップ用」で、実際にアニメするのは別のリグだと、動かないメッシュやずれた骨になる。

さらに、骨の名前が Mixamo 流(`mixamorig:Spine`)とアバター流(`spine`)で違う。

やったこと

GPU が扱える骨格は 1 つなので、`SelectModelSkin` で「どれを使うか」を決める。アニメーションの対象になっている関節が最も多いリグを選び①、同数ならメッシュの頂点数が多いほう②。先頭にあるからという理由では選ばない。

名前の違いは、骨を改名せず呼び出し側の名前だけを解決する③。完全一致を先に探し、見つからなければ対応表で別名を引く④。

ポイント

- メッシュ・骨・アニメーション・バイクが同じ選択関数を使う。
どこかだけ別のリグを見て食い違う、ということが起きない。
- この修正でバイクの版数を上げ、古い .mdl は
自動で作り直される。

gltf_skin_selection.h

```
19 std::unordered_set<const cglTF_node*> animated;
20 for (size_t a = 0; a < data->animations_count; ++a) {
21     const cglTF_animation& animation = data->animations[a];
22     for (size_t c = 0; c < animation.channels_count; ++c) {
23         if (animation.channels[c].target_node) animated.insert(animation.channels[c].target_node)
24     }
25 }
26
46 size_t active = 0;
47 for (size_t j = 0; j < skin->joints_count; ++j) active += animated.count(skin->joints[j]); ①
48 if (active > bestAnimated || (active == bestAnimated && vertices > bestVertices)) { ②
49     selected = skin;
50     bestAnimated = active;
51     bestVertices = vertices;
52 }
```

gltf_skin_selection.h L19-L25 / L46-L52

rig_bone_names.h

```
15 inline constexpr Alias ALIASES[] = {
16     { "mixamorig:Hips",      "hips" },
17     { "mixamorig:Spine",    "spine" },
18 }
19
38 inline bool Matches(const std::string& actual, const std::string& requested) {
39     if (actual == requested) return true; ③
40     for (const Alias& alias : ALIASES) { ④
41         if ((requested == alias.mixamo || requested == alias.avatar)
42             && (actual == alias.mixamo || actual == alias.avatar)) return true;
43     }
44     return false;
45 }
```

rig_bone_names.h L15-L17 / L38-L44

モデルを使わず、曲線から武器を組み立てる — 虚装

— 課題

「体から伸びる影の武器(虚装)」は形が決まっていない。固定のメッシュだと、動きの表現に合わない。

— やったこと

モデルを持たず、毎フレーム頂点を組み立てて描く。振り・溜め・突きは、2次ベジエ曲線の制御点を動かすだけで表す①。
物理シミュレーションは持たない。

曲線の各点に断面の座標系を運び②、平たい楕円の断面を掃引してチューブにする。カメラ正対の板にすると、どこから見ても厚みが無くなるため。

— ポイント

- 長さ方向に等間隔の瘤を立てて節を作る③。
- 周期の違う正弦を掛け合わせた痙攣を重ねる④。単純な正弦1本だと、きれいに振動しているだけに見えるため。
- シェーダーはライティング無し(UNLIT)なので、陰影を頂点カラーに焼き込む(ランバート・リム・縁)。

```
phantom_arm.cpp

31 // 2次ベジエ。root → ctrl → tip の3点で曲線を作る。
32 // 制御点を動かすだけで振りも溜めも表現できるので、物理シミュを持たずに済む。
33 static Vector3 Bezier(const Vector3& p0, const Vector3& p1, const Vector3& p2, float t) { ①
34     const float u = 1.0f - t;
35     ...
332 // 前の u から接線成分を抜いて、直交させ直す
333 U = carriedU - tan * Vector3::dot(carriedU, tan); ②
351 float barb = sinf(u * m_BarbFreq + phase * 1.7f);
352 barb = (barb > 0.0f) ? powf(barb, BARB_SHARP) : 0.0f;
353 const float swell = 1.0f + m_BarbAmp * barb; ③
364 const float twEnv = powf(u, 0.70f) * m_Deploy;
365 const float twitch = sinf(u * TWITCH_FREQ + m_Time * TWITCH_SPEED + phase * 3.1f) ④
366     * sinf(m_Time * TWITCH_BEAT + phase)
367     * TWITCH_AMP * twEnv;
```

phantom_arm.cpp L31-L34 / L332-L333 / L351-L353 / L364-L367

手応えを組み合わせで作る — ヒットストップと被弾演出

— 課題

当たった手応えは、ダメージ数字ではなく、画面と音と振動で決まる。1つの演出だけ強くしても、まとまった手応えにならない。

— やったこと

ヒットストップは、時間の進み方(`TimeScale`)を一瞬だけ下げて作る①。カウントダウンは

`UnscaledDeltaTime` で行うので、止まっている間も戻る時刻がずれない③。重なったときは、長いほうを採る②。

被弾のときは、ヒットストップ・カメラの揺れ・パッドの振動・カメラごと押される動き・画面の効果・音を、同じ入口から同時に出す④。斬撃・ジャスト回避・覚醒でも、同じ部品を強さだけ変えて使う。

