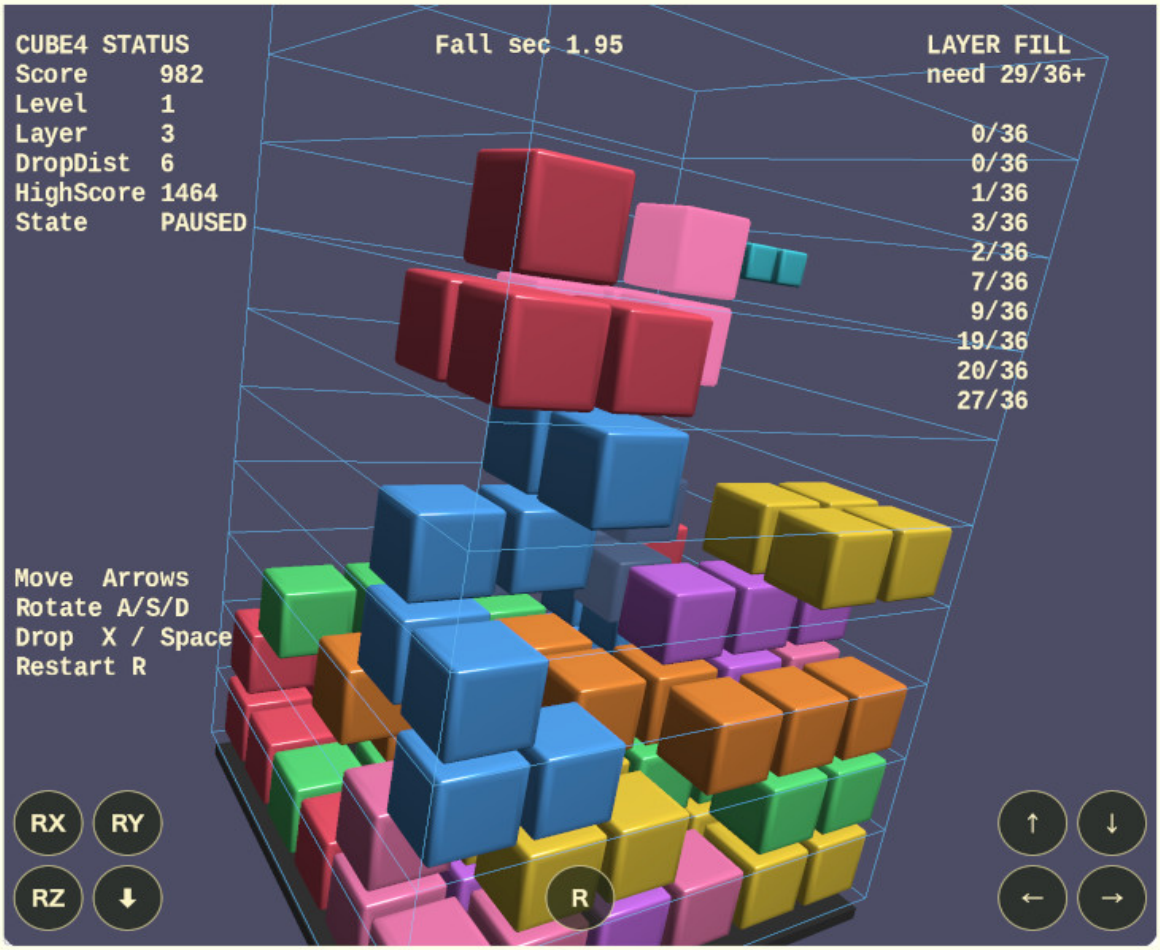
webgではじめる
WebGPU 3D アプリ開発



水谷 純

webg ではじめる
WebGPU 3D アプリ開発

水谷 純　著

2026-07-03 版　　発行

まえがき

本書は、JavaScript と WebGPU を用いて 3D アプリケーションを構築するためのライブラリ `webg` を題材にした解説書です。`webg` は、ブラウザが標準で備える WebGPU、WebAudio、Compression Streams、JavaScript を利用して、`webg` とブラウザさえあれば 3D アプリケーションを開発したり実行したりできる環境です。外部のライブラリもビルドツールも不要なので、インターネットに接続する必要も、環境構築で挫折することもありません。外部の大規模な 3D エンジンに依存せずに、3D 描画、シーン管理、モデル読み込み、アニメーション、入力、UI、診断情報、音声、ポストプロセス、物理エンジンまでを一つのフォルダ内で扱えます。入力についても、タッチ、マウス、ペンを Pointer Events として同じ gesture 仕様へまとめることで、スマートフォン向けの直接操作を PC ブラウザ上でも同じ考え方で試せるようにしています。

Logo 言語のタートルグラフィックス（Turtle Graphics）という、画面上の亀に前進、回転、ペンの上げ下げを指示して絵を描かせるというものがありました。1991 年頃にアセンブラで 16 ビットの固定小数点の 3D パッケージを作って、タートルグラフィックスの 3 次元版を作ったのが筆者の 3 次元歴の始まりです。しかし最初に立方体を描こうとして、90 度上を向くところでジンバルロックにぶち当たりました。真上を向くと次に進む方角が不安定になるという、角度で回転を指定する限り避けられない問題です。そこから行列、四元数（Quaternion）と調べていって、少なくとも計算の原理には詳しくなりました。`webg` の Quat.js や Matrix.js は、いわば「30 年間継ぎ足して使っている秘伝のタレ」のようなものです。使った言語も Pascal、C++、Python、Lua、JavaScript と渡り歩いてきました。

この「秘伝のタレ」を現代のコーディング AI に継承させ、AI に最適な形で 3D アプリケーションを構築してもらうためのコンテキストを整理した結果、このライブラリと本書が生まれました。特に、最新の WebGPU 機能であるコンピュートシェーダーを用いた高度な画面効果の統合に至るまでを体系化したことで、AI が最も推論しにくい「複雑な処理順序」や「リソースの依存関係」という暗黙知を、明示的なコンテキストとして AI に提供できるようになりました。したがって、本書は「人間の開発者」だけでなく「コーディング AI」をも想定した、一風変わった構成の書籍となっています。

本書が想定する読者

本書は、すでに WebGPU に詳しい人だけを対象にした本ではありません。JavaScript の基本的な読み書きができ、ブラウザ上で 3D 表現を動かしてみたい人、既存の巨大なエンジンを使うだけでなく中の仕組みも理解したい人も対象です。また、最初は初心者であっても、コーディング AI に相談すれば 3D アプリケーションを作ってもらいながら成長していくことができるでしょう。

3D グラフィックスには、座標系、行列、クォータニオン、カメラ、シェーダー、アセット形式、アニメーション、衝突判定、物理挙動など、多くの専門用語が出てきます。本書では、それらを一度にすべて暗記することを求めません。まずは「何がどの役割を持つのか」をつかみ、必要な場面で該当する章へ戻れるように構成しています。たとえば、座標系や視野角で迷ったときは第 3 章へ、`WebgApp` の起動順序で迷ったときは第 5 章へ、カメラ操作で迷ったときは第 6 章へ、レイキャストや重なり判定で迷ったときは第 17 章へ、物理挙動で迷ったときは第 26 章へ戻る、という読み方ができます。

また、本書は人間の読者だけでなく、コーディング AI も読者として想定しています。近年、アプリケーション開発では AI にコードの作成や修正を依頼する場面が増えています。しかし AI が正確に支援するためには、対象プロジェクトの設計思想、用語、処理の順序、参照すべきサンプルが文書として明確になっている必要があります。`webg` では、この書籍の原稿すべて、100 本を超える実行可能なサンプルとテスト用アプリ、webg のコアをできるだけ対応づけ、AI が一般論だけで推測せず、プロジェクト内の一次情報を根拠に判断できるようにしています。

コーディング AI と一緒に読む場合

コーディング AI に `webg` を使ったアプリケーション作成を依頼する場合は、最初に `book/付録A_コーディングAIの皆さまへ.md` を読んでもらうことをおすすめします。この付録には、AI がどの章やサンプルを優先して参照すべきか、一般的な WebGPU や他の 3D エンジンの知識をどのように扱うべきか、`webg` の設計を崩さないために何を守るべきかがまとめられています。

AI に依頼するときは、「3D のアプリを作って」とだけ伝えるよりも、「第 5 章の `WebgApp` の構成に従って」「第 6 章のカメラ制御を参考にして」「samples の該当サンプルと説明テキス

トを読んでから」といった形で、参照してほしい文書やサンプルを指定すると精度が高くなります。webg は、AI が実装を進める際にも、本文、サンプル、unittest、コア実装をたどりやすいように構成されています。

一方で、AI の出力は常に実行して確認することが重要です。3D アプリケーションでは、文法上は正しいコードでも、初期化順序、描画順序、座標系、アセットの向き、入力状態の扱いが少しずれるだけで、意図した見え方にならないことがあります。本書では、サンプルや unittest を通じて、実際にブラウザで確認しながら理解を進めることを重視しています。

この本で大切にしていること

本書では、API の使い方だけでなく、その背景にある理由や仕組みをできるだけ説明します。うまく動かないときこそチャンスです。苦労して解決できたことは忘れません。なぜ `Shape.endShape()` が必要なのか。なぜ `clear`、`draw`、`present` の順序が重要なのか。なぜ `WebgApp` は `cameraRig`、`cameraRod`、`eye` という構成を持つのか。なぜ座標系や回転用語を本書全体で統一する必要があるのか。こうした理由を理解しておくと、問題が起きたときに原因を切り分けやすくなります。

また、本書では webg を巨大なブラックボックスとして扱いません。`WebgApp` のような便利な入口は用意しつつ、必要に応じて内部構造を読み解けるようにします。これは学習のためだけではなく、実際の開発で問題に直面したときにも重要です。表示がおかしい、入力が効かない、モデルが想定と違う向きになる、アニメーションが再生されない、という問題は、たいてい一つの API 名だけでは解決できません。どの層で何が起きているのかを追えることが、安定した開発につながります。

webg は、ブラウザ上で 3D アプリケーションを作るための実験場であり、学習教材であり、実用的なアプリケーション基盤でもあります。本書が、その三つの側面を行き来するための地図となり、読者が自分の手で 3D 表現を組み立てる助けになれば幸いです。

目次

第 1 章

はじめに

webg は、Chrome、Safari、Firefox といったブラウザが標準で持っている JavaScript、WebGPU API、WebAudio API を利用して 3D アプリケーションを構築するためのライブラリです。PC やスマートフォンのブラウザ上で高速に動作する 3D アプリの作成を目的として設計されており、キー入力、タッチジェスチャー、マウス操作、ピンチ操作、効果音や BGM の再生、GLB アニメーションのインポート、パーティクル、被写界深度、フォグ、物理演算に加えて、G-buffer、SSAO、Shadow Map、SSR、Toon、Edge のような Compute Shader を使った画面効果まで、3D 表現に必要となる機能を網羅しています。

外部ライブラリや複雑なビルドツールを導入する必要はなく、実装に必要なコードと詳細な解説はすべて webg 自身に含まれています。そのため、学習目的での利用はもちろん、AI を活用した効率的なアプリ開発にも適しています。また、WebGPU による 3D 開発で不可欠な座標計算や GPU 制御といったローレベル（低レイヤー）なプログラミングから、多くの機能が抽象化された WebgApp クラスによる高水準（高レイヤー）なプログラミングまで、柔軟に使い分けることが可能です。

図 1.1: circular_breaker

webg では、低レイヤーと高レイヤーの両方を扱うことができます。学習の初期段階で重要なのは、個別のメソッド名を暗記することではなく、「どのような用途に向いたライブラリなのか」「どの文書から読み進めれば理解が早いのか」「サンプルと unittest をどう使い分ければよいのか」といった、学習の指針をつかむことです。

本章では、webg の全体像、設計思想、プロジェクト内のファイル配置、そして学習のロードマップについて解説します。

1.1 webg が提供する価値と設計思想

WebGPU を用いた 3D 実装では、描画処理が完成した後、アプリケーションの規模が大きくなるにつれて、シーングラフ、アセット管理、カメラ制御、入力処理、診断情報、音声再生といった周辺機能が必要になります。特に AI と共同で実装を進める場合、「どの機能がどこに定義されているか」「現在の挙動を理解するために何を読めばよいか」が曖昧であると、調査や修正の効率が著しく低下します。

webg はこの問題を解決するため、3D アプリケーションを構成する各レイヤーをリポジトリ内で明確に分離し、サンプルコードと解説文書を対にして読める構造を採用しています。巨大なエンジンをブラックボックスとして利用するのではなく、プロジェクトに必要な部分を順に読み解き、組み上げていくスタイルを重視しています。

入力については、スマートフォン向けの操作を別系統として扱うのではなく、タッチ、マウス、ペンを Pointer Events として同じ処理フローへ集めます。これにより、スマートフォンで使う tap、double tap、long press、flick のような gesture を PC ブラウザ上でも同じ条件で確認でき、3D ビューアや編集ツールに対して、デバイスをまたいだ直接操作の UI を設計しやすくなります。

具体的に、webg は以下のような場面で真価を発揮します。

- WebGPU を使った 3D アプリケーションの内部構造を理解しながら構築したいとき
- サンプル実装とドキュメントを往復しながら効率的に学びたいとき
- 診断情報、HUD、タッチ入力、音声処理までを単一のリポジトリで完結させたいとき
- スマートフォン向け gesture UI を PC ブラウザ上でも確認しながら調整したいとき
- コーディング AI に 3D アプリ開発の支援をさせたいとき

1.2 本書の構成

webg の最大の特徴は、3D の基礎からアプリケーション構成までが一つのリポジトリで完結している点です。本書では、まず実行環境を整備し、3D の基礎と最小限の描画フローを押さえた後、WebgApp、アセット、アニメーション、UI、診断情報、ポストプロセス、ローレベル API、物理エンジン、そして Compute Shader を使った空間表現の構築と統合へと段階的に進む構成をとっています。

導入と 3D の基礎（第 1 章 〜 第 3 章）

まずは環境構築と、3D 空間における共通概念を学びます。

- 環境構築： WebGPU が動作するブラウザの準備とローカルサーバーの起動（第 2 章）。
- 3D 数学の基礎： 座標系（EUS）、UV マップ、行列（Matrix）とクォータニオン（Quat）の基本概念（第 3 章）。

webg のコア機能と基本実装（第 4 章 〜 第 6 章）

webg の基本構造を理解し、簡単なシーンを表示させます。

- 低レベル API： Screen, Space, Shape を用いた描画ループの構築（第 4 章）。
- WebgApp（高レベル API）： アプリケーション全体を管理する WebgApp クラスの使い方と、HUD/UI の基本（第 5 章）。
- カメラ制御（EyeRig）： Orbit, First-person, Follow といった多様なカメラ挙動の実装（第 6 章）。

シェーダーとアセットの活用（第 7 章 〜 第 13 章）

見た目の質を向上させ、外部モデルを導入します。

- マテリアルとシェーダー： `SmoothShader` による Phong 反射モデルとフォグ効果（第 7 章）。
- WGSL 入門： WebGPU のシェーディング言語 WGSL の基本と TBN 行列（第 8 章）。
- カスタムシェーダー： NormPhong や BonePhong など、独自のシェーダー実装（第 9 章）。
- モデル読み込み： glTF/GLB 形式のモデルを `ModelAsset` として扱う方法（第 10 章）。
- シーン定義： JSON 形式でシーンを記述する `Scene JSON` の活用（第 11 章）。
- アニメーション： Clip → Action → State へと至るアニメーション状態遷移の構築（第 12〜13 章）。

応用機能と高度な実装（第 14 章 〜 第 26 章）

インタラクティブな機能と高度な視覚効果を追加します。

- ユーザーインターフェース： HUD、OverlayPanel、タッチ操作の実装（第 14〜16 章）。
- 空間判定： レイキャスト（Raycast）と衝突判定（Picking）（第 17 章）。
- オーディオ： `ToneSynth`、`AudioSynth`、`GameAudioSynth` による単音、BGM、SE の制御（第 18 章）。
- デバッグと最適化： `DebugDock` と Diagnostics による実行時解析（第 19 章）。
- ポストプロセス： Bloom, DOF（被写界深度）, Vignette などの画面効果（第 20 章）。
- パーティクル： `ParticleEmitter` によるエフェクト実装（第 21 章）。
- 詳細 API と高度な形状： 形状生成 API、手続き的形状（Icosphere 等）の構築（第 22〜24 章）。
- スキニング： ボーンウェイトとスケルトンアニメーションの内部構造（第 25 章）。
- 物理演算： `PhysicsSpace` による剛体物理と衝突応答（第 26 章）。

Compute Shader と空間表現の統合（第 27 章 〜 第 29 章）

本書の集大成として、Compute Shader を用いて高品質な空間表現を構築し、それらを一つのパイプラインへと統合します。

- **コンピュートシェーダーの基礎（第 27 章）**：`ComputePass` を基礎に、ストレージバッファやワークグループ、データ同期などの実行モデルを学びます。

- **個別パスの実装（第 28 章）**: GeometryBufferPass、SsaoPass、ShadowMapPass、ComputeSsrPass など、空間情報を扱う個別パスの構成と処理フローを理解します。
- **リアルタイム 3D 表現の統合（第 29 章）**: ComputeEffectPipeline を用い、SSAO、Shadow Map、SSR、Toon、DoF、Bloom、Edge を標準 Space と Shape に効率的に重ね合わせる、実用的かつ高水準な統合手法を学びます。

1.3 プロジェクトの構成

webg を読み解く際は、まずファイル配置と役割を把握してください。

- webg/（コアライブラリ）: Screen、Space、Node、Shape、WebgApp など、ライブラリの本体となる実装が含まれています。
- samples/（教材サンプル）: 機能確認とデモを兼ねたサンプル群です。各サンプルは main.js（実装）と *.txt（解説）で構成されており、samples/index.html がその目次となっています。
- unittest/（単体テスト）: 特定の機能を切り出して検証するためのテストアプリ集です。問題の切り分けや最小単位での動作確認に適しています。
- book/examples/（本書のコード例）: 本書で出てくるコードを実行可能な形にしたサンプルです。

1.4 webg で実現できること

webg は、WebGPU 上で 3D アプリケーションを構築するための基盤を包括的に備えています。

- 描画基盤: Screen、Shader、Shape、Space、Node、RenderTarget
- アプリ入口: WebgApp
- アセット・ローダー: ModelAsset、ModelLoader、ModelBuilder、SceneAsset、SceneLoader
- アニメーション: Animation、Action、AnimationState、Skeleton
- カメラ・入力: EyeRig、Touch、InputController
- UI・診断情報: Message、OverlayPanel、Diagnostics、DebugDock、DebugProbe
- 音声: ToneSynth、AudioSynth、GameAudioSynth

- ポストプロセス: FullscreenPass、BloomPass、DofPass、VignettePass など
- Compute Shader 系効果: GeometryBufferPass、SsaoPass、ShadowMapPass、SpotShadowMapPass、ComputeShadowPass、ComputeSpotShadowPass、ComputeSsrPass、ComputeToonPass、ComputeDofPass、ComputeBloomPass、ComputeEdgePass、ComputeEffectPipeline
- 物理: PhysicsNode、PhysicsSpace、Collider など

1.5 コーディング AI に作ってもらう

この本と webg のリポジトリは、コーディング用の AI（コーディングエージェント）も読者として意識して設計されています。人間にとっての読みやすさと、AI にとっての解析しやすさの両立を追求しています。

コーディングエージェントに webg をプロジェクトフォルダーとして読み込ませ、「book/付録A_コーディングAIの皆さまへ.md を読んで、my_webg (例) フォルダー以下に XXXX を作成して」と指示することで、効率的にアプリケーションを構築させることが可能です。

1.6 問題調査の考え方

3D アプリケーションの開発では、単なるコンソール出力だけでなく、「なぜその結果になったか」を画面上で可視化して追跡することが重要です。webg では、HUD やパネルに情報を表示させ、必要に応じて診断情報レポートや probe を出力する手法を標準としています。後の章で解説する Diagnostics や DebugDock は、この「可視化によるデバッグ」という思想に基づいています。

1.7 まとめ

本章で最も重要な点は、webg を単なる API 集としてではなく、WebGPU の低レイヤーな描画から WebgApp による高レイヤーな構成、さらに Compute Shader を使った空間表現の統合までを体系的に学べるフレームワークとして捉えることです。

まずは第 2 章で環境を整え、第 3 章で基礎を学び、第 4 章で最初の描画に挑戦してください。本書が、WebGPU を用いた 3D アプリケーション開発の地図として役立つことを願っています。

第 2 章

インストールと実行環境

本章では、webg を手元のマシンに配置し、サンプルや unittest を実際に起動するまでの手順を解説します。第 1 章でプロジェクトの全体像を把握し、第 4 章以降でアプリケーションの組み立て方を学ぶための前提として、まずは「正しく動作する実行環境」を整えます。

webg の利用においては、ライブラリをパッケージとしてインストールするのではなく、リポジトリをそのまま配置して利用する運用を基本としています。また、サンプルや unittest を実行する際は、ファイルをブラウザで直接開くのではなく、ローカルサーバーを経由してアクセスしてください。

もし、実装が正しいにもかかわらずファイルを直接開いてアセットの読み込みに失敗した場合、「WebGPU のコードに問題があるのか」「ブラウザが未対応なのか」「パスの設定が間違っているのか」という切り分けが困難になります。そのため、まずは「サーバーの問題」と「ブラウザの WebGPU 対応状況」を明確に分けて考えることが重要です。

2.1　ローカルサーバーが必要な理由

webg のサンプルは、単一の HTML ファイルで完結しているわけではありません。main.js や webg/*.js といったスクリプトに加え、JSON、テクスチャ、glTF / GLB、Collada といった外部アセットを組み合わせて構成されています。

ブラウザでこれらのファイルを読み込む際、セキュリティ上の重要な基準となるのが Same-Origin Policy (SOP: 同一オリジンポリシー) と CORS (Cross-Origin Resource Sharing) です。SOP は、ページがアクセスできるリソースを「同一オリジン（プロトコル、ホスト、ポートがすべて一致するもの)」に制限することでセキュリティを確保する仕組みです。

ファイルをブラウザで直接開いた場合（file:// プロトコル）、多くのブラウザではセキュリティ制限が厳しくなり、ES Modules の import や fetch() によるアセット読み込みが制限されます。その結果、ページ自体は表示できても、テクスチャやモデルを読み込んだ瞬間にエラーが発生します。

具体的に webg 内で fetch() を利用しているのは以下のクラスです。* webg/Texture.js * webg/ModelAsset.js * webg/SceneAsset.js * webg/Gltf.js * webg/GltfShape.js

これらを正常に動作させるには、http://localhost:8000/ のようなローカルサーバーを立て、HTTP 配信経由でアクセスする必要があります。これにより、相対パスによるモジュールの読み込みやアセットの取得が、標準的な Web ページと同様に動作するようになります。

2.2 ファイルの配置手順

まずは webg リポジトリを手元に用意してください。

git clone を利用する場合

更新履歴を管理し、最新の状態を維持したい場合に推奨される方法です。

```
git clone https://github.com/jun-mizutani/webg.git
cd webg
```

フォルダコピー（zip）を利用する場合

GitHub のリポジトリから zip ファイルをダウンロードし、展開して配置してください。（ダウンロード先: https://github.com/jun-mizutani/webg/archive/refs/heads/main.zip）

展開した webg フォルダ内に、例えば my_webg という作業用フォルダを作成し、その中でプロジェクトを構築すると、サンプルと同じインポートパスでコア機能を利用できるため便利です。

```
webg/
  book/
```

```
samples/
unittest/
webg/
my_webg/
```

なお、webg/samples/ や webg/ フォルダだけを切り出すと相対パスが崩れるため、必ずリポジトリのルートディレクトリを保持したまま運用してください。

2.3 サンプルの実行

webg に付属するサンプルを実行するには、後述するローカルサーバーを起動してから、ブラウザで以下の URL にアクセスして下さい。

```
http://localhost:8000/samples/
```

2.4 実行環境の使い分け

リポジトリ内のファイルには、「直接開いても問題ないもの」と「サーバー経由で開くべきもの」があります。

直接開いても動作しやすいもの

以下のような静的なページは、主に一覧や説明の表示が目的であり、アセット読み込みやモジュール実行に依存しないため、file:// で直接開くことが可能です。* webg/samples/index.html * webg/unittest/index.html * 各種ドキュメントファイル

必ずローカルサーバーで開くべきもの

以下のような実行系ページは、<script type="module"> による ES Modules の利用や、fetch() による外部ファイルの読み込み、および WebGPU の初期化（navigator.gpu）に依存しているため、必ずローカルサーバーで起動してください。* 最小コードの実行ページ* we

bg/samples/*/index.html の大部分* webg/unittest/*/index.html の大部分* webg/book/examples/*.html の大部分

2.5　ネットワーク環境とセキュアコンテキスト

LAN 内の既存 Web サーバー（例: http://192.168.1.20/webg/）にリポジトリを配置して実行することも可能です。この場合もモジュール import や fetch() は正常に動作します。

ただし、WebGPU はブラウザによって セキュアコンテキスト（Secure Context） を要求します。localhost は例外的にセキュアな環境として扱われますが、LAN 内の IP アドレスやホスト名で HTTP 配信した場合、navigator.gpu が利用できないことがあります。

LAN 内サーバーで実行し、アセット読み込みはできているのに WebGPU が動作しない場合は、配信プロトコルを HTTPS に変更するか、動作確認のみを localhost で行うことで、問題が「サーバーの設定」にあるのか「WebGPU の対応状況」にあるのかを切り分けてください。

2.6　ローカルサーバーの起動方法

リポジトリのルートディレクトリをカレントディレクトリとして、以下のコマンドを実行してください。

```
cd /path/to/webg
```

OS ごとの起動コマンド例

Mac / Linux

Python 3 の簡易サーバーを利用する方法が最も手軽です。

```
# Python 3 を利用する場合
python3 -m http.server 8000

# Node.js (npx) を利用する場合
```

```
npx http-server . -p 8000

# PHP を利用する場合
php -S 127.0.0.1:8000
```

Windows

Windows 環境では、Python ランチャー（py）の利用が簡単です。

```
# Python Launcher を利用する場合
py -m http.server 8000

# Python 3 を直接利用する場合
python -m http.server 8000

# Node.js（npx）を利用する場合
npx http-server . -p 8000
```

サーバーを起動後、ブラウザで以下の URL にアクセスしてください。`http://127.0.0.1:8000/samples/index.html` または `http://localhost:8000/samples/index.html`

※ 他のアプリケーションとポートが衝突する場合は、8000 の部分を 9000 など任意の番号に変更してください。

2.7 推奨される確認手順

環境構築後、いきなり複雑なサンプルを動かすのではなく、以下の順序で段階的に確認することをお勧めします。これにより、問題が発生した際の切り分けが容易になります。

1. `samples/index.html` を開くサーバーが正常に動作し、ファイルが配信されているかを確認します。
2. `low_level` サンプルで最小描画を確認するローレベル（低レイヤー）な最小構成で、WebGPU の描画が機能するかを確認します。

3. high_level サンプルで WebgApp の構成を確認する高水準（高レイヤー）な標準構成で、アプリ基盤が動作するかを確認します。
4. scene サンプルで Scene JSON の読み込みを確認する外部アセットの読み込みとシーン構成が正しく動作するかを確認します。
5. 目的に応じた詳細サンプル（sound, bloom, dof など）へ進む

2.8 トラブルシューティング

ローカルサーバーを起動しても動作しない場合は、まず「サーバーの問題」と「ブラウザの WebGPU 対応状況」を切り分けて確認してください。

- サーバーの問題が疑われる場合
  - URL で一覧ページが表示されるか。
  - ブラウザのコンソールに 404 Not Found や CORS error が出ていないか。
  - リポジトリの途中階層（例: webg/samples/scene/）をサーバーのルートにしていないか。
- WebGPU の対応状況が疑われる場合
  - ブラウザのコンソールで navigator.gpu が undefined になっていないか。
  - webg/Screen.js が WebGPU の初期化に失敗していないか。

最小構成である low_level サンプルまで戻って動作を確認することで、問題が「アセット読み込み」にあるのか、「WebGPU の実行環境」にあるのかを迅速に特定できます。

2.9 まとめ

本章では、webg を利用するための基本運用として、「リポジトリをそのまま配置し、ローカルサーバー経由で実行する」手順を確認しました。

静的なページは file:// で閲覧できても、実際の描画処理を含むページは ES Modules や fetch()、WebGPU の初期化を必要とするため、ローカルサーバーの利用が不可欠です。この実行環境の基準を明確にしておくことで、今後の学習において「実装の問題」と「実行条件の問題」を混同することなく進めることができます。

次章では、この実行環境の上で、webg が採用している 3D グラフィックスの基礎定義（座標系、回転、行列、クォータニオン、UV、ノーマルマップなど）について詳しく解説します。

第 3 章

3D グラフィックスの基礎

3D グラフィックスの実装において、座標系、回転、行列、クォータニオン、UV、ノーマルマップといった基礎概念は、互いに密接に関連しています。これらのうち、どれか一つの定義だけを理解していても、実際にコードを書く段階で他の定義と整合性が取れなければ、実装は困難になります。

特に、「Y 軸が上か、Z 軸が上か」「Y 軸まわりの回転をどの用語で呼ぶのか」「行列は行優先か列優先か」「ベクトルを左に置くのか右に置くのか」「クォータニオンの要素順序はどうなっているか」といった定義が曖昧なままだと、コードの意図は理解できても、具体的な実装で手が止まってしまいます。

そこで本章では、`webg` で採用している定義について詳しく解説します。まず、`webg` は右手系を採用し、`+X=右`、`+Y=上`、`+Z=前` を基本軸とします。回転の用語には `yaw / pitch / roll` を用います。行列は列優先（Column-major）で保持し、ベクトルは列ベクトルとして右側に配置し、数式としては `M * v` の形式で処理します。頂点はローカル座標からワールド座標、view 座標、clip 座標、画面座標へ順に変換されます。クォータニオンの内部順序は `[w, x, y, z]` です。また、面の表裏は頂点を登録する順序で決まり、表側から見て反時計回りになるように並べます。UV 座標は概念上の基準として「左下（bottom-left）」を採用し、ノーマルマップは RGB の色情報を `-1..1` の方向ベクトルとして扱います。

本章でこれらの基準を揃えておくことで、`Node` の姿勢制御、カメラの orbit 操作、GLB モデルの読み込み、ノーマルマップの適用、そしてスケールの計算に至るまで、すべてを同一の土台の上で理解できるようになります。

3.1 空間の定義と回転の基準

webg では右手系座標系を採用しています。右手の親指を +X、人差し指を +Y、中指を +Z と向けたとき、その方向がそのまま基本軸となります。

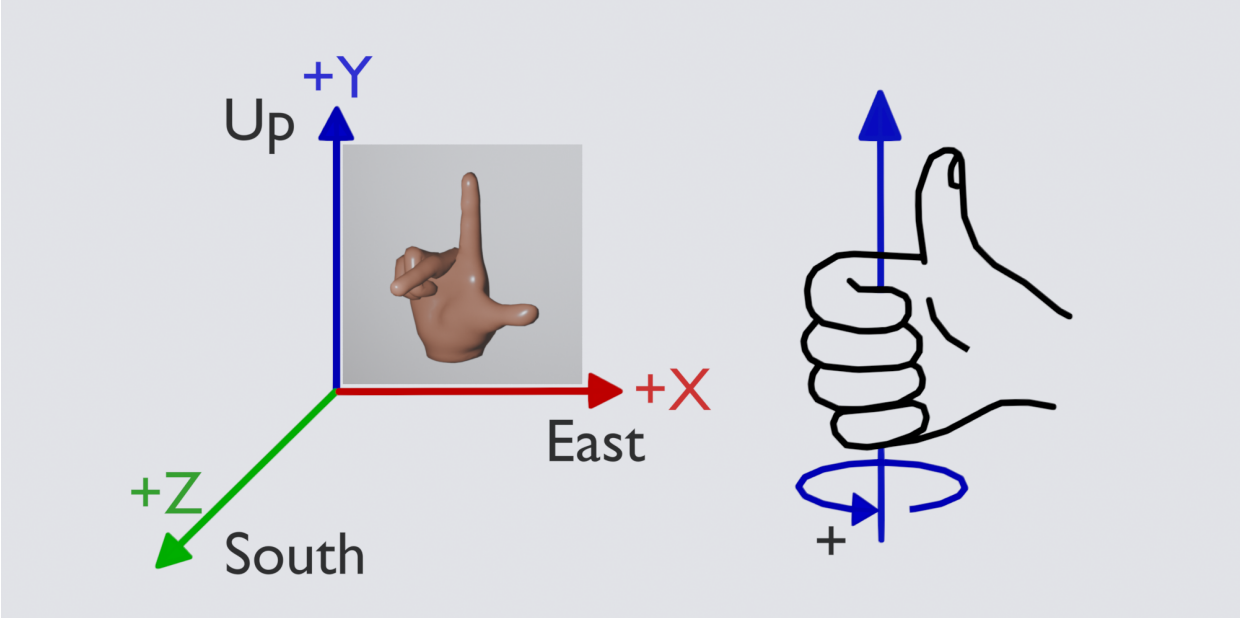


図 3.1: 座標系と回転の基準

webg は右手系を採用し、$+X$=右、$+Y$=上、$+Z$=前 を基本軸とします。回転の順序は *yaw* / *pitch* / *roll* = Y / X / Z の対応で整理されます。

3D 空間において「どちらが前か」という定義は、視点や文脈によって異なります。例えば、航空分野で用いられる NED 座標系（North-East-Down）では、X 軸を真北（進行方向）、Y 軸を真東、Z 軸を真下（地球の中心方向）と定義します。

これに対し、webg の座標系は、いわゆる EUS 座標系（East-Up-South）に相当します。これは X 軸を真東、Y 軸を真上、Z 軸を真南に向ける定義です。この系において Z 座標の値が増える方向を「前」と定義すると、結果として +X は右、+Y は上、+Z は前となります。カメラが向いている方向を「前」と定義すれば、自然と画面の奥方向が +Z となります。

回転の定義についても、用語を統一します。webg では回転を yaw / pitch / roll で表し、回転軸はそれぞれ yaw=Y、pitch=X、roll=Z に対応します。

用語	回転軸	動作のイメージ
yaw	Y 軸	左右に首を振るような回転
pitch	X 軸	上下に見上げるような回転
roll	Z 軸	頭を左右に傾けるような回転

回転の方向は「右ネジの法則」に従います。回転軸に沿って右ネジを回したとき、ネジが進む方向が正の回転方向です。軸の正方向に向かって見ると正の回転は「時計回り」に見え、逆に軸の正方向側から原点を見た場合は「反時計回り」に見えます。

例えば roll（Roll）回転の場合、「前方向（画面奥）」を Z 軸の負方向としたとき、その軸周りの正の回転は、原点から奥を見た視点で時計回りとなります。右手の親指を軸の正方向に向け、他の指を軽く曲げたときの指先の方向が、回転角の正の向きです。

```
node.setAttitude(90, 0, 0);  // yaw +90（Y 軸まわりに 90 度回転）
```

pitch は X 軸まわりの回転です。右側から見たとき、正の pitch は前方向 +Z を下方向 -Y へと回転させます。例えば pitch = 90 に設定すると、前方に向いていた軸は真下を向きます。

```
node.setAttitude(0, 90, 0);  // pitch +90（X 軸まわりに 90 度回転）
```

roll は Z 軸まわりの回転で、一般的に roll と呼ばれるものです。前方から見たとき、正の roll は右方向 +X を上方向 +Y へと回転させます。これは航空機が機体を傾ける動作に相当します。

```
node.setAttitude(0, 0, 90);  // roll +90（Z 軸まわりに 90 度回転）
```

webg の Matrix.setByEuler(yaw, pitch, roll) および Quat.eulerToQuat(yaw, pitch, roll) は、Y → X → Z の順で回転を適用するオイラー角を扱います。3DCG の資料では XYZ や ZYX など様々な表記が混在しますが、webg を利用する際は「yaw(Y) → pitch(X) → roll(Z)」という順序で統一されていると捉えてください。

3.2　表面の定義：テクスチャ座標とノーマルマップ

3D モデルの表現において、形状だけでなく、どちら側を表として扱うか、表面にどのように画像を貼り付けるか、光に対してどのような凹凸を見せるかを定義する必要があります。webg では、面の表裏、UV 座標、ノーマルマップを同じ表面定義の一部として捉えます。

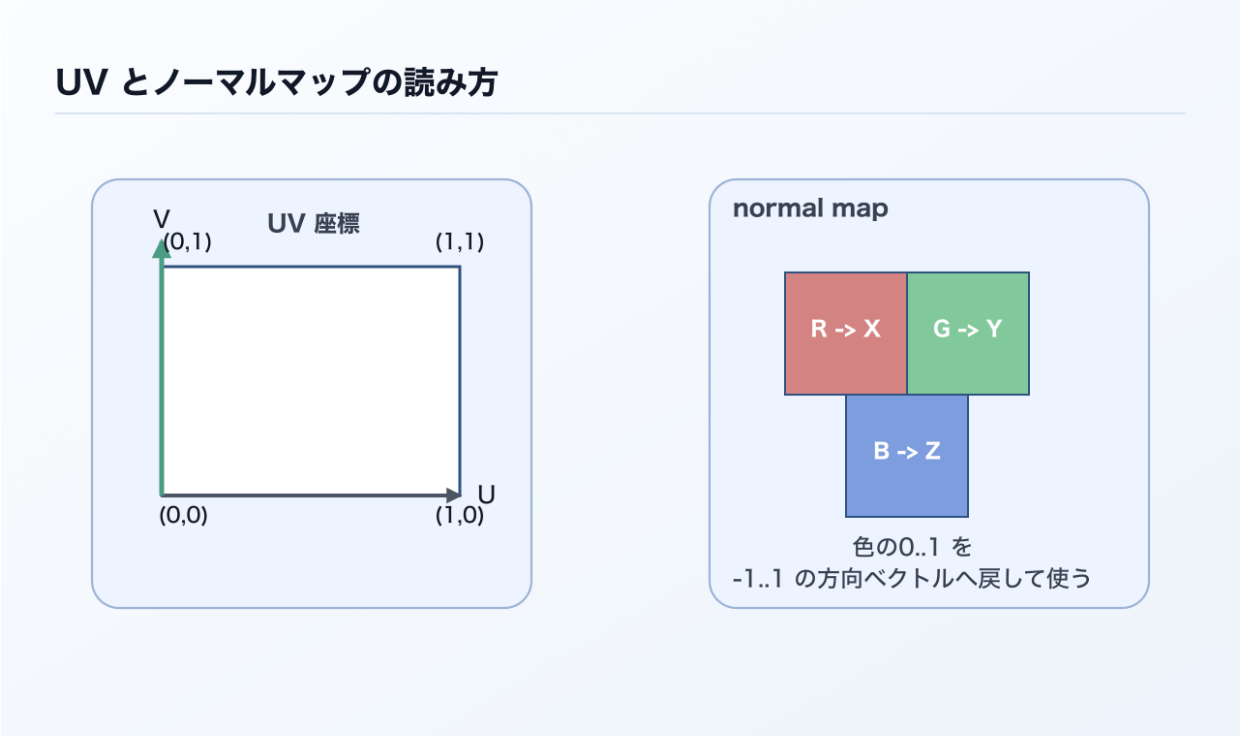


図 3.2: UV とノーマルマップの読み方

UV は概念上の基準として左下（*bottom-left*）を採用し、ノーマルマップは *RGB* の色情報を X / Y / Z の方向ベクトルとして扱います。

面の表裏と頂点順

3D の面は、単に「3 個または 4 個の頂点を結んだもの」ではありません。描画やライティングでは、その面のどちら側を表とみなすかが重要になります。webg では、ポリゴンを表側から見たときに頂点が反時計回りに並ぶように登録します。これは一般に winding order と呼ばれる規約で、面法線の向き、裏面カリング、ノーマルマップの見え方に直接影響します。

例えば、XY 平面上にあり、+Z 側を表にした四角形を作る場合は、+Z 側から面を見たときに反時計回りになるよう、左下、右下、右上、左上の順に頂点を並べます。

```
const v0 = shape.addVertexUV(-1.0, -1.0, 0.0, 0.0, 0.0) - 1; // 左下
const v1 = shape.addVertexUV( 1.0, -1.0, 0.0, 1.0, 0.0) - 1; // 右下
const v2 = shape.addVertexUV( 1.0,  1.0, 0.0, 1.0, 1.0) - 1; // 右上
const v3 = shape.addVertexUV(-1.0,  1.0, 0.0, 0.0, 1.0) - 1; // 左上

shape.addPlane([v0, v1, v2, v3]);
```

この順序では、addPlane() が内部で v0, v1, v2 と v0, v2, v3 の三角形へ分解したとき、外積（v1 - v0）×（v2 - v0）は +Z 方向を向きます。つまり、この面の表側は +Z 側になります。逆に [v3, v2, v1, v0] のように時計回りで登録すると、法線は -Z 方向を向き、同じ位置にある面でも裏返った面として扱われます。

この規約は、手動で Shape.addTriangle()、Shape.addPlane()、Shape.addPolygon() を使う場合だけでなく、ModelAsset の indices や polygonLoops を作る場合にも同じです。四角形を保持したい場合は addPolygon([a, b, c, d]) や addPlane([a, b, c, d]) のように face loop を残せますが、その場合も頂点順は「表側から見て反時計回り」を基本にします。頂点順が逆になると、SmoothShader の通常描画では裏面として扱われたり、backfaceDebug で裏面色が出たりします。

面の表裏と UV は別の概念ですが、実際の見え方では密接に関係します。面を裏返すために UV を反転するのではなく、表裏は頂点順で決め、画像の上下左右は UV と画像読み込み側の設定で調整します。この分離を守ると、法線、ノーマルマップ、ワイヤーフレーム表示、GLB / ModelAsset の読み書きが同じ規則で理解できます。

面法線と頂点法線

面の表裏を理解したら、次に区別しておきたいのが「面法線」と「頂点法線」です。法線とは、面や表面がどちらを向いているかを表す方向ベクトルです。ライティングでは、光の方向と法線の向きを比較して明るさを決めるため、法線の向きが間違うと、表側が暗く見えたり、裏側が明るく見えたりします。

面法線は、ポリゴンの頂点順から計算できる法線です。三角形 a, b, c の場合、概念的には（b - a）×（c - a）の外積で求められます。前節の規約どおり、表側から見て反時計回

りに頂点を並べていれば、この面法線は表側を向きます。したがって、面法線は「そのポリゴン自体がどちらを表としているか」を確認するための最も基本的な手がかりになります。

一方、頂点法線は、各頂点に持たせる法線です。頂点法線は必ずしも、その頂点を含む 1 枚の面法線と同じではありません。例えば、角張った箱を描きたい場合は面ごとに異なる法線を使うことで平らな面として見せます。反対に、球や円柱を滑らかに見せたい場合は、隣り合う面の法線を平均したような頂点法線を使うことで、ポリゴンの境界が目立たない陰影になります。

`SmoothShader` のようなシェーダーでは、頂点法線を補間しながらフラグメントごとの明るさを計算します。そのため、面の頂点順が正しくても、頂点法線が逆向きであればライティングは不自然になります。逆に、頂点法線が正しくても、頂点順が逆で面が裏返っていれば、裏面カリングやデバッグ表示で期待と異なる結果になります。面の向きと法線は密接に関係しますが、同じデータではないことを意識しておくと、モデル読み込みやシェーダー実装の問題を切り分けやすくなります。

裏面カリングと両面描画

3D の描画では、カメラから見て裏側を向いている面を描画しないことがあります。これを裏面カリングと呼びます。閉じた立体では、通常、外側の面だけが見えればよいため、内側を向いた裏面を描かないことで描画量を減らせます。また、間違って裏返った面を見つけるための手がかりにもなります。

裏面カリングでは、面の頂点順から「この面はカメラに対して表を向いているか、裏を向いているか」を判定します。そのため、表側から見て反時計回りという規約が崩れていると、本来見えるはずの面が消えたり、反対側からだけ見えたりします。テクスチャが貼られているのに面が見えない、スキニングした形状の一部だけが裏向きに見える、という症状は、頂点順や法線の向きが原因であることがあります。

一方で、板、布、葉、編集用の選択面、デバッグ用の面など、用途によっては両面を描画したい場合もあります。両面描画を使うと裏側からも見えるようになりますが、面の向きの誤りを隠してしまうこともあります。形状データを作る段階では、まず表裏の規約を正しく守り、必要な箇所だけ意図して両面描画にする、と考えると安全です。

UV 座標の基準

`webg` のテクスチャ座標（UV）は、概念的に bottom-left（左下）基準です。これは Blender

等の主要な 3D ツールと整合性を確保するための設計となっています。U は右方向へ、V は上方向へ行くほど値が増加するため、左下が (0, 0)、右上が (1, 1) となります。この定義は、Shape に手動で UV を設定する場合や、プロシージャルに生成したメッシュへ UV を配置する場合など、ライブラリ全体で共通です。

例えば、四角形に画像を 1 枚貼り付ける場合、概念的な UV 配置は以下のようになります。

```
[
  { pos: [-1, -1, 0], uv: [0, 0] },
  { pos: [ 1, -1, 0], uv: [1, 0] },
  { pos: [ 1,  1, 0], uv: [1, 1] },
  { pos: [-1,  1, 0], uv: [0, 1] }
]
```

ただし、GPU にアップロードした際の画像の上下方向が、概念上の UV と一致しない場合があります。その際は、UV 座標自体を変更するのではなく、画像入力側の flipV 設定を用いて調整します。UV 座標を top-left 基準に変更してしまうと、ジオメトリの定義と画像データの定義が混在し、ノーマルマップの処理などで混乱の原因となるためです。UV は「ジオメトリにどう貼るか」を決め、flipV は「テクスチャをどう読み込むか」を決める、という役割分担で整理します。

ノーマルマップの方向

表面の微細な凹凸を表現するノーマルマップは、色そのものではなく「法線方向をどちらへ傾けるか」というベクトル情報を保持しています。サンプルプログラム book/examples/NormPhong.js では、以下のように読み込んでいます。

```
let ntex = textureSampleLevel(uNormalTexture, uSampler, input.vTexCoord, 0.0).xyz
         * 2.0
         - vec3f(1.0, 1.0, 1.0);
```

ここでは R → X、G → Y、B → Z に対応しています。色情報としての 0..1 の値を -1..1 の範囲に変換することで、接空間（Tangent Space）における方向ベクトルとして利用します。例えば、R = 0.5, G = 0.5, B = 1.0 のピクセルは、方向ベクトル (0, 0, 1) となり、「表面の正面を向いている」ことを意味します。

NormPhong.js では、フラグメントシェーダー内で dpdx / dpdy を用いて tangent（接線）を再構成し、bitangent = cross(nnormal, tangent) によって右手系の TBN 行列を構築しています。つまり、ノーマルマップの X / Y / Z は接空間の T / B / N に対応し、最終的にビュー空間の法線へと変換されてライティングに利用されます。

3.3 計算の流儀：行列とベクトル

webg の Matrix クラスは 4 × 4 行列を使用し、内部配列 mat[16] は列優先（Column-major）で保持されています。set(row, column, val) メソッドが column * 4 + row というインデックス計算を行っているため、メモリ上の並びは以下の通りとなります。

```
| m[0]  m[4]  m[8]  m[12] |
| m[1]  m[5]  m[9]  m[13] |
| m[2]  m[6]  m[10] m[14] |
| m[3]  m[7]  m[11] m[15] |
```

この構成では、平行移動成分は最後の列に格納され、m[12] = tx、m[13] = ty、m[14] = tz となります。また、左上の 3 × 3 部分は主に回転基底として利用されます。CoordinateSystem は position（位置）、quat（回転）、uniform scale（一様スケール）からローカル行列を再構築します。なお、本ライブラリでは、X、Y、Z 軸によって倍率が異なる非一様スケール（non-uniform scale）はサポート対象外としています。

webg では、ベクトルを列ベクトルとして右側に配置する数学的な作法を採用しています。数式としては以下のように表現されます。

```
v' = M * v
```

これを 4 次元同次座標で記述すると、以下の計算になります。

```
|x'|   | m00 m01 m02 tx |   |x|
|y'| = | m10 m11 m12 ty | * |y|
|z'|   | m20 m21 m22 tz |   |z|
|w'|   |  0   0   0   1 |   |1|
```

Matrix.mulVector([x, y, z]) は、この考えに基づき w=1 とした点を掛けています。

```
let x = m[0] * v[0] + m[4] * v[1] + m[8]  * v[2] + m[12];
let y = m[1] * v[0] + m[5] * v[1] + m[9]  * v[2] + m[13];
let z = m[2] * v[0] + m[6] * v[1] + m[10] * v[2] + m[14];
```

webg では行ベクトル（v * M）形式は採用していません。行列合成においても、Matrix.mul(mb) は this = this * mb、Matrix.lmul(mb) は this = mb * this となります。親子階層の計算では world = parent * local となるため、列ベクトル形式では「右側の変換から先に作用する」と理解してください。つまり、parent * local * v という式では、最初に local が点 v に作用し、その結果に parent が作用します。

座標空間の変換フロー

3D のコードを読むときは、値が「どの座標空間にあるのか」を常に意識する必要があります。同じ [1, 2, 3] という数値でも、それがローカル座標なのか、ワールド座標なのか、カメラから見た座標なのかによって意味が変わります。webg の描画では、頂点はおおむね次の順序で変換されます。

```
ローカル座標
  -> ワールド座標
  -> view 座標
  -> clip 座標
  -> NDC
  -> 画面座標
```

ローカル座標は、形状やノード自身から見た座標です。Shape に登録した頂点は、最初はこの座標系にあります。例えば、立方体の中心を原点にして頂点を [-1, -1, -1] から [1, 1, 1] に置く場合、それは「立方体自身から見た座標」です。

ワールド座標は、シーン全体の中での座標です。ノードの位置、回転、スケール、親子構造を反映すると、ローカル座標の頂点はワールド座標へ移ります。親ノードが動けば子ノードのワールド座標も変わりますが、子ノードのローカル座標そのものが書き換わるわけではありません。

view 座標は、カメラから見た座標です。実際にはカメラを原点へ戻し、カメラの向きを基準方向へ戻すような逆変換を、シーン全体にかけた座標系です。webg では視点ノードのワー

ルド行列から view 行列を作り、ワールド座標を view 座標へ変換します。

clip 座標は、透視投影行列を掛けた直後の座標です。この段階ではまだ 2D の画面座標ではなく、`x`, `y`, `z`, `w` の 4 成分を持つ同次座標です。GPU はこの値を使って視錐体の外側にある頂点や三角形をクリップします。

NDC (Normalized Device Coordinates) は、clip 座標を `w` で割った後の正規化された座標です。概念的には、画面の左が `-1`、右が `+1`、下が `-1`、上が `+1` という範囲になります。WebGPU では、深度に対応する NDC の `z` は `0..1` の範囲として扱われます。最後に viewport 変換によって、NDC は実際の canvas 上のピクセル位置、つまり画面座標へ変換されます。canvas のピクセル座標では一般に左上が原点になり、下方向へ `y` が増えるため、3D 空間や NDC の `+Y=上` とは見方が変わる点に注意します。

この変換フローを意識すると、デバッグ時の質問が具体的になります。例えば「モデルの頂点が壊れている」のか、「ノードのワールド変換が間違っている」のか、「view 行列でカメラ基準にした時点で外へ出ている」のか、「projection 後に clip されている」のかを分けて考えられます。矩形選択やマーカ表示のように 3D の点を 2D 画面上へ投影する処理でも、この順序がそのまま使われます。

同次座標と w の役割

3D の行列計算では、位置を `[x, y, z]` の 3 成分だけでなく、`[x, y, z, w]` の 4 成分で扱います。この表現を同次座標と呼びます。通常の点は `w=1` として扱います。これにより、4 × 4 行列の中に回転、拡大縮小、平行移動をまとめて表現できます。

```
point = [x, y, z, 1]
```

一方、方向ベクトルは `w=0` として扱います。

```
direction = [x, y, z, 0]
```

この違いは重要です。点には平行移動が作用しますが、方向には平行移動が作用してはいけません。例えば「光がどちらから来るか」や「法線がどちらを向いているか」は方向であり、カメラやモデルの位置が変わっても、単なる平行移動で向きが変わるわけではありません。`w=0` にすると、4 × 4 行列の平行移動成分が計算に入らないため、方向として正しく変換でき

ます。

透視投影では、この w がさらに重要になります。投影行列を掛けた後の clip 座標は [x, y, z, w] のままで、GPU は最終的に x / w、y / w、z / w を計算します。これを透視除算と呼びます。遠くの点ほど w の影響を受け、画面上では小さく、中心に近づくように写ります。これが遠近感の正体です。

w は普段のアプリケーションコードでは直接意識しないことも多い値ですが、シェーダー、投影、画面座標への変換、選択処理を理解する上では避けて通れません。vec4(position, 1.0) は「これは点である」、vec4(direction, 0.0) は「これは方向である」と読むと、行列計算の意味を追いやすくなります。

3.4 親子構造とローカル座標系

Node やボーンを使い始めたときに混乱を招きやすいのが、「位置や回転はどの座標系で指定しているのか」という点です。webg では、Node もボーンも、単体で空間に浮いているのではなく、親から子へと連なる階層構造の中で扱います。このとき各要素が直接持っているのはワールド座標ではなく、「親から見た自分の位置と向き」、つまりローカル座標系です。

ここを曖昧にしたままコードを書くと、「地球を回転させた際、月だけでなく太陽まで一緒に動いてしまう」といった、意図しない挙動に直面しやすくなります。逆に、親子構造とローカル座標系の意味がつかめると、Node の配置、カメラリグ、ボーン階層、アニメーションのどれも同じ考え方で理解できるようになります。

天体の階層構造による理解

親子構造を最も直感的に理解しやすい題材が、太陽、地球、月の関係です。太陽はそれ自体が自転し、地球は太陽の周りを公転しながら自転し、月は地球の周りを公転しながら自転します。このとき「月の位置」は、月だけの情報では決まりません。地球が今どこにいるか、地球から見て月がどこに置かれているか、その両方が合わさって初めて決まります。

Node で書くと、次のような親子関係になります。

```
sun
└── earthOrbit
    └── earth
```

```
        └──── moonOrbit
            └──── moon
```

ここで earth のローカル位置を [3.5, 0, 0] に設定したとすると、それは「太陽から見て地球が X 方向へ 3.5 離れている」という意味ではありません。正確には、「earthOrbit という親ノードから見て、地球本体が X 方向へ 3.5 離れている」という意味です。earthOrbit 自体を回転させれば、その子である earth は半径 3.5 の円を描いて公転して見えます。

行列として書くと、月のワールド行列は次のように合成されます。

```
sunWorld = sunLocal
earthOrbitWorld = sunWorld * earthOrbitLocal
earthWorld = earthOrbitWorld * earthLocal
moonOrbitWorld = earthWorld * moonOrbitLocal
moonWorld = moonOrbitWorld * moonLocal
```

列ベクトル形式では右側の変換から先に作用するため、moonWorld * v を計算するとき、まず月自身のローカル形状 v に moonLocal が作用し、その後に地球まわりの公転、地球自身の配置、太陽系全体の向きが順番に重なります。「子は親の変換結果の上に積み重なる」と考えると理解しやすくなります。

実際のコードでは、回転のための pivot と見た目の本体を分けると構造的に整理されます。

```
const sun = app.space.addNode(null, "sun");

const earthOrbit = app.space.addNode(sun, "earthOrbit");
const earth = app.space.addNode(earthOrbit, "earth");
earth.setPosition(3.5, 0.0, 0.0);

const moonOrbit = app.space.addNode(earth, "moonOrbit");
const moon = app.space.addNode(moonOrbit, "moon");
moon.setPosition(1.2, 0.0, 0.0);

app.start({
  onUpdate() {
    sun.rotateY(0.2);        // 太陽の自転
    earthOrbit.rotateY(0.8); // 地球の公転
    earth.rotateY(2.5);      // 地球の自転
```

```
    moonOrbit.rotateY(2.2);  // 月の公転
    moon.rotateY(1.6);       // 月の自転
  }
});
```

この構造では、earthOrbit を回しても太陽は動きません。逆に sun を動かせば、その子孫である地球と月はまとめて影響を受けます。これは「親の変換は子へ伝わるが、子の変換は親へ戻らない」という階層構造の基本です。

人体関節への応用と座標伝播

同じ原理は人体の関節でもそのまま使えます。腰が動けば肩も腕も手も一緒に動きます。肩を回せば上腕、前腕、手が動きます。肘を曲げれば前腕と手が動きますが、肩や腰の位置は変わりません。この関係は、まさに親子構造そのものです。

例えば右腕を単純化すると、次のような階層になります。

```
waist
└── torso
    └── shoulder
        └── upperArm
            └── foreArm
                └── hand
```

ここで重要なのは、hand の位置を直接「世界のどこ」として考えるのではなく、「前腕の先端からどれだけ離れているか」として定義することです。そうしておくと、腰の移動、胴体のひねり、肩の回転、肘の曲げがすべて自然に手先へ伝わります。

```
const waist = app.space.addNode(null, "waist");
waist.setPosition(0.0, 0.0, 0.0);

const torso = app.space.addNode(waist, "torso");
torso.setPosition(0.0, 1.8, 0.0);

const shoulder = app.space.addNode(torso, "shoulder");
shoulder.setPosition(0.9, 0.9, 0.0);
```

```
const upperArm = app.space.addNode(shoulder, "upperArm");
upperArm.setPosition(1.2, 0.0, 0.0);

const foreArm = app.space.addNode(upperArm, "foreArm");
foreArm.setPosition(1.1, 0.0, 0.0);

const hand = app.space.addNode(foreArm, "hand");
hand.setPosition(0.6, 0.0, 0.0);
```

この状態で shoulder.rotateZ(...) を呼べば、上腕、前腕、手はまとめて円弧を描いて動きます。さらに foreArm.rotateZ(...) を呼べば、肘から先だけが追加で曲がります。これは「手のワールド行列」が、腰から手までのすべてのローカル行列を掛け合わせた結果だからです。

```
handWorld
  = waistWorld
  * torsoLocal
  * shoulderLocal
  * upperArmLocal
  * foreArmLocal
  * handLocal
```

この見方に慣れておくと、見た目が崩れたときにも「手の local 値が悪いのか」「肩の回転が想定より大きいのか」「腰の向きまで含めた合成結果が問題なのか」という切り分けが容易になります。

ボーン階層における座標変換

ボーンは特別な仕組みに見えますが、階層構造という意味では Node と同じです。スキニングで使うボーンも、「親ボーンから見た子ボーンの位置と向き」をローカル座標で持ち、親の変換を受け継いでワールド姿勢を作ります。

```
const skeleton = new Skeleton();

const waistBone = skeleton.addBone(null, "waistBone");
```

```
const spineBone = skeleton.addBone(waistBone, "spineBone");
const shoulderBone = skeleton.addBone(spineBone, "shoulderBone");
const upperArmBone = skeleton.addBone(shoulderBone, "upperArmBone");
const foreArmBone = skeleton.addBone(upperArmBone, "foreArmBone");
const handBone = skeleton.addBone(foreArmBone, "handBone");

waistBone.setRestPosition(0.0, 0.0, 0.0);
spineBone.setRestPosition(0.0, 1.2, 0.0);
shoulderBone.setRestPosition(0.9, 0.8, 0.0);
upperArmBone.setRestPosition(1.1, 0.0, 0.0);
foreArmBone.setRestPosition(1.0, 0.0, 0.0);
handBone.setRestPosition(0.6, 0.0, 0.0);
```

Node と異なるのは、このボーン階層がそのまま表示物になるのではなく、頂点ごとのウェイトと組み合わされてメッシュ変形に使われる点です。しかし、「親の回転が子へ伝わる」「ワールド姿勢は親と自分のローカル変換の積で決まる」という原理は同一です。第 25 章のスキニングでは、この階層に逆バインド行列と頂点ウェイトが加わることで、メッシュが関節に追従して曲がる仕組みを扱います。

階層構造を読み解くポイント

親子構造でコードを読むときは、次の 3 点を先に確認すると整理しやすくなります。

1. そのノードやボーンの `setPosition()` は「親から見た距離」を置いているのか
2. 回しているノードは「見た目の本体」なのか、それとも「公転や関節回転の pivot」なのか
3. いま知りたい値はローカル座標なのか、`getWorldPosition()` などで得るワールド座標なのか

webg のサンプルやコアを読むときも、この区別を意識すると理解が速くなります。例えば `WebgApp` の camera では `cameraRig -> cameraRod -> eye` という親子構造を使って orbit を実現していますし、スキニングでは root bone から child bone へ姿勢が伝わります。見た目は違っても、座標変換の仕組みは共通です。

本文の内容を実際に動かして確認したい場合は `book/examples/03_01.html` を参照してください。このページでは、左側に太陽・地球・月、右側に腰・肩・腕・手の階層を置き、同じ

「親の変換が子へ伝わる」という原理が別の題材でも同様に成立することを確認できます。さらに、ボーン階層の実際の変形は第 25 章の `book/examples/25_01.html` と `25_02.html` で確認できます。

3.5 透視投影行列と視野角

3D カメラにおいて、位置と姿勢に加えて「どの範囲を写すか」を決定するのが「透視投影行列」です。webg では `Matrix.makeProjectionMatrix(near, far, vfov, ratio)` を使用してこれを生成します。FOV（Field of View）は「視野角」のことで、一度に見える範囲の角度のことです。

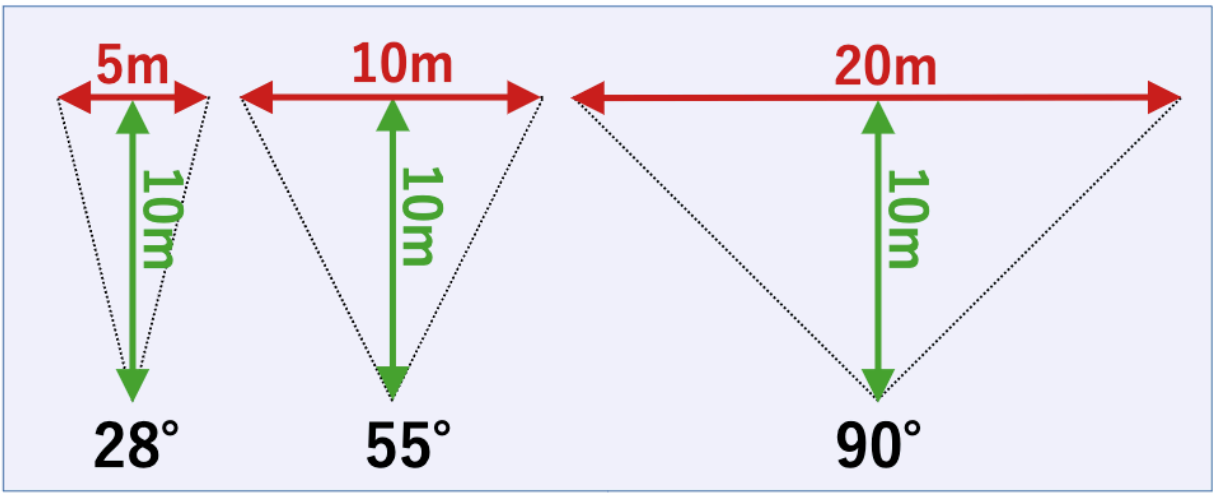


図 3.3: 視野角と見える範囲

FOV が小さいほど望遠寄りで写る範囲は狭くなり、*FOV* が大きいほど広角寄りで写る範囲は広くなります。

```
makeProjectionMatrix(near, far, vfov, ratio) {
  let h = 1.0 / Math.tan(vfov * 0.5 * Math.PI / 180);
  let w = h / ratio;
  let q = 1.0 / (near - far);

  this.makeUnit();
  this.mat[0] = w;
  this.mat[5] = h;
```

```
  this.mat[10] = far * q;
  this.mat[11] = -1.0;
  this.mat[14] = far * near * q;
  this.mat[15] = 0.0;
}
```

ここで重要なのは、vfov が縦方向の視野角である点です。h = 1 / tan(vfov / 2) となるため、視野角を小さくすると h が大きくなり、結果として画面に写る範囲が狭まります。これは写真の焦点距離を長くした状態に近く、望遠寄りの圧縮された見え方になります。逆に視野角を大きくすると、写る範囲が広がり、広角寄りの強い遠近感が生じます。

写真や映像に慣れている人にとっては、視野角をレンズの焦点距離に置き換えると感覚をつかみやすくなります。フルサイズカメラのセンサーは一般に 36mm x 24mm として扱われるため、短辺方向の FOV をフルサイズ換算の焦点距離へ変換する場合は、短辺 24mm を使って次の式で求められます。

```
focalLengthMm = 24 / (2 * tan(fovShort / 2))
```

fovShort は短辺方向の視野角です。横長画面では縦方向が短辺であり、縦長画面では横方向が短辺になります。次の表は、代表的なフルサイズ換算の焦点距離と画角の関係です。

焦点距離 (mm)	水平画角 (度)	垂直画角 (度)
8	132.1	112.6
12	112.6	90.0
16	96.7	73.7
24	73.7	53.1
35	54.4	37.9
50	39.6	27.0
85	23.9	16.0
135	15.2	10.1
200	10.3	6.9

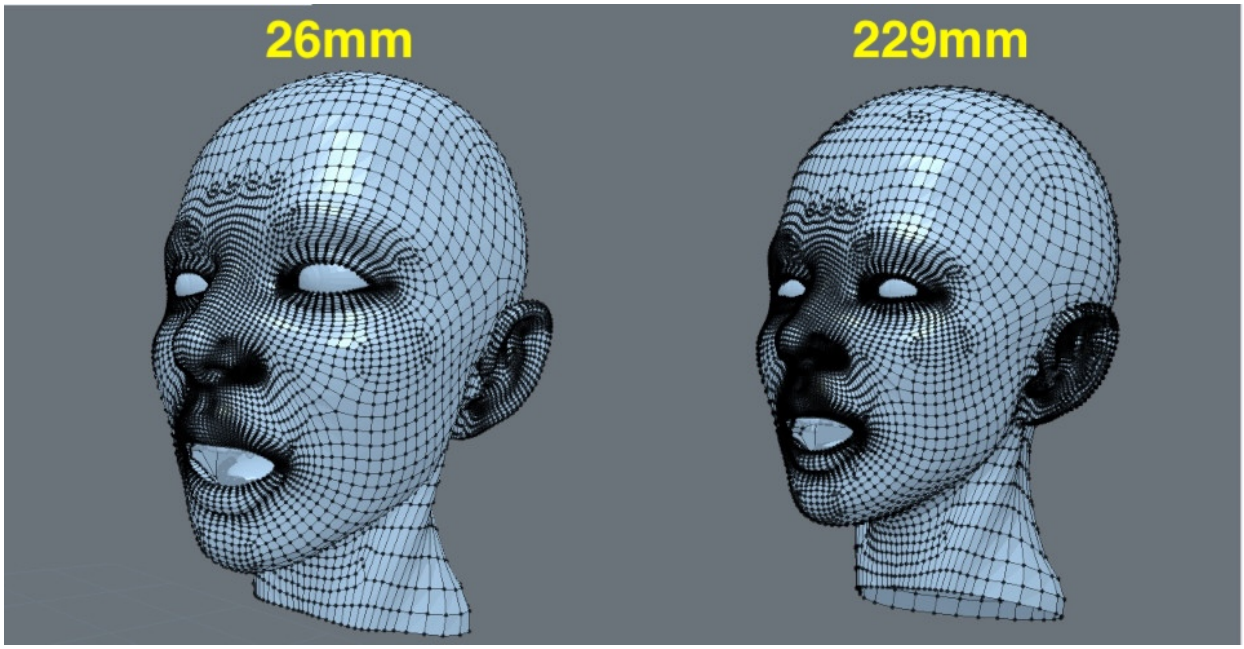


図 3.4: 焦点距離と見え方の違い

焦点距離は、同じ対象を同じ大きさに写したときの遠近感を考える上でも役立ちます。広角側では近くのものが大きく、遠くのものが小さく見えやすくなり、望遠側では奥行き方向の差が圧縮されたように見えます。webg の viewAngle は物理レンズそのものではありませんが、短辺 FOV をこの表に対応させると、ビューアやモデラーで「いまどの程度のレンズ感で見ているか」を判断しやすくなります。

人間の視野は非常に広く、上下方向でもおよそ 120 度、水平方向では 180 度程度まで外界を感じ取れます。しかし、これは「何かが視界に入っている」範囲であり、画面上の大きさや奥行きを自然に判断できる範囲とは同じではありません。写真や映像では、フルサイズ換算で 55mm 程度の標準レンズが、人が対象の大きさを無理なく捉える感覚に近いものとして扱われます。

そのため、3D アプリケーションでは、人間の視野が広いからといって常に大きな FOV を使えばよいわけではありません。モデラー、キャラクターの確認のように狭い範囲を注視するアプリケーションでは、望遠側の FOV にすると形状の比率を読み取りやすくなります。一方、広大な地形、レースゲームといった俯瞰的なシーン演出では、広角寄りの FOV にすることで周囲の情報や移動感を得やすくなりますが、画面隅の歪みは大きくなります。カメラの viewAngle は一度決めたら固定する値ではなく、アプリケーションが何を見せたいかに応じて積極的に調整すべき値です。

また、横方向の範囲は ratio = width / height（アスペクト比）を用いて w = h / rat

io として決定されます。アスペクト比とは画面の横幅を縦幅で割った値です。例えば 16:9 の PC 画面は約 1.78、正方形は 1.0、縦長のスマートフォン画面は 0.5 前後になることがあります。同じ縦 FOV を使っても、横長画面では横方向に広く写り、縦長画面では縦方向に広く写るため、アスペクト比は構図と操作感に強く影響します。

webg の Screen.getRecommendedFov(base) は、アスペクト比の違いによって短辺方向の見える範囲が大きく変わらないようにするためのヘルパー関数です。base は短辺方向の FOV として扱われます。横長画面では縦方向が短辺なので base をそのまま縦 FOV として使い、縦長画面では横方向が短辺になるため、横 FOV が base になるように縦 FOV を逆算します。この仕組みの詳しい式と実装上の扱いは、第 6 章の「視野角と投影行列の管理」で説明します。

near（近面）と far（遠面）は、描画対象とする奥行きの範囲を決定するクリッピング面（裁断面）です。この範囲外のオブジェクトはクリップされ、描画されません。また、この値は深度バッファの精度に影響します。near を極端に小さく、far を極端に大きく設定すると、深度精度が低下し、「Z ファイティング」と呼ばれる面がちらつく現象が発生しやすくなります。シーンの実際の規模に合わせて適切な値に設定する必要があります。

なお、「カメラを近づけて対象を大きく見せること」と「FOV を狭めて大きく見せること」は異なります。カメラを近づけると遠近感が強調されますが、FOV を狭めると位置を変えずに写る範囲だけを制限するため、望遠レンズのような見え方になります。この違いを理解しておくことは、EyeRig 等によるカメラ制御を設計する上で非常に重要です。

深度バッファと描画順

3D シーンでは、手前のものが奥のものを隠します。この前後関係を GPU が判定するために使うのが深度バッファです。深度バッファには、各ピクセルについて「これまでに描かれた最も手前の深度」が記録されます。新しいフラグメントを描こうとしたとき、その深度がすでに記録されている値より手前であれば描画され、奥であれば破棄されます。この判定が depth test です。

深度バッファがあるため、多くの不透明オブジェクトは描画順に強く依存せず、正しく前後関係を保って表示できます。例えば、奥の立方体を先に描いてから手前の立方体を描いても、手前の立方体を先に描いてから奥の立方体を描いても、depth test と depth write が正しく有効であれば、最終的な見え方は大きく変わりません。

ただし、すべての描画が同じ扱いになるわけではありません。選択面、頂点マーカ、ワイヤーフレーム、半透明の面のような補助表示では、depth test を使うか、depth write を行うか、alpha blend を使うかによって見え方が変わります。例えば、通常のシーンを描いた後に

選択面を重ねる場合、選択面が深度を書き込むと、その後に描くマーカや辺の表示を邪魔することがあります。反対に、深度をまったく見ない補助表示は、奥にある要素まで手前に浮いて見えることがあります。

そのため、3D の描画では「どの順序で描くか」と「depth test / depth write / blend をどう設定するか」を組み合わせて考えます。不透明な通常シーンは depth test と depth write を有効にして描き、半透明や編集用 overlay は目的に応じて depth write を無効にする、といった設計がよく使われます。深度バッファは単なる内部処理ではなく、見え方と操作感を決める重要な基礎です。

3.6 視線方向と view 行列

透視投影行列を理解するときは、「カメラがどちらを向いているか」と「その向きに対して視錐体（画面に表示される範囲）がどのように広がるか」を一緒に捉えることが重要です。webg では空間全体の基本軸を +X=右、+Y=上、+Z=前 と定義していますが、視点ノードをカメラとして使うときは、そのノードのローカル座標系における -Z 方向が視線方向になります。したがって、視点ノードに回転が入っていない初期状態では、「カメラの前方」はローカル -Z、「カメラの右」はローカル +X、「カメラの上」はローカル +Y です。

ここは、概念の切り替えが必要なため、混乱を招きやすいポイントです。空間全体の定義では +Z を「前」と呼んでいるのに、カメラは -Z を見るからです。しかしこれは 3D グラフィックスでは一般的な考え方で、カメラを「世界を見るための座標変換」として扱うと自然に理解できます。例えばカメラをワールド座標で z=20 に置き、原点方向を見せたいとき、視点ノードは +Z 側に配置され、そこから -Z 方向へ向いていると考えると整理しやすくなります。

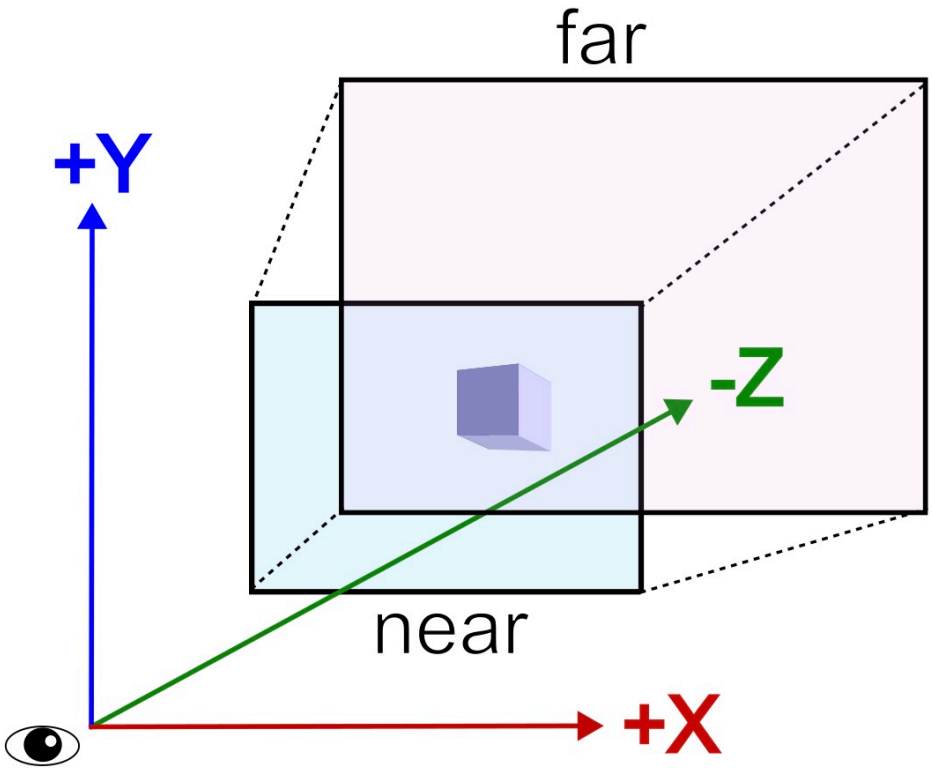


図 3.5: 座標系と視線方向

webg の座標系とは右手系を採用し、$+X$=右、$+Y$=上、$+Z$=前 を基本軸としますが、視線方向は Z 軸の負の方向です。

視錐台は、この視線方向 `-Z` に向かって広がる「先端を切り落とした四角錐」です。視点ノードの近くにある面が `near`、遠くにある面が `far` で、`vfov` は縦方向の開き具合を決めます。`vfov` を大きくすると視錐体は大きく開き、広い範囲が写ります。逆に `vfov` を小さくすると視錐台は細くなり、狭い範囲を望遠寄りに写します。透視投影行列は、この視錐台の内側にある点を画面へ写すための変換です。

`webg` のローレベル（低レイヤー）な実装では、この考え方を `Space.draw(eye_node)` と `Matrix.makeView()` がそのまま受け持っています。描画の開始時には、まず視点ノードのワールド行列を作り、その後でその逆行列を view 行列として使います。

```
eye_node.setWorldMatrix();
let view_matrix = new Matrix();
view_matrix.makeView(eye_node.worldMatrix);
```

Matrix.makeView() の中身は非常に単純で、渡されたワールド行列をコピーして逆行列へ変換しているだけです。

```
makeView(w) {
  this.copyFrom(w);
  this.inverse();
}
```

ここで意味しているのは、「カメラ自身を動かす代わりに、シーン全体をカメラ基準の座標系へ変換する」ということです。例えば、視点ノードがワールド空間で右へ 5 動いていれば、view 行列ではシーン全体が左へ 5 動いたのと同じ効果になります。視点ノードが上を向けば、view 行列ではシーン全体が下を向いたように見える変換になります。つまり view 行列は、「視点ノードの姿勢を打ち消して、世界をカメラから見た座標へ並べ替える行列」と考えると理解しやすくなります。

この変換を通したあとで透視投影行列を掛けると、視錐体の内側にある点だけが正しく画面へ投影されます。流れとしては、**ローカル座標 -> ワールド座標 -> view 座標 -> 透視投影 -> 画面** です。Node の親子構造やボーン階層でどれだけ複雑な変換を行っていても、最終的にはまずワールド座標へ集約され、次に view 行列で「カメラから見た座標」へ変換され、そのあと投影行列で 2D 画面へ写されます。透視投影行列だけを単独で覚えるのではなく、view 行列とセットで理解しておくと、カメラ制御やデバッグ時の見通しが大きく良くなります。