— ポイント

- 演出は「フレームワーク層」の部品(`engine/feedback`)。ゲーム側は強さと長さを渡すだけ。

hit_stop.cpp

```
17 void HitStop::Trigger(float duration, float scale) {
18     if (s_Timer <= 0.0f) {
19         // 新規発動：現在のTimeScaleを保存
20         s_OriginalScale = Time::GetTimeScale();
21     }
22     // 既存のヒットストップより長ければ上書き（重なった時の対応）
23     if (duration > s_Timer) {
24         s_Timer = duration;
25     }
26     Time::SetTimeScale(scale);
27 }
...
33 s_Timer -= Time::GetUnscaledDeltaTime();
```

hit_stop.cpp L17-L27 / L33

player_vitals.cpp

```
68 HitStop::Trigger(0.15f, 0.05f);
69 CameraShake::Trigger(1.2f, 0.42f);
70 InputManager::StartVibration(0.8f, 0.32f); // 画面の揺れと同じ強さ・長さで振動させる
71 CameraPush::Trigger(knockbackDir, 0.7f, 0.30f); // 殴られた方向へカメラごと押される
72 ScreenFX::TriggerDamage(1.0f, 0.95f);
73 ScreenFX::TriggerFlash(0.18f, 0.09f); // 一瞬白く抜いて食らった瞬間を強く
```

player_vitals.cpp L68-L73

ゲームを起動せずに確かめる — 画面に依存しないテスト

— 課題

手触りやゲームの流れは、遊んで確かめるしかない部分が多い。一方で、「時間の計算」「当たりの幾何」「数値の増減」は、間違えても気づきににくく、直したつもりで壊しやすい。

— やったこと

判定や計算のロジックを、描画・入力・ウィンドウに依存しない小さなヘッダに切り出し、ゲームを起動せずに確かめるコンソールのテストを書いた。①は「条件」と「名前」を渡す `Check` だけ。失敗するとその名前を出して異常終了する。

例えば敵の攻撃は、大きな `dt` で全発生をちょうど 1 回ずつ跨ぐこと②、中断したら発生しないことを確かめている。

— ポイント

- テストできる形にするために、ロジックを分けた。分けた結果、描画や物理の差し替えにも強くなった。
- 手触りそのもの(ヒット感・操作感)は、自動では確かめられないので、実際に遊んで確認する。

— tests/ の内容

<code>enemy_tactics_tests</code>	敵の攻撃 7 種の時間軸(予告・発生・追撃の順序、大きな <code>dt</code> 、中断)、出現する役割、同時に攻撃できる数、高さを含む範囲判定
<code>triangle_mesh_tests</code>	三角形メッシュの床・壁・屋上・掃引・箱の判定。実物のビルの BVH を、総当たりのレイ計算と比べる
<code>player_health_tests</code>	HP・回復・HUD の時間変化
<code>handgun_tests</code>	拳銃の発射制御・強化・レベルアップ特典
<code>refactor_tests</code>	整理で切り出した部品(HP の通知・攻撃のセッション・特典)

```
enemy_tactics_tests.cpp
11 static void Check(bool value,const char* name) { ++checks; if (!value) throw std::runtime_error("n
    :
35     EnemyAttackClock skipped; skipped.Start(plan);
36     Check(skipped.Advance(plan.duration+1.0f)==((1u<<plan.eventCount)-1u),"large dt re2ai
37     Check(Close(skipped.DashSeconds(),plan.dashDuration),"large dt dash movement only use
    :
38     EnemyAttackClock canceled; canceled.Start(plan); canceled.Advance(plan.startup*0.25f)
39     Check(canceled.Advance(plan.duration)==0,"interrupted cast cannot release");
40     Check(canceled.Finished() && canceled.WarningProgress()<0 && Close(canceled.DashSecon
```

enemy_tactics_tests.cpp L11 / L35-L37 / L38-L40

担当のまとめと、次の課題

1.86 → 1.25秒

Release の起動時間(バイクと VFS の導入)

12.7 → 3.3秒

Debug の起動時間(ミップ生成を GPU へ)

30 → 60 FPS

衝突判定をブロード/ナローに分けた結果(開発機)

5 本

画面に依存しないテスト

— 担当の要点

- 個人開発として、フレームワーク(描画・物理・アセット・音)からゲームの中身まで設計・調整した。コードの記述は主に AI(Claude Code・Codex)に依頼し、仕様の指示・調整・動作確認は私が行った。
- 「追加・調整が 1 か所で済む形」を優先した(X-Macro、レイヤーの表、State ごとのキャンセル、ISkill)。
- 速度の話は計測してから手を入れ、効かなかったものも記録に残した。

— 次に手を入れたいところ

- **区画グリッドの効果の数値化:** 30 → 60FPS は確認できたが、ms 単位の前後比較は取っていない。計測して残す。
- **OBB×OBB:** 未実装のまま。武器どうしの判定など、必要になった時に足す。
- **Release 側の計測:** GPU ミップの計測は Debug のみ。分けた部品(State・ISkill)を含めた最新の起動時間も取り直す。
- **手触りの確認:** State や技を分けたあとの操作感は、テストでは見られない。遊んで確かめ続ける。