3.7 クォータニオンによる回転表現

クォータニオン（四元数）は、3D の回転を 4 つの要素で表現する手法です。webg の Quat クラスでは、要素の並び順を [w, x, y, z] と定義し、w を実部、x, y, z を虚部として保持します。回転がない状態は単位クォータニオン [1, 0, 0, 0] となります。

```
this.q = [1.0, 0.0, 0.0, 0.0];
```

クォータニオンを採用する主な理由は、オイラー角で発生するジンバルロックを回避でき、回転の合成や球面線形補間（Slerp）を効率的に行えるためです。webg の CoordinateSystem も、内部では姿勢をクォータニオンで保持しており、setAttitude() や rotate() の呼び出し結果を最終的に Quat へ変換して格納します。

実行時（ランタイム）においては、クォータニオンを回転行列へ変換して利用します。CoordinateSystem.setMatrix() では、Quat から 3×3 の回転成分を Matrix.setByQuat() で生成し、そこに平行移動成分を加えてローカル行列を構築します。また、CoordinateSystem.setByMatrix(matrix) では、行列から Quat.matrixToQuat(matrix) を用いてクォータニオンを復元します。

単軸回転の場合、回転軸 $\text{axis} = (ax, ay, az)$ と回転角 θ を用いて、概念的に次のように定義されます。

```
q = [ cos(theta / 2),
      ax * sin(theta / 2),
      ay * sin(theta / 2),
      az * sin(theta / 2) ]
```

webg の setRotateX()、setRotateY()、setRotateZ() は、この定義を各軸に特化させたものです。クォータニオンが回転を正しく表すためには、長さが 1 の「単位クォータニオン」である必要があります。計算を繰り返すと丸め誤差が蓄積するため、Quat.normalize() を用いて長さを補正します。Quat.slerp() の後にも normalize() が呼ばれるのは、回転表現を維持するための基本処理です。

数学的には、ベクトル v をクォータニオン q で回転させる操作は $v' = q * v * q^{-1}$ と表現されますが、webg の通常フローでは、クォータニオンを回転行列に変換して計算する手法が一般的です。

3.8 glTF / GLB との整合性

glTF / GLB 形式のデータを扱う際は、クォータニオンの要素順序に注意が必要です。webg は数学的な表記に基づき実部を先頭とする [w, x, y, z] 形式を採用していますが、glTF 規格では [x, y, z, w] 形式が採用されています。

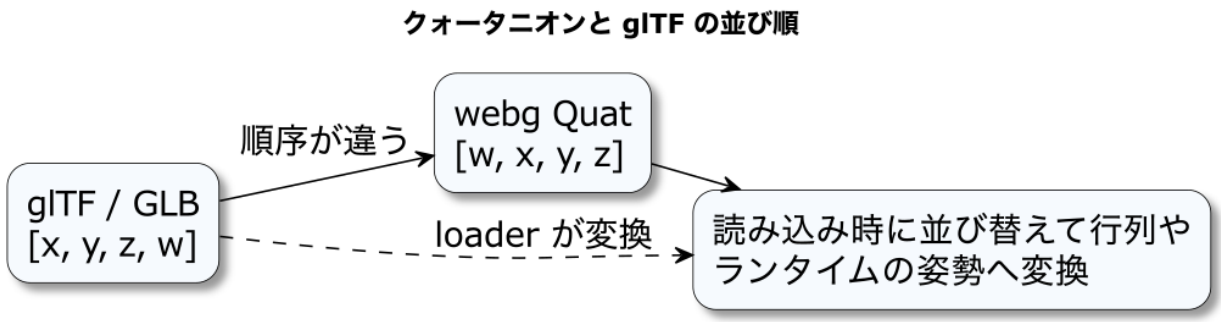


図 3.6: クォータニオンと glTF の並び順

webg のクォータニオンは *[w, x, y, z]*、*glTF / GLB* は *[x, y, z, w]* であるため、読み込み時に順序を変換して *runtime* 姿勢に適用します。

このため、`SceneLoader` や `ModelBuilder` では、glTF のデータを読み込んだ後に要素の並べ替えを行い、`webg` 形式の `Quat` へ変換しています。glTF の値をそのまま `Quat.q` に代入すると回転が正しく適用されないため、ローダー側でこの変換が行われていることを理解しておく必要があります。

また、glTF の TRS（Translation, Rotation, Scale）情報はローダーを通じて読み込まれます。`webg` の `Node` / `CoordinateSystem` は一様スケール（uniform scale）を保持できるため、glTF の変換情報をそのままシーングラフのローカル行列として扱うことが可能です。ただし、非一様スケール（non-uniform scale）は本ライブラリのサポート対象外であるため、この点に留意して実装する必要があります。

3.9 まとめ

本章で解説した通り、`webg` では 3D グラフィックスの各概念を以下のように統一して定義しています。

- 座標系: 右手系（`+X=右`、`+Y=上`、`+Z=前`）
- 回転: `yaw(Y) / pitch(X) / roll(Z)` の順序
- 面の表裏: 表側から見て反時計回りに頂点を登録
- 法線: 面法線は頂点順から決まり、頂点法線は滑らかな陰影に利用する
- 裏面カリング: 頂点順から表裏を判定し、裏側の面を描画対象から外せる
- 行列: 列優先（Column-major）、列ベクトル形式（`M * v`）
- 座標空間: ローカル座標からワールド座標、view 座標、clip 座標、画面座標へ変換する

- 同次座標: 点は w=1、方向は w=0 として扱い、投影後に透視除算を行う
- 親子構造: world = parent * local で合成し、親の変換が子へ伝わる
- クォータニオン: 要素順序 [w, x, y, z]
- UV 座標: 概念上の基準は左下（bottom-left）
- ノーマルマップ: RGB の色情報を方向ベクトル（XYZ）として処理
- 透視投影: near、far、縦 FOV、アスペクト比で定義
- 深度バッファ: depth test / depth write によって前後関係と overlay の見え方が決まる

これらの個別概念をバラバラに捉えるのではなく、相互に関連する一つのシステムとして理解することで、今後の Node 制御、カメラ操作、モデル読み込み、スキニングなどの実装が非常にスムーズになります。

次章では、これらの基礎定義を前提として、Screen、Space、Shape、Matrix、SmoothShader を連携させた、最初の描画フローについて解説します。

第 4 章

WebGPU と webg の最小描画

この章では、webg を用いて最初の 3D アプリケーションを動作させるまでの流れを具体的に解説します。

webg を使い始める際に重要なのは、API を網羅的に暗記することではなく、「どのような順序でプログラムを組み立てればよいか」という全体像を理解することです。そのため本章では、まず「描画の基礎的な構造（ローレベル）」を実装して動作を確認し、その後に「WebgApp を用いた標準的なアプリケーション構成」へと進む段階的なアプローチをとります。

これにより、どこまでが描画の土台であり、どこからがアプリケーションとしての補助機能なのかを明確に切り分けることができます。

なお、サンプルの起動方法やリポジトリの配置、ローカルサーバーの構築方法を確認したい場合は、先に第 2 章「インストールと実行環境」を参照してください。

4.1 最初のアプリを構築するアプローチ

3D アプリケーションの開発では、描画の土台、カメラ、入力、HUD、診断情報、アセットの読み込みなど多くの要素を最初から盛り込むと、何も映らないなどの不具合が発生した場合に原因の切り分けが困難になります。

そこで webg では、以下の段階的な手法を推奨しています。

1. ローレベル（低レイヤー）構成: Screen + Space + Node + Shape という最小単位で描画の仕組みを理解する。

2. 標準構成 (WebgApp): 補助機能（カメラリグ、HUD、入力管理など）を統合し、効率的に開発する。

特に、await screen.ready による待機と shape.endShape() によるバッファの確定は、非常に重要なステップです。これらを怠ると、描画が表示されない、あるいは GPU バッファが正しく確定しないといった初歩的な問題に直面することになります。

4.2 WebGPU API から見た webg の低レイヤー

したがって、本章の最小描画を理解するうえでは、「ブラウザ標準の WebGPU API で何をしなければならないか」と、「それを webg がどのクラスへ分担しているか」を対応づけて読むと全体像がつかみやすくなります。WebGPU のネイティブ API では、まず GPU アダプタとデバイスを取得し、canvas に対応するコンテキストを作成し、そこへレンダーパイプライン、バッファ、シェーダー、描画コマンドを順に接続していきます。概念的には「描画先を準備する」「GPU に渡すデータを作る」「どのシェーダーでどう描くかを決める」「コマンドを発行して表示する」という流れになります。たとえば最小描画においても、WebGPU 側では次のような処理が行われています。

- GPUDevice の取得: GPU へコマンドを送るための本体を用意する
- GPUCanvasContext の設定: canvas を描画先として結び付ける
- シェーダーとパイプラインの作成: 頂点処理とピクセル処理の流れを GPU 側へ定義する
- 頂点バッファやインデックスバッファの作成: 形状データを GPU メモリへ渡す
- コマンドエンコーダとレンダーパスの発行: どのフレームで何を描くかを記録し、最後に送信する

これらをすべてアプリケーション側で直接扱うのは、単純な立方体を描画する段階であっても非常に冗長です。さらに、実用的な 3D アプリケーションでは、canvas の初期化だけでなく、リサイズへの追従、深度バッファの用意、投影行列の更新、シーン全体の描画順序、複数形状の管理まで必要になります。webg のローレベル (低レイヤー) は、こうした煩雑な処理をクラス単位で分担しています。

具体的に、本章で最初に使用する Screen は、WebGPU における GPUCanvasContext、描画サイズ、クリア、プレゼント、レンダーパス開始といった「描画先まわり」の処理を引き受ける入口です。await screen.ready を待つという操作は、内部で WebGPU の利用準備が

整うまで待機していることを意味しています。また、screen.clear() と screen.present( ) は、WebGPU でいうところのフレームごとのレンダーパスの開始と表示処理に相当します。

Shape は、WebGPU に渡す頂点データや材質設定をまとめる層です。Primitive.cube() のような形状生成結果を Shape へ適用し、endShape() を呼ぶことで、CPU 側で保持していた形状情報が GPU バッファとして確定します。したがって shape.endShape() を忘れると描画されないのは、WebGPU へ渡すべき頂点データが未完成であるためです。シェーダーについては、SmoothShader のようなクラスが WebGPU のパイプライン設定と WGSL シェーダーをまとめて扱います。アプリケーション側からは shader.init()、setP rojectionMatrix()、setLightPosition() といったメソッドを呼び出すだけで操作できま すが、その背後では WebGPU に必要なシェーダーモジュール、パイプライン、uniform 相当 の更新が行われています。

Node は物体の 3D 空間での位置や姿勢や形状、さらに移動や回転といった 3 次元の動作を担当します。

Space は WebGPU の特定の API に直接対応するものではありませんが、シーン内にある Node と Shape を集約し、どの視点から何を描画するかを整理するための描画管理層です。WebGPU のネイティブ API は「何をいつ描くか」を命令として積み上げる仕組みですが、webg ではそれをシーン単位で扱えるように設計されています。space.draw(eye) は、その時点のシーン構造を走査して必要な描画命令を発行する入口となります。

このように、本章の最小描画は WebGPU の物理的な処理を Screen、Shape、シェーダークラス、Node、Space というローレベル (低レイヤー) の部品へ整理して実装しているものです。

4.3　最初のアプリを構築する標準フロー

図 4.1: 最小描画から WebgApp への流れ

webg でアプリケーションを構築する際は、以下の順序で進めることでスムーズに組み立てられます。

1. Screen を生成し、await screen.ready で準備完了を待機する。
2. シェーダーを初期化し、プロジェクション行列（projection）を設定する。
3. Space を生成し、視点となるノード（eye）を用意する。
4. Shape を生成し、endShape() を呼び出して形状を確定させる。
5. clear → draw → present の描画ループを構築する。

4.4　ローレベル（低レイヤー）の最小実装例

まずは、最小構成で立方体を 1 つ描画する例を確認します。ここでは「webg で描画を実現するために本当に必要な最小限の骨格は何か」を明らかにします。

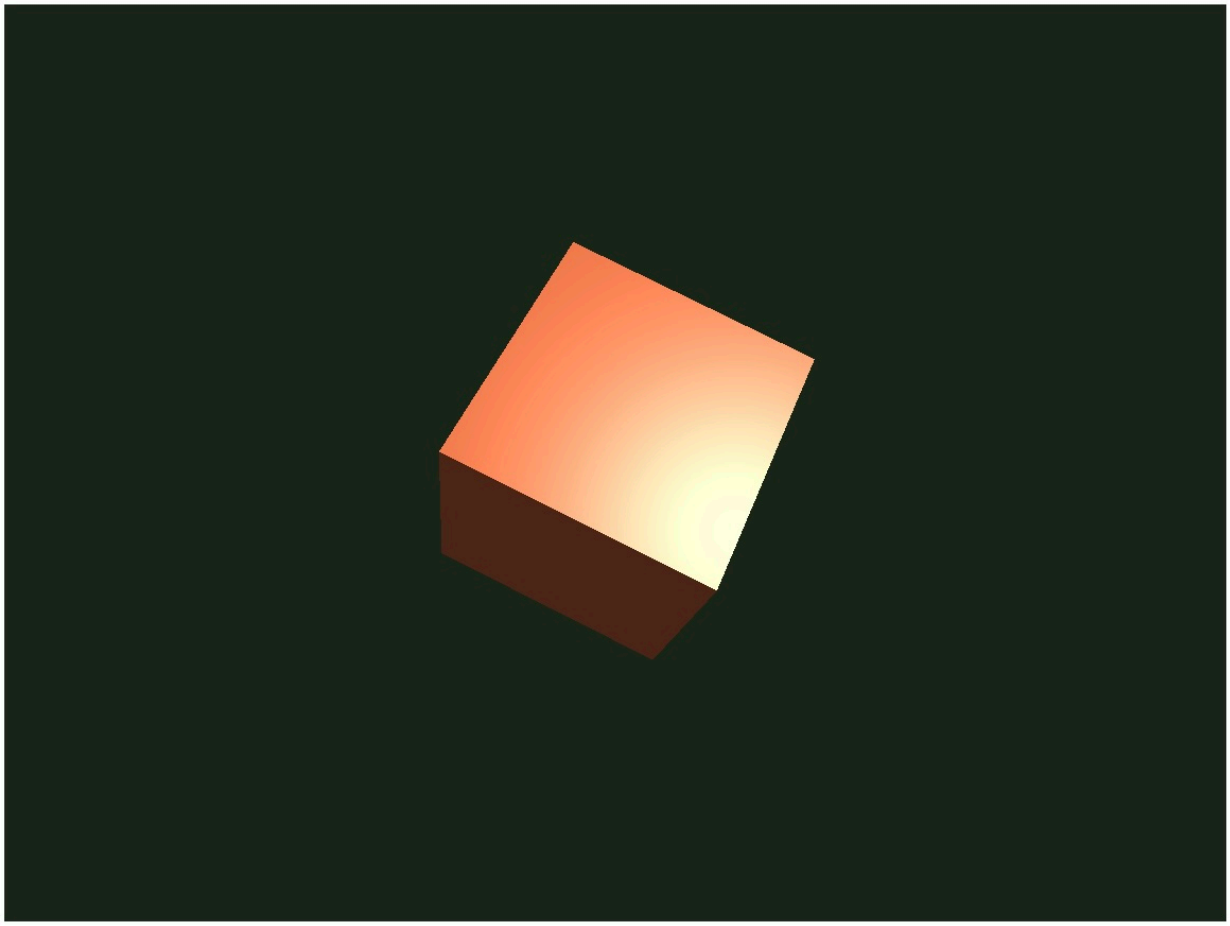


図 4.2: ローレベルの最小実装例

ここでは webg を展開したフォルダの中に「user」というフォルダを作成することにします。

ファイル配置

webg フォルダの中に user/lowlevel フォルダを作成して、index.html と main.js を配置します。

```
--+-- webg/
  |
  +-- user/
       |
       +-- lowlevel/
            |
            +- index.html
```

```
            +- main.js
```

index.html

表示するページを index.html という名前で用意します。html のファイル名は任意のファイル名で構いません。

```
<!DOCTYPE html>
<html lang="ja">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Low Level</title>
</head>
<body style="margin:0; overflow:hidden;">
  <canvas id="canvas" width="720" height="540"></canvas>
  <script type="module" src="./main.js"></script>
</body>
</html>
```

main.js

JavaScript のファイルを main.js をいう名前で用意します。 ファイルを置くフォルダの構成を変更する場合は、import 文の相対パスを適切に修正して下さい。

```
import Screen from "../../webg/Screen.js";
import Space from "../../webg/Space.js";
import Primitive from "../../webg/Primitive.js";
import Shape from "../../webg/Shape.js";
import Matrix from "../../webg/Matrix.js";
import SmoothShader from "../../webg/SmoothShader.js";

const screen = new Screen(document);
await screen.ready;
screen.setClearColor([0.1, 0.15, 0.1, 1.0]);
```

```
const shader = new SmoothShader(screen.getGPU());
await shader.init();
Shape.prototype.shader = shader;

const projection = new Matrix();
projection.makeProjectionMatrix(0.1, 1000.0, 55.0, screen.getAspect());
shader.setProjectionMatrix(projection);
shader.setLightPosition([120.0, 180.0, 140.0, 1.0]);

const space = new Space();
const eye = space.addNode(null, "eye");
eye.setPosition(0.0, 0.0, 28.0);

const shape = new Shape(screen.getGPU());
shape.applyPrimitiveAsset(Primitive.cube(8.0, shape.getPrimitiveOptions()));
shape.endShape(); // 重要：ここで GPU バッファを確定させる
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [1.0, 0.5, 0.3, 1.0]
});
const node = space.addNode(null, "obj");
node.addShape(shape);

const loop = () => {
  node.rotateY(0.8);
  node.rotateX(0.4);
  screen.clear();
  space.draw(eye);
  screen.present();
  requestAnimationFrame(loop);
};
loop();
```

このコードを読み解く際は、以下の順序を意識してください。まず Screen とシェーダーを初期化し、次にビューポートとプロジェクション行列を同期させ、その後に Space と視点（eye）を作成します。続いて Shape を定義し、endShape() で GPU バッファを確定させ、最後に clear → draw → present の順で毎フレームの描画処理を行います。

この骨格を理解しておくことで、後述する WebgApp を利用した際に「どの処理が自動化さ

れているのか」を明確に切り分けることができます。

4.5 WebgApp による標準実装例

次に、先ほどの骨格を WebgApp で実装した例を見てみましょう。WebgApp を使うと、Screen、Space、入力、HUD などの土台部分が内部に集約され、記述が大幅に簡略化されます。

```
import WebgApp from "../../webg/WebgApp.js";
import Shape from "../../webg/Shape.js";
import Primitive from "../../webg/Primitive.js";

const app = new WebgApp({
  messageFontTexture: "../../webg/font512.png",
  clearColor: [0.1, 0.15, 0.1, 1.0],
});
await app.init();

const orbit = app.createOrbitEyeRig({
    target: [0.0, 0.0, 0.0],
    distance: 8.0,
    yaw: 24.0,
    pitch: -12.0,
    minDistance: 4.0,
    maxDistance: 18.0,
    wheelZoomStep: 1.0
});

const shape = new Shape(app.getGPU());
shape.applyPrimitiveAsset(Primitive.cube(2.0, shape.getPrimitiveOptions()));
shape.endShape();

shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [1.0, 0.5, 0.3, 1.0]
});

const obj = app.space.addNode(null, "obj");
obj.addShape(shape);

app.start({
```

```
  onUpdate: ({ deltaSec }) => {
    obj.rotateY(0.8);
    obj.rotateX(0.4);
  }
});
```

この例では、Screen、Space、カメラリグ、入力、HUD などの土台部分が WebgApp 内部に集約されています。この例では立方体が回転しているだけではなく、マウスのドラッグで視点が立方体の周りを周回したり、平行移動することも可能です。また、「F9」キーの後に「M」キーを押すことでデバッグモードへ切り替えて詳細な情報を取得することができます。

実用的なアプリケーションを構築する場合は、この WebgApp の構成をベースにするのが自然です。WebgApp の詳細については次章で解説します。

4.6　サンプルの効果的な読み方

webg のサンプルを確認する際は、以下の順序で読み解くことで理解が深まります。

1. webg/samples/index.html で全体像を把握する。
2. 対応する webg/samples/*/*.txt（解説ファイル）を先に読み、そのサンプルの目的と仕様を理解する。
3. main.js を開き、「初期化 → 入力 → 更新 → 描画 → HUD」の順に処理を追う。

特におすすめのサンプルは、low_level（最小構成）、high_level（WebgApp 構成）、scene（統合例）、shapes（形状比較）です。

4.7　開発時の留意事項（チェックリスト）

実装の初期段階で陥りやすい注意点をまとめました。動作しない場合はここを確認してください。

- ☒ await screen.ready の待機を忘れていないか。
- ☒ shape.endShape() を呼び出し、GPU バッファを確定させているか。
- ☒ ウィンドウのリサイズ後にプロジェクション行列（projection）を更新しているか。

- ☒ `event.key` を小文字化して比較しているか。
- ☒ 操作説明を画面（HUD）に表示し、ユーザーが操作方法を把握できる状態にあるか。
- ☒ `console.log()` だけでなく、診断情報レポート（Diagnostics）を活用しているか。
- ☒ サンプルの `*.txt` ファイルを読み、実装の意図を正しく理解しているか。

4.8 まとめ

本章で最も重要なのは、「最初の描画を出すための骨格」と「そこから標準的なアプリケーション構成へ移行する流れ」を分けて理解することです。

ローレベルの最小例では、`Screen` → `Shader` → `Space` → `Shape` → `Loop` という一連の順序が土台となります。一方で `WebgApp` は、その土台の上にカメラ、入力、HUD、診断情報といった実用的な機能を統合的に提供します。

この構造を理解することで、以降の章で扱うモデル読み込み、アニメーション、UI、物理エンジンなどの各機能が、アプリケーションを構成する階層的な要素として理解しやすくなるはずです。

次は、第 5 章「WebgApp によるアプリ構成」へ進み、標準構成の詳細を深掘りしましょう。

第 5 章

WebgApp によるアプリ構成

5.1 WebgApp で何が簡単になるのか

第 4 章では、Screen、SmoothShader、Space、Shape、eye を順番に用意し、最後に clear -> draw -> present を呼び出すことで、WebGPU 画面へ 3D オブジェクトを描画しました。

この流れは webg の土台を理解するうえで重要です。一方で、実際のアプリケーションでは、画面の初期化、標準シェーダー、シーン、カメラ、入力、HUD、診断情報、リサイズ対応、フレームループといった周辺処理を毎回組み立てることになります。

WebgApp は、この共通部分をまとめた高レイヤー（ハイレベル）の入口です。WebgApp を使うと、開発者は「何を置くか」「毎フレームどう動かすか」に集中しやすくなります。ただし、WebgApp は何でも持った巨大な便利 API ではありません。アプリケーションの土台を作ることが主な役割です。ヘルプ、エラー表示、チュートリアル、ゲームルール、メニュー構造などは、OverlayPanel やサンプル側の controller / helper と組み合わせて作ります。

5.2 最小の WebgApp アプリ

WebgApp を使う最小の流れは、次の 4 段階です。

1. new WebgApp(...) で設定を渡す。
2. await app.init() で GPU、シーン、カメラ、入力、HUD を準備する。
3. app.space にオブジェクトを置く。

4. app.start({ onUpdate }) でフレームループを開始する。

```
import WebgApp from "../../webg/WebgApp.js";
import Shape from "../../webg/Shape.js";
import Primitive from "../../webg/Primitive.js";

const app = new WebgApp({
  document,
  messageFontTexture: "../../webg/font512.png",
  clearColor: [0.1, 0.15, 0.1, 1.0],
  camera: {
    target: [0.0, 0.0, 0.0],
    distance: 8.0,
    yaw: 24.0,
    pitch: -12.0
  }
});

await app.init();

const shape = new Shape(app.getGPU());
shape.applyPrimitiveAsset(Primitive.cube(2.0, shape.getPrimitiveOptions()));
shape.endShape();
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [1.0, 0.5, 0.3, 1.0]
});

const box = app.space.addNode(null, "box");
box.addShape(shape);

app.start({
  onUpdate({ deltaSec }) {
    box.rotateY(0.8 * deltaSec);
    box.rotateX(0.3 * deltaSec);
  }
});
```

この例で開発者が直接書いているのは、立方体の形状を作る処理、シーンへ配置する処理、毎フレーム回転させる処理です。Screen、標準シェーダー、Space、カメラ、入力管理、HUD、

診断機能、requestAnimationFrame の予約は WebgApp がまとめて扱います。

5.3 基本ライフサイクル

WebgApp の利用順序は、次の形で覚えておくと安全です。

```
const app = new WebgApp(options);
await app.init();

// app.getGPU(), app.space, app.eye, app.input, app.message が使える
// Shape や Node を作り、シーンを組み立てる

app.start({
  onUpdate(ctx) {
    // 毎フレームの更新
  }
});
```

constructor は設定を保持する

new WebgApp(options) の時点では、GPU デバイス、Screen、Space、標準シェーダー、eye はまだ利用できません。constructor は、後続の init() で使う設定と内部状態を用意する段階です。

代表的な option は次の通りです。

option	役割
document	DOM 操作に使用する document
messageFontTexture	HUD 文字表示に使うフォントテクスチャ
clearColor	背景色
camera	標準カメラの初期状態
viewAngle	投影行列の視野角
light	標準ライト設定
fog	標準フォグ設定
attachInputOnInit	init() 内で入力を接続するか
autoDrawScene	フレーム内で space.draw(app.eye) を自動実行するか
autoDrawBones	スケルトンボーンを自動描画するか
layoutMode	viewport または embedded
renderMode	ondemand または continuous
uiTheme	DebugDock / OverlayPanel のテーマ

init の後にアプリの土台が使える

await app.init() は、WebgApp における大きな境界線です。ここで Screen の準備を待ち、シェーダー、Space、標準カメラリグ、入力、HUD、診断機能などを作成します。

init() 後によく使うものは次の通りです。

プロパティ / メソッド	用途
app.getGPU()	Shape や低レイヤーリソース作成に使う GPU context
app.space	ノード、形状、モデルを配置するシーン
app.eye	現在の視点ノード
app.input	入力状態
app.message	HUD / メッセージ表示
app.shader	標準シェーダー
app.screen	描画先の Screen

app.getGPU()、app.space、app.eye などは、必ず await app.init() の後に使ってください。init() 前には実体がまだ作られていないためです。

5.4 app.start と onUpdate

app.start() は、WebgApp のフレームループを開始します。内部では requestAnimationFrame から渡される時刻を受け取り、前フレームからの経過時間を計算し、更新、描画、HUD、提示を順番に進めます。

onUpdate は、その中で「描画前にシーンの状態を 1 フレームぶん進める場所」です。画面へ描く処理そのものではなく、次に描かれるべき状態を作る処理を書きます。

```
app.start({
  onUpdate(ctx) {
    box.rotateY(0.8 * ctx.deltaSec);
  }
});
```

たとえば次のような処理は、onUpdate に書くのが自然です。

- 入力を見てプレイヤーを動かす。
- ノードの位置や回転を更新する。
- タイマーやクールダウンを進める。
- アプリのフェーズに応じて処理を分ける。
- HUD に表示する数値や状態を更新する。

一方で、次の処理は別の場所に置くと整理しやすくなります。

処理	書く場所
初期化、Shape 作成、モデル読み込み	await app.init() の後、app.start() の前
毎フレームの状態更新	onUpdate
3D シーン描画前の特殊な描画	onBeforeDraw
3D シーン描画後のポストプロセス	onAfterDraw3d
HUD 描画後の追加表示	onAfterHud

onUpdate() が true を返すと、フレームループは停止します。

```
app.start({
  onUpdate() {
    if (gameOver) {
      return true;
    }
    return false;
  }
});
```

5.5 フレームコンテキスト ctx

onUpdate(ctx) の ctx は、そのフレームで使いやすい情報をまとめたフレームコンテキストです。WebgApp が毎フレーム作成し、onUpdate、onBeforeDraw、onAfterDraw3d、onAfterHud に渡します。

主なプロパティは次の通りです。

プロパティ	意味と使いどころ
app	現在の WebgApp 自身。HUD、フェーズ、スクリーンショットなどに使う
scenePhase	"title"、"gameplay"、"result" などの大まかな進行状態
timeMs	requestAnimationFrame から渡された時刻。単位はミリ秒
timeSec	timeMs を秒にした値。周期演出に使いやすい
deltaSec	前フレームからの経過秒数。移動、回転、タイマー更新に使う
screen	描画先の Screen
shader	標準シェーダー
space	シーン内のノードや形状を管理する Space
eye	現在のカメラ視点
cameraRig	カメラ全体の土台
cameraRod	カメラ距離を表すアーム
cameraTarget	現在のカメラ注視点の配列コピー
cameraFollow	カメラ追従状態
input	キー、ポインター、アクションの入力状態
projection	現在の projection matrix

最初のうちは、deltaSec、timeSec、input、space、app を中心に見ると十分です。

```
app.start({
  onUpdate(ctx) {
    if (ctx.input.has("arrowright")) {
      player.move(3.0 * ctx.deltaSec, 0.0, 0.0);
    }

    box.rotateY(1.2 * ctx.deltaSec);

    ctx.app.message.setLines("status", [
      `phase: ${ctx.scenePhase}`,
      `time: ${ctx.timeSec.toFixed(1)}`
    ]);
```

```
  }
});
```

onUpdate({ deltaSec }) という書き方

サンプルでは、次のような書き方もよく使います。

```
app.start({
  onUpdate({ deltaSec }) {
    box.rotateY(0.8 * deltaSec);
  }
});
```

これは webg 独自の構文ではなく、JavaScript の分割代入です。次のコードと同じ意味です。

```
app.start({
  onUpdate(ctx) {
    const deltaSec = ctx.deltaSec;
    box.rotateY(0.8 * deltaSec);
  }
});
```

複数の値を取り出すこともできます。

```
app.start({
  onUpdate({ deltaSec, timeSec, input, app }) {
    if (input.has("space")) {
      app.setScenePhase("jump");
    }

    box.setPosition(0.0, Math.sin(timeSec * 2.0) * 0.5, 0.0);
    box.rotateY(1.0 * deltaSec);
  }
});
```

ctx 全体を何度も使う場合は onUpdate(ctx)、必要な値が少ない場合は onUpdate({ deltaSec, input }) のように書くと読みやすくなります。

deltaSec を使う理由

deltaSec は、前フレームから現在のフレームまでに経過した秒数です。

毎フレーム固定値で回転させると、フレームレートによって速度が変わります。

```
// 1 フレームごとに 0.02 回す。
// 30fps と 144fps では 1 秒あたりの回転量が変わる。
box.rotateY(0.02);
```

deltaSec を掛けると、「1 秒あたりどれだけ進むか」という指定になります。

```
// 1 秒あたり 1.2 回す。
box.rotateY(1.2 * deltaSec);
```

移動やタイマーも同じ考え方です。

```
app.start({
  onUpdate({ deltaSec, input }) {
    if (input.has("arrowright")) {
      player.move(3.0 * deltaSec, 0.0, 0.0);
    }

    cooldown -= deltaSec;
  }
});
```

timeSec は「アプリ全体の時刻」に基づく演出に向いています。

```
app.start({
  onUpdate({ timeSec }) {
    box.setPosition(0.0, Math.sin(timeSec * 2.0) * 0.5, 0.0);
```

```
  }
});
```

deltaSec は「このフレームで何秒ぶん進めるか」、timeSec は「いま全体時間の何秒目か」と考えると使い分けやすくなります。

物理エンジンに渡す時間

PhysicsSpace.step(deltaMs) や Scene JSON runtime の sceneRuntime.stepPhysics(deltaMs) は、名前の通りミリ秒単位の deltaMs を受け取ります。一方、WebgApp の ctx には秒単位の deltaSec が入っています。

そのため、onUpdate から物理を進める場合は、deltaSec を 1000 倍して渡します。

```
import PhysicsSpace from "../../webg/PhysicsSpace.js";

const physics = new PhysicsSpace({
  fixedTimeStepMs: 1000.0 / 60.0,
  maxSubSteps: 5
});

app.start({
  onUpdate({ deltaSec }) {
    const deltaMs = deltaSec * 1000.0;
    physics.step(deltaMs);
  }
});
```

PhysicsSpace.step(deltaMs) は、渡された可変の deltaMs を内部の accumulator に溜め、fixedTimeStepMs ごとに stepFixed(dtSec) を必要回数だけ実行します。つまり、onUpdate 側では「前フレームから何ミリ秒経ったか」を渡し、物理空間側が安定しやすい固定ステップへ分配します。

step() の戻り値は、実際に進めた fixed step の回数です。長い停止から復帰した場合などは、maxSubSteps によって一度に進める回数が抑えられます。

```
app.start({
  onUpdate({ deltaSec }) {
    const deltaMs = Math.min(deltaSec * 1000.0, 80.0);
    const stepCount = physics.step(deltaMs);

    app.message.setLine("physics", `physics steps: ${stepCount}`);
  }
});
```

通常の移動や回転は deltaSec、PhysicsSpace.step() は deltaMs、stepFixed(dtSec) の内部処理は秒単位、という単位の違いに注意してください。物理の詳細は第 26 章で扱います。

5.6 入力を扱う

WebgApp は内部に InputController を保持しています。attachInputOnInit が true（既定値）の場合、init() 内で app.attachInput() が自動的に呼ばれます。

ctx.input を使うと、押されているキーを毎フレーム確認できます。

```
app.start({
  onUpdate({ deltaSec, input }) {
    if (input.has("arrowleft")) {
      player.move(-3.0 * deltaSec, 0.0, 0.0);
    }
    if (input.has("arrowright")) {
      player.move(3.0 * deltaSec, 0.0, 0.0);
    }
  }
});
```

キー名で直接判定するのではなく、抽象化されたアクション名で扱いたい場合は registerActionMap() を使います。

```
app.registerActionMap({
  jump: ["space", "enter"],
```

```
  reset: ["r"]
});

app.start({
  onUpdate() {
    if (app.wasActionPressed("jump")) {
      player.jump();
    }
    if (app.wasActionPressed("reset")) {
      resetStage();
    }
  }
});
```

wasActionPressed() は「押された瞬間」を扱う用途に向いています。押されている間ずっと移動させたい場合は、input.has() のような継続入力を使います。

app.attachInput() は、単に入力を接続するだけでなく、F9 を接頭辞とするデバッグキーも処理します。既定では次の順次入力が使えます。

操作	意味
F9 -> M	デバッグ / リリースモードの切り替え
F9 -> C	診断サマリーをコピー
F9 -> V	診断 JSON をコピー

独自のキーハンドラを追加する場合も、デバッグキーを維持したいなら app.attachInput() を使うのが基本です。

```
app.attachInput({
  onKeyDown: (key, ev) => {
    if (ev.repeat) return;
    if (key === "s") {
      app.takeScreenshot({ prefix: "sample" });
    }
  }
});
```

5.7 カメラの基本

WebgApp の標準カメラは、cameraRig -> cameraRod -> eye の 3 段構成です。

- cameraRig: 注視点や全体回転を持つ土台。
- cameraRod: カメラまでの距離を表すアーム。
- eye: 実際の視点ノード。

constructor の camera option で初期状態を指定できます。

```
const app = new WebgApp({
  document,
  camera: {
    target: [0.0, 0.0, 0.0],
    distance: 8.0,
    yaw: 24.0,
    pitch: -12.0,
    roll: 0.0
  }
});
```

init() の中でこの構成が作られ、app.eye が app.space の視点として登録されます。

マウスやタッチで周回できる Orbit カメラが必要な場合は、await app.init() の後に createOrbitEyeRig() を呼び出します。

```
app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0,
  minDistance: 4.0,
  maxDistance: 18.0
});
```

このメソッドは標準リグの上に EyeRig を構築し、ポインター操作も接続します。WebgApp

が毎フレーム EyeRig を更新するため、サンプル側で個別に update() を呼ぶ必要はありません。

位置追従、即時位置合わせ、シェイクには次の補助機能があります。ここでの followNode() は cameraRig の基準位置を対象へ近づける機能です。独立したカメラ位置から対象へ視線だけを向ける EyeRig Follow とは責務が異なります。

- followNode(): 対象ノードの位置へ cameraRig を滑らかに追従させる。
- lockOn(): 対象位置へ cameraRig を即時に合わせる。
- clearCameraTarget(): 追従やロックオンを解除する。
- shakeCamera(): 短い衝撃演出を発生させる。

カメラ制御の詳細は第 6 章で扱います。

5.8 ライトとフォグ

WebgApp の標準ライトは、教材サンプルやビューアで対象を確認しやすい eye-fixed が既定です。

```
const app = new WebgApp({
  document,
  light: {
    mode: "eye-fixed",
    position: [120.0, 180.0, 140.0, 1.0],
    type: 1.0
  }
});
```

シーン内の特定ノードに結び付いたライトにしたい場合は world-node を使います。

```
const app = new WebgApp({
  document,
  light: {
    mode: "world-node",
    nodeName: "sunLight",
    position: [80.0, 120.0, 60.0, 1.0],
```

```
    type: 1.0
  }
});
```

init() 後に設定を切り替える場合は、setEyeLight() または setWorldLight() を使用します。フォグは constructor option と setFog() の両方で指定できます。

5.9 HUD と Overlay

WebgApp は、Canvas 上へ描く HUD 用に app.message と app.hudMessage を持っています。

短い状態表示には app.message.setLine() / setLines() が便利です。

```
app.start({
  onUpdate({ app, deltaSec }) {
    app.message.setLines("status", [
      "WebgApp sample",
      `delta: ${deltaSec.toFixed(3)} sec`
    ], {
      anchor: "top-left",
      x: 0,
      y: 0
    });
  }
});
```

項目名と値を並べたい場合は setHudRows() や setControlRows() を使います。短時間の通知には pushToast() や flashMessage() を使います。

長い説明、ヘルプ、エラー、選択肢付きのパネルには DOM ベースの OverlayPanel を使います。

```
app.showOverlayPanel({
  id: "help",
  title: "Help",
```

```
  lines: [
    "Drag: orbit",
    "R: reset"
  ],
  anchor: "top-left",
  collapsible: true
});
```

同じ id で showOverlayPanel() を呼ぶと、既存のパネルが更新されます。隠す場合は hideOverlayPanel()、削除する場合は removeOverlayPanel() を使います。

ヘルプやエラー表示の定型オプションが必要な場合は、OverlayPanelPresets.js の helper を使います。

```
import { buildHelpPanelOptions } from "../../webg/OverlayPanelPresets.js";

app.showOverlayPanel(buildHelpPanelOptions({
  id: "help",
  lines: ["Drag: orbit", "R: reset"]
}));
```

WebgApp はヘルプ専用 API、エラー専用 API、会話専用 API を持ちません。表示の枠は OverlayPanel、文章のキューや分岐はアプリ側 controller、という分担にします。

5.10 必要になったら使う機能

WebgApp には、最小アプリを越えた機能も統合されています。最初からすべてを覚える必要はありません。必要になったところから使います。

機能	用途
`layoutMode: "viewport"`	canvas と overlay を画面全体基準で配置する
`layoutMode: "embedded"`	書籍や教材ページ内に canvas を埋め込む
Diagnostics	環境チェックやランタイム警告を記録する
DebugDock	診断情報を開発用 UI として表示する
`loadModel()`	glTF / Collada / ModelAsset JSON を読み込む
`loadScene()`	Scene JSON を読み込む
`validateScene()`	Scene JSON の妥当性を確認する
`createTween()`	値を時間で補間する
`createParticleEmitter()`	軽量なパーティクル演出を作る
`scenePhase`	`"title"`、`"gameplay"` などの進行状態を持つ
`saveProgress()` / `loadProgress()`	進行状況を保存 / 読み込みする
`takeScreenshot()`	描画後の canvas を画像として保存する

`loadModel()` の例です。

```
const runtime = await app.loadModel("./assets/robot.glb", {
  format: "gltf"
});
```

`format` には `"gltf"`、`"collada"`、`"json"` を指定できます。シーン全体を宣言的に読み込む場合は `loadScene()` を使います。

5.11　フレーム処理の詳しい順序

基本的なアプリでは、`onUpdate` に更新処理を書き、`autoDrawScene: true` のまま使えば十分です。ここから先は、ポストプロセスや独自レンダーパスを組み込むときに必要になります。

`WebgApp.frame()` は概ね次の順序で進みます。

1. ページが非表示または非フォーカスなら、`ondemand` モードでは休止する。
2. 前フレームからの `deltaSec` を計算する。
3. 管理中の `EyeRig` を更新する。
4. フレームコンテキスト `ctx` を作成する。
5. `onUpdate(ctx)` を呼び出す。
6. Tween を更新する。

7. Space のアニメーションを更新する。
8. パーティクルエミッターを更新する。
9. カメラ追従、ロックオン、シェイクを反映する。
10. screen.clear() を呼び出す。
11. onBeforeDraw(ctx) を呼び出す。
12. autoDrawScene が true なら space.draw(app.eye) を実行する。
13. autoDrawBones が true ならボーンを描画する。
14. onAfterDraw3d(ctx) を呼び出す。
15. パーティクルを描画する。
16. HUD / メッセージ / トーストを描画する。
17. onAfterHud(ctx) を呼び出す。
18. screen.present() を呼び出す。
19. 入力のワンショット状態を次フレームへ進める。
20. 継続中なら次フレームを予約する。

ポストプロセスやオフスクリーンレンダーターゲットを使用する場合は、autoDrawScene: false を指定し、描画順を自前で管理します。

```
const app = new WebgApp({
  document,
  autoDrawScene: false
});

await app.init();

app.start({
  onBeforeDraw({ space, eye }) {
    offscreenRenderer.drawScene(space, eye);
  },
  onAfterDraw3d() {
    postprocess.composeToScreen();
  }
});
```

5.12 典型的な構成例

ここまでの要素を組み合わせると、標準的なサンプルの骨格は次のようになります。

```
import WebgApp from "../../webg/WebgApp.js";
import Shape from "../../webg/Shape.js";
import Primitive from "../../webg/Primitive.js";
import { buildHelpPanelOptions } from "../../webg/OverlayPanelPresets.js";

const app = new WebgApp({
  document,
  messageFontTexture: "../../webg/font512.png",
  clearColor: [0.06, 0.08, 0.12, 1.0],
  debugTools: {
    mode: "release",
    system: "sample",
    source: "samples/sample/main.js"
  },
  camera: {
    target: [0.0, 0.0, 0.0],
    distance: 8.0,
    yaw: 24.0,
    pitch: -12.0
  }
});

await app.init();

app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0
});

const shape = new Shape(app.getGPU());
shape.applyPrimitiveAsset(Primitive.cube(2.0, shape.getPrimitiveOptions()));
shape.endShape();
shape.setMaterial("smooth-shader", {
  has_bone: 0,
```

```
  use_texture: 0,
  color: [1.0, 0.5, 0.3, 1.0]
});

const node = app.space.addNode(null, "box");
node.addShape(shape);

app.showOverlayPanel(buildHelpPanelOptions({
  id: "help",
  lines: [
    "Drag: orbit",
    "R: reset",
    "F9 then M: debug mode"
  ]
}));

app.registerActionMap({
  reset: ["r"]
});

app.start({
  onUpdate({ deltaSec, app }) {
    if (app.wasActionPressed("reset")) {
      node.setAttitude(0.0, 0.0, 0.0);
    }

    node.rotateY(0.8 * deltaSec);
    node.rotateX(0.3 * deltaSec);

    app.message.setLines("status", [
      "WebgApp sample",
      `debug=${app.getDebugMode()}`
    ], {
      anchor: "top-left",
      x: 0,
      y: 0
    });
  }
});
```

5.13 実装時のチェックリスト

WebgApp を使っていて何も表示されない、動かない、入力が効かない場合は、まず次を確認してください。

- await app.init() の完了後に app.getGPU()、app.space、app.eye を使っているか。
- Shape に頂点やプリミティブを追加した後、shape.endShape() を呼んでいるか。
- 作成した Shape を Node に追加し、その Node を app.space 上に作っているか。
- 毎フレームの移動や回転に deltaSec を使っているか。
- PhysicsSpace.step() に渡す値は deltaSec ではなく deltaSec * 1000.0 の deltaMs になっているか。
- 入力の「押された瞬間」と「押されている間」を使い分けているか。
- Orbit カメラが必要な場合、app.createOrbitEyeRig() を呼んでいるか。
- 長い説明やヘルプを HUD ではなく OverlayPanel に出しているか。
- カスタムレンダーパスを使う場合、autoDrawScene: false が必要か確認しているか。
- デバッグキーを維持したい独自入力は app.attachInput() 経由で接続しているか。

5.14 まとめ

WebgApp は、webg でアプリケーションを作るための標準的な土台です。Screen、シェーダー、Space、カメラ、入力、HUD、診断機能、フレームループをまとめ、開発者がシーン構築と更新処理に集中できるようにします。

最初に覚える流れは、new WebgApp()、await app.init()、app.space への配置、app.start({ onUpdate }) の 4 つです。

onUpdate では、描画前にシーンの状態を 1 フレームぶん進めます。移動や回転には ctx.deltaSec を使い、必要な場合は onUpdate({ deltaSec, input, app }) のように分割代入で取り出します。物理エンジンへ渡す場合は、deltaSec をミリ秒へ変換して physics.step(deltaSec * 1000.0) とします。

次章では、この WebgApp が作る cameraRig -> cameraRod -> eye を土台として、EyeRig による Orbit、Follow、First-person などのカメラ制御を詳しく扱います。

第 6 章

カメラ制御と EyeRig

3D アプリケーションで最初に考えるべきことの一つは、「ユーザーに世界をどう見せるか」です。同じ 3D 空間でも、展示物を見るアプリ、建物の中を歩くアプリ、キャラクターや乗り物を操作するアプリでは、気持ちよく感じるカメラの動きがまったく異なります。

展示物を見るなら、対象を中心に回転する orbit カメラが使いやすくなります。建物の中を移動するなら、自分の目で見ているような first-person カメラが自然です。移動する対象を見失わないようにするなら、対象の後ろから追いかける follow カメラが必要になります。

カメラ制御を選ぶときには、視点の動かし方だけでなく視野角も重要になります。人間は周辺視野まで含めると水平に 180 度前後の広い範囲を感じ取れますが、特定の物体を詳しく見ているときに使う中心視野はそれよりずっと狭くなります。そのため、モデルビューアのように対象を落ち着いて観察する用途では、一般的に標準レンズに近い 30 度から 40 度程度の画角が自然に感じられることがあります。一方、建物内を歩く first-person のような用途では、狭すぎる視野角は操作しにくさにつながります。たとえば幅 0.8m から 0.9m 程度のドアを 1m 手前から見るだけでも、ドア全体を画面内に収めるには 40 度を超える水平画角が必要になります。視野角が狭いと、近くの壁や入口、曲がり角の位置関係がつかみにくくなり、空間を歩いている感覚も弱くなります。このように、orbit、first-person、follow のどれを選ぶかだけでなく、視野角もアプリケーションの目的に合わせて決める必要があります。対象物を観察するアプリでは狭めの画角が有効な場合があり、ウォークスルーや操作対象を追うアプリでは、周囲の状況を把握しやすい少し広めの画角が適しています。

以上のように、カメラ制御はアプリケーションの目的と強く結びついています。EyeRig は、その違いを cameraRig、cameraRod、eye という 3 段構造に整理して扱うための仕組みです。

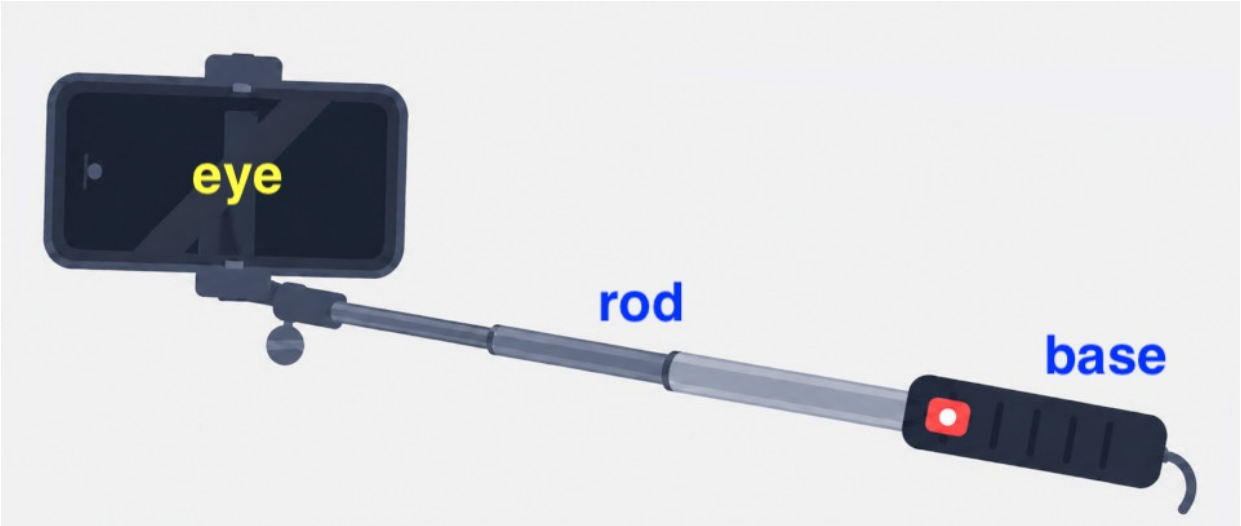


図 6.1: 自撮り棒

本章では、webg/EyeRig.js と webg/WebgApp.js をベースに、各ノードの役割や 3 段構成を採用している理由、そして利用者側で制御すべきポイントについて詳述します。第 5 章で解説した WebgApp の標準リグの上に、どのように視点操作を実装するかを順に紐解いていきましょう。

まず理解しておくべき重要な点は、視点の本体は EyeRig クラスそのものではなく、Space.setEye(node) によって指定された eye ノードであるということです。EyeRig はカメラを描画するクラスではなく、cameraRig、cameraRod、eye という 3 つの Node に対して、一定の規則に基づいた位置と回転を与えるための支援的な役割を担います。最終的に画面に何が映るかを決定しているのは、あくまで eye ノードです。

また、base -> rod -> eye という階層構造は、回転と距離の役割を分離しやすくするための設計です。軌道視点では注視点、追従視点では追従対象、一人称視点では身体の向きと視線の向きを個別に制御したい場面が多くあります。ここで重要になるのが、setAngles() と setLookAngles() の使い分けです。setAngles() は base や rod の向きを変更し、視点の土台そのものを動かす操作です。対して setLookAngles() は eye 側の独立した視線を制御するもので、進行方向とは異なる方向を向かせるための補助的な操作となります。

さらに、new EyeRig(...) で直接構成する場合は、attachPointer() を呼び出しただけでは視点制御は完結しません。attachPointer() はマウス、タッチ、ペンなどの入力インターフェースを接続する処理であり、実際のモード切り替え、追従対象への追随、キーボード操作の反映などは update(deltaSec) メソッドによって実行されます。一方、標準の Orbit で WebgApp.createOrbitEyeRig() を使う場合は、pointer 入力の接続と毎フレームの update(d

eltaSec) が WebgApp によって管理されます。この標準構成にアプリケーション側の手動 update() を追加すると、1 フレームに 2 回視点が更新されるため呼び出さないでください。

6.1　視点制御の基盤となる 3 段構成の設計思想

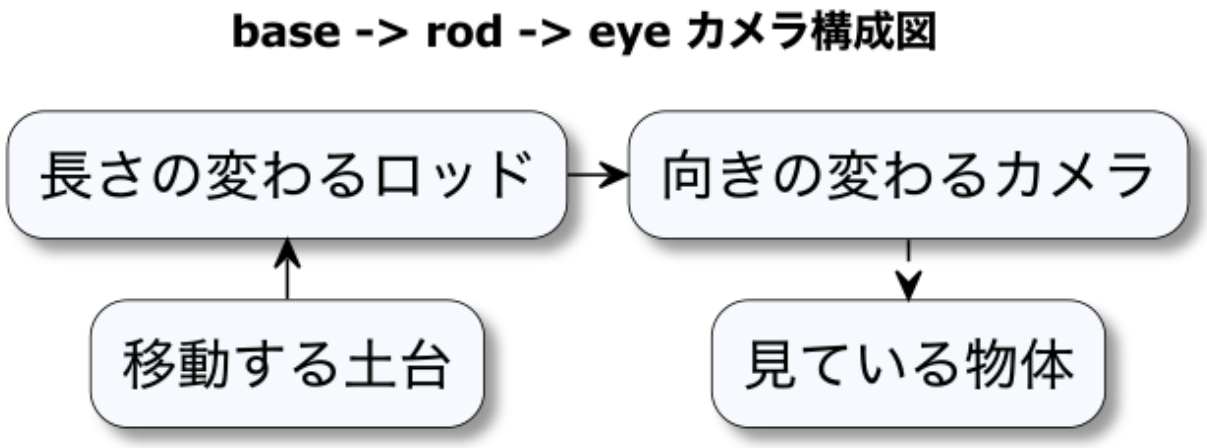


図 6.2: base rod eye カメラ構成図

EyeRig は *base*、*rod*、*eye* の *3* 段に分けることで、水平回転、高低角、最終視点を独立して扱いやすくしています。

3D アプリケーションでは、「どこを見るか」だけでなく、「どこを中心に回転するか」「誰を追跡するか」「身体の向きと視線の向きを分けるか」といった要求を場面に応じて切り替える必要があります。視点を単一の座標のみで管理すると、軌道視点と一人称視点を共存させた際に制御概念が混在し、設計が複雑になります。webg が base -> rod -> eye という 3 段構成を採用しているのは、これらの役割分担を明確に保つためです。

カメラ全体の基準位置を base、そこからの回転や構図を rod、そして最終的な位置と独立視線を eye に割り当てることで、モードが切り替わっても一貫した考え方で制御が可能になります。WebgApp.createCameraRig() もこの設計思想に基づいて標準のカメラノードを作成します。

```
createCameraRig() {
  this.cameraRig = this.space.addNode(null, this.camera.rigName);
  this.cameraRig.setPosition(...this.camera.target);
  this.cameraRig.setAttitude(this.camera.yaw, this.camera.pitch, this.camera.roll);
```

```
  this.cameraRod = this.space.addNode(this.cameraRig, this.camera.rodName);
  this.cameraRod.setPosition(0.0, 0.0, 0.0);
  this.cameraRod.setAttitude(0.0, 0.0, 0.0);
  this.eye = this.space.addNode(this.cameraRod, this.camera.eyeName);
  this.eye.setPosition(0.0, 0.0, this.camera.distance);
  this.eye.setAttitude(0.0, 0.0, 0.0);
  this.space.setEye(this.eye);
}
```

ここで重要なのは、EyeRig がなければ視点を作成できないわけではないということです。EyeRig は、既存の 3 段構成に対して orbit / first-person / follow という意味付けを与えるヘルパーであると理解してください。WebgApp.init() は標準の cameraRig、cameraRod、eye を作成し、space.setEye(this.eye) までを完了させるため、利用者はその構造の上に EyeRig を適用させるだけで十分です。

モードごとの役割分担を整理すると以下のようになります。- 軌道視点 (orbit): base が注視点、rod が yaw / pitch（回転）、eye が distance（距離）を担います。- 一人称視点 (first-person): base が身体の位置と body yaw、rod が eye height（目の高さ）、eye が独立視線を担います。- 追従視点 (follow): base がカメラ側の基準位置、rod が比較的安定した基準構図、eye が距離と対象を向く動的な追跡姿勢を担います。

このように共通構造を持つことで、「全景を俯瞰する」「主人公の視点で歩く」「対象を後方から追う」といった異なる視点操作を、同一の型で効率的に扱うことができます。

ここで、base の親子関係は EyeRig が決めるものではない点に注意してください。移動・回転する乗り物の動きをそのまま継承したい場合は、base を乗り物または取り付け用ノードの子にします。同じ位置へ移動しても乗り物の自転やロールを継承したくない場合は、乗り物とは独立した回転しないノードを用意し、必要な位置だけをアプリケーション側で反映します。

各ノードへ設定される位置と姿勢は、その親を基準とするローカル変換です。したがって、最終的なワールド位置とワールド姿勢は、アプリケーションが構成した親子階層と、EyeRig が設定するローカル変換を合成した結果になります。EyeRig は対象オブジェクトを勝手に移動させず、カメラの親子関係も暗黙に変更しません。

アプリケーションと EyeRig の役割を分けると、次のようになります。

- アプリケーション

  - base の親、継承する移動と回転、対象オブジェクトの移動、モード切り替え、入力の割り当て
- EyeRig
  - モードごとの状態、base / rod / eye へのローカル変換、入力値の変換、Follow の注視姿勢と補間

6.2 視野角と投影行列の管理

カメラを理解する上で注意すべき点は、EyeRig が制御するのはあくまで「位置と姿勢」までであり、「どれくらい広く写すか（画角）」は別の制御領域であるということです。webg では WebgApp が viewAngle、projectionNear、projectionFar を保持し、updateProjection() メソッドを通じて現在のシェーダーへ投影行列を転送します。

```
updateProjection(viewAngle = this.viewAngle) {
  const proj = new Matrix();
  const vfov = this.screen.getRecommendedFov(viewAngle);
  proj.makeProjectionMatrix(
    this.projectionNear,
    this.projectionFar,
    vfov,
    this.screen.getAspect()
  );
  this.projectionMatrix = proj;
  if (this.shader?.setProjectionMatrix) {
    this.shader.setProjectionMatrix(proj);
  }
  return proj;
}
```

ここで viewAngle は、短辺方向の見え方を決める基準視野角です。投影行列そのものは縦方向の FOV (vfov) を受け取りますが、現代の画面は PC の横長画面とスマートフォンの縦長画面でアスペクト比が大きく異なります。もし常に同じ縦 FOV を使うと、横長画面では縦方向、縦長画面では横方向の見える範囲が意図せず変わり、同じ viewAngle でも体感的なズーム量が揃いません。

webg ではこの差を抑えるため、Screen.getRecommendedFov(base) が base を「短辺方向の FOV」として解釈し、現在のアスペクト比から投影行列へ渡す縦 FOV を計算します。

アスペクト比は次のように定義されます。

```
aspect = width / height
```

aspect >= 1.0 の横長または正方形の画面では、短辺は縦方向です。この場合、短辺方向の FOV と縦 FOV は同じなので、実際に使う vfov はそのまま base になります。

```
vfov = base
```

一方、aspect < 1.0 の縦長画面では、短辺は横方向です。この場合、横方向の FOV (hfov) が base になるように、縦 FOV を逆算します。透視投影では、距離 d における半分の見える幅または高さは d * tan(fov / 2) で表せます。横 FOV と縦 FOV の関係は次の式になります。

```
tan(hfov / 2) = aspect * tan(vfov / 2)
```

縦長画面で hfov = base を保ちたいので、vfov は次の式で求めます。

```
vfov = 2 * atan(tan(base / 2) / aspect)
```

これにより、例えば base = 50° のとき、PC の横長画面で aspect = 1.8 なら vfov = 50° のままです。一方、スマートフォンの縦長画面で aspect = 0.5 なら vfov は約 86° になります。この値だけを見ると広角化しているように見えますが、横方向の FOV は 50° に保たれます。つまり、画面の短辺方向で見える範囲を維持するために、縦長画面では縦方向を大きく広げている、ということです。

この設計は、短辺方向の見える範囲を「ズーム 1 段階程度の調整で収まる範囲」に保つためのものです。画面の長辺方向は端末によって広くなったり長くなったりしますが、短辺方向が大きく変わらなければ、対象が極端に窮屈になったり、逆に小さくなりすぎたりする問題を避けやすくなります。特に、モバイル縦画面と PC 横画面の両方で同じサンプルを動かす場合、この基準は構図の安定に大きく効きます。

したがって、カメラの運用においては、「どこに配置し、どちらを向かせるか」は EyeRig や Node で制御し、「どれくらい広く写すか」「遠近感をどのように表現するか」は viewAngle

と投影行列側で制御するという切り分けを明確にすることが肝要です。最も基本的な設定方法は、WebgApp の生成時に viewAngle および projectionNear、projectionFar を指定することです。

```
const app = new WebgApp({
  document,
  messageFontTexture: "./webg/font512.png",
  viewAngle: 54.0,
  projectionNear: 0.1,
  projectionFar: 160.0,
  camera: {
    target: [0.0, 6.0, 0.0],
    distance: 46.0,
    yaw: 28.0,
    pitch: -18.0
  }
});
await app.init();
```

このとき、camera.distance はカメラの物理的な位置を決定し、viewAngle は短辺方向におけるレンズの広さに相当します。同じ distance であっても、viewAngle を小さくすれば望遠的な視覚効果となり、大きくすれば広角的な視覚効果となります。ズーム演出などの実装では、実行中に viewAngle を変更して updateProjection() を呼び出す手法が有効です。

```
app.viewAngle = 40.0;
app.updateProjection();
```

updateProjection() は、現在の viewAngle から投影行列を作り直す処理です。したがって、以後のリサイズやレイアウト更新後も同じ画角を保ちたい場合は、上のように app.viewAngle を更新してから呼び出します。

```
app.updateProjection(40.0);
```

引数付きの updateProjection(40.0) は、その呼び出しで使う基準 FOV を直接渡す形です。ただし app.viewAngle 自体を書き換えるわけではありません。継続的なズーム状態として扱う値は app.viewAngle に保持し、投影行列だけを一時的に作り直したい場合に引数を使

う、と分けて考えると安全です。

この操作はカメラの位置を変更しません。構図を維持したまま、望遠または広角の見え方に切り替えたい場合に適しています。一方で、対象に実際に近づいた感覚を出したい場合は、EyeRig 側の distance や position を変更するのが自然です。また、projectionNear と projectionFar の設定も重要です。広大なシーンを表示するために far を過剰に広げると、深度バッファの精度が低下し、Z ファイティング（描画のちらつき）が発生しやすくなるため注意してください。

短辺 FOV は、フルサイズカメラの焦点距離に換算して表示することもできます。フルサイズセンサーの短辺を 24mm とすると、短辺 FOV fovShort に対応する焦点距離は次の式で求められます。

```
focalLengthMm = 24 / (2 * tan(fovShort / 2))
```

この換算は、編集用ビューアやモデラーのように、見え方の違いを利用者へ短い言葉で伝えたい場面で特に役立ちます。50° や 24° といった角度表示よりも、26mm、56mm、114mm のようなレンズ相当表示の方が、広角、標準、望遠の感覚を共有しやすいためです。ただし、これは実在するカメラレンズをシミュレートしているという意味ではありません。あくまで、viewAngle が作る見え方を、写真でよく使われる焦点距離の言葉へ置き換えるための目安です。

6.3　軌道視点（Orbit Camera）の実装とパン操作

まずは、最も基本的な軌道（orbit）視点から解説します。注視点と距離を定義し、ドラッグやホイール操作で視点を回転させることで、シーン全体の空間的な位置関係を容易に確認できます。samples/high_level（高レイヤー（ハイレベル）） でもこの構成が最小例として採用されています。

```
import WebgApp from "./webg/WebgApp.js";

const app = new WebgApp({
  document,
  messageFontTexture: "./webg/font512.png",
  clearColor: [0.1, 0.15, 0.1, 1.0],
  camera: {
    target: [0.0, 0.0, 0.0],
```

```
    distance: 8.0,
    yaw: 0.0,
    pitch: 0.0,
    roll: 0.0
  }
});
await app.init();

const orbit = app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0,
  minDistance: 4.0,
  maxDistance: 18.0,
  wheelZoomStep: 1.0
});

app.start();
```

この例では、WebgApp が生成した cameraRig、cameraRod、eye の上に、createOrbitEyeRig() で軌道用の EyeRig を作成しています。createOrbitEyeRig() は、pointer 入力の接続、毎フレームの update(deltaSec)、そして EyeRig の orbit state と WebgApp の camera state の同期をまとめて扱います。これにより、サンプル側で orbit.update(deltaSec) や app.camera.target への手動コピーを書く必要がなくなり、パン（PAN）操作が app.camera.target で上書きされる事故を避けやすくなります。

返される orbit は通常の EyeRig なので、必要に応じて setTarget()、setAngles()、setDistance() などもそのまま使えます。target が base の位置に、yaw / pitch が rod の向きに、distance が eye の Z 軸位置にそれぞれ反映されます。

ここで重要なのは、軌道視点が「回転とズームだけのカメラ」ではないという点です。実際のモデルビューアやエディタでは、見たい対象を画面の中央へ寄せ直したい場面が頻繁に発生します。たとえば、キャラクター全体を確認したあとに手元だけを拡大したい場合や、建物全景から一部の窓まわりへ視線を移したい場合に、回転だけでは目的の箇所を中央へ持ってきにくいことがあります。このような場面のために、EyeRig の orbit にはパン（PAN）が実装されています。

パン（PAN）は、カメラ自体を別の場所へ瞬間移動させるのではなく、orbit.target を

視線のスクリーン平面に沿って平行移動する操作です。これにより、現在の yaw / pitch / distance を大きく崩さずに、「見ている中心」だけを横や上へずらすことができます。

設計上は、right / up の向きを eye のワールド行列から取り出し、ドラッグ量からワールド空間の移動量を求めます。ただし、orbit.target は base の親を基準とするローカル座標です。base に回転した親がある場合、ワールド空間で求めた移動量を親のローカル座標へ逆変換してから target へ加えます。

ワールド方向をそのままローカル座標へ加えると、親の回転が後からもう一度適用され、画面上の操作方向と実際の移動方向がずれます。ワールド方向とローカル状態を明示的に変換することで、乗り物や惑星の子にカメラを置いた場合でも、画面上の左右上下と PAN の見え方を一致させています。

EyeRig の pointer 操作では、orbit モードで Shift を押しながらドラッグするとパン（PAN）操作が有効になります。また、タッチ操作では 2 本指操作の中心移動がパン（PAN）に割り当てられています。さらに、キーボード操作でも Shift + Arrow によって同じ平行移動を行えます。createOrbitEyeRig() を使うと、この入力処理と WebgApp camera state への同期が標準で接続されるため、サンプルごとに個別の実装を行うことなく、軌道カメラの挙動を共通化できます。

このパン（PAN）機能は、単に操作性を向上させるだけでなく、構図の決定において非常に有用です。詳細部へ寄ったときに target が対象の中心から外れていると、少し回転させただけで見たい箇所が画面外へ出やすくなります。パン（PAN）を併用して関心点を中央へ戻してから回転やズームを続けることで、ビューアやアセット検証、ライティング確認などの作業効率が大きく向上します。gltf_loader、collada_loader、json_loader などのローダーサンプルで Shift + Arrow と Shift + Drag を有効にしたのも、まさにこの用途を想定してのことです。

createOrbitEyeRig() の利点は、視点位置の初期化だけでなく、キーバインディングの管理にもあります。キーマップの既定値は WebgApp 側で管理されるため、利用者は「すべてのキー設定を書き直す」のではなく、「既定値に対して差分だけを指定する」という形で調整が可能です。

たとえば、回転キーだけを W / A / S / D へ変更し、ズームキーは既定値のままにする場合は次のように記述します。

```
const orbit = app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
```

```
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0,
  orbitKeyMap: {
    left: "a",
    right: "d",
    up: "w",
    down: "s"
  }
});
```

この場合、zoomIn と zoomOut は既定の [と] がそのまま使われます。また、パン（PAN）に使用する修飾キーは panModifierKey で変更可能です。たとえば Alt + Drag と Alt + W / A / S / D をパン（PAN）にしたい場合は、次のように指定します。

```
const orbit = app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0,
  orbitKeyMap: {
    left: "a",
    right: "d",
    up: "w",
    down: "s"
  },
  panModifierKey: "alt"
});
```

panModifierKey を変更すると、キーボードのパン判定とポインタドラッグのパン判定の両方が同時に更新されます。指定可能な修飾キーは以下の通りです。

役割	指定できる名称
Shift	shift
Control	control、ctrl
Alt / Option	alt、option
Meta / Command	meta、command、cmd

この一覧は 16 章 の特殊キー一覧と一致しています。

ドラッグボタンと代替入力の調整

ビューアのみを構築する場合、左ドラッグを軌道回転に割り当てても問題ありません。しかし、モデラーやエディタでは、左ドラッグを矩形選択や頂点移動などのツール操作に割り当てたいため、カメラ操作を別のボタンへ移す必要があります。EyeRig はこの用途のために、カメラドラッグを開始する dragButton を設定できます。

dragButton はポインターイベントの button 値を使用します（左: 0、中: 1、右: 2）。エディタ等で中ボタンに変更すると、左ボタンを編集操作に開放できます。

```
const orbit = app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0,
  dragButton: 1
});
```

さらに、Blender のような操作感を実現したい場合は、dragZoomModifierKey を指定することで、ホイールとは別のドラッグによるズーム操作を実装できます。

```
const orbit = app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0,
  dragButton: 1,
  panModifierKey: "shift",
  dragZoomModifierKey: "control",
  dragZoomSpeed: 0.04
});
```

この設定では、中ボタンドラッグが回転、Shift + 中ボタンドラッグ がパン（PAN）、Ctrl + 中ボタンドラッグ がドラッグズームとなります。

一方で、macOS のトラックパッド環境などでは中ボタンドラッグがブラウザに届かない場合があります。これを補完するため、EyeRig には修飾キー付きの代替ドラッグ開始条件を指定できる alternateDragButton と alternateDragModifierKey が用意されています。

```
const orbit = app.createOrbitEyeRig({
  target: [0.0, 0.0, 0.0],
  distance: 8.0,
  yaw: 24.0,
  pitch: -12.0,
  dragButton: 1,
  panModifierKey: "shift",
  dragZoomModifierKey: "control",
  dragZoomSpeed: 0.04,
  alternateDragButton: 0,
  alternateDragModifierKey: "alt"
});
```

この設定では、通常の中ボタンドラッグに加えて、Option + 左ドラッグ もカメラドラッグとして認識されます。alternateDragModifierKey が押されているときのみ代替入力として扱うため、左ドラッグ単体での選択操作と衝突させずに導入可能です。これは、「左ドラッグは選択に使い、macOS では Option + 左ドラッグを中ボタン相当として使う」編集系アプリケーションに適した構成です。

なお、現在の標準設定を確認したい場合は getDefaultOrbitEyeRigBindings() を使用してください。

```
const defaults = app.getDefaultOrbitEyeRigBindings();

console.log(defaults.keyMap.left);       // "arrowleft"
console.log(defaults.panModifierKey);    // "shift"
console.log(defaults.alternateDragButton); // null
```

また、生成後の EyeRig インスタンスに対しても動的に設定を変更することが可能です。

```
orbit.orbit.keyMap.left = "j";
orbit.orbit.keyMap.right = "l";
orbit.orbit.keyMap.up = "i";
orbit.orbit.keyMap.down = "k";
```

```
orbit.orbit.panModifierKey = "control";
```

このように、createOrbitEyeRig() は単なるヘルパーではなく、入力設定を既定値付きで管理するエントリーポイントとして機能します。

コード上で注視点を明示的に変更したい場合は、setTarget() を使用します。たとえば、モデルのバウンディングボックスに基づいて初期表示を決めた後、特定の部位を中央に寄せたい場合に有効です。

```
orbit.setTarget(
  orbit.orbit.target[0] + 0.4,
  orbit.orbit.target[1] + 0.8,
  orbit.orbit.target[2]
);
```

ただし、スクリーン平面に沿った自然なパン（PAN）を毎回手作業で実装する必要はありません。createOrbitEyeRig() を使う標準構成では、ポインタ接続、毎フレーム更新、camera state 同期が WebgApp 側で管理されます。new EyeRig(...) で直接構成する場合は、attachPointer() と update(deltaSec) を適切に呼び出すことで、ポインタ、タッチ、キーボードのすべての経路で統一されたパン挙動が得られます。

6.4　一人称視点：身体の向きと視線の方向の独立制御

一人称（first-person）視点は、移動可能な base と、進行方向から独立した最終視線を持つカメラを構成するモードです。名称は First Person ですが、用途をキャラクターの目の位置だけに限定しません。キャラクターの少し後方、少し上方、または少し右側にカメラを置き、キャラクターと同じ方向へ進みながら周囲を見る肩越し視点にも利用できます。

ここでの設計上の要点は、身体または親オブジェクトの進行方向と、利用者が見ている方向を同一視しないことです。EyeRig では、base に身体の位置と姿勢、rod に目の高さ、eye に独立した視線を配置します。

```
base.position    = firstPerson.position
base.attitude    = body yaw / pitch / roll
```

```
rod.position     = [0, eyeHeight, 0]
rod.attitude     = identity
eye.position     = [0, 0, 0]
eye.attitude     = look yaw / pitch / roll
```

bodyYaw は何を表すか

bodyYaw は「カメラが現在見ている方角」そのものではなく、base が持つ身体基準をローカル Y 軸まわりに回す角度です。EyeRig のカメラ前方はローカル -Z、右方はローカル +X です。したがって、bodyYaw は身体基準のローカル -Z を水平面内のどちらへ向けるかを決めます。

```
body forward = rotateY(bodyYaw) * [0, 0, -1]
body right   = rotateY(bodyYaw) * [1, 0, 0]
```

基準となる向きは次のようになります。ここで示す方向は base の親を基準とするローカル方向です。親ノード自体が回転している場合は、その親姿勢がさらに合成されて最終的なワールド方向になります。

bodyYaw	身体の前方	身体の右方
0°	-Z	+X
90°	-X	-Z
180°	+Z	-X
-90°	+X	+Z

この対応を理解すると、bodyYaw: 180.0 の意味も明確になります。たとえば、キャラクターや車両のモデルがローカル +Z を前方として作られている場合、カメラ身体の前方であるローカル -Z とは初期状態で逆を向きます。そこで bodyYaw に 180 度を与えると、カメラ身体の前方を親モデルの +Z 前方へ一致させられます。モデルもローカル -Z を前方としているなら、同じ軸合わせに 180 度は必要なく、bodyYaw: 0.0 が基準になります。

つまり、bodyYaw の初期値は一律に 0 度または 180 度と決めるものではありません。まず親モデルがどのローカル軸を前方としているかを確認し、その前方とカメラ身体のローカル -Z を一致させる角度を選びます。実行中に身体を左右へ旋回させる場合は、この初期軸合わせを基準として bodyYaw を増減させます。

```
const eyeRig = new EyeRig(app.cameraRig, app.cameraRod, app.eye, {
  document,
  element: app.screen.canvas,
  input: app.input,
  type: "first-person",
  firstPerson: {
    position: [1.0, 2.2, -4.0],
    bodyYaw: 180.0,
    bodyPitch: 0.0,
    bodyRoll: 0.0,
    lookYaw: 0.0,
    lookPitch: -8.0,
    lookRoll: 0.0,
    eyeHeight: 0.0,
    moveSpeed: 12.0,
    runMultiplier: 2.2
  }
});
eyeRig.attachPointer();

app.start({
  onUpdate: ({ deltaSec }) => {
    eyeRig.update(deltaSec);
  }
});
```

firstPerson.position は base の親を基準とするローカル位置です。キャラクターや乗り物の子に base を置いた場合、この値は取り付け位置のオフセットになります。上の例は親モデルがローカル +Z を前方としている想定で、position: [1.0, 2.2, -4.0] によって右、上、後方へ取り付け位置をずらし、bodyYaw: 180.0 によってカメラ身体のローカル -Z 前方を親の +Z 前方へそろえています。

bodyYaw / bodyPitch / bodyRoll は base に適用され、身体または取り付け部の基準方向を表します。lookYaw / lookPitch / lookRoll は eye に適用され、身体方向から独立した最終視線を表します。

標準のポインター操作では、水平ドラッグを lookYaw、垂直ドラッグを lookPitch へ反映します。lookYaw は bodyYaw で決まった身体基準に対する追加の見回し角であり、標準の移動方向は変更しません。そのため、「上を見ながら前進する」「身体の進行方向を維持したまま

横を見る」といった動きを、親オブジェクトや身体の進行方向を変更せずに実現できます。

アプリケーションがキャラクターの移動と方向転換を管理する場合、EyeRig の移動キーを使用する必要はありません。親オブジェクトをアプリケーション側で移動・回転させ、EyeRig にはローカルな取り付け位置と見回しだけを担当させることができます。

一方、自由移動カメラとして使用する場合は、W / A / S / D / Q / E によって firstPerson.position を更新できます。W は bodyYaw で回転した body のローカル -Z、S はその反対、D は body のローカル +X、A はその反対へ進みます。Q / E は上下移動です。標準の WASD 移動は水平面上に保たれるため、bodyPitch / bodyRoll および独立視線の lookYaw / lookPitch / lookRoll は移動方向へ混ぜません。Shift に割り当てられた run 入力を使うと、runMultiplier に従って移動速度が増加します。

コードから明示的に姿勢を変更したい場合は、setPosition()、setAngles()、setLookAngles() を使用します。

```
eyeRig.setType("first-person");
eyeRig.setPosition(0.0, 0.0, 12.0);
eyeRig.setAngles(180.0, 0.0, 0.0);
eyeRig.setLookAngles(0.0, -10.0, 0.0);
```

ここでの setAngles() は身体（body）側の向きを、setLookAngles() は視点（eye）側の向きを制御します。キャラクターの方向転換と利用者の見回しを別の入力へ割り当てたい場合は、この二つを明確に分けて更新します。

samples/eye_rig では、青い camera vehicle の後方、上方、右側へ base を配置し、水平ドラッグ前後の bodyYaw と lookYaw を HUD へ表示します。ドラッグ後も bodyYaw が変わらず、lookYaw だけが変化することで、身体方向と独立視線が分離されていることを確認できます。

6.5　追従視点：追従対象と挙動の分離

追従（follow）視点は、カメラの基準位置とは独立して移動する対象を、滑らかな視線変化で見続けるためのモードです。

典型例は、ジェットコースターのような乗り物にカメラを取り付け、前方を走る別の車両を追跡する場面です。カメラを載せた車両と対象車両が急カーブ、上り下り、ジャンプを行って

も、カメラは対象を見失わず、急激に不自然な角度へ切り替わらないように追跡します。

ここで最も重要なのは、Follow が対象位置へカメラを移動する機能ではないことです。カメラの基準位置は、アプリケーションが base の親子階層、basePosition、baseAttitude によって決定します。Follow が毎フレーム動的に変更する主対象は、対象を見るための eye のローカル姿勢です。

ノードの分担は次のようになります。

```
base.position    = アプリケーションが決める基準位置
base.attitude    = アプリケーションが決める基準姿勢
rod.position     = [0, 0, 0]
rod.attitude     = アプリケーションが決める基準構図
eye.position     = [0, 0, distance]
eye.attitude     = 対象追跡姿勢 * 手動 look 補正
```

Orbit と同様に、eye の基準位置は rod のローカル +Z 方向へ distance だけ離れた位置です。カメラが映す方向は eye のローカル -Z です。rod は、前方、斜め前方、側方、見下ろしなど、アプリケーションが意図する比較的安定した基本構図を保持します。

```
const followRig = new EyeRig(app.cameraRig, app.cameraRod, app.eye, {
  document,
  element: app.screen.canvas,
  input: app.input,
  type: "follow",
  follow: {
    targetNode: targetVehicle,
    targetOffset: [0.0, 2.3, 0.0],
    basePosition: [0.0, 2.2, -2.0],
    baseAttitude: [0.0, 0.0, 0.0],
    distance: 16.0,
    yaw: 0.0,
    pitch: -12.0,
    roll: 0.0,
    minDistance: 6.0,
    maxDistance: 40.0,
    response: 6.0,
    maxAngularSpeed: 240.0,
    upReference: "base"
  }
```

```
});
followRig.attachPointer();

app.start({
  onUpdate: ({ deltaSec }) => {
    followRig.update(deltaSec);
  }
});
```

対象位置と targetOffset

追跡対象は targetNode で指定します。対象の注視位置は、対象ノードの原点へ単純なワールド座標のずれを加えた位置ではありません。targetOffset は対象ノードのローカル座標として扱われ、対象のワールド行列によって変換されます。

たとえば車両の中心ではなく運転席、キャラクターの足元ではなく胸や頭を見る場合に使用します。対象ノードが回転した場合、ローカルオフセットも対象の姿勢に従うため、車体上の同じ位置を追跡できます。

```
followRig.setTargetOffset(0.0, 1.8, 0.0);
followRig.setTargetNode(nextTarget);
```

setTargetNode() または setTargetOffset() を呼ぶと、追跡状態は初期化されます。次の update(deltaSec) で新しい対象方向から初期姿勢を計算し、その後のフレームで滑らかな追跡を続けます。

Follow は対象の実際のワールド位置を毎フレーム取得しますが、その位置へ base を移動しません。対象位置は、目標視線方向を求めるためだけに使用します。

追跡姿勢を求める処理

Follow の目標姿勢は、次の順序で求めます。

1. targetNode のワールド行列で targetOffset を変換し、注視点のワールド位置を求める
2. rod のワールド行列を逆変換し、注視点を rod のローカル座標へ移す
3. eye のローカル位置から注視点へ向かう forward を求める
4. 選択した上方向を同じ rod ローカル座標へ変換する
5. forward と上方向から right、cameraUp、back の直交基底を作る
6. 直交基底を、eye に設定するローカル quaternion へ変換する
7. 現在の追跡 quaternion から目標 quaternion へ球面線形補間する

対象と eye が同じ位置にある場合、視線方向を定義できません。この状態を任意の単位ベクトルで補うと、設定誤りを隠してしまいます。そのため EyeRig はゼロ長の追跡方向を例外として検出します。

上方向とロール

対象方向だけでは、視線方向を軸とするロールを一意に決められません。急な上り下りや bank を含む乗り物では、どの方向をカメラの上とするかを明示する必要があります。upReference はこの基準を指定します。

```
upReference = "base"    base のワールド上方向を使用
upReference = "rod"     rod のワールド上方向を使用
upReference = "world"   ワールド +Y を使用
```

"base" はカメラを載せた乗り物の傾きを基準にします。乗り物が bank すれば、その傾きを含む上方向で対象を追います。"rod" はアプリケーションが決めた基準構図を上方向へ含めたい場合に使用します。"world" は乗り物のロールから離れ、ワールドの水平線を基準にしたい場合に使用します。

追跡方向を forward、基準上方向を up とすると、カメラの right 軸は概念的に次の外積から求めます。

```
right = normalize(cross(forward, up))
cameraUp = cross(-forward, right)
```

forward と up が平行またはほぼ平行な場合、外積はゼロベクトルになり、ロールを決めら

れません。これはオイラー角で起きるジンバルロックではありません。目標 quaternion を作る前に、look-at の直交基底そのものを一意に構築できない特異条件です。

初期姿勢でこの条件が発生した場合、EyeRig は例外にします。初期状態にはロールを引き継ぐ直前姿勢がないため、別の上方向または初期配置をアプリケーションが明示する必要があります。

追跡開始後に一時的に forward と up がほぼ平行になった場合は、直前の追跡姿勢が持つ right 軸を新しい視線平面へ直交投影します。これにより直前のロールを連続的に維持し、補助 up を突然切り替えてカメラが 90 度回転するような不連続を避けます。これは異常値を隠すサイレントフォールバックではなく、連続追跡において直前姿勢を正規の状態として使う仕様です。

フレームレートに依存しない姿勢補間

追跡の滑らかさは、対象位置を遅らせるのではなく、視線姿勢の補間によって実現します。対象位置を平滑化して仮想的な注視点を作ると、急カーブやジャンプのときに実際の対象とは異なる場所を見るため、標準方式にはしていません。

現在姿勢から目標姿勢への補間には quaternion の球面線形補間を使用します。補間係数は次の式で求めます。

```
t = 1 - exp(-response * deltaSec)
```

response が大きいほど対象へ素早く向き、小さいほどゆっくり収束します。この式は deltaSec を含むため、フレームレートが変化しても追跡速度の体感が大きく変化しにくくなります。

急激な方向転換で視線が過度に速く回転しないよう、maxAngularSpeed は 1 秒あたりの最大回転角を制限します。response は目標への収束速度、maxAngularSpeed は瞬間的な回転速度の上限であり、役割が異なります。

初回の update() では対象を即座に向いて有効な初期姿勢を確定し、2 フレーム目以降を滑らかに追跡します。対象またはモードを切り替えた後に初期追跡をやり直す場合は、resetFollowTracking() を使用できます。

基準構図と手動視線補正

アプリケーションが決める、あまり動かない基本構図は `rod` の yaw、pitch、roll として保持します。Follow による動的な追跡姿勢は `eye` の tracking quaternion として保持します。

この分担により、アプリケーションは `rod` を使って「通常は前方を見る」「少し右側から見る」「やや見下ろす」といった構図を制御でき、Follow はその構図から対象を向くために必要な `eye` のローカル姿勢を計算できます。

`setLookAngles()` で与える look 角は、自動 tracking quaternion とは別の補正として保持され、最終的な `eye` 姿勢へ合成されます。自動追跡結果そのものを手動入力で直接上書きしないため、次フレームの追跡計算と競合しません。

Follow の入力と含めない機能

標準入力では、ドラッグと矢印キーが `rod` の基準 yaw / pitch を変更し、ホイール、pinch、`[` / `]` が `eye` の基準 distance を変更します。この distance は `rod` から `eye` までの取り付け距離であり、注視対象とのワールド距離そのものではありません。

Follow には PAN を含めません。PAN によって targetOffset や base 位置を動かすと、注視姿勢追跡とカメラ位置追従の責務が混ざるためです。対象の後方へカメラ位置そのものを移動したい場合は、アプリケーションが `base` の親または取り付けノードを移動します。

同じ理由から、Follow は対象 yaw を `base` へコピーせず、対象位置を中心とした Orbit も行いません。これらが必要な場合は、アプリケーション側の階層構造または位置追従処理と組み合わせます。

WebgApp の位置追従との違い

`WebgApp.followNode()`、`lockOn()`、`clearCameraTarget()` にも「追従」に関係する補助機能があります。ただし、これらは `WebgApp` の camera target と `cameraRig` の基準位置を対象へ追従させる機能です。

`EyeRig` Follow は、独立したカメラ基準位置から対象を見続けるための姿勢追跡です。`WebgApp.followNode()` は位置を更新し、`EyeRig` Follow は姿勢を更新します。両者を組み合わせる場合も、どの処理が位置を決め、どの処理が視線を決めるかを分けて設計してください。

samples/eye_rig では、青い camera vehicle へ base を取り付け、別に動くオレンジ色の target vehicle を追跡します。HUD の follow dot は eye のワールド前方と target 方向の内積で、1 に近いほど正確に対象を見ています。U キーでは base / rod / world の上方向基準を切り替え、bank 時のロールの違いを確認できます。

6.6 モード切り替えと状態の扱い

Orbit、First Person、Follow は、それぞれ独立した状態を保持します。setType() は有効なモードを切り替えて、そのモードの状態を base / rod / eye へ反映します。別モードの状態を暗黙に破棄したり、同じ意味の値へ自動変換したりはしません。

```
eyeRig.setType("orbit");
eyeRig.setType("first-person");
eyeRig.setType("follow");
```

モードごとに 3 ノードの役割が異なるため、切り替え前と同じワールド位置・姿勢を自動的に維持することは標準動作ではありません。たとえば Orbit の target と First Person の position は、どちらも base.position へ反映されますが、前者は注視中心、後者は身体または取り付け位置という異なる意味を持ちます。

切り替え時の見た目を連続させたい場合は、アプリケーションが切り替え前のワールド変換を読み、切り替え先のローカル state へ明示的に変換します。値の意味が異なる状態を自動的にコピーすると、親階層が異なる場合や、Follow の自動姿勢を含む場合に不具合を隠しやすいためです。

Follow へ切り替えた直後は、対象を即座に向いて有効な初期姿勢を確定し、その後のフレームを滑らかに追跡します。対象を変更したときも同じ初期化が行われます。

6.7 サンプルとテストでの確認

samples/eye_rig は、Orbit、First Person、Follow を同じ 3 次元 track 上で比較するサンプルです。モードを切り替えるだけでなく、アプリケーションが作る親子階層と EyeRig のローカル state が、最終的なワールド視点へどのように合成されるかを確認できます。

- Orbit では、H によって camera vehicle の回転を継承する階層と、位置だけを共有する

独立 camera anchor を切り替えます
- First Person では、水平ドラッグ後も `bodyYaw` が維持され、`lookYaw` だけが変化することを HUD で確認できます
- Follow では、camera vehicle に取り付けた base が target 位置へ移動せず、`follow dot` が 1 へ近づくことを確認できます
- `U` によって `base / rod / world` の upReference を切り替え、bank に対する roll 基準の違いを確認できます

描画を使わない数値契約は `unittest/eye_rig/headless_probe.js` にあります。このテストは、Follow の base 独立、targetOffset のローカル変換、First Person の body/look 分離、複数の `bodyYaw` における W/D 移動と body 姿勢軸の一致、回転親下の Orbit PAN、Follow 初期特異姿勢の例外を確認します。

6.8 EyeRig の補助 API

`EyeRig` はモードの切り替え以外に、運用に便利な補助 API を提供しています。

- `setType(type)`: `"orbit"`、`"first-person"`、`"follow"` の間でモードを切り替えます。
- `setDistance(distance)` / `setRodLength(length)`: Orbit または Follow の `eye` 取り付け距離を変更します。許容範囲外の値は設定誤りとして例外になります。
- `setTarget(x, y, z)`: Orbit の中心となる `base` のローカル位置を設定します。
- `setTargetNode(node)` / `setTargetOffset(x, y, z)`: Follow の注視対象と、対象ノード内のローカル注視位置を設定します。
- `setAngles(...)`: `base` や `rod` の向きを変更し、視点の土台を制御します。
- `setLookAngles(...)`: `eye` の独立視線、または Follow の自動追跡へ合成する look 補正を変更します。
- `resetFollowTracking()`: Follow の追跡 quaternion と診断値を初期化し、次の更新で初期姿勢を再計算します。
- `getFollowTargetWorldPosition()`: `targetNode` とローカル `targetOffset` から現在のワールド注視点を返します。

Orbit の `setTarget()` と Follow の `setTargetOffset()` は、似た名前でも意味が異なります。`setTarget()` は Orbit の `base` 位置を変更します。`setTargetOffset()` は Follow 対象の内部で「どの場所を見るか」を変更し、camera base は移動しません。Orbit の PAN は前

者を入力操作で変更する機能ですが、Follow には同じ PAN 処理はありません。

6.9 実装上の留意点

EyeRig を利用する際は、特に以下のポイントに留意してください。

1. new EyeRig(...) で直接構成する場合は、attachPointer() の呼び出しだけでなく、毎フレーム update(deltaSec) を実行すること。標準の Orbit では WebgApp.createOrbitEyeRig() がこの更新を管理します。
2. EyeRig は位置と姿勢を制御するものであり、投影行列（画角など）は変更しないこと。
3. 軌道視点の標準利用では WebgApp.createOrbitEyeRig() を使い、入力更新と camera state 同期を WebgApp 側に任せること。
4. First Person では、親モデルの前方軸と body のローカル -Z の関係を確認して bodyYaw の基準値を決めること。
5. setAngles() と setLookAngles() の役割を混同しないこと。
6. 詳細部を追いたい場合は、回転だけで解決しようとせず、パン（PAN）によるターゲット調整を併用すること。
7. エディタで左ドラッグを選択操作に使う場合は、dragButton: 1 などでカメラドラッグを別ボタンへ移し、macOS 向けには alternateDragButton と alternateDragModifierKey による代替操作を用意すること。
8. 視野角や near / far を変更したい場合は、WebgApp.viewAngle および updateProjection() を使用すること。
9. base の親が回転する場合、Orbit の target や PAN 量がローカル座標であることを意識し、ワールド座標を直接代入しないこと。
10. Follow では camera base と target を独立させ、位置追従が必要ならアプリケーション側の取り付けノードまたは WebgApp.followNode() と役割を分けること。
11. Follow の初期配置で追跡方向と upReference が平行にならないことを確認すること。

EyeRig は「視点をどこに置き、どちらに向かせるか」というレイヤーであり、「どのように写すか」というレイヤーではありません。この分離を意識することで、カメラ制御に関する原因の切り分けを迅速に行うことが可能になります。

また、WebgApp.init() は標準の cameraRig、cameraRod、eye を自動的に作成します。ルート直下の標準的な Orbit では、この階層をそのまま利用できます。一方、乗り物の回転を継承する構成や、位置だけを共有して回転を継承しない構成では、アプリケーションが base

の親を目的に合わせて選びます。独自階層が必要かどうかは、どの移動と回転を camera へ継承したいかで判断してください。

6.10 まとめ

本章で最も重要なのは、EyeRig を「カメラそのもの」として捉えないことです。視点の本体は Space.setEye(node) で指定された eye ノードであり、EyeRig は cameraRig、cameraRod、eye という 3 段構成に対して、意味的な制御を与えるヘルパーです。この階層構造があるため、注視点中心の回転、身体と視線を分けた一人称視点、対象を追う追従視点を、共通の概念で扱うことが可能になります。

また、EyeRig が管理するのは「位置と姿勢」であり、視野角や投影行列は WebgApp 側の viewAngle、projectionNear、projectionFar および updateProjection() が管理します。つまり、「どこに置くか」と「どのように写すか」は完全に別の層として分離されています。この設計思想を理解しておくことで、今後の複雑なシーン構築においても、カメラ制御の問題を的確に切り分けることができるでしょう。

次章では、視点操作の上に乗る視覚的な層として、シェーダーとマテリアルの考え方について解説します。

第 7 章

シェーダーとマテリアル

この章では、webg で 3D オブジェクトを描画する際に、「どのシェーダーを選択し、どのような値をどのタイミングで転送すべきか」という判断基準について解説します。

webg における標準的な起点は webg/SmoothShader.js です。このシェーダーは非常に汎用的であり、静的メッシュからノーマルマップ付きメッシュ、さらにはスキニング付きメッシュまで、ほとんどの描画ニーズをこれ一つでカバーできます。

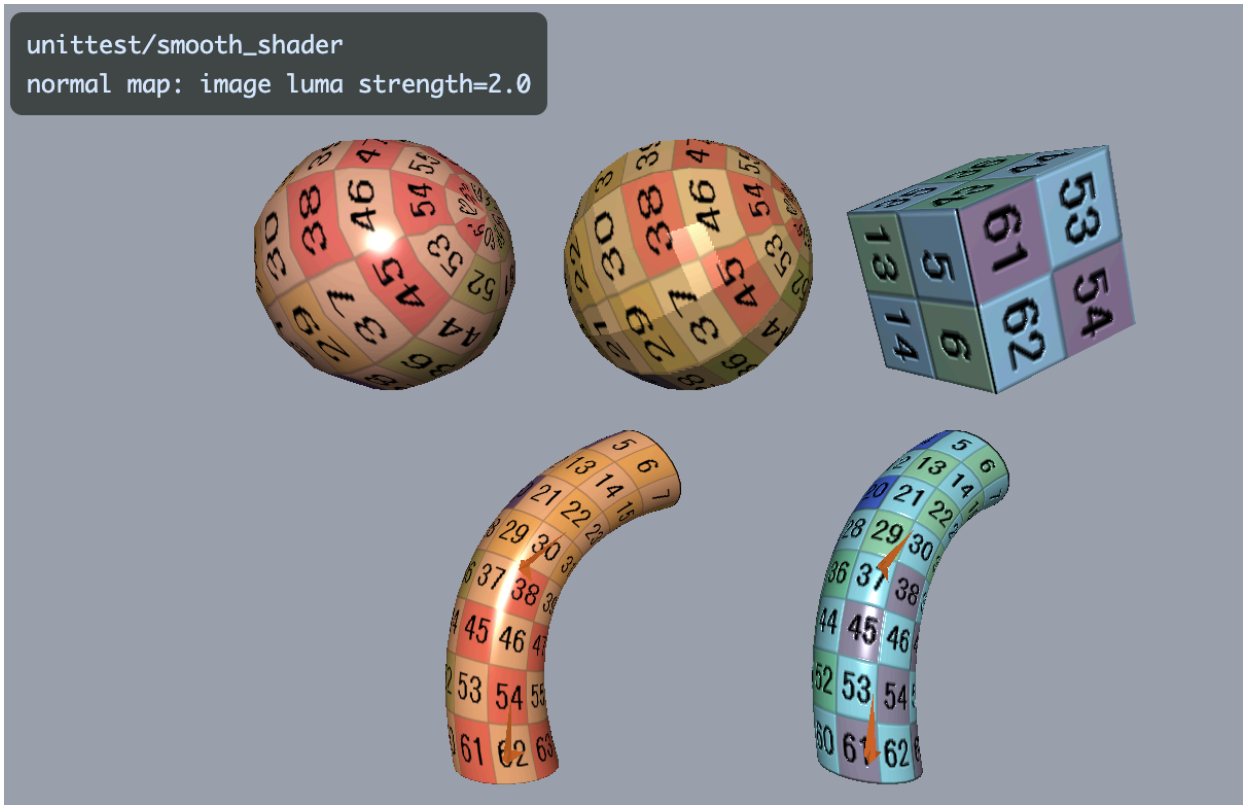


図 7.1: Smooth Shader サンプル

SmoothShader は テクスチャ、法線マップ、スキニング、フォグ、フラットシェーディングをサポートしています。

とはいえ、常に `SmoothShader` だけを用いればよいわけではありません。例えば `Phong` クラスは、極めてシンプルに構成されたシェーダーであるため、陰影計算の基礎を学習する際に最適です。つまり、実用的なアプリケーション開発においては `SmoothShader` を標準とし、内部構造の学習や基礎の理解には `Phong` 系を活用するという使い分けが最も効率的です。

また、`SmoothShader` は多くの機能を統合していますが、非スキニング描画においてパフォーマンスが低下しないよう最適化されています。具体的には、静的メッシュやノーマルマップ、スキニングを同一のインターフェースで扱いながら、ボーン行列パレットを WebGPU のバインドグループ（bind group）として分離し、更新タイミングを切り分けています。これにより、ユーザーインターフェースの簡潔さを維持しながら、「必要なときだけ大きなデータを GPU に転送する」という効率的な処理を実現しています。

WGSL 自体の記述方法については次章「WGSL の読み方」で、シェーダークラスの CPU 側実装については「シェーダーの実装」で詳しく解説します。

7.1 SmoothShader の基本コンセプト

通常の 3D オブジェクトを描画する場合、まずは SmoothShader の利用を検討してください。静的メッシュであっても、ノーマルマップやスキニングを利用する場合であっても、利用する入口は共通です。

```
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [0.25, 0.62, 1.0, 1.0],
  ambient: 0.26,
  specular: 0.86,
  power: 44.0,
  emissive: 0.0
});
```

このように、has_bone、use_texture、use_normal_map、flat_shading といったフラグを切り替えるだけで、同一のシェーダークラスのまま描画経路を柔軟に変更できます。例えば、面単位の陰影（フラットシェーディング）を表現したい場合も、別のクラスへ切り替えるのではなく、SmoothShader の flat_shading パラメータで制御します。

対して Phong や、その派生クラスである NormPhong、BonePhong、BoneNormPhong は、機能ごとにクラスが分離されています。これらは、特定の機能がどのように実装されているかを個別に追跡したい場合に非常に有効です。

マテリアル設定の仕組み

Shape.setMaterial() の挙動について正しく理解しておくことは重要です。このメソッドは、独立したマテリアル管理層へ値を送るための API ではなく、描画時のパラメータを予約するためのものです。

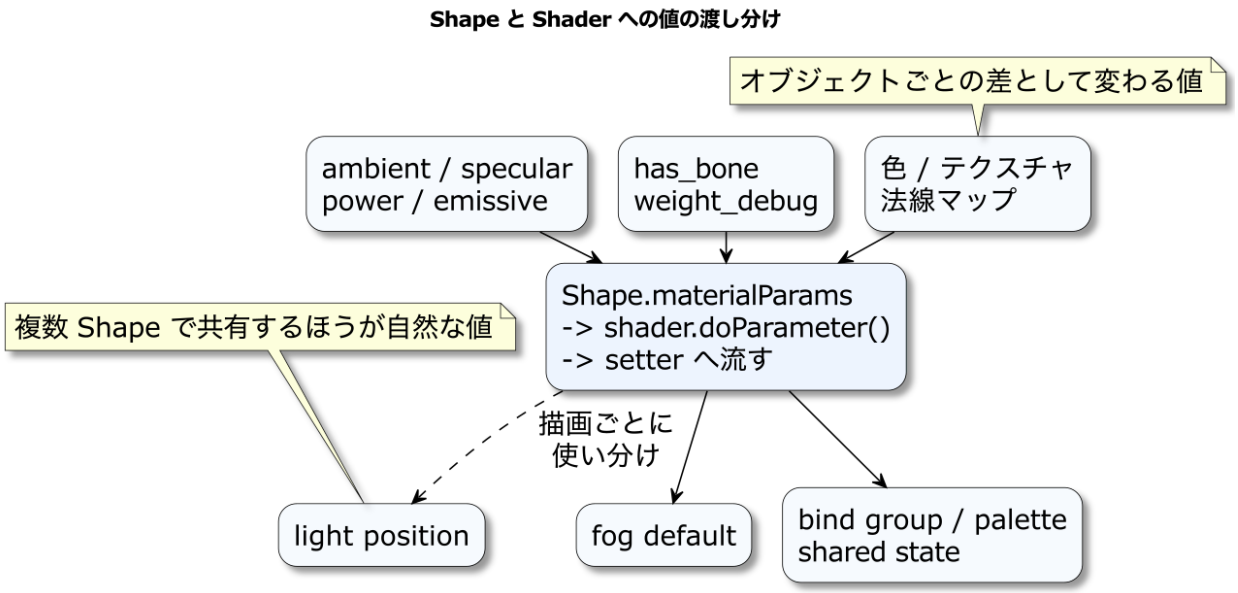


図 7.2: Shape と Shader への値の渡し分け

setMaterial() で渡された値は一旦 *Shape* 側に保持され、描画直前にシェーダーの *setter* へ流し込まれます。一方、ライトやフォグのような共有値は、シェーダーインスタンス側で保持することで管理を効率化しています。

実際には、`Shape` がパラメータ辞書を保持し、描画直前に `shader.doParameter()` を通じてシェーダーのセッターへ値を渡しています。この流れを把握しておくことで、設定した値がどこに保存され、どのタイミングで GPU に反映されるのかが明確になります。

したがって、`setMaterial("smooth-shader", {...})` を使用する際は、その値が「オブジェクトごとの差異」として保持すべきものか、そしてそれが `SmoothShader` のどのフラグや係数に対応しているかを意識して設定してください。

色、テクスチャ、ノーマルマップ、鏡面反射係数、`has_bone`、`weight_debug` などは `Shape` 側で管理します。一方で、光源位置やフォグの初期値のように、同じシェーダーを共有する複数の Shape で共通して利用したい値は、シェーダーインスタンス側で設定します。

ここで、`color` の 4 番目の値を alpha として小さくすれば、すぐに曇りガラスのような半透明表現になる、と考えたくなるかもしれません。しかし、背景をぼかして見せるタイプの半透明表現は、単一の Shape の material parameter だけでは完結しません。背後のシーンを一度 texture として描き出し、その texture をぼかし、ガラス面の領域だけに合成する必要があります。そのため、すりガラスや曇りガラスのような表現は、本章の `SmoothShader` の設定ではなく、第 20 章で扱う `FrostedGlassPass` と `GlassMaskShader` の組み合わせとして理解

してください。

柔軟な描画条件の切り替え

SmoothShader では、主に以下の 4 つの描画パターンを同一のインターフェースで扱うことができます。

経路	主なパラメータ
静的メッシュ + テクスチャ	has_bone: 0, use_texture: 1, use_normal_map: 0
静的メッシュ + テクスチャ + ノーマルマップ	has_bone: 0, use_texture: 1, use_normal_map: 1
スキニング + テクスチャ	has_bone: 1, use_texture: 1, use_normal_map: 0
スキニング + テクスチャ + ノーマルマップ	has_bone: 1, use_texture: 1, use_normal_map: 1

さらに、weight_debug: 1 を指定すればスキニングウェイトの色分布を確認でき、backface_debug を指定すれば裏面の確認が可能です。「別のシェーダーへ切り替える」のではなく、「同一のシェーダーに対して必要な条件を明示する」という設計により、実装の迷いを排除し、効率的な開発を可能にしています。

7.2 SmoothShader の実践的な使い方

ローレベル（低レイヤー）API での実装

Screen や Shape を直接操作するローレベル（低レイヤー）API を利用する場合、自身で SmoothShader インスタンスを生成し、Shape.prototype.shader に割り当てます。ここでは、ライトやフォグなど、シェーダー全体で共有する値を設定します。

```
import SmoothShader from "./webg/SmoothShader.js";
import Shape from "./webg/Shape.js";

const shader = new SmoothShader(gpu);
await shader.init();
Shape.prototype.shader = shader;

shader.setLightPosition([40.0, 90.0, 150.0, 1.0]);
shader.setDefaultParam("fog_color", [0.82, 0.90, 1.0, 1.0]);
shader.setDefaultParam("fog_near", 28.0);
```

```
shader.setDefaultParam("fog_far", 88.0);
shader.setDefaultParam("fog_mode", 1.0);
```

WebgApp を利用した実装

WebgApp を使用する場合、内部で標準的に SmoothShader が利用されるため、通常は app.init() の後に Shape.setMaterial("smooth-shader", ...) を記述するだけで十分です。面単位の陰影を表現したい場合も、まずはシェーダーを差し替えずに flat_shading: 1 を Shape 側へ渡してください。shape.setShader(otherShader) を使用するのは、学習用の Phong 系や独自のカスタムシェーダーへ切り替える場合に限られます。

```
const shape = new Shape(app.getGPU());
shape.applyPrimitiveAsset(asset);
shape.endShape();
shape.setShader(app.shader);
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [0.24, 0.64, 1.0, 1.0],
  ambient: 0.26,
  specular: 0.86,
  power: 46.0,
  emissive: 0.0
});
```

setMaterial() と shaderParameter() の使い分け

初期状態をまとめて設定する際は setMaterial() を使い、実行中に特定のパラメータのみを動的に変更したい場合は shaderParameter() を使用します。

setMaterial() は「その Shape が持つ材質条件の一式」を定義する API であり、shaderParameter() は「一部の値を差分更新する」ための API です。このように使い分けることで、初期化コードと更新処理の意図が明確になり、コードの可読性が向上します。

```
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 1,
  texture: colorTexture,
  use_normal_map: 1,
  normal_texture: normalTexture,
  normal_strength: 1.0,
  color: [1.0, 1.0, 1.0, 1.0],
  ambient: 0.24,
  specular: 0.92,
  power: 56.0,
  emissive: 0.0
});

shape.shaderParameter("normal_strength", 1.4);
shape.shaderParameter("emissive", 0.18);
```

7.3 パラメータの割り当て先

どの値を Shape に渡し、どの値をシェーダーインスタンスに渡すべきかを整理します。

Shape 側で管理する値（オブジェクトごとの個別設定）

以下のパラメータは、オブジェクトごとに異なる値を持つのが自然であるため、Shape 側で管理します。

パラメータ	意味	典型値
color	ベースカラー	[1, 1, 1, 1]
use_texture	カラーテクスチャの使用有無	0 or 1
texture	カラーテクスチャ本体	Texture instance
use_normal_map	ノーマルマップの使用有無	0 or 1
normal_texture	ノーマルマップ本体	Texture instance
normal_strength	元の法線とノーマルマップの混合比	0.8 〜 2.0
ambient	環境光（暗い部分の明るさ）	0.2 〜 0.4
specular	鏡面反射（ハイライト）の強度	0.3 〜 1.0
power	ハイライトの鋭さ（光沢感）	16 〜 96
emissive	自己発光の強度	0.0 〜 1.0
has_bone	ボーンパレット（スキニング）の使用有無	0 or 1
weight_debug	ウェイト分布の可視化	0 or 1
backface_debug	裏面のデバッグ表示	0 or 1
backface_color	裏面デバッグ時の色	[1, 0, 1, 1]

シェーダーインスタンス側で管理する値（共通設定）

以下のパラメータは、同一シェーダーを共有するすべての Shape で共通とするため、シェーダーインスタンス側で設定します。

パラメータ	設定メソッド
light	shader.setLightPosition(...)
fog_color	shader.setDefaultParam("fog_color", ...)
fog_near	shader.setDefaultParam("fog_near", ...)
fog_far	shader.setDefaultParam("fog_far", ...)
fog_density	shader.setDefaultParam("fog_density", ...)
fog_mode	shader.setDefaultParam("fog_mode", ...)

代表的な 4 つの設定パターン

具体的にどのように設定するか、代表的なケースを挙げます。

1. 単色の静的メッシュ

```
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
```

```
  use_normal_map: 0,
  color: [0.25, 0.62, 1.0, 1.0],
  ambient: 0.26,
  specular: 0.86,
  power: 44.0,
  emissive: 0.0
});
```

1. テクスチャ付き静的メッシュ

```
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 1,
  texture: colorTexture,
  use_normal_map: 0,
  color: [1.0, 1.0, 1.0, 1.0],
  ambient: 0.24,
  specular: 0.82,
  power: 40.0,
  emissive: 0.0
});
```

1. テクスチャとノーマルマップを併用する静的メッシュ

```
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 1,
  texture: colorTexture,
  use_normal_map: 1,
  normal_texture: normalTexture,
  normal_strength: 1.0,
  color: [1.0, 1.0, 1.0, 1.0],
  ambient: 0.24,
  specular: 0.92,
  power: 56.0,
  emissive: 0.0
```

```
});
```

1. スキニングとノーマルマップを併用する場合

```
shape.setMaterial("smooth-shader", {
  has_bone: 1,
  use_texture: 1,
  texture: colorTexture,
  use_normal_map: 1,
  normal_texture: normalTexture,
  normal_strength: 1.0,
  color: [1.0, 1.0, 1.0, 1.0],
  ambient: 0.35,
  specular: 0.80,
  power: 40.0,
  emissive: 0.0,
  weight_debug: 0
});
```

これらの例を比較すると、主な違いは has_bone、use_texture、use_normal_map、およびテクスチャの指定のみであることがわかります。これが SmoothShader の設計による利便性です。

7.4 SmoothShader の内部構造と WebGPU の利点

SmoothShader は一つのシェーダーでありながら、内部的に役割の異なるデータを 3 つのバインドグループ（bind group）に分離しています。これにより、描画負荷を最適化しています。

- group(0) は 射影行列、Model-View 行列、Normal 行列、ライト、マテリアル係数、flags、fog を draw ごとに更新します。
- group(1) は ベーステクスチャ、ノーマルテクスチャ、sampler をテクスチャの組み合わせごとに更新します。
- group(2) は ボーンパレット（ボーン行列）を スケルトンごとに更新します。

パフォーマンス最適化の仕組み

WebGPU のバインドグループ分離を活用することで、静的メッシュの描画において不要なデータ転送を排除しています。

group(0) に含まれる描画ごとのユニフォーム（uniform）データは 84 floats（336 bytes）と非常に軽量です（dynamic offset 用の stride は 256 バイト境界に合わせて 512 bytes としています)。一方、ボーンパレット（group(2)）は、最大 320 ボーンを想定すると 320 × 12 floats × 4 bytes = 15,360 bytes という大きなサイズになります。

もしこれらを同一のユニフォームバッファに混在させていた場合、たとえ has_bone: 0 の単純な立方体を描画する場合であっても、毎回 15KB 以上のデータを GPU に送るか、あるいは巨大なバッファを抱えたまま処理することになり、効率が悪くなります。

SmoothShader では以下のように処理を切り分けています。* 静的メッシュの場合: has_bone: 0 で描画し、Shape.draw() は setHasBone(0) のみを呼び出します。group(2) には共有の空ボーンバインドグループが割り当てられるため、大きなデータの書き込み（writeBuffer）は発生しません。* スキニングメッシュの場合: skeleton.updateMatrixPalette() を呼び出し、shader.setMatrixPalette(...) を通じてスケルトン専用のバッファにパレットを書き込みます。このボーン用バインドグループは WeakMap でキャッシュされ、スケルトンごとに再利用されます。

つまり、「スキニング機能を持っているから常に重い」のではなく、「スキニングを利用する描画に対してのみ、必要なデータを転送する」構造になっています。このようにバインドグループを用途ごとに分けることで、API の単純さを維持しながら、CPU から GPU への転送コストを最適化できるのが WebGPU を採用した webg の大きなメリットです。

Wireframe とスキニング

Shape.setWireframe(true) は、通常の面描画を Wireframe シェーダーによる line-list 描画へ切り替えるための入口です。Wireframe は質感を表現するマテリアルシェーダーではないため、テクスチャ、ノーマルマップ、ライティングは扱いません。しかし、頂点入力とボーンパレットの扱いは SmoothShader と互換になるよう設計されています。

具体的には、Wireframe は position だけでなく、normal、UV、bone index、weight の頂点入力を受け取ります。線描画では normal と UV は使用しませんが、SmoothShader と同

じ vertex buffer 構成を受け入れることで、Shape.draw() は通常描画と wireframe 描画を同じ処理フローで切り替えられます。スキニングメッシュでは group(2) にスケルトンごとのボーンパレットが渡され、Wireframe の頂点シェーダーも SmoothShader と同じ考え方で頂点位置を変形してから線を描きます。

Wireframe はテクスチャやライト計算を行いませんが、fog は SmoothShader と同じパラメータで扱えます。fog_color、fog_near、fog_far、fog_density、fog_mode、use_fog は通常描画と同じ意味を持ち、線色がカメラ空間での距離に応じて fog 色へ補間されます。Shape.setWireframe(true) で通常シェーダーから切り替える場合は、通常シェーダーの default に設定された fog が wireframe 側にも反映されます。そのため、遠景を fog でなじませているシーンでも、wireframe 表示だけが手前に浮いて見える状態を避けられます。

このため、静的メッシュだけでなく、ボーンで変形するメッシュに対しても shape.setWireframe(true) を使用できます。四角面を対角線なしで表示したい場合は、メッシュ構築時に Shape.addPolygon([a, b, c, d]) や addPlane() を使い、polygonLoops を残してください。Wireframe はこの面ループから外周 edge を作るため、描画用の三角形分割に含まれる対角線を表示しません。一方、最初から addTriangle() で作られた三角形データは、入力された三角形メッシュとしてそのまま wireframe 表示されます。

7.5 そのほかのシェーダー

webg で提供されている主なシェーダークラスを整理します。

クラス名	主な用途	texture	normal	bone	備考
Shader	基底クラス	-	-	-	派生クラスが継承
SmoothShader	標準的な 3D 描画	Yes	Yes	Yes	"smooth-shader"
Phong	基本的な陰影シェーダー	Yes	No	No	"phong"
NormPhong	ノーマルマップ対応の Phong	Yes	Yes	No	"norm-phong"
BonePhong	スキニング対応の Phong	Yes	No	Yes	"bone-phong"
BoneNormPhong	スキニング + ノーマルマップ	Yes	Yes	Yes	"bone-norm-phong"
Background	背景画像および背景矩形	Yes	No	No	new Background(gpu)
Font	文字描画	Yes	No	No	Text / Message で使用
BillboardShader	カメラ追従ビルボード描画	Yes	No	No	Billboard で使用
Wireframe	ワイヤーフレーム表示	No	No	Yes	shape.setWireframe()
FullscreenPass	ポストプロセス	Yes	No	No	ポストプロセス用

7.6 実装時の注意点と解決策

実装時に陥りやすい混乱について解説します。

- テクスチャが反映されない: use_texture: 1 に設定していても、texture インスタンスを正しく渡していない場合は描画されません。
- ノーマルマップで凹凸が出ない: use_normal_map: 1 に加え、normal_texture が正しく渡されているか、また normal_strength が極端に小さい値になっていないかを確認してください。
- スキニングでモデルが曲がらない: has_bone: 1 を指定しただけでは動作しません。Shape.setSkeleton() でスケルトンが関連付けられており、描画時に updateMatrixPalette() が正しく生成される状態である必要があります。
- デバッグモードの挙動: weight_debug を有効にするとライティングが無効になり、ウェイト分布の色が優先して表示されます。また、backface_debug を有効にした場合、観察しやすさを優先して cullMode: "none"（カリングなし）が設定されるため、通常描画時とは見え方が異なります。これらはデバッグ専用のモードとして切り分けて利用してください。

7.7 まとめ

本章で最も重要な点は、通常の 3D 描画においては SmoothShader を標準的な起点として利用することです。静的メッシュ、スキニング、ノーマルマップのすべてを一つのインターフェースで扱い、has_bone や use_normal_map といったフラグで切り替える設計になっています。これにより、シーンに応じて個別のシェーダーを使い分ける手間が省けます。

一方で、基礎を学ぶ際はシンプルな Phong 系シェーダーが有効です。また、SmoothShader が多機能でありながら高いパフォーマンスを維持できているのは、WebGPU のバインドグループを活用し、更新頻度の異なるデータ（軽量なユニフォームと重量なボーンパレット）を分離して管理しているためです。

この構造を理解しておくことで、今後のサンプルコードやテストコードの読み解きがよりスムーズになります。次章では、これらのシェーダーの内部実装を理解するために不可欠な WGSL の読み方について解説します。

第 8 章

WGSL の読み方

本章では、webg の標準的なシェーダーである SmoothShader.js を基準に、WGSL の読み方と CPU から GPU へのデータフローを解説します。

WGSL を読み解く際の要諦は、単に構文を追うことではなく、「JavaScript 側で設定した値が、どのような経路を通って GPU のどの変数に到達するのか」というデータフローを把握することにあります。WebGPU では、この経路が @group、@binding、@location というアノテーションによって厳格に管理されています。

8.1　CPU から GPU へのデータフロー

WebGPU におけるデータの流れは、大きく分けて以下の 3 段階のステップで構成されています。

1. CPU 側での値の保持（JavaScript）

まず、shape.setMaterial() や shader.setLightPosition() などのメソッドが呼ばれると、値はシェーダークラス内の Float32Array（this.uniformData）の特定のオフセット（OFF_LIGHT など）に書き込まれます。この時点では、データはまだ CPU のメモリ上にあります。

2. GPU メモリへの転送（writeBuffer）

次に、gpu.queue.writeBuffer() を介して、CPU 上のデータが GPU 上の専用バッファ（GPUBuffer）へとコピーされます。これにより、GPU が高速にアクセス可能なメモリ領域にデータが配置されます。

3. シェーダーでの参照（WGSL）

最後に、描画直前に passEncoder.setBindGroup() が呼ばれることで、GPU バッファと WGSL 内の変数が紐付けられます。これにより、WGSL 側で u.lightPos と記述した際に、先ほど転送したバッファの特定の位置から値が読み出されます。

8.2　リソース接続の仕組み：Group, Binding, Location

WebGPU のリソース管理は、従来のグラフィックス API とは異なる「階層構造」を持っています。これを理解するために、「フォルダ（グループ） → スロット（バインディング） → データ」というイメージで捉えてください。

@group と @binding（外部リソースの接続）

@group と @binding は、CPU 側で作成したリソース（バッファやテクスチャ）をシェーダーに接続するための「住所」のようなものです。

- @group(g)（フォルダ）: リソースをまとめた大きなグループです。webg では、「データの更新頻度」によってグループを分けています。
- @binding(b)（スロット）: グループ内の個別の接続口です。

SmoothShader では、以下のように役割を分離して管理しています。

グループ	役割	更新頻度	具体的なリソース
group(0)	描画ごとの設定	非常に高い	行列、ライト位置、マテリアル係数、フラグ
group(1)	テクスチャ設定	中程度	サンプラー、ベーステクスチャ、ノーマルマップ
group(2)	スケルトン設定	低〜中	ボーンパレット（骨行列の配列）

具体的な WGSL 例

```
// group(0) の 0 番スロットに DrawUniforms 構造体を接続
@group(0) @binding(0) var<uniform> u : DrawUniforms;

// group(1) の 1 番スロットにベーステクスチャを接続
@group(1) @binding(1) var myTexture : texture_2d<f32>;

// group(2) の 0 番スロットにボーンパレットを接続
@group(2) @binding(0) var<uniform> skin : SkinUniforms;
```

@location（パイプライン内部の受け渡し）

@group が「外部（CPU）から GPU へ」の接続であるのに対し、@location は 「GPU 内部のステージ間」 でデータを運ぶための「配線」です。

1. 頂点バッファ → 頂点シェーダー: CPU から送られた頂点データ（座標、法線など）を @location(n) で受け取ります。
2. 頂点シェーダー → フラグメントシェーダー: 頂点シェーダーが計算した結果を @location(n) に乗せて送り出すと、GPU がピクセルごとに値を補間し、フラグメントシェーダーへ届けます。

具体的なデータフロー例

- 頂点入力: @location(0) position : vec3f → 頂点バッファの 0 番目の属性を読み込む。
- 頂点出力 → フラグメント入力: 頂点シェーダーで output.vNormal = ... とし、それを @location(1) vNormal : vec3f で受け取る。

8.3　ボーンパレット分離のメリットと実装

SmoothShader の設計で最も重要なのが、ボーンパレット（group(2)）を他のユニフォーム（group(0)）から分離している点です。

なぜ分離するのか（パフォーマンスの視点）

ボーンパレットはデータ量が非常に大きく、最大 320 本のボーンを扱う場合、約 15KB のデータになります。一方、行列やライト情報などの DrawUniforms は数百バイト程度と非常に軽量です。

もしこれらを一つのグループ（group(0)）にまとめていた場合、「スキニングを使わない単純な立方体」を描画するときでさえ、毎回 15KB のボーンデータを GPU に転送するか、巨大なバッファを抱えて処理することになり、効率が悪くなります。

分離による最適化の仕組み

SmoothShader では、描画対象に応じて group(2) に割り当てるリソースを切り替えています。

- スキニングメッシュの場合: そのスケルトン専用のボーンバッファを group(2) に接続します。
- 静的メッシュの場合: 全ての静的メッシュで共有する 「中身が空の共通ボーンバッファ（defaultBoneBindGroup）」 を group(2) に接続します。

これにより、静的メッシュの描画においては、ボーンデータの転送コストを完全にゼロにしつつ、シェーダー側では常に skin.bones という同じインターフェースで記述できるため、コードの共通化と高速化を同時に実現しています。

8.4 WGSL の文法：構文と演算

シェーダー内のロジックを理解するために、基本となる文法を整理します。

変数の宣言と寿命

WGSL では、変数の性質に応じて 3 つの宣言キーワードを使い分けます。

キーワード	性質	説明
`let`	不変 (Immutable)	一度代入すると変更できない。計算結果の一時保存に多用される。
`var`	可変 (Mutable)	値の書き換えが可能。関数内での計算状態の保持に使用する。
`const`	定数 (Constant)	コンパイル時に決定される値。不変であり、メモリ効率が良い。

制御構文

条件分岐やループなどの制御フローは、JavaScript や C 言語に近い構文です。

条件分岐 (`if`, `else`)

```
if (condition) {
    // 真の場合の処理
} else {
    // 偽の場合の処理
}
```

条件選択 (`select`) シェーダーでは、`if` による分岐がパフォーマンスに影響を与えるため、`select` 関数を用いて値を切り替える手法がよく使われます。`select(false_value, true_value, condition)` → `condition` が真なら `true_value` を、偽なら `false_value` を返します。

関数定義

関数は `fn` キーワードを用いて定義し、戻り値の型を `->` で指定します。

```
fn calculateLighting(normal: vec3f, lightDir: vec3f) -> f32 {
    return max(dot(normal, lightDir), 0.0);
}
```

シェーダーで多用される数学関数

WGSL には 3D 計算に特化した組み込み関数が豊富に用意されています。

関数名	役割	用途
dot(a, b)	内積	2 つのベクトルのなす角（ライティングの強度計算など）
cross(a, b)	外積	2 つのベクトルに垂直なベクトル（面法線の算出など）
normalize(v)	正規化	ベクトルの長さを 1 にする（方向ベクトルの作成）
reflect(i, n)	反射	入射ベクトル i を法線 n で反射させる（鏡面反射）
mix(a, b, t)	線形補間	a と b を t (0〜1) で混ぜ合わせる（色のブレンド）
clamp(v, min, max)	範囲制限	値を一定の範囲内に収める（色の飽和防止）

8.5 シェーダーパイプラインの構造

WebGPU の描画は、頂点シェーダーからフラグメントシェーダーへとデータが流れるパイプラインとして構成されます。

エントリーポイントの役割

頂点シェーダー (@vertex) は、頂点一つひとつを計算し、画面上のどこに配置するかを決定します。頂点バッファから @location を通じて座標や UV、法線を受け取り、最終的に @builtin(position) としてクリップ空間座標を出力します。

フラグメントシェーダー (@fragment) は、塗りつぶされるピクセル一つひとつの色を決定します。頂点シェーダーから補間されて送られてきたデータを受け取り、最終的に @location(0) へピクセルの色を出力します。

リソースの受け渡しとデータフロー

SmoothShader では、複数のパラメータを params や flags といった vec4f 型の束にまとめて格納しています。例えば u.flags.x が 0.0 かどうかでスキニングの有無を判定しています。読み解く際は、u.flags.w（ノーマルマップ使用フラグ）のような記述を見かけたら、「JavaScript 側の useNormalMap() メソッドがこの値を書き換えている」と結びつけてください。

また、WGSL 側の @location 番号は、CPU 側の shaderLocation 設定と完全に一致している必要があります。この番号が不整合を起こすと、ライティング計算が不正な結果となるため、注意が必要です。

8.6　ユニフォームバッファと CPU 側の対応付け

SmoothShader では、ユニフォーム構造体の中に、行列、カラー、パラメータ、フラグが厳密なオフセットで定義されています。

```
struct DrawUniforms {
  projMatrix : mat4x4<f32>,    // OFF_PROJ (0)
  viewMatrix : mat4x4<f32>,    // OFF_VIEW (16)
  normalMatrix : mat4x4<f32>,  // OFF_NORM (32)
  lightPos : vec4<f32>,        // OFF_LIGHT (48)
  color : vec4<f32>,           // OFF_COLOR (52)
  // ...
};
```

読み解く際のポイントは、JavaScript 側の OFF_ 定数と WGSL の構造体の順番を照らし合わせることです。例えば、setLightPosition() が呼ばれると、this.uniformData の OFF_LIGHT (48 番目の float) が更新され、それが WGSL の u.lightPos として現れます。

8.7　テクスチャサンプリングの読解

SmoothShader では、ベーステクスチャとノーマルマップを group(1) でまとめて管理しています。

サンプリングの基本

WGSL でテクスチャから色を取得する際は、以下のように記述します。

```
let texColor = textureSample(myTexture, mySampler, input.vTexCoord);
```

ここで重要なのは、myTexture（画像データ）と mySampler（拡大・縮小の補間方法）の 2 つをセットで指定することです。

textureSampleLevel の使い分け

SmoothShader のノーマルマップ処理では、textureSample ではなく textureSampleLevel(..., 0.0) が使用されています。これは WebGPU の「非一様制御フロー（Non-uniform Control Flow）」という制約を避けるためです。

if (u.flags.w != 0.0) のような条件分岐の中で通常の textureSample を使用すると、GPU のハードウェアによっては描画エラーやパフォーマンス低下を招くことがあります。textureSampleLevel を使い、ミップマップレベルを 0.0（最高解像度）に固定することで、どのような分岐条件下でも安全にテクスチャをサンプリングできる実装になっています。

バインドグループのキャッシュ戦略

JavaScript 側の getBindGroup1() メソッドでは、WeakMap を使用してテクスチャの組み合わせをキャッシュしています。テクスチャのバインドグループ作成はコストが高い処理であるため、「同じテクスチャの組み合わせであれば、以前作成したグループを使い回す」ことで、描画ループ中の CPU 負荷を最小限に抑えています。

8.8　ノーマルマップと TBN 行列の仕組み

SmoothShader の最大の特徴は、メッシュ側に「接線（Tangent）データ」を持っていなくても、ノーマルマップや面単位の陰影（フラットシェーディング）を実現できる点にあります。ここで重要になるのが TBN という概念です。

TBN とは何か

TBN とは、Tangent（接線）、Bitangent（従接線）、Normal（法線） の 3 つのベクトルで構成される座標系（基底ベクトル）のことです。

ノーマルマップのピクセルに保存されている RGB 値は、「その面から見て、どちらに法線が傾いているか」というローカルな方向（接線空間 / Tangent Space）を記録しています。- ノーマルマップの中での「上」: 常に面に対して垂直な方向（ベクトル [0, 0, 1]）です。-

3D 空間での「上」: モデルが回転したり曲がったりしているため、場所によって方向が異なります。

もし、ノーマルマップの値をそのまま 3D 空間で使ってしまうと、モデルを回転させたときに光の当たり方が不自然な結果となります。そこで、「テクスチャ上の方向」を「現在のモデルの向きに合わせた方向」に変換するための変換行列（基底変換）が必要になります。それが TBN 行列 です。

T, B, N それぞれの役割

TBN は、面の上に置かれた「局所的な座標系」のようなものです。- N (Normal / 法線): 面の表面から垂直に突き出しているベクトル。- T (Tangent / 接線): テクスチャの U 方向（横方向）に沿ったベクトル。- B (Bitangent / 従接線): テクスチャの V 方向（縦方向）に沿ったベクトル。N と T の両方に垂直な方向です。

この 3 つのベクトルを並べて行列（`mat3x3`）にすることで、接線空間からビュー空間（またはワールド空間）への変換が可能になります。

微分による TBN 行列の再構成

通常、TBN を構築するにはモデルデータに最初から「接線（Tangent）」の情報を持たせておく必要があります。しかし、`SmoothShader` ではフラグメントシェーダー内で `dpdx` と `dpdy` という微分関数を使用しています。

```
let dp1 = dpdx(input.vPosition); // 画面上の X 方向への位置変化
let dp2 = dpdy(input.vPosition); // 画面上の Y 方向への位置変化
```

これは、「隣のピクセルと比べて、UV 座標と位置がどう変化したか」を計算することで、接線（T）と従接線（B）をその場で算出し、TBN 行列を構築する手法です。この特筆すべき実装上の工夫により、ユーザーは複雑な頂点データの準備をすることなく、あらゆるメッシュにノーマルマップを適用させることができます。

シェーダー内での処理フロー

SmoothShader.js では、具体的に以下の手順で計算が行われています。1. ベクトルの抽出: テクスチャから色を読み込み、それを [-1, 1] の範囲に変換して、接線空間での法線ベクトル $\vec{n}_{tangent}$ を得ます。2. TBN 行列の構築: 微分関数から得た T, B, N を組み合わせて行列を作成します。3. 座標変換: $\vec{n}_{view}$ = TBN 行列 × $\vec{n}_{tangent}$ により、ノーマルマップの「傾き」を実際の 3D 空間での正しい法線方向へ変換します。4. ライティング計算: この変換後の法線 $\vec{n}_{view}$ を使い、光との角度（内積）を計算して陰影を決定します。

フラットシェーディングの実現

同様の微分によるアプローチを使い、u.debugFlags.y（flat_shading フラグ）が有効な場合は、頂点法線の補間を無視して、三角形の面法線を直接算出します。

```
nnormal = normalize(cross(dpdy(input.vPosition), dpdx(input.vPosition))) * facing;
```

2 つの微分ベクトルの外積（cross）を取ることで、その面の正確な法線が得られます。これにより、ローポリゴンモデルのようなカクカクとした面単位の陰影を、シェーダー側の設定一つで切り替えることが可能です。

8.9 スキニングの読解

スキニング処理は、group(2) に配置されたボーンパレットを利用して、頂点シェーダーで行われます。

圧縮パレット表現

SmoothShader では、1 本のボーン行列（4x4）をそのまま保持せず、vec4 x 3 の形式で圧縮して保持しています。これは、4 行目の [0, 0, 0, 1] という固定値を除去し、転送データ量を削減するためです。

頂点変形のフロー

頂点シェーダー (vs_main) では、以下の手順で頂点を変形させています。1. ボーンの抽出: 頂点属性 index（ボーン番号）を使い、skin.bones 配列から最大 4 本のボーン行列を抽出します。2. ウェイト合成: 各ボーン行列に weight（影響度）を掛け合わせ、その頂点専用の合成行列 skinMat を構築します。3. 座標変換: 元の頂点座標に skinMat を掛け、その後 viewMatrix でビュー空間へ変換します。

8.10　ダイナミックオフセットによる大量描画の最適化

最後に、SmoothShader が大量のオブジェクトを高速に描画できる核心的な仕組みである「ダイナミックオフセット」について解説します。

巨大な単一バッファの運用

通常、オブジェクトごとにユニフォームバッファを作成すると、描画のたびに setBindGroup を呼び出す必要があり、CPU 負荷が高くなります。SmoothShader では、maxUniforms (2048 個分) のデータを格納できる一つの巨大なバッファをあらかじめ作成しています。

オフセットによる参照切り替え

描画時に、そのオブジェクトがバッファの「どこからデータが始まっているか」というオフセット（開始位置）だけを指定してバインドします。

- JavaScript 側: passEncoder.setBindGroup(0, this.uniformBindGroup, [offset])
- WGSL 側: 指定されたオフセットから DrawUniforms 構造体としてデータを読み出す。

この手法により、バッファの切り替え（バインド）回数を劇的に減らし、GPU のパイプラインを効率的に稼働させています。SmoothShader の uniformStride が 256 バイト境界に合わせられているのは、WebGPU の仕様でダイナミックオフセットがこの境界に従う必要があるためです。

8.11 まとめ：読解のチェックリスト

SmoothShader を読み解く際は、以下の流れで確認してください。

1. データの入り口: Setter → uniformData (JS) → writeBuffer → GPUBuffer
2. リソースの接続:
   - group(0): 描画ごとの設定（ダイナミックオフセットで切り替え）
   - group(1): テクスチャの組み合わせ（キャッシュして使い回し）
   - group(2): ボーンパレット（スケルトンごとに分離）
3. 内部の配線: @location を通じて頂点データ → 頂点シェーダー → フラグメントシェーダーへと流れる。
4. 特殊処理: dpdx/dpdy による TBN 再構成や、textureSampleLevel による安全なサンプリングが行われているか。

この構造を理解していれば、独自のパラメータを追加したり、描画ロジックをカスタマイズしたりすることが容易になります。

第 9 章

シェーダーの実装

本章では、webg のシェーダーを単に「機能として利用する」段階から一歩進め、Shader.js およびその派生クラスの実装をどのように読み解くべきかを解説します。

本章では、webg/Shader.js をはじめ、book/examples/ に用意された Phong 系のサンプル、および webg/Background.js や webg/Wireframe.js といった特殊目的のシェーダーファイルを対象に解説します。

第 8 章では WGSL の読解方法を扱いましたが、本章では CPU 側のクラス設計に重点を置きます。具体的には、「JavaScript 側で何を準備し、どのように GPU へ転送し、どのクラスがどのような役割を担っているか」という構造を明らかにします。

まず理解しておくべき点は、Shader.js が単に WGSL のソースコードを管理するだけのクラスに留まらないということです。webg のシェーダークラスは、パイプラインの作成、ユニフォームバッファの管理、バインドグループ（bind group）の生成、デフォルトテクスチャの提供、パラメータの反映、そして描画時の更新処理までを包括的に管理しています。そのため、シェーダーの実装を分析する際は、WGSL の関数を追う前に、「クラスがどのような状態を保持しているか」を確認することが効率的です。

特に重要なのが、Shape.setMaterial() で設定したパラメータが最終的に各シェーダーの doParameter() メソッドへ渡され、そこで各セッター（setter）メソッドに振り分けられ、必要に応じて初期値へリセットされるというフローです。この仕組みを把握することで、そのシェーダーがユーザーに対してどのような操作インターフェースを提供しているかが明確になります。

また、実装の比較には Phong 系 4 種類（Phong、NormPhong、BonePhong、BoneNormPhong）が最適です。これらは「静的メッシュ」「ノーマルマップ」「スキニング」、そして「その両

方の組み合わせ」という段階的な差分として構成されており、webg の 3D マテリアルを理解する上での核心部分となります。最小限の構成を持つ Phong を起点として、機能がどのように拡張されていくかを追うことで、実装の変更点を容易に把握できます。

9.1 Shader.js の共通構造

webg の多くのシェーダーは共通の処理フローを共有しています。具体的には、プロジェクション（projection）行列やモデルビュー（model-view）行列を受け取り、ユニフォームバッファを更新し、必要なテクスチャとサンプラーをバインドグループにまとめ、描画直前に GPU へ反映させるという流れです。これらの処理を個別のシェーダーごとに実装すると、機能追加やデバッグオプションの導入のたびにコードの重複と不整合が発生します。そこで、Shader.js はこれらの共通処理を引き受ける基底クラスとして機能しています。

派生クラスは、この基底クラスを継承し、固有の WGSL コード、ユニフォームレイアウト、初期パラメータ、バインドグループレイアウト、およびパラメータを反映させるためのセッターメソッドを定義します。

Shader.js を分析する際は、以下の順序で確認することをお勧めします。

1. constructor でどのような変数が初期化されているか。
2. createResources() で GPU 側にどのようなリソースを生成しているか。
3. getBindGroup() でどのリソースをどのように結び付けているか。
4. updateUniforms() や各セッターメソッドが、バッファのどの位置にデータを書き込んでいるか。
5. doParameter() が派生クラスでどのように拡張され、パラメータが処理されているか。

この流れで追跡することで、各シェーダークラスの「共通部分」と「固有部分」を体系的に理解できます。

特に注目すべきは、default と change という 2 つのパラメータ管理テーブルです。default はシェーダーの初期値を保持し、change は Shape.shaderParameter() によって現在上書きされている値を保持します。doParameter(param) メソッドは、指定されたキーをセッターへ流し、直前の呼び出しで上書きされていたものの現在は指定されていないキーを初期値へ戻します。この設計により、ユーザーは毎回すべてのパラメータを再設定する必要がなく、必要な項目だけを更新することが可能です。

シェーダーを拡張して新しいパラメータを追加する場合は、単に WGSL を書き換えるだけ

では不十分です。初期値の設定、セッターメソッドの実装、doParameter() 内での反映処理、そして必要に応じて setDefaultParam() 側のインターフェース整備を行い、文書とサンプルコードに反映させる必要があります。実装を追加する際は、WGSL だけでなく、CPU 側のインターフェースと検証手段までを一貫して整備することが、webg の開発思想に沿った進め方です。

9.2 標準シェーダーとしての SmoothShader

第 7 章では、ユーザーが通常選択する標準シェーダーとして SmoothShader を紹介しました。本章では、実装読解の教材として Phong 系 4 種類を詳しく扱いますが、実際のアプリケーションでは SmoothShader が最も広い範囲を受け持ちます。SmoothShader は、静的メッシュ、テクスチャ、フォグ、スキニング、ウェイト可視化、ノーマルマップ、面単位の陰影、裏面デバッグを 1 本のシェーダーに統合した実用向けのシェーダーです。

SmoothShader は、単に Phong 系の機能をすべて 1 ファイルへ足し合わせたものではありません。Phong 系 4 種類が学習用に分離されているのに対し、SmoothShader はアプリ側で shaderClass: SmoothShader を指定するだけで、Shape ごとの material parameter によって描画経路を切り替えることを目的としています。たとえば、通常の形状では has_bone: 0 のまま描画し、スキンドメッシュでは has_bone: 1 を設定し、ノーマルマップを持つ形状だけ use_normal_map: 1 と normal_texture を指定します。つまり、クラスを切り替えるのではなく、同じ SmoothShader の中で Shape ごとに必要な機能を選択する設計となっています。

SmoothShader の主な material parameter とその役割は以下の通りです。

色と質感の制御

color は material の基本色です。glTF の base color や、手動で設定したベース色として扱います。

multiplyColor は、color と テクスチャを合成した後に掛け合わせる乗算色です。元マテリアルの色差を保ったまま、ユニットの色や状態を示す色を重ねる用途に向いています。同じ GLB データを複数のキャラクターとして表示し、キャラクターごとに少し色味だけ変えたい場合は、color を上書きするよりも multiplyColor を使う方が、元の質感を損なわずに調整できるため安全です。

addColor は、color と texture を合成した後に加算する色です。選択中の強調、ダメージ表示、警告表示、発光寄りのハイライトなど、元の色を残したまま明るさや色味を足したい場合に使用します。

これらの色の合成順序は、概念的に以下のようになります。base = color * texture → adjusted = base * multiplyColor + addColor → lit = lighting(adjusted)

use_texture と texture は、ベーステクスチャを使用するか、およびどの texture を参照するかを指定します。

ambient、specular、power、emissive は、環境光、鏡面反射、ハイライトの鋭さ、発光寄りの見え方を調整します。emissive は照明の影響を弱めて自発光に近い見え方へ寄せるための値で、現在は 0.0 から 1.0 の範囲で扱います。

機能的な切り替えとデバッグ

has_bone は、スキニング用の bone palette を使うかを指定します。スキンドメッシュでは 1 に設定し、通常の静的メッシュでは 0 のままにします。

weight_debug は、頂点ウェイトを色で表示するデバッグ機能です。スキニングの影響範囲を確認したいときに使用し、有効な間は通常の material 色やライティングよりもウェイト可視化が優先されます。

use_normal_map、normal_texture、normal_strength は、ノーマルマップの有無、使用する texture、法線の効き具合を指定します。normal_strength を下げると元の頂点法線に近づき、上げるとノーマルマップによる凹凸表現が強く現れます。

flat_shading は、補間済み頂点法線ではなく、fragment 微分から面法線を再構成して描画するための指定です。低ポリゴン表現や、面の向きを確認するデバッグで使用します。通常のスムーズシェーディングでは頂点法線が三角形内で補間されるため、低ポリゴンの形状でも境界がなめらかに見えます。一方 flat_shading: 1 のときは、fragment shader が dpdx / dpdy からその fragment が属する三角形の面法線を再構成するため、面ごとの向きがはっきりした見え方になります。

backface_debug と backface_color は、裏面を指定色で表示し、面の向きや culling の問題を確認するための指定です。モデルの一部が裏返って見える場合や、両面表示中に面の向きを確認したい場合に役立ちます。

fog_color、fog_near、fog_far、fog_density、fog_mode は、距離に応じた fog の色と

減衰を指定します。シーンの奥行き感を出すだけでなく、遠景の情報量を抑えて視認性を上げる目的にも利用できます。

実装例：flat_shading の利用

webg には現在、FlatShader.js という独立した標準シェーダーファイルは存在しません。面単位の陰影を使いたい場合は、SmoothShader の flat_shading を 1 に設定します。これは別のシェーダークラスへ切り替えるのではなく、同じ SmoothShader の fragment shader 内で法線の算出方法を切り替える方式です。

```
shape.setMaterial("smooth-shader", {
  color: [0.85, 0.72, 0.48, 1.0],
  ambient: 0.28,
  specular: 0.35,
  power: 28.0,
  flat_shading: 1
});
```

flat_shading はメッシュの頂点データを変更せず、描画時の法線計算だけを切り替えるため、同じ geometry を「滑らかに見せる」状態と「面を立てて見せる」状態で容易に比較できます。

9.3 Phong シェーダーの進化と差分解析

Phong 系 4 種類のシェーダーを段階的に比較することで、その構造がより鮮明になります。基本形である Phong に、ノーマルマップを加えたのが NormPhong、スキニングを加えたのが BonePhong、そしてその両方を統合したのが BoneNormPhong です。

コンピュータグラフィックスにおいて照明を表現するには、物理現象を適切にモデル化（簡略化）する必要があります。webg では、物体を照らす光の計算を次の 3 つに分解して計算し、最後にそれらを合計する「Phong shading」を採用しています。

環境光 (ambient light)

直接光源から来る光ではなく、周囲（環境）に散乱して存在する光です。実際にはあらゆる光が反射や拡散の結果として存在しますが、それらを一様にまとめたものを環境光として表します。曇りの日には環境光の影響が強く、宇宙空間では非常に弱くなります。なお、環境光の強さにかかわらず、物体自体が明るく見える場合は「発光（emissive）」として扱われます。

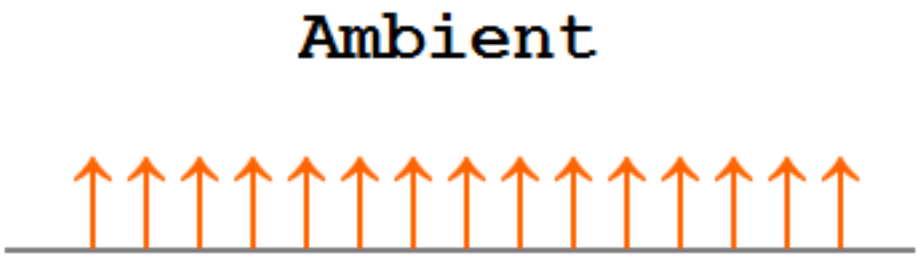


図 9.1: ambient light

拡散光 (diffuse light)

光に照らされた物体の表面からあらゆる方向に反射（拡散）する光です。表面を照らす光の強さは、光の方向に対して垂直な面が最も強く、傾いた面ほど弱くなります。これは、物体の表面に対する入射光の垂直成分の大きさに比例します。

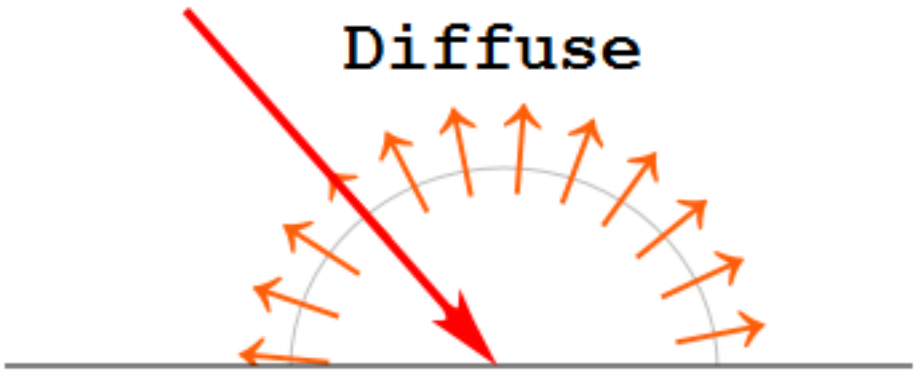


図 9.2: diffuse light

鏡面反射光 (specular light)

スペキュラ（specular）またはハイライト（highlight）とも呼ばれます。なめらかな面に入射した光の一部が、鏡のように特定の方向に反射する光です。入射光の角度によって反射する方向と強さが変化し、表面の性質によって、金属のように鋭く反射する場合と、プラスチックのように鈍く反射する場合があります。パラメータの `power` が大きくなるほど、反射光の範囲が狭まり、鋭いハイライトになります。

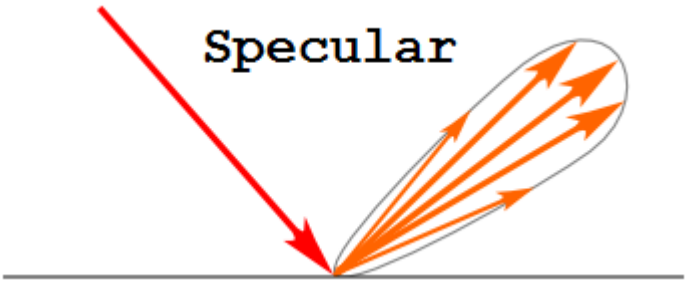


図 9.3: specular light

実際の計算手法

物体の色や照明は、最終的にスクリーンに表示される点（ピクセル）の色として決定されます。個々のピクセルに対応する物体表面の点をベクトル $\mathbf{V}$、点光源の座標をベクトル $\mathbf{L}$ とすると、表面の点から光源に向かう入射光ベクトルは $\mathbf{L} - \mathbf{V}$ で表せます。視点が原点（視点座標系に変換済み）にある場合、表面の点から視点に向かう視線ベクトルは $-\mathbf{V}$ となります。

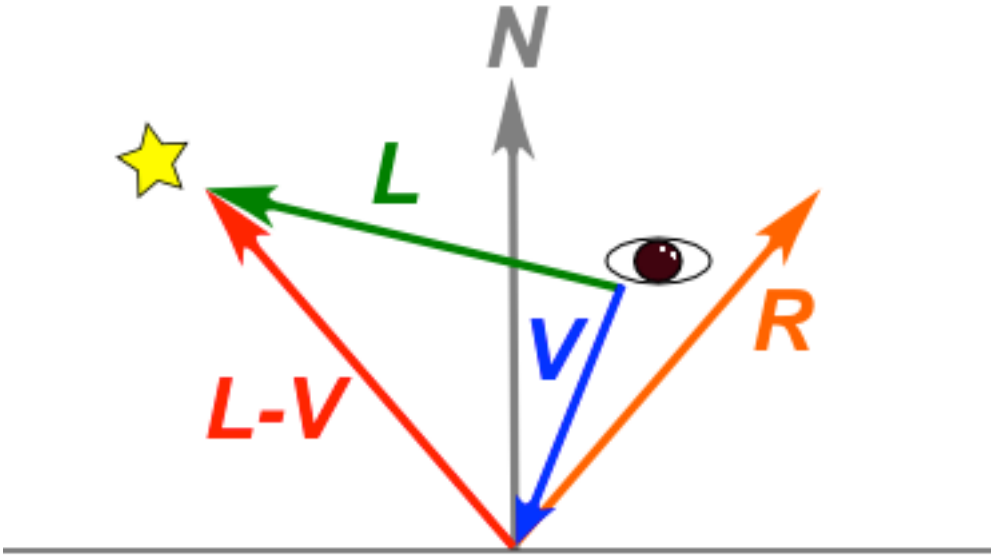


図 9.4: Phong shading

面の法線ベクトル $\mathbf{N}$ と入射光ベクトル $\mathbf{L}-\mathbf{V}$ がともに正規化されている場合、その内積 $\mathrm{dot}(\mathbf{N},\mathbf{L}-\mathbf{V})$ が拡散光の強度となります。また、反射光の強度は、視線ベクトル $-\mathbf{V}$ と反射光ベクトル $\mathbf{R}$ の角度 ϕ の関数として求められます。この関数として $\cos\phi$ を用いることで、特定の角度でキラリと光る反射を表現でき、その変化の鋭さを `power` パラメータで制御しています。

Phong の実装ポイント

`Phong` はすべての比較基準となるシェーダーです。分析の際は、まず `Uniforms` 構造体が保持するデータ、`doParameter()` が呼び出すセッターメソッド、そして WGSL 側で `params0` / `params1` がどのように利用されているかを確認してください。

`Phong` のパラメータは、効率的な転送のために `vec4f` 型にまとめられています。例えば `params0` には `ambient`, `specular`, `power`, `emissive` が、`params1` には `useTexture`, `backfaceDebug` などが格納されています。JavaScript 側の `setAmbientLight()` が `params0.x` に、`setSpecular()` が `params0.y` に対応するというように、セッターと WGSL フィールドの対応関係を整理して読むと理解がスムーズになります。

`Phong` は実用的な基本シェーダーであり、立方体や球、床などの基本オブジェクトに広く適用されます。ライティング、テクスチャ、フォグ、エミッシブ、スペキュラといった基本機能を網羅しており、スキニングやノーマルマップを必要としない場面に最適です。その役割は、

モデルビュー変換された position と normal を用いて拡散反射と鏡面反射を計算し、ベーステクスチャを合成し、最終的にフォグを適用して色を決定することです。

利用者側の最小実装例は以下の通りです。

```
import Phong from "./book/examples/Phong.js";
import Shape from "./webg/Shape.js";

const shader = new Phong(gpu);
await shader.init();
Shape.prototype.shader = shader;

shader.setLightPosition([40.0, 90.0, 150.0, 1.0]);

const shape = new Shape(gpu);
shape.applyPrimitiveAsset(asset);
shape.endShape();

shape.setMaterial("phong", {
  color: [1.0, 0.9, 0.7, 1.0],
  ambient: 0.28,
  specular: 0.82,
  power: 34.0,
  use_texture: 1,
  texture
});
```

この実装におけるポイントは、シーン全体で共有する光源情報はシェーダー側に保持し、オブジェクトごとの質感（マテリアル）は shape.setMaterial() を通じて Shape 側に渡している点です。実装上のポイントは、setProjectionMatrix() や setModelViewMatrix() などの共通セッターと、Shape ごとの個別パラメータを区別して整理することです。

以下に、book/examples/Phong.js の頂点シェーダー vsMain とフラグメントシェーダー fsMain を掲載します。

```
@vertex
fn vsMain(input : VSIn) -> VSOut {
  var output : VSOut;

  // 1. モデル空間の position を model-view 行列で 視点空間（視点座標系）へ変換する
```

```
  //    ここで得た worldPos は、このシェーダーでは「カメラから見た座標」として
  //    以降のライティングとフォグの基準になる
  let worldPos = uniforms.modelView * vec4f(input.position, 1.0);

  // 2. 視点空間の position をさらに projection 行列でクリップ空間へ変換する
  //    この値だけが最終的な画面上の頂点位置としてラスタライザへ渡る
  output.position = uniforms.proj * worldPos;

  // 3. フラグメントシェーダー側で点光源方向やフォグ距離を計算できるように、
  //    視点空間の位置をそのまま補間用 varyings として渡す
  output.vPosition = worldPos.xyz;

  // 4. 法線も normal 行列で 視点空間へ変換してから渡す
  //    法線は位置と違って平行移動の影響を受けないため w=0.0 で掛ける
  output.vNormal = (uniforms.normalMat * vec4f(input.normal, 0.0)).xyz;

  // 5. UV はこの段階では加工せず、そのままフラグメント側へ渡す
  output.vTexCoord = input.texCoord;
  return output;
}
```

```
@fragment
fn fsMain(input : FSIn) -> @location(0) vec4f {
  // 1. frontFacing を見て、裏面では法線方向を反転する
  //    両面描画時にそのままの法線を使うと、裏面側だけライティングが反転して
  //    不自然に暗くなるため、ここで向きをそろえる
  let facing = select(-1.0, 1.0, input.frontFacing);
  let nnormal = normalize(input.vNormal) * facing;
  let backfaceDebug = uniforms.params1.y;
  var litVec : vec3f;

  // 2. 光源ベクトルを作る
  //    w != 0.0 のときは点光源として「光源位置 - フラグメント位置」を使い、
  //    w == 0.0 のときは平行光方向をそのまま正規化して使う
  if (uniforms.lightPos.w != 0.0) {
    litVec = normalize(uniforms.lightPos.xyz - input.vPosition);
  } else {
    litVec = normalize(uniforms.lightPos.xyz);
  }
```

```
// 3. 視線方向と反射ベクトルを求める
//    視点空間ではカメラは原点にいる前提なので、
//    フラグメント位置の反対向きが視線方向になる
let eyeVec = normalize(-input.vPosition);
let refVec = normalize(reflect(-litVec, nnormal));

// 4. ユニフォームへパックしてある材質値を取り出す
let ambient = uniforms.params0.x;
let specular = uniforms.params0.y;
let power = uniforms.params0.z;
let emissive = uniforms.params0.w;
var diff : f32;
var ispec : f32;

// 5. 発光材質でなければ通常の拡散反射 + 鏡面反射を計算する
//    emissive != 0.0 のときは照明を受けない発光物として扱い、
//    拡散項だけを強制的に 1.0 - ambient 相当へ寄せる
if (emissive == 0.0) {
  diff = max(dot(nnormal, litVec), 0.0) * (1.0 - ambient);
  ispec = specular * pow(max(dot(refVec, eyeVec), 0.0), power);
} else {
  diff = 1.0 - ambient;
  ispec = 0.0;
}

// 6. テクスチャを使う場合はサンプリングした色をベース色へ掛け合わせる
//    alpha には「ベース色だけで塗るか」「テクスチャ色を採用するか」の
//    混ぜ比率としての意味も持たせている
let texFlag = uniforms.params1.x;
var color = uniforms.color;
if (texFlag > 0.5) {
  let tex = textureSample(uTexture, uSampler, input.vTexCoord);
  color = uniforms.color * tex;
  color = mix(diff * uniforms.color, color, uniforms.color.w);
}

// 7. ambient + diffuse + specular を合成してライティング後の色を作る
let lit = vec4f(color.rgb * (ambient + diff) + vec3f(1.0) * ispec, 1.0);

// 8. フォグ係数を求める
//    linear fog のときは near/far 区間で線形補間し、
//    exp fog のときは density を使った指数減衰にする
```

```
  let fogDistance = length(input.vPosition);
  let fogNear = uniforms.fogParams.x;
  let fogFar = uniforms.fogParams.y;
  let fogDensity = uniforms.fogParams.z;
  let fogMode = uniforms.fogParams.w;
  var fogFactor = 1.0;
  if (fogMode > 0.5 && fogMode < 1.5) {
    let fogRange = max(fogFar - fogNear, 0.0001);
    let linearFactor = clamp((fogFar - fogDistance) / fogRange, 0.0, 1.0);
    let linearWeight = clamp(fogDensity * 50.0, 0.0, 1.0);
    fogFactor = 1.0 - (1.0 - linearFactor) * linearWeight;
  } else if (fogMode >= 1.5) {
    fogFactor = clamp(exp(-fogDensity * fogDistance), 0.0, 1.0);
  }

  // 9. 裏面デバッグが有効なら、裏面だけを固定色で返す
  //    ライティング計算よりも「裏面が出ているか」の確認を優先する用途
  if (backfaceDebug > 0.5 && !input.frontFacing) {
    return vec4f(1.0, 0.0, 1.0, 1.0);
  }

  // 10. 最後にフォグ色とライティング後の色を fogFactor で補間して出力する
  return vec4f(mix(uniforms.fogColor.rgb, lit.rgb, fogFactor), lit.a);
}
```

NormPhong による詳細表現の追加

NormPhong は Phong を理解した後に、その差分として分析するのが最も効率的です。注目すべき点は、ノーマルマップ用テクスチャのバインディングが追加され、フラグメントシェーダー内でサンプリングした法線と元の法線を合成する処理が導入されていることです。

WGSL 側では、まず `@group(...) @binding(...) var uNormalTexture : texture_2d<f32>;` という記述でノーマルマップ用リソースが定義されています。その後、フラグメントシェーダー内で「ノーマルマップのサンプリング」→ 「dpdx / dpdy による接線ベクトルおよび従法線ベクトルの再構成」→ 「TBN 行列による接空間からワールド空間への変換」→「normal_strength による補間」という順で処理が進みます。

book/examples/NormPhong.js では、この差分が次のように実装されています。ここでは

Phong と共通の部分を省き、ノーマルマップ対応で新しく増えた箇所と、差し替わったフラグメント側の処理を掲載します。

```
@group(0) @binding(0) var<uniform> uniforms : Uniforms;
@group(0) @binding(1) var uTexture : texture_2d<f32>;
@group(0) @binding(2) var uSampler : sampler;
// ‘Phong‘ との差分:
// - binding(3) に normal map 用 texture を追加する
@group(0) @binding(3) var uNormalTexture : texture_2d<f32>;

@fragment
fn fsMain(input : FSIn) -> @location(0) vec4f {
  // ‘Phong‘ との差分 1:
  // - params1 の割り当てを normal map 用に拡張する
  //   x=useTexture, y=useNormalMap, z=normalStrength, w=backfaceDebug
  let ambient = uniforms.params0.x;
  let specular = uniforms.params0.y;
  let power = uniforms.params0.z;
  let emissive = uniforms.params0.w;

  let texFlag = uniforms.params1.x;
  let normalFlag = uniforms.params1.y;
  let normalStrength = uniforms.params1.z;
  let backfaceDebug = uniforms.params1.w;

  // ‘Phong‘ との差分 2:
  // - まず補間された頂点法線を基準法線として正規化する
  var nnormal = normalize(input.vNormal);
  if (normalFlag > 0.5) {
    // ‘Phong‘ との差分 3:
    // - normal map を読み、TBN 行列で tangent space から eye space へ戻す
    // WGSL 制約回避:
    // dpdx/dpdy 由来の非一様制御フロー内では textureSample が使えないため、
    // 明示 LOD の textureSampleLevel(..., 0.0) で法線マップを読む
    let ntex = textureSampleLevel(uNormalTexture, uSampler,
               input.vTexCoord, 0.0).xyz * 2.0 - vec3f(1.0, 1.0, 1.0);

    // screen-space 微分から position と UV の変化量を取り出し、
    // tangent / bitangent を再構成する
    let dp1 = dpdx(input.vPosition);
    let dp2 = dpdy(input.vPosition);
    let duv1 = dpdx(input.vTexCoord);
```

```
  let duv2 = dpdy(input.vTexCoord);

  // UV ヤコビアンの行列式
  // - UV が退化していると 0 付近になり、接線空間を安全に作れない
  let det = duv1.x * duv2.y - duv1.y * duv2.x;
  if (abs(det) > 1.0e-8) {
    let invDet = 1.0 / det;

    // 接線 T を再構成する
    var tangent = (dp1 * duv2.y - dp2 * duv1.y) * invDet;
    let tlen0 = length(tangent);
    if (tlen0 > 1.0e-8) {
      tangent = tangent / tlen0;

      // Gram-Schmidt で法線 N に直交化し、TBN を安定化する
      tangent = tangent - nnormal * dot(nnormal, tangent);
      let tlen1 = length(tangent);
      if (tlen1 > 1.0e-8) {
        tangent = tangent / tlen1;

        // B は外積で構成する
        var bitangent = cross(nnormal, tangent);
        let blen = length(bitangent);
        if (blen > 1.0e-8) {
          bitangent = bitangent / blen;
          let tbn = mat3x3f(tangent, bitangent, nnormal);
          let mapped = normalize(tbn * ntex);

          // normalStrength で元法線と normal map 法線の混ぜ比率を調整する
          let w = clamp(normalStrength, 0.0, 2.0);
          nnormal = normalize(mix(nnormal, mapped, w));
        }
      }
    }
  }
}

// 以降の lighting / fog / texture 合成は ‘Phong‘ と同じ流れを使う
var litVec : vec3f;
if (uniforms.lightPos.w != 0.0) {
  litVec = normalize(uniforms.lightPos.xyz - input.vPosition);
} else {
```

```
    litVec = normalize(uniforms.lightPos.xyz);
  }
```

NormPhong は、表面の微細な凹凸感を表現するためのマテリアルです。実際のポリゴン数を増やさずに詳細な陰影を表現することを目的としています。ユーザー側では use_normal_map などのパラメータが追加されます。normal_texture を指定しない場合は、シェーダーが用意する平坦な既定テクスチャを明示的に使い、ノーマルマップを持たない材質として描画します。これはエラーを隠す代替処理ではなく、ノーマルマップを使わない状態を表す標準設定です。

BonePhong による動的変形の導入

BonePhong では、特に頂点シェーダーの処理に注目してください。Phong とは異なり、頂点位置をそのまま使用せず、ボーンパレットを用いて変形させた後に後段へ渡します。

コード内には let i0 = i32(input.index.x) * 3; といった式が現れます。webg では 1 本のボーンを vec4 x 3 の形式で保持する最適化されたパレット表現を採用しているため、標準的な 4x4 行列の配置とは記述が異なります。この設計を理解することで、BONE_VECTOR_COUNT やオフセット計算の意図が明確になります。

分析の順序としては、「頂点入力にボーンインデックス（bone index）とウェイト（weight）が含まれているか」→「ユニフォームにボーンパレットが定義されているか」→「最大 4 本のボーン影響をどのように合成しているか」→「変形後の法線をどう処理しているか」→「weight_debug の切り替えロジックはどこにあるか」という流れで追跡してください。

BonePhong の役割は、頂点ごとの boneIndex と weight を参照し、行列パレットからボーン行列を合成して頂点を変形させることです。変形後のライティング計算は Phong とほぼ同様です。また、weight_debug を有効にすると、ウェイトの分布を RGB 色で可視化でき、スキニングの正当性を検証できます。

ユーザー側では、has_bone: 1 を設定し、Shape 側にボーンインデックスとウェイトのデータが含まれていることが前提となります。Shape.endShape() は、スキニングメッシュの場合に専用のバッファを生成し、BonePhong のパイプラインはこれを前提とした構成になっています。

book/examples/BonePhong.js では、Phong に対して「ボーンインデックス / ウェイトの

頂点入力を追加し、ユニフォームにボーンパレットを並べ、頂点ごとに 4 本までのボーン行列を合成してから通常のライティングへ渡す」という差分が実装されています。

```
struct Uniforms {
  projMatrix : mat4x4<f32>,
  viewMatrix : mat4x4<f32>,
  normalMatrix : mat4x4<f32>,
  lightPos   : vec4<f32>,
  color      : vec4<f32>,
  params     : vec4<f32>,   // amb / spec / power / emit
  flags      : vec4<f32>,   // hasBone / texFlag / weightDebug / backfaceDebug
  fogColor   : vec4<f32>,
  fogParams  : vec4<f32>,
  // ‘Phong‘ との差分:
  // - bone palette を vec4 x 3 単位でまとめて持つ
  bones      : array<vec4<f32>, BONE_VECTOR_COUNT>,
};

struct VertexInput {
  @location(0) position : vec3<f32>,
  @location(1) normal   : vec3<f32>,
  @location(2) texCoord : vec2<f32>,
  // ‘Phong‘ との差分:
  // - skinning 用の bone index / weight を追加する
  @location(3) index    : vec4<f32>,
  @location(4) weight   : vec4<f32>,
};

struct VertexOutput {
  @builtin(position) position : vec4<f32>,
  @location(0) vPosition : vec3<f32>,
  @location(1) vNormal   : vec3<f32>,
  @location(2) vTexCoord : vec3<f32>,
  // ‘weight_debug‘ 用に、先頭 3 本の weight をそのまま渡す
  @location(3) vWeight   : vec3<f32>,
};

@vertex
fn vs_main(input : VertexInput) -> VertexOutput {
  var output : VertexOutput;
  var mat : mat4x4<f32>;
```

```
if (u.flags.x == 0.0) {
  mat = mat4x4<f32>(
    vec4<f32>(1.0, 0.0, 0.0, 0.0),
    vec4<f32>(0.0, 1.0, 0.0, 0.0),
    vec4<f32>(0.0, 0.0, 1.0, 0.0),
    vec4<f32>(0.0, 0.0, 0.0, 1.0)
  );
} else {
  // 最大 4 本の bone index から、palette 上の開始位置を引く
  let i0 = i32(input.index.x) * 3;
  let i1 = i32(input.index.y) * 3;
  let i2 = i32(input.index.z) * 3;
  let i3 = i32(input.index.w) * 3;

  var v0 : vec4<f32>;
  var v1 : vec4<f32>;
  var v2 : vec4<f32>;

  // 各 bone の 3 行ぶんを weight 付きで混ぜ、頂点専用の skin 行列を作る
  v0  = u.bones[i0]     * input.weight.x + u.bones[i1]     * input.weight.y;
  v0 += u.bones[i2]     * input.weight.z + u.bones[i3]     * input.weight.w;

  v1  = u.bones[i0 + 1] * input.weight.x + u.bones[i1 + 1] * input.weight.y;
  v1 += u.bones[i2 + 1] * input.weight.z + u.bones[i3 + 1] * input.weight.w;

  v2  = u.bones[i0 + 2] * input.weight.x + u.bones[i1 + 2] * input.weight.y;
  v2 += u.bones[i2 + 2] * input.weight.z + u.bones[i3 + 2] * input.weight.w;

  mat[0] = vec4<f32>(v0.x, v1.x, v2.x, 0.0);
  mat[1] = vec4<f32>(v0.y, v1.y, v2.y, 0.0);
  mat[2] = vec4<f32>(v0.z, v1.z, v2.z, 0.0);
  mat[3] = vec4<f32>(v0.w, v1.w, v2.w, 1.0);
}

output.vTexCoord = input.texCoord;
output.vWeight = input.weight.xyz;

let pos4 = u.viewMatrix * mat * vec4<f32>(input.position, 1.0);
output.vPosition = pos4.xyz;

let normMat = u.normalMatrix * mat;
output.vNormal = (normMat * vec4<f32>(input.normal, 0.0)).xyz;
```

```
  output.position = u.projMatrix * pos4;
  return output;
}

@fragment
fn fs_main(input : FragmentInput) -> @location(0) vec4<f32> {
  // ‘Phong‘ との差分:
  // - weight_debug が有効なら、lighting より先に weight 色を返す
  if (u.flags.z != 0.0) {
    let c = clamp(input.vWeight, vec3<f32>(0.0), vec3<f32>(1.0));
    return vec4<f32>(c, 1.0);
  }

  // この先の texture / lighting / fog は ‘Phong‘ と同じ
```

BoneNormPhong による機能統合

BoneNormPhong は、BonePhong と NormPhong の機能を統合したものです。頂点シェーダーは BonePhong の構造を、フラグメントシェーダーは NormPhong の構造を踏襲しており、パラメータ群は両方の機能を保持しています。

本シェーダーの役割は、「スキニングによる頂点変形」と「ノーマルマップによる法線制御」を同時に実現することです。また、weight_debug と backface_debug の両方を備えており、変形と陰影の両面から検証が可能です。

9.4　2D / オーバーレイ / ヘルパー系シェーダーの解析

Background、Font、BillboardShader、Wireframe、FullscreenPass といったシェーダーは、3D マテリアルとは異なる視点で分析する必要があります。これらは「質感を決定する」ことよりも、「特定の表示経路（描画パイプライン）を構築する」ことに主眼が置かれています。

- Background: 画面全体や背景矩形の描画、およびテクスチャの向き補正（V 反転など）に注目してください。

- Font: グリフ用フォントアトラスの参照と色の管理が中心であり、Text や Message クラスの内部実装の基盤となります。
- BillboardShader: 常にカメラに向く四角形をどのように生成するかに注目してください。
- Wireframe: メッシュの構造をデバッグ表示するための描画手法を定義しています。
- FullscreenPass: レンダーターゲットのテクスチャを画面いっぱいに描画するパスの構成を確認してください。

例えば Font シェーダーでは、文字ごとに個別の四角形を描画するのではなく、同一のパイプラインを使い回し、ダイナミックオフセット（dynamic offset）を用いて「文字番号」「色」「位置」を高速に切り替える設計になっています。この設計を理解することで、HUD やメッセージ表示のパフォーマンス特性を把握できます。

GlassMaskShader も、この系統に含めて読むべきシェーダーです。名前に glass とありますが、ライトを計算してガラスの質感を描くためのシェーダーではありません。FrostedGlassPass の mask target へ、ガラス面の画面上の領域、tint 色、blur の効き具合を示す alpha を書き込むためのシェーダーです。頂点入力は通常の Shape.draw() の流れに乗れるよう position / normal / UV を受け取りますが、fragment shader は material 色をそのまま返します。実装を読むときは、見た目の美しさを作っている箇所ではなく、beginMask() で scene depth を使いながら mask を生成し、その mask を FrostedGlassPass の composite shader がどう解釈するかを追うと理解しやすくなります。

Wireframe の実装詳細

Wireframe は、メッシュの面を塗るのではなく、line-list topology で edge を描くためのシェーダーです。描画入口は shape.setWireframe(true) であり、Shape.draw() は通常のシェーダーを使う代わりに、GPU ごとにキャッシュされた Wireframe インスタンスへ切り替えます。

ここで重要なのは、Wireframe が単に position だけを読む最小シェーダーではなく、SmoothShader と差し替えやすい vertex buffer layout と bind group layout を持つように設計されている点です。

現在の Wireframe は、slot0 に position / normal / UV、slot1 に bone index / weight を受け取ります。線描画そのものには normal や UV は使いませんが、通常描画と同じ Shape.draw() の処理フローで vertex buffer を設定できるように、あえて SmoothShader と同じ入力

形式を受け入れます。非スキニングメッシュでは slot1 に 0 埋めのダミーバッファを渡し、スキニングメッシュでは Shape が作成した bone index / weight バッファをそのまま渡します。

bind group も同様の考え方です。`group(0)` は projection、modelView、線色、`has_bone` flag、fog 色、fog パラメータを含む draw ごとの uniform です。`group(1)` は `Wireframe` 自身では使用しませんが、`SmoothShader` の texture group と番号をそろえるため、空の bind group layout として pipeline layout に含めています。`group(2)` は bone palette で、`SmoothShader` と同様にスケルトンごとの uniform buffer / bind group を `WeakMap` でキャッシュします。

WGSL 側の頂点シェーダーでは、`has_bone` が 0 の場合に identity 行列を使い、`has_bone` が 1 の場合に `input.index` と `input.weight` から最大 4 本の bone を合成します。bone palette は `SmoothShader` と同じく 1 bone を `vec4 x 3` として並べる形式なので、`i32(input.index.x) * 3` のようなオフセット計算で 3 行ぶんを取り出します。

```
let worldPos = uniforms.modelView * skinMat * vec4f(input.position, 1.0);
output.position = uniforms.proj * worldPos;
output.vPosition = worldPos.xyz;
```

fragment shader では、`SmoothShader` と同じ `fog_color`、`fog_near`、`fog_far`、`fog_density`、`fog_mode` を使い、線色を fog 色へ補間します。`Wireframe` はライトやテクスチャを扱わないため、fog の入力色はライティング済みの面色ではなく `color` で指定された線色です。一方で距離計算は同じく modelView 後の `vPosition` を使うため、通常描画と wireframe 表示を切り替えても遠景の見え方が大きく変わりません。

この構成により、静的メッシュとスキニングメッシュのどちらでも `Shape.setWireframe(true)` を同じ API で使えます。特に skinned mesh では、通常描画で見えている変形後の姿に wireframe が追従するため、不整合を避けられます。また、`Shape.draw()` は通常シェーダーの default にある fog 設定をキャッシュ済みの `Wireframe` インスタンスへ反映するため、シーン全体の fog 設定も維持されます。

wireframe の edge list は `Shape._buildWireIndexBuffer()` が作ります。`polygonLoops` がある場合は、三角形化された描画用 index ではなく、元の face loop の外周から線を作ります。そのため `Shape.addPolygon([a, b, c, d])` や `addPlane()` で作った四角面は、内部の対角線を表示せず、四角形の外周だけが表示されます。

9.5 シェーダー拡張時の整合性確保

シェーダーの実装を変更する際、最も注意すべき点は「WGSL の修正だけで完結させない」ことです。例えば、新しいパラメータを 1 つ追加する場合、以下の項目すべてに整合性を取る必要があります。

1. WGSL 側: `Uniforms` 構造体にフィールドを追加する。
2. JavaScript 側: `uniformData` のサイズとオフセットを更新する。
3. リソース管理: バインドポイント（binding）数を変更した場合は、バインドグループレイアウトと `getBindGroup()` の実装を修正する。
4. 初期設定: コンストラクタオプションや初期パラメータを適切に設定し、既存の動作を破壊しないようにする。
5. 検証: 変更内容を確認できる単体テストやサンプルコードを整備する。

特に `doParameter()` への反映を忘れると、`Shape.setMaterial()` でパラメータを指定しても反映されないという不具合が発生します。これは視覚的に気づきにくく、原因の切り分けに時間を要するため注意が必要です。`webg` のシェーダーは、「CPU 側のクラス設計 → ユニフォーム更新 → バインドグループ → GPU 描画」までが一連のパイプラインとして機能しています。

9.6 AI を用いた実装・修正時の注意点

コーディング AI を用いてシェーダーを修正・拡張させる場合は、単に WGSL コードを生成させるのではなく、以下のチェックリストを同時に提示し、整合性を検証させる必要があります。

- メモリレイアウトの整合性: WGSL の `Uniforms` フィールドを増やした場合、JavaScript 側の `uniformData` のサイズとオフセットも正しく更新されているか。
- バインドポイントの整合性: バインド数を変更した場合、レイアウト定義と `getBindGroup()` の実装が一致しているか。
- パラメータ管理: 新しいデバッグオプションを追加した際、初期値を破壊せず、コンストラクタまたは初期パラメータで適切に管理されているか。

- 言語仕様の制約: `textureSample()` の呼び出し位置が、WGSL の非一様制御（non-uniform control）に関する制約に抵触していないか。
- 検証可能性: 変更結果を客観的に確認できるテストケースやサンプルコードが提示されているか。

コード単体が正しくても、ユーザーや開発者がどこを確認すれば実装内容を判断できるか分からない状態は避けるべきです。シェーダーを拡張する際は、「どのパラメータを操作すれば、どのような視覚的変化が起きるか」という説明を併記させることが重要です。

9.7 推奨される読解順序と関連章

初めてシェーダー実装を分析される方は、以下の順序で進めることをお勧めします。

1. 第 7 章「シェーダーとマテリアル」で用途とインターフェースを確認する。
2. 第 8 章「WGSL の読み方」で WGSL の基本記法を習得する。
3. `Shader.js` を読み、共通構造と基底クラスの役割を把握する。
4. `Phong` を起点とし、`NormPhong` → `BonePhong` → `BoneNormPhong` の順に差分を分析する。
5. 必要に応じて `Background`、`Font`、`FullscreenPass` などの特殊シェーダーを確認する。

さらに詳細な実装へ進む場合は、以下の章が関連します。- アプリ全体の構成の中での利用方法 → 第 5 章「WebgApp によるアプリ構成」- モデルアセットやシーンとの連携 → 第 10 章「モデルアセットとランタイム」および 第 11 章「シーン構成と SceneJSON」- ローレベル（低レイヤー）の `Screen`、`Shader`、`Node`、`Matrix` の実装 → 第 25 章「ローレベル API の基礎」- `Shape` とメッシュ構築の詳細手順 → 第 26 章「Shape とメッシュ構築」- プロシージャル形状の生成手法 → 第 24 章「プロシージャル形状の作り方」

9.8 まとめ

本章で最も重要なのは、シェーダーを単なる「WGSL の断片」としてではなく、「CPU 側のクラス設計、ユニフォーム更新、バインドグループ、そして最終的な描画結果」までを含んだ一つの統合的な実装として捉えることです。

`Shader.js` が共通構造を抽象化し、`doParameter()` がユーザー向けのインターフェースを整理し、`Phong` 系 4 種類が機能的な段階を示すという構造を理解することで、シェーダーの

カスタマイズやデバッグの効率は飛躍的に向上します。

第 10 章

モデルアセットとランタイム

本章では、ModelAsset がどのような役割を持ち、どの段階で検証され、最終的にランタイムの Node、Shape、Skeleton、Animation へと展開されるのかを解説します。

10.1 ModelAsset による抽象化の設計思想

3D モデルの読み込みにおいて、glTF や Collada、あるいは独自形式の JSON など、入力形式によってデータの保持方法には大きな差が生じます。これらの形式をそのままランタイムに直結させると、利用方法まで形式ごとに依存してしまい、実装の共通化が困難になります。そこで webg では、形式ごとの差異を一度 ModelAsset という共通形式で吸収する設計を採用しています。

これにより、その後の build()、instantiate()、およびアニメーション再生といった処理を、入力形式に関わらず同一の手順で扱うことが可能になります。利用者は「現在扱っているのがインポート済みのデータ（設計図）なのか」、あるいは「描画可能なランタイム状態（実体）なのか」を明確に区別して管理できます。

ここで重要なポイントは、ModelAsset は「描画オブジェクト」ではなく、CPU 側で正規化された「共通データ」であるということです。ModelAsset 自体は GPU バッファや描画状態を保持せず、materials、meshes、skeletons、animations、nodes といった JSON 互換のデータ構造を保持します。実際の Shape、Skeleton、Animation、Node といったランタイムオブジェクトは、後述する build() と instantiate() の段階で初めて生成されます。

このような構造を採用することで、主に以下の 4 つのメリットが得られます。

1. 形式の正規化: ファイル形式ごとの差異を単一の JSON 互換構造へ集約し、インポート処理と利用処理を完全に切り離すことができます。
2. 早期検証: `ModelValidator` を用いて、id 参照の不整合や配列長の不一致などを `build()` 前に検出でき、ランタイム側で原因不明の表示不具合が発生するリスクを低減できます。
3. 効率的なリソース構築: ビルダーが共有リソースとランタイム定義を効率的に組み立てられる形式でデータを渡せます。
4. 共通インターフェースの提供: クリップ一覧の取得、JSON 保存、スケール調整などを、入力形式を問わず共通の操作体系で実行できます。

特に、同一モデルをシーン内に複数配置する場合、この設計が大きな威力を発揮します。`build()` を 1 回だけ行い、その結果を使い回して `instantiate()` を必要な回数だけ呼び出すことで、GPU バッファの消費を抑えつつ、個々のモデルが独立したアニメーション状態を持つことが可能になります。

また、インポート後のデータを JSON として保存し、後から比較・検証できる点も実用的です。`downloadJSON()` を使用して glTF インポーターの出力を `modelasset.json` として保存すれば、インポート結果の検証やデータ構造の仕様確認に活用できます。

なお、利用にあたっては以下の点に注意してください。- `ModelAsset` 自体はシーンに配置されるオブジェクトではなく、描画には `build()` と `instantiate()` が必須です。- `build()` の戻り値は「インスタンス化可能な状態のランタイム」であり、まだ `Space` の Node ツリーには組み込まれていません。- `runtime.instantiate()` は呼び出しごとに新しいインスタンスを返すため、インスタンス間でアニメーション状態は共有されません。- `scaleUniform()` はジオメトリだけでなく、スケルトンとアニメーションの translation にも影響を与えます。- `downloadJSON()` と `downloadJSONGz()` はブラウザ環境向けのヘルパーであり、Node.js 等の環境には適していません。- `.json.gz` は ModelAsset の構造を変える形式ではなく、JSON 文字列を gzip 圧縮した保存・転送用のバイナリストリームです。- `node.matrix` が定義されている場合は `node.transform` より優先されます。

10.2　標準フローと 2 つのエントリーポイント

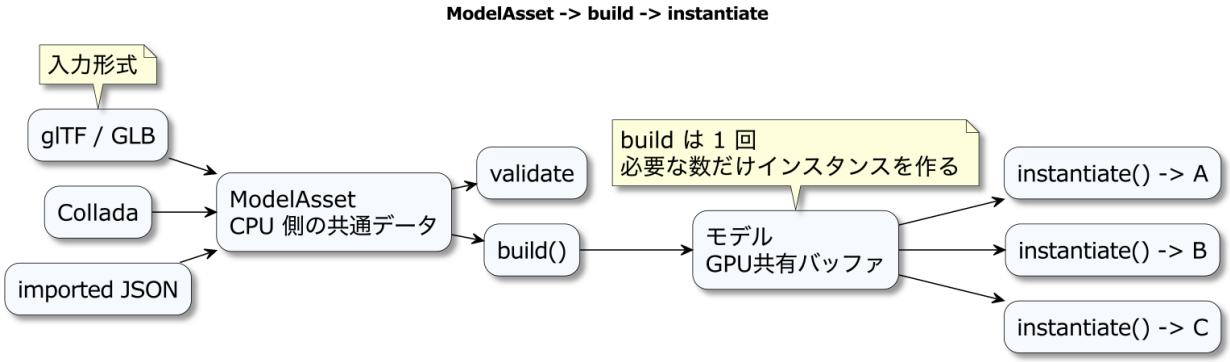


図 10.1: ModelAsset build instantiate

ModelAsset は異なる入力形式を共通の *build() / instantiate()* フローへ統合する入口であり、通常 *build()* は *1* 回、*instantiate()* は必要な数だけ行います。

`ModelAsset` を利用するフローには、大きく分けてハイレベル（高レイヤー）経路とローレベル（低レイヤー）経路の 2 つが存在します。

通常のサンプル実装では、`WebgApp.loadModel()` または `ModelLoader.load()` を使用するハイレベル（高レイヤー）経路が推奨されます。この経路では、形式判定、インポート、整合性検証、`build()`、そして必要に応じた `instantiate()` までを一括して実行できます。

```
const result = await app.loadModel("./human.glb", {
  instantiate: false,
  validate: true
});

const runtime = result.runtime;
const actorA = runtime.instantiate(app.space);
const actorB = runtime.instantiate(app.space);
```

この例では、`human.glb` のインポートと `build()` を 1 回だけ行い、得られた `runtime` から 2 体のインスタンスを生成しています。`actorA` と `actorB` はそれぞれ独立した `Node`、`Shape` インスタンス、`Skeleton`、`Animation` を持ちますが、メッシュ由来の共有リソースは

build 結果側で共有されます。

一方で、アセットを手動で編集する場合や、整合性検証および build() の各段階を詳細に制御したい場合は、ModelAsset を直接操作するローレベル（低レイヤー）経路が適しています。

```
import ModelAsset from "./webg/ModelAsset.js";

const asset = await ModelAsset.load("./modelasset.json");
asset.assertValid();

const runtime = asset.build(app.getGPU());
const instantiated = runtime.instantiate(app.space);
runtime.startAllAnimations();
```

この経路では ModelAsset オブジェクトを直接保持するため、build() を実行する前に getClipNames()、downloadJSON()、scaleUniform() などのヘルパーメソッドを利用できます。

また、ModelAsset.load() は、通常の modelasset.json だけでなく、gzip 圧縮された modelasset.json.gz も読み込めます。.json.gz の場合は、fetch で取得した Blob を DecompressionStream("gzip") に通し、復元された文字列を解析します。読み込み後に得られる ModelAsset は通常 JSON と同一であるため、その後の validate()、build()、instantiate() の使い方は変わりません。

```
const asset = await ModelAsset.load("./modelasset.json.gz");
asset.assertValid();

const runtime = asset.build(app.getGPU());
runtime.instantiate(app.space);
```

ModelLoader.detectFormat() も .json.gz を ModelAsset JSON として扱うため、ハイレベル（高レイヤー）経路でも同様に読み込むことが可能です。

まとめると、ModelAsset を利用する際は以下の順序で処理を構成するのが効率的です。1. インポーターまたは JSON ファイルから ModelAsset を取得する。2. validate() または assertValid() でデータの整合性を確認する。3. build() を実行し、共有リソースを含むランタイムを構築する。4. instantiate() を呼び出し、新しいシーンインスタンスを Space

へ配置する。5. 必要に応じて startAnimation() や startAllAnimations() でクリップを再生する。

10.3 runtime と instantiation の寿命管理

ModelAsset を build し、runtime.instantiate() で Space へ配置した後は、リソースの寿命管理（いつ片付けるか）について意識する必要があります。モデルの読み込みは容易ですが、不要になったリソースを適切に破棄しないと、画面遷移やモデル差し替えのたびに不要な Node や GPU リソースが蓄積し、メモリリークの原因となります。

ここで重要なのは、runtime と instantiate() の結果は異なる役割を持つということ点です。- runtime: build 済みの共有資源を持つ基盤層です。メッシュ、共有 ShapeResource、Node 定義、スケルトン定義、アニメーション定義を再利用可能な形で保持します。- instantiate() の結果: runtime を現在の Space に実体化したものです。個別の Node、Shape インスタンス、Skeleton、Animation を持つシーン上の実体です。

この違いにより、寿命管理の単位を明確に分けることができます。

```
const result = await app.loadModel("./robot.glb", {
  instantiate: false
});

const runtime = result.runtime;
const instantiated = runtime.instantiate(app.space);

runtime.startAllAnimations();

// シーン上のこの実体だけを片付けたい場合
instantiated.destroy();

// runtime 自体も今後使わない場合
runtime.destroy();
```

instantiated.destroy() は、現在の Space に配置された Node 群と Shape インスタンスを破棄します。同じ runtime から複数体を生成している場合でも、この API を使えば特定の実体のみを削除できます。

対して runtime.destroy() は、runtime が保持していた共有リソースを含めてすべてを終

了させます。「このモデル定義自体を今後一切使用しない」と判断した際に呼び出します。例えば、モデルビューアのように読み込んだモデルを次々と入れ替える用途や、ステージ全体を切り替えて旧ステージのリソースを一括で破棄する場合に有効です。

GPU バッファのような GPU 側リソースは、JavaScript のガベージコレクションだけでは即座に解放されないため、webg では destroy() による明示的な寿命管理を基本としてください。

10.4　実践的な活用例

glTF を読み込み、同じモデルを複数配置する

同一の glTF モデルを複数配置する場合、loadModel() を複数回呼び出すのではなく、一度読み込んで runtime.instantiate() を複数回呼び出す手法が効率的です。

```
const result = await app.loadModel("./hand.glb", {
  instantiate: false,
  validate: true
});

const player = result.runtime.instantiate(app.space);
const cpu = result.runtime.instantiate(app.space);

const playerNode = player.nodeMap.get("HandRoot");
const cpuNode = cpu.nodeMap.get("HandRoot");

playerNode.setPosition(-1.2, 0.0, 0.0);
cpuNode.setPosition( 1.2, 0.0, 0.0);

player.startAllAnimations();
cpu.startAllAnimations();
```

この実装では、ジオメトリや GPU バッファは build 側で共有されます。一方で、player と cpu は別々のインスタンスであるため、アニメーションの再生位置や一時停止状態などの状態管理は互いに干渉しません。

インポート済み JSON を検証してから build する

手書きの JSON やエクスポーターの出力を利用する場合、描画前に validate() を実行することで、不整合によるランタイムエラーを未然に防ぐことができます。

```
const asset = await ModelAsset.load("./modelasset.json");
const report = asset.validate();

if (!report.ok) {
  console.error(report.errors);
  throw new Error("Invalid ModelAsset");
}

const runtime = asset.build(app.getGPU());
runtime.instantiate(app.space);
```

validate() は { ok, errors, warnings } 形式のレポートを返します。エラーが検出された場合に build() を中断させることで、id 参照ミスや行列データの不足を早期に発見できます。

インポート済みモデル全体へ一定倍率を焼き込む

アセット全体のサイズを調整したい場合は、ジオメトリだけでなくスキンとアニメーションも含めて同一の規則でスケールさせる必要があります。scaleUniform() はそのための専用ヘルパーです。

```
const asset = await ModelAsset.load("./human.json");
asset.assertValid();

const scaledAsset = ModelAsset.fromData(
  asset.cloneJSONValue(asset.getData())
);

scaledAsset.scaleUniform(0.70);

const runtime = scaledAsset.build(app.getGPU());
```

```
runtime.instantiate(app.space);
```

scaleUniform(0.70) は、メッシュ頂点、Node の translation、Skeleton のジョイント行列、Animation のポーズの translation を一括して 0.70 倍します。スキニングモデルにおいてジオメトリのみを縮小させるとポーズが崩れるため、アセット全体に同一倍率を適用する手法が最も安全です。

クリップ一覧を確認し、要約情報を取得する

複数のアニメーションを含むモデルを扱う際は、事前にクリップ名や長さを把握しておくことで、実装をスムーズに進めることができます。

```
const result = await app.loadModel("./robot.glb", {
  instantiate: false
});

for (const clipName of result.getClipNames()) {
  const info = result.getClipInfo(clipName);
  console.log(clipName, info.durationMs, info.trackCount);
}
```

getClipInfo() は id、targetSkeleton、keyCount、trackCount、durationMs を返します。ランタイムをシーンに配置する前でもこれらの情報を取得できるため、UI への表示やドキュメント作成に活用できます。

インポーターの出力を JSON として保存する

glTF や Collada インポーターが内部的にどのようにデータを処理したかを確認したい場合は、downloadJSON() を使用して ModelAsset をファイルとして保存できます。

```
const result = await app.loadModel("./vehicle.glb", {
  instantiate: false
});
```

```
result.downloadJSON("vehicle.modelasset.json");
```

保存された JSON を解析することで、インポート後のメッシュ、スケルトン、アニメーション、Node がどのように正規化されたかを詳細に追跡でき、デバッグや仕様策定に役立ちます。

また、モデルが大きくなり JSON ファイルが肥大化する場合は、downloadJSONGz() を使用して gzip 圧縮した .json.gz として保存することが可能です。

```
const result = await app.loadModel("./vehicle.glb", {
  instantiate: false
});

await result.downloadJSONGz("vehicle.modelasset.json.gz");
```

.json.gz は単に JSON 文字列を gzip 圧縮したものであり、復元すれば通常の ModelAsset JSON と同一の内容になります。

ここで GLB/glTF 形式との重要な違いについて触れます。glTF のメッシュプリミティブは三角形を基本単位として扱うため、四角面を含むモデルを GLB として書き出すと、強制的に三角形に分割されます。一方、ModelAsset JSON では、描画用の三角形 indices に加えて、編集用の面ループを geometry.polygonLoops として保持できます。これにより、webg の編集データや Blender などの DCC ツールとの間で、四角面を維持したままデータをやり取りできます。

したがって、外部ビューアや一般的な glTF ツールチェーンと連携する場合は GLB が向いており、webg 側の編集情報を保持したまま往復させたい場合は .json または .json.gz を使用するのが最適です。

Blender との ModelAsset JSON 受け渡し

Blender と webg の ModelAsset JSON は、専用の Blender アドオンを用意することで相互変換できます。アドオン側では File > Import / Export に Webg ModelAsset JSON のような項目を追加し、.json および .json.gz の読み書きを扱います。

本アドオンは主にメッシュジオメトリの相互変換を目的としています。Import 時には positions、indices、polygonLoops、uvs などを読み取り、Blender の Mesh オブジェクトを

作成します。この際、nodes の階層構造にあるトランスフォームは頂点位置へベイク（固定化）されるため、Blender 上では編集しやすい単一のメッシュとして扱えます。

Export 時には、選択したメッシュを ModelAsset として出力します。この際もワールド変換は頂点へベイクされます。なお、複雑なリグやアニメーションを完全に保持したい場合は、本アドオンではなく glTF / GLB 形式を ModelLoader で読み込む経路を利用してください。

また、Blender（Z-up）と ModelAsset（Y-up）では上方向の軸が異なるため、アドオン内部で座標軸の変換を自動的に行います。

```
Import : ModelAsset (X, Y, Z) -> Blender    (X, -Z, Y)
Export : Blender    (X, Y, Z) -> ModelAsset (X,  Z, -Y)
```

Shape.createInstance() による共有ジオメトリの再利用

ModelAsset を介さない低レイヤーな実装であっても、「ジオメトリは一度だけ構築し、色や位置、スケールのみを変更したインスタンスを複数配置する」構成にすることで、GPU バッファの重複生成を回避できます。

webg/samples/dof では、半径 1.0 のユニットスフィアを一度だけ作成し、Shape.createInstance() と node.setScale(radius) を組み合わせて異なるサイズの球体を表現しています。

```
const sphereSource = new Shape(app.getGPU());
sphereSource.applyPrimitiveAsset(
  Primitive.sphere(1.0, 28, 20, sphereSource.getPrimitiveOptions())
);
sphereSource.endShape();

function addSphere(name, options) {
  const shape = sphereSource.createInstance();
  shape.setMaterial("smooth-shader", {
    use_texture: 0,
    color: options.color,
    ambient: 0.70,
    specular: 1.10,
    power: 58.0
  });
```

```
  const node = app.space.addNode(null, name);
  node.setPosition(options.x, options.y, options.z);
  node.setScale(options.radius);
  node.addShape(shape);
  return node;
}
```

この手法により、見た目の差分（マテリアルやスケール）のみをインスタンス側に持たせ、負荷の高いメッシュ構築を共有側に集約できます。これにより、頂点数やメッシュ数の大幅な削減が可能となり、パフォーマンスが劇的に向上します。

10.5 build() の結果と ModelAsset の内部構造

ModelAsset.build(gpu) の戻り値である runtime オブジェクトは、以下の 3 層構造で構成されています。

1. アセット由来の定義: runtime.nodes および runtime.nodeMap に格納される、アセットから変換された Node 定義。
2. build 済みの共有リソース: runtime.shapeResources に格納される、複数の Node で共有される描画リソース群。
3. instantiate 後のランタイムインスタンス: runtime.instantiate(space) によって生成される、個別の Node、Shape インスタンス、Skeleton、Animation。

この構造により、build() の結果をキャッシュし、必要なタイミングで instantiate() を呼び出す効率的な運用が可能になります。

ModelAsset の JSON スキーマは、主に以下の項目で構成されています。

```
{
  "version": "1.0",
  "type": "webg-model-asset",
  "meta": {},
  "materials": [],
  "meshes": [],
  "skeletons": [],
```

```
  "animations": [],
  "nodes": []
}
```

- materials: メッシュが参照する材質定義。shaderParams で色や光沢などを指定します。
- meshes: ジオメトリ、マテリアル参照、スキン情報を保持します。positions と indices が必須です。
- skeletons: ジョイント階層、レストポーズ、Inverse Bind Matrix を定義し、スキニングメッシュの骨格となります。
- animations: 特定のスケルトンを動作させるためのクリップ定義です。
- nodes: モデル内の配置単位。使用するメッシュ、結び付けるスケルトン、およびローカル変換を定義し、親子関係を構築します。

validate() は、これらのデータ構造における id の重複や参照整合性、配列長などを検証し、{ ok, errors, warnings } 形式の結果を返します。errors は build() の実行を妨げる致命的な不整合であり、warnings は確認が推奨される項目です。

最終的に instantiate() が呼ばれた時点で、初めて Space.addNode() が実行され、個別の Skeleton と Animation が生成されます。これにより、同一の build 結果から生成された複数のインスタンスであっても、それぞれが独立したアニメーション再生状態を保持できます。

10.6 例を通して確かめる

ModelAsset の役割を具体的に確認するには、以下のサンプルおよびテストケースの参照が有効です。

- webg/samples/gltf_loader: glTF のインポート、クリップ情報の取得、および JSON 保存の動作を確認できます。
- webg/samples/collada_loader: Collada 形式から ModelAsset へ正規化されるフローを確認できます。
- webg/samples/json_loader: 手書きの JSON を ModelAsset として直接読み込む例です。

- webg/samples/mmodeler: ModelAsset JSON / .json.gz の読み書きと、編集用 polygonLoops を持つ geometry の保存を確認できます。
- webg/samples/janken: 同一の build 済みランタイムから複数の instantiate() を行い、アニメーション状態を独立して制御する実例です。
- webg/unittest/primitive_modelasset: Primitive → ModelAsset → validate → build という最小構成のパイプラインを確認できます。
- webg/unittest/compression: Compression Streams API による gzip 圧縮・復元および .json.gz の読み戻しを確認できます。

10.7 変更時の注意点

ModelAsset 関連の機能を変更・拡張する際は、影響範囲が広いため、以下の要素をセットで確認してください。

- スキーマ変更時: ModelValidator および ModelBuilder の修正が必要です。
- メッシュ・スキン変更時: ジオメトリ処理およびスキニング反映ロジックを確認してください。
- スケルトン変更時: レストポーズ、Inverse Bind Matrix、およびアニメーションの結び付け処理を確認してください。
- アニメーション変更時: トラックの整合性検証およびランタイムのクリップ生成ロジックを確認してください。
- 高レイヤー API 変更時: ModelLoader および各サンプルの実装への影響を確認してください。
- 保存形式変更時: .json と .json.gz の両方で ModelAsset.load()、downloadJSON()、downloadJSONGz()、ModelLoader.detectFormat()、unittest/compression を確認してください。

特に、「build() 結果で共有すべきリソース」と「instantiate() ごとに独立させるべき状態」の分離を厳格に維持することが重要です。GPU バッファやジオメトリは共有側に集約し、アニメーション再生状態やスケルトンの現在ポーズ、マテリアルの個別差分などは必ずインスタンス側で独立して保持させてください。

10.8　参考用の最小例

最小構成の ModelAsset は、type、version、meshes、nodes を必須項目として持ち、mesh.geometry.positions と mesh.geometry.indices が正しく定義されていれば build() が可能です。

```
{
  "version": "1.0",
  "type": "webg-model-asset",
  "meta": {
    "name": "triangle"
  },
  "materials": [
    {
      "id": "mat0",
      "shaderParams": {
        "color": [0.4, 0.8, 1.0, 1.0],
        "ambient": 0.3,
        "specular": 0.4,
        "power": 24.0
      }
    }
  ],
  "meshes": [
    {
      "id": "mesh0",
      "name": "triangle",
      "material": "mat0",
      "geometry": {
        "vertexCount": 3,
        "polygonCount": 1,
        "positions": [-1, -1, 0, 1, -1, 0, 0, 1, 0],
        "normals": [0, 0, 1, 0, 0, 1, 0, 0, 1],
        "uvs": [0, 0, 1, 0, 0.5, 1],
        "indices": [0, 1, 2]
      }
    }
  ],
  "nodes": [
    {
```

```
      "id": "node0",
      "name": "triangleNode",
      "parent": null,
      "mesh": "mesh0",
      "transform": {
        "translation": [0, 0, 0],
        "rotation": [0, 0, 0, 1],
        "scale": [1, 1, 1]
      }
    }
  ]
}
```

この最小例は、手書き JSON によるテストの出発点として適しています。まずはアニメーションやスケルトンを含まない基本経路を確認し、その後段階的にスキンやクリップを追加することで、問題の切り分けを容易に行うことができます。

10.9　関連文書と次章へのつながり

本章の内容は、以下の文書と密接に関連しています。- 第 1 章「はじめに」- 第 3 章「3D グラフィックスの基礎」- 第 4 章「WebGPU と webg の最小描画」- 第 12 章「アニメーション」- 第 13 章「アニメーションとアセット」- 付録「API 一覧」- `webg/samples/index.html` - `webg/unittest/index.html` - `webg/ModelAsset.js` - `webg/ModelValidator.js` - `webg/ModelBuilder.js` - `webg/ModelLoader.js` - `webg/unittest/compression`

本章で解説した `ModelAsset` は、単一のモデルを共通形式として扱うためのレイヤーです。しかし、実際のアプリケーション開発では、カメラ、HUD、入力設定、プリミティブ、タイルマップなどを含めた「シーン全体の初期状態」を定義する必要があります。次章では、そのための仕組みである `SceneAsset` と Scene JSON について解説します。

第 11 章

シーン構成と Scene JSON

この章では、Scene JSON が何を表し、どの段階で検証され、どのようにして WebgApp 上の実体へと変換されるのかを解説します。第 10 章で扱った ModelAsset が「1 つのモデル」に関する共通表現であったのに対し、Scene JSON は、モデルやプリミティブ、カメラ、HUD、入力を統合した「シーン全体の初期状態」を表現するものです。

3D アプリケーションの初期状態は、単一のモデルだけでは決定しません。カメラの初期方向、HUD の表示内容、キー入力とアクションの対応関係、プリミティブやモデルの配置など、複数の要素を同時に決定する必要があります。webg では、これら「シーン全体の初期状態」を JSON 形式でまとめたものを Scene JSON と呼び、それを保持するクラスを SceneAsset、実際にアプリケーションへと構築するクラスを SceneLoader と定義しています。

ここで最も重要な点は、Scene JSON は「ゲームロジック全体」を記述するものではなく、あくまで「シーンの初期状態を宣言するデータ」であるということです。具体的には、カメラ、HUD、入力、プリミティブ、モデルを「起動時にどのように配置するか」という観点から定義します。ゲームルールや毎フレームの条件分岐などのロジックは持たせず、それらは JavaScript 側に記述し、Scene JSON には初期配置と宣言のみを持たせるのが基本設計となります。

SceneAsset は保存、読み込み、検証、およびビルドの入り口としての役割を担い、SceneLoader はプリミティブとモデルを同一のシーン構築フローへと統合します。入力についても、input.bindings では宣言のみを行い、実際の処理は sceneRuntime.createInputHandler(actionHandlers) を通じて JavaScript 側へ渡されます。さらに、top-level の physicsSpace と各 entry の physics を使うことで、PhysicsSpace と PhysicsNode を Scene JSON から立ち上げる入口も持ちます。つまり、Scene JSON は「配置と初期状態を宣言する層」として理解するのが最も効率的です。物理設定そのものの意味、回転付き OBB、接触応答、制限

事項は第 26 章「物理エンジン」で詳しく扱います。

Scene JSON を利用する際は、以下のフローで検討するとスムーズです。まず、Scene JSON または JavaScript オブジェクトを `SceneAsset` として読み込み、`validate()` または `assertValid()` で構造を検証します。次に、`build(target)` または `app.loadScene(scene)` を実行してシーンを実体化し、`sceneRuntime.getEntry(id)` や `sceneRuntime.createInputHandler()` を用いて JavaScript 側の処理へと接続します。必要に応じて `sceneRuntime.update()` を毎フレーム呼び出すことで、アニメーションを進行させます。物理を宣言した scene では、`sceneRuntime.physicsSpace` と `sceneRuntime.stepPhysics(deltaMs)` を用いて物理更新を明示的に進めます。このように、Scene JSON は「シーン全体の初期配置表」であり、`SceneLoader` はその配置表をランタイムへと変換する役割を担っています。この役割分担を明確にすることで、`ModelAsset` と `SceneAsset` の混同を防ぐことができます。

11.1　Scene JSON 導入の目的とメリット

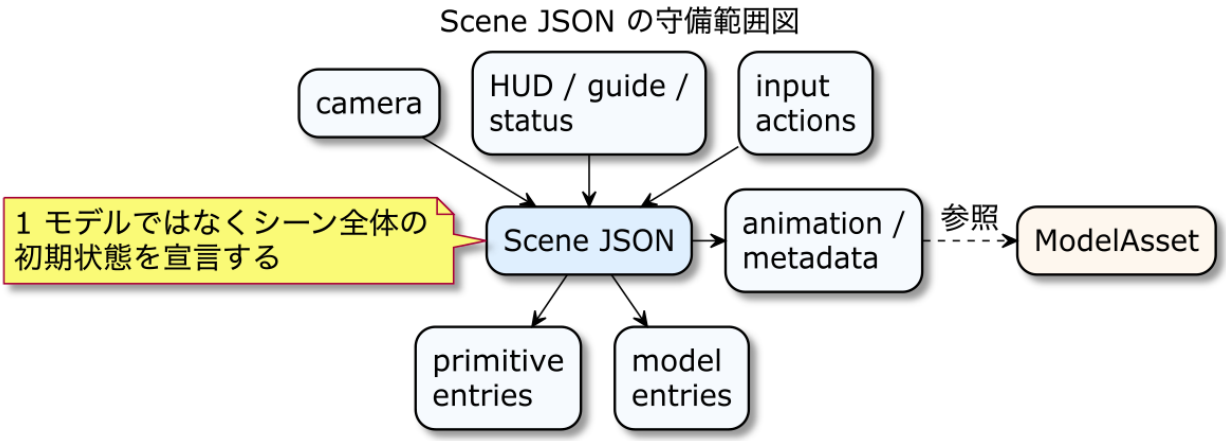


図 11.1: Scene JSON の守備範囲図

Scene JSON は、単一のモデルではなく、*camera*、*HUD*、*input*、*primitive*、*model* を含むシーン全体の初期状態を定義します。

3D アプリケーションにおいて、単にモデルを表示できるだけでは不十分です。カメラの向きや画面上のガイドテキスト、操作キーの割り当て、地面や背景の配置といった「シーン全体の初期状態」を定義する必要があります。これらをすべて JavaScript の初期化コードとして記述すると、サンプルごとに実装方法が異なり、保守性が低下します。`webg` ではこの問題を解決するために Scene JSON を導入しています。これにより、人間が構造を把握しやすくな

るだけでなく、生成 AI がシーン構成を生成する際にも、JSON 形式であれば直接的に構造を制御できるためです。

Scene JSON、SceneAsset、および SceneLoader の役割は、大きく分けて以下の 4 点に集約されます。1. シーン全体の初期状態を JSON 互換データとして保持すること。2. カメラ、HUD、入力、プリミティブ、モデルの構造をビルド前に検証すること。3. 検証済みシーンを WebgApp や { gpu, space } 上のランタイムへ変換すること。4. ビルド後のエントリ参照や入力配線を、JavaScript から扱いやすいヘルパー関数として提供すること。

ここで注意すべきは、Scene JSON は ModelAsset の代替ではないということです。ModelAsset が「1 モデル分」の共通表現であるのに対し、Scene JSON はそのモデルを含む「1 シーン分」の共通表現であり、管理する粒度が異なります。

Scene JSON を導入する最大の利点は、シーンの初期状態を JavaScript のロジックから切り離せる点にあります。これにより、サンプルコードは「シーンをどう活用するか」という本質的な説明に集中できます。また、バリデータによってカメラや HUD、入力設定の整合性を事前に検証できるため、実行後にタイプミスや設定不足に気づくといった手間を削減できます。さらに、SceneLoader がプリミティブとモデルを同一の構築フローにまとめることで、「プリミティブは手書きコード、モデルはローダー」といった実装のばらつきを解消し、利用者はどちらも「シーンエントリを配置する」という統一的な視点で扱うことが可能になります。

具体的には、サンプルの初期設定を 1 つのファイルにまとめたい場合や、プリミティブとモデルを混在させたい場合に非常に有効です。特に webg/samples/scene では、「Scene JSON で初期状態を宣言し、JavaScript 側でアクションハンドラーと update 処理のみを記述する」という標準的な構成例を示しています。

なお、Scene JSON はあくまで初期状態の宣言であり、ゲームロジック全体を記述する場所ではないことに注意してください。SceneAsset.build() はシーンランタイムを返しますが、すべての挙動を自動化するわけではありません。例えば input.bindings はアクション名の宣言のみを行い、実処理は createInputHandler() 側で定義します。また、シーンエントリの transform はアセット内部の座標ではなく、シーン上の配置座標を指します。HUD については、バリデータで文字列の短縮記法を許容していますが、SceneLoader.normalizeHudLines() はオブジェクト形式を前提にビルドするため、実用的には { x, y, text, color } を明示的に指定することを推奨します。

11.2　シーン実体化の 2 つの経路

Scene JSON を利用するフローは、高レイヤー（ハイレベル）な経路と低レイヤー（ローレベル）な経路の 2 つに分けられます。

通常のサンプル開発では、`WebgApp.loadScene(scene)` を使用するのが最も効率的です。この経路では、`SceneAsset` が JSON の読み込みと検証を担当し、`WebgApp.loadScene()` が `SceneLoader` を経由してシーンを実体化させます。カメラや HUD の設定をアプリケーションに即座に反映させたい場合に最適です。

```
import SceneAsset from "./webg/SceneAsset.js";

const sceneAsset = await SceneAsset.load("./scene.json");
sceneAsset.assertValid();

const sceneRuntime = await app.loadScene(sceneAsset.getData());
```

一方で、シーンをオブジェクトとして保持したい場合や、`WebgApp` を介さずに `{ gpu, space }` のみでビルドしたい場合は、`SceneAsset.build(target)` を直接利用します。この経路ではカメラや HUD のアプリ反映は行われませんが、プリミティブ、モデルのビルドおよび `inputMap` の生成は可能です。アプリケーションを持たないユニットテストや、シーンデータの一部のみを検証したい場合に適しています。

```
import SceneAsset from "./webg/SceneAsset.js";

const sceneAsset = SceneAsset.fromData(sceneObject);
sceneAsset.assertValid();

const sceneRuntime = await sceneAsset.build({
  gpu: app.getGPU(),
  space: app.space
});
```

11.3 実装例

Scene JSON の読み込みと入力ハンドラーの利用

以下は、シーンを読み込む最小の実用例です。Scene JSON にはアクション名のみを記述し、JavaScript 側でそのアクションに対応する処理を紐付けます。これにより、シーン定義（データ）とロジック（処理）を明確に分離できます。

```
const sceneAsset = await SceneAsset.load("./scene.json");
sceneAsset.assertValid();

const sceneRuntime = await app.loadScene(sceneAsset.getData());
const input = sceneRuntime.createInputHandler({
  "reset-camera": () => resetCamera(),
  "toggle-pause": () => togglePause()
});

window.addEventListener("keydown", (event) => {
  input.onKeyDown(event.key, event);
});
```

プリミティブとモデルの混在配置

SceneLoader はプリミティブとモデルを同一の「シーンエントリ」としてビルドします。これにより、開発者は個別の処理の違いを意識せず、同一の配置単位として管理できます。

```
{
  "primitives": [
    {
      "id": "floor",
      "type": "cube",
      "args": [16],
      "transform": {
        "translation": [0.0, -10.0, 0.0],
        "rotation": [0.0, 0.0, 0.0, 1.0],
```

```
        "scale": [1.0, 0.08, 1.0]
      }
    }
  ],
  "models": [
    {
      "id": "hero",
      "source": "../json_loader/modelasset.json",
      "transform": {
        "translation": [0.0, 0.0, 0.0],
        "rotation": [0.0, 0.0, 0.0, 1.0],
        "scale": [1.5, 1.5, 1.5]
      },
      "bindAnimations": true,
      "startAnimations": true,
      "playOnUpdate": true
    }
  ]
}
```

ビルド済みエントリの操作

ビルド後に特定のシーンエントリを操作したい場合は、`getEntry(id)` を使用します。戻り値には `placementNode`、`runtime`、`nodeMap` などが含まれており、初期化後の座標調整や可視化の変更を JavaScript 側で容易に行うことができます。

```
const sceneRuntime = await app.loadScene(sceneAsset.getData());

const heroEntry = sceneRuntime.getEntry("hero");
const floorEntry = sceneRuntime.getEntry("floor");

heroEntry.placementNode.move(2.0, 0.0, 0.0);
floorEntry.runtime.shapes[0].setWireframe(true);
```

sceneRuntime.update() によるアニメーション制御

playOnUpdate が有効なモデルエントリが含まれるシーンでは、毎フレーム sceneRuntime.update() を呼び出すことで、対象となるすべてのアニメーションを一括して進行させることができます。個別のクリップを詳細に制御したい場合は getEntry(id).runtime のアニメーションヘルパーを利用し、シーン全体を大まかに進行させたい場合は update() を利用するという使い分けが可能です。

```
app.start({
  onUpdate() {
    sceneRuntime.update();
  }
});
```

シーン構成の保存

現在のシーン構成を保存したい場合は、downloadJSON() を利用できます。保存された JSON は、構成の比較や診断情報の解析に活用でき、サンプル制作中の状態固定にも便利です。

```
const sceneAsset = SceneAsset.fromData(sceneObject);
sceneAsset.downloadJSON("scene-export.json");
```

11.4　シーンランタイムの構造

SceneLoader.build(scene) が返すシーンランタイムは、単なるエントリの配列ではなく、以下の 4 つの機能を備えたオブジェクトです。

- entries: ビルド済みのシーンエントリ一覧。
- inputMap: 小文字化したキーでアクションを検索できる対応表。
- update(): playOnUpdate が有効なモデルアニメーションを一括更新するヘルパー。

- createInputHandler() / getEntry(id): JavaScript からシーンランタイムを効率的に操作するためのヘルパー。

この構造により、Scene JSON は「初期状態の宣言」に専念し、ビルド後の JavaScript 側では「エントリの取得」「入力の配線」「アニメーションの更新」といった実務的な操作に集中できる設計となっています。

11.5 Scene JSON のデータ構成

Scene JSON のトップレベルは、主に以下の項目で構成されます。個々のスキーマを記憶するよりも、「どのような役割のデータか」という視点で把握してください。

```
{
  "version": "1.0",
  "type": "webg-scene",
  "meta": {},
  "camera": {},
  "hud": {},
  "input": {},
  "primitives": [],
  "models": []
}
```

- camera: アプリ起動時のカメラ状態（target, distance, yaw, pitch, roll, viewAngle, near, far）を定義します。WebgApp を対象にビルドした場合、これらの値がアプリのカメラに反映されます。
- hud: ガイドテキストとステータステキストの初期表示を定義します。実用的には各行を { x, y, text, color } で明示するオブジェクト形式を推奨します。
- input: キーからアクションへの対応表です。ここではアクション名のみを定義し、実処理は createInputHandler() で紐付けます。
- primitives: Primitive ファクトリーによるシーンエントリの定義です。type, args, transform のほか、必要に応じて material や wireframe を指定します。
- models: ModelAsset をシーンエントリとして配置する定義です。source（または埋め込み asset）、transform、bindAnimations, startAnimations, playOnUpdate などを指定します。

11.6　検証（validate）とビルド（build）の動作

SceneValidator.validate(scene) は、トップレベルの構造、カメラの数値、HUD の行配列、入力バインディング、プリミティブ、モデルの構造を包括的に検証します。戻り値は以下の形式で返されます。

- errors: ビルドを停止させるべき致命的な不整合。
- warnings: ビルドは可能だが、見直しを推奨する項目。

読み込み失敗を早期に検知したい場合は、assertValid() を使用してください。

```
{
  ok: true,
  errors: [],
  warnings: []
}
```

SceneAsset.build(target) および SceneLoader.build(scene) は、検証を通過したシーンデータからランタイムを組み立てます。target が WebgApp であればカメラと HUD も反映され、{ gpu, space } のみの場合はシーンエントリのビルドに特化します。

また、プリミティブとモデルの双方において、SceneLoader はまず「配置ノード」を作成し、その配下にランタイムのルート Node 群を接続します。これにより、アセット内部の原点や骨格構造を維持したまま、シーンエントリ単位で外部から配置トランスフォームを適用できます。

アニメーションに関しては、モデルエントリで startAnimations が false でない場合はビルド後に全クリップを開始し、playOnUpdate が false でない場合は sceneRuntime.update() によって毎フレーム更新されます。Scene JSON では「再生方針」までを宣言し、個別のクリップ制御などの詳細なロジックは JavaScript 側で実装する運用となります。

11.7　動作確認のためのリファレンス

Scene JSON の挙動を具体的に確認したい場合は、以下のサンプルを参照してください。

- webg/samples/scene: Scene JSON を読み込み、カメラ、HUD、プリミティブ、モデル、入力を統合的に確認できる標準サンプルです。入力アクションを createInputHandler() で接続する流れや、SceneAsset.downloadJSON() による再保存、診断情報の確認まで一通り実装されており、最初に読むべき例として最適です。
- webg/samples/json_loader: Scene JSON 内で参照される ModelAsset の最小構成を確認できます。

11.8　変更時の注意点と最小構成例

Scene JSON 関連の機能を変更する際は、整合性を保つため、以下のセットを併せて確認してください。

- スキーマの変更: SceneValidator と SceneLoader
- カメラ仕様の変更: validateCamera() と applyCamera()
- HUD 書式の変更: validateHud() と normalizeHudLines()
- 入力処理の変更: createInputMap() と createInputHandler()
- エントリ種類の追加: バリデータとビルドフローの両方

特に HUD の文字列短縮記法については、バリデータとローダーの間で許容範囲に一部差異があるため、ドキュメントやサンプルを更新する際はビルドが完全に通るオブジェクト形式を優先して使用してください。

以下に、最小構成の Scene JSON の例を示します。camera, hud, input, primitives, models を含めることで、シーンの基本構造を網羅できます。type と version は、保存や比較を行う際に推奨される項目です。

```
{
  "version": "1.0",
  "type": "webg-scene",
  "meta": {
    "name": "triangle-scene"
  },
  "camera": {
    "target": [0, 0, 0],
    "distance": 30,
    "yaw": 0,
```

```
    "pitch": 0,
    "roll": 0,
    "viewAngle": 55,
    "near": 0.1,
    "far": 1000
  },
  "hud": {
    "guideLines": [
      {
        "x": 0,
        "y": 1,
        "text": "scene ready",
        "color": [1.0, 0.9, 0.6]
      }
    ]
  },
  "input": {
    "bindings": [
      { "key": "r", "action": "reset-camera", "description": "reset orbit camera" }
    ]
  },
  "primitives": [
    {
      "id": "floor",
      "type": "cube",
      "args": [12],
      "transform": {
        "translation": [0, -7, 0],
        "rotation": [0, 0, 0, 1],
        "scale": [1, 0.08, 1]
      }
    }
  ],
  "models": [
    {
      "id": "hero",
      "source": "./modelasset.json",
      "transform": {
        "translation": [0, 0, 0],
        "rotation": [0, 0, 0, 1],
        "scale": [1, 1, 1]
      },
```

```
      "bindAnimations": true,
      "startAnimations": true,
      "playOnUpdate": true
    }
  ]
}
```

11.9 まとめ

本章で最も重要な点は、Scene JSON を「ゲームロジックを格納する箱」ではなく、「シーン全体の初期状態を宣言する共通表現」として捉えることです。`ModelAsset` が単一モデルの表現であったのに対し、Scene JSON はカメラ、HUD、入力、プリミティブ、モデルを包括する「シーン全体の初期配置表」として機能します。

`SceneAsset` が保存・読み込み・検証・ビルドのインターフェースとなり、`SceneLoader` がそのデータを実際のランタイムへと変換します。この構造を理解することで、シーンの初期化コードとアプリケーションのロジック本体をきれいに分離することが可能になります。

また、高レイヤー（ハイレベル）な経路と低レイヤー（ローレベル）な経路の使い分け、`createInputHandler()` や `getEntry(id)`、`update()` といったビルド後の操作手法についても確認しました。プリミティブとモデルを同一のシーンエントリとして扱い、入力の宣言を JSON に集約させることで、開発効率と保守性が向上します。

次章では、このように読み込まれたモデルやシーンに対して、具体的な動きを与えるためのアニメーションについて解説します。

第 12 章

アニメーション

本章では、webg/Animation.js、webg/Action.js、webg/AnimationState.js、webg/Schedule.js、webg/Task.js の役割と使い方について解説します。ここで扱う「アニメーション」とは、単にオブジェクトが動くことだけを指すのではありません。webg では、アニメーションを clip → pattern → action → state という階層構造で捉えます。このように層を分けて定義することで、アセット側で保持すべきデータ、ランタイム側で組み立てる仕組み、そしてサンプルやゲームコード側で制御すべきロジックを明確に切り分けることができます。

本章でまず理解していただきたいのは、アニメーションの制御を単一の仕組みとして捉えず、役割ごとに分離して考えることです。clip はインポーターが取り込んだ生のキーフレーム列であり、pattern はその内部の特定区間、action はその区間の集合、そして state はどの action または clip を使用するかを決定する上位制御を担います。たとえば、AnimationState を導入したからといってキーフレーム自体が増えるわけではありません。動きの滑らかさは、Action が選択する fromKey / toKey や entryDurationMs、そして下層の時間管理に強く依存します。この構造を理解しておくことで、「アセット側の問題」と「ランタイム側の制御の問題」を混同せずに切り分けることが可能になります。

また、提供されているサンプルの構成も、この区別に準拠しています。collada_loader や gltf_loader は、インポーターが clip 全体を正しく復元できているかを確認するためのサンプルであり、読み込んだ clip を最初から最後まで再生してキーフレーム列そのものの見え方を確認します。一方で hand、janken、animation_state は、Action を用いて clip の一部を切り出して利用するサンプルです。そのため、後者は fromKey / toKey の設定に強く依存しており、Blender のエクスポート設定によってキー数や配置が変わると、同じアセット名であっても見え方が変化します。これが、ローダーサンプルとポーズ切り替えサンプルの根本的な違いです。

さらに、fromKey / toKey と entryDurationMs の意味を混同しないことが重要です。fromKey / toKey は 0 始まりのキーインデックスであり、現在の Action.js の実装では fromKey < toKey である必要があります。つまり、「1 つのキーを選択する」というよりは、「短い保持区間を切り出す」と考えるのが適切です。対して entryDurationMs は、ソースファイルのキーフレーム間隔を指すものではありません。これはランタイム側で、「現在の姿勢から、指定した pattern の開始姿勢へ何ミリ秒かけて遷移するか」を指定する値です。結果としてアセット側のキー時刻と似た見え方になることはありますが、自動的に一致するわけではない点に注意してください。

アセット作成時の前提についても整理しておきます。Blender からエクスポートするアセットは Y-up であることを前提としており、webg 側で軸補正を行う運用は想定していません。Collada をキーフレームベースで利用する場合は、全フレームを出力するのではなく、Sampling 間隔を大きくし、Keep Keyframes を有効にしたエクスポートが推奨されます。glTF / GLB を利用する場合は、Sampling を無効にし、補間を LINEAR に設定して出力するのが安全です。現在の webg/GltfShape.js は CUBICSPLINE を受け取った場合、前後の tangent を除いて中央値のみを抽出し、LINEAR のキー列として近似します。そのため、CUBICSPLINE を厳密に再生することを前提とする場合は注意が必要です。また、hand 系アセットについては、現状では 0: rest, 1: goo, 2-3: N1, 4-5: N2, 6-7: N3, 8-9: N4, 10-11: N5, 12-14: N0 と解釈すると整理しやすくなります。同じ姿勢が続いている区間は単なる重複ではなく、その姿勢を保持して見せる時間や、次の状態へ遷移するための余白として機能しています。

本章では、まず各層の役割を明確に定義し、その後に Animation.js、Action.js、AnimationState.js、Schedule.js、Task.js の順に詳細を解説します。最後に、目的に応じてどの層までを利用すべきか、および誤解しやすいポイントをまとめます。本章の目的は、実装の構造を正しく読み解き、サンプルで何が起きているかを把握するための「地図」を提供することです。

12.1　各層の役割を分けて考える

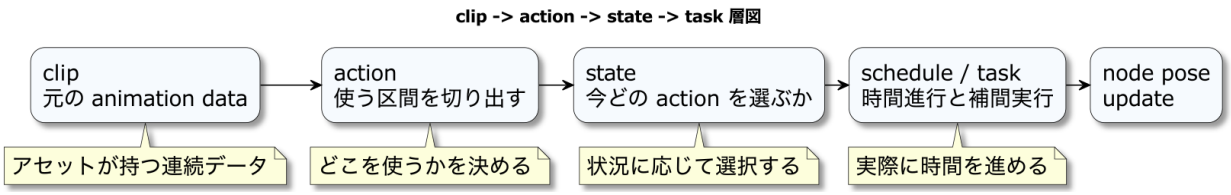


図 12.1: clip action state task 層図

clip、*action*、*state*、*task* は似て見えますが、それぞれ役割が異なります。アニメーションに関する問題を解決する際は、まずどの層の話であるかを確定させることが重要です。

まず clip は、1 本の連続したアニメーションを指します。ランタイムでは Animation.js のオブジェクトがこれに相当します。特定のスケルトンに対して、「どの時刻に、どのボーンが、どの姿勢を取るか」というデータ列をまとめて保持しています。DCC ツール側で Idle、Walk、Jump を別々に出力した場合、通常はそれぞれが別々の clip となります。clip は最下層のデータであるため、「再生する」「停止する」「特定のキー区間へ移行する」といった低レイヤー（ローレベル）な役割を担いますが、「ゲームの状態に応じて Walk を使う」といった判断は行いません。あくまで再生される対象としてのデータです。

pattern は、その clip の内部にある特定の区間を指します。Action.js では、特定のキー範囲を以下のような形式で登録します。

```
// 特定のキー区間を定義し、ランタイムでの入り込み時間を指定
{
  id: "step_left",
  fromKey: 2,
  toKey: 5,
  entryDurationMs: 120
}
```

ここで重要なのは、pattern は新しいアニメーションを作成する仕組みではなく、既存の clip のどこを使用するかを宣言する仕組みであるということです。fromKey は開始キー、toKey は終了キーであり、entryDurationMs は現在の姿勢からその開始地点へ何ミリ秒で遷移するかをランタイム側で決定する値です。したがって、pattern は「区間の選択」と「遷移時間」の定義であり、キーフレーム列そのものではありません。

action は、利用者やサンプル側で扱いやすい単位にするため、pattern を束ねたものです。たとえば、「左足の区間」と「右足の区間」を walk_loop という名前でまとめておけば、サンプル側は詳細な区間名を意識せず、walk_loop という動作名だけで制御できます。

```
// 複数の pattern を束ねて、ループ設定を含めて定義
{
  id: "walk_loop",
  patterns: ["step_left", "step_right"],
  loop: true
}
```

Action.js は、この action を再生しながら現在の pattern を管理し、終端に達した際に「次へ進むか」「先頭に戻るか」「停止するか」を制御します。つまり action は、「どの区間を、どのような順番で見せるか」を定義する層です。

state は、「今どの action または clip を使用すべきか」を決定する高レイヤー（ハイレベル）な制御層です。AnimationState.js では、状態名と遷移条件を持つオブジェクトとして管理します。

```
// 状態名、使用する action、遷移条件を定義
{
  id: "walk",
  action: "walk_loop",
  transitions: [
    { to: "idle", test: (ctx) => ctx.moveSpeed <= 0.1 },
    { to: "jump", test: (ctx) => ctx.jumpPressed }
  ]
}
```

ここでの state はアニメーションデータそのものは保持しません。保持しているのは、「この状態のときに何を再生するか」という参照と、「どのような条件で次の状態へ移るか」という遷移ルールです。このように、action は「再生対象のまとまり」であり、state は「再生対象を選択する判断単位」であるという明確な役割の違いがあります。

12.2 Animation.js は clip をそのまま再生する層

Animation.js は、最下層の clip を再生するためのオブジェクトです。名称からは汎用的なアニメーションマネージャーのように見えますが、実際には「1 本の clip をスケルトンに適用して進行させる」ことに特化しています。setTimes(times) と setBonePoses(bonePoses) でキー情報を保持し、setData(skeleton, bindShapeMatrix) でランタイム再生に必要な内部タスクを生成し、start() や play() で進行させます。つまり Animation は、「アニメーション列を時間軸に沿ってどう進めるか」を担当する層です。

ローダーサンプルなどで clip 全体の正しさを確認したい場合は、この層を直接利用するのが最もシンプルです。

```
// ローダーサンプル：clip 全体を確認するための基本的な使い方
const model = await app.loadModel("model.dae", {
  format: "collada",
  instantiate: true,
  startAnimations: false
});

const clipId = model.asset.getClipNames()[0];
const runtime = model.runtime;

runtime.restartAnimation(clipId);

function update() {
  runtime.playAnimation(clipId);
}
```

この手法は、アセット側のアニメーションが正しく作成されているかを確認したい場合や、Action や AnimationState による制御を加える前に、インポーターの結果を純粋に確かめたい場面に適しています。つまり、「まず clip が正しいか」を確認するための入り口が Animation.js です。

12.3 Action.js は clip の中を切り出して使いやすくする層

Action.js は、1 本の Animation を参照し、その内部区間を pattern や action として再利用しやすくするためのランタイムコントローラーです。その存在理由は、clip 全体ではなく、その一部分だけを分かりやすい名前で利用したいというニーズに応えるためです。特に、ハンドポーズのように 1 本の clip 内に複数のポーズが格納されているアセットでは、この層が不可欠になります。

```
// Action を初期化し、pattern と action を定義
const action = new Action(animation);

action.addPattern({
  id: "step_left",
  fromKey: 2,
  toKey: 5,
```

```
  entryDurationMs: 120
});

action.addPattern({
  id: "step_right",
  fromKey: 6,
  toKey: 9,
  entryDurationMs: 120
});

// action を定義して再生
action.addAction("walk_loop", ["step_left", "step_right"]);
action.start("walk_loop");
```

再生時は、毎フレーム play() を呼び出して進行させます。

```
// 毎フレーム呼び出して進行させる
action.play();
```

ここで重要なのは、Action が clip のデータ形式を変換しているわけではないという点です。既存の clip の中から「左足」「右足」「グー」「チョキ」といった区間を切り出し、利用者が扱いやすい名前で管理できるようにしているだけです。Action 層があることで、サンプル側のコードで毎回キーフレーム番号を意識する必要がなくなります。

12.4 AnimationState.js は再生対象の選択を分離する層

AnimationState.js は、現在どの action または clip を再生すべきかを決定する上位制御層です。その役割は、「再生対象の選択ロジックを、サンプル側のメイン処理から切り離すこと」にあります。条件分岐が増えると、サンプルコードの中で「どの状態で何を始めるか」という判断と、「実際に再生を進める処理」が混在し、可読性が低下します。そこで AnimationState を導入することで、「何を再生するか」という判断のみを独立させることができます。

最小限のステート定義は、以下のようになります。

```
// state の基本定義：id、action、遷移条件
{
  id: "idle",
  action: "idle",
  transitions: [
    { to: "walk", test: (ctx) => ctx.moveSpeed > 0.1 }
  ]
}
```

また、遷移定義には優先度やクールダウン時間を設定することも可能です。

```
// transition の詳細設定：優先度、クールダウン時間
{
  to: "jump",
  test: (ctx) => ctx.jumpPressed,
  priority: 10,
  cooldownMs: 80
}
```

ここでの state はアニメーションデータそのものではなく、あくまで「再生対象の選択ルール」です。同じ action を複数の state から参照することも可能ですし、state が直接 clip を指すこともできます。

標準的な導入手順としては、まず Action または clip 再生側の仕組みを用意し、その上に AnimationState を構築します。

```
// Action で区間を切り、それを state で参照
const clipId = model.asset.getClipNames()[0];
const animation = model.runtime.getAnimation(clipId);
const action = new Action(animation);

action.addPattern({ id: "pose0", fromKey: 2, toKey: 3, entryDurationMs: 220 });
action.addPattern({ id: "pose1", fromKey: 4, toKey: 5, entryDurationMs: 220 });
action.addAction("pose0", ["pose0"]);
action.addAction("pose1", ["pose1"]);

const sm = new AnimationState(action, {
  initialState: "pose0"
```

```
});

sm.addState({ id: "pose0", action: "pose0" });
sm.addState({ id: "pose1", action: "pose1" });
```

更新時は、毎フレーム update(context) を呼び出します。

```
// 毎フレームコンテキストを渡して更新
sm.update({
  moveSpeed,
  jumpPressed,
  grounded,
  nowMs
});
```

AnimationState には大きく分けて 2 つの利用形態があります。1 つは、上記のように毎フレーム update(context) を呼び出し、条件に基づいて遷移先を決定する「条件評価型」です。もう 1 つは、入力イベントやゲームイベントが発生した瞬間に setState() を直接呼び出す「コマンド型」です。webg/samples/animation_state は前者の、webg/samples/janken は後者の代表例です。

たとえば janken のように、ユーザー入力があった瞬間に特定のポーズを出したい場合は、以下のような実装が自然です。

```
// 入力イベントに応じて状態を切り替える
function onJankenInput(choiceId) {
  handState.setState(choiceId, {
    startOptions: {
      entryDurationMs: 250
    },
    force: true
  });
}
```

この形式では、毎フレーム状態を計算する必要がなく、入力イベントを高レベルなコマンドとして扱えます。また、force: true を指定することで、現在と同じ state であっても再始

動させることが可能です。これが条件評価型との大きな違いです。

12.5 Schedule.js と Task.js は補間実行を担当する層

AnimationState や Action が再生対象を決定した後、実際の移動と回転の処理はさらに下層へと流れます。ここを担当するのが Schedule.js と Task.js です。構造的に見ると、AnimationState は「何を再生するか」を決めるだけであり、補間式やボーンの座標値に直接触れることはありません。下層では、選択された区間が時間付きのコマンド列に変換され、それが各ボーンへ実行されます。

処理の流れを追うと、以下のようになります。まず AnimationState.update() が遷移を決定し、Action.start() が現在の pattern を選択します。その後、transitionToKey(entryDurationMs, fromKey, toKey) が Animation.startTimeFromTo() を呼び出し、Animation.setData() がボーンごとの Task を組み立てます。そして Schedule.directExecution() がコマンド列を流し、最終的に Task.execute(delta_msec) が CoordinateSystem.prototype.putRotTransByMatrix や CoordinateSystem.prototype.doRotTrans を対象ボーンに適用します。つまり、実際にボーンを動かしているのは最下層の Task です。

この分離構造があるため、ステートマシン（状態管理）と補間実行（物理的な計算）を混在させずに済みます。AnimationState は「今どの動きを見せたいか」という意図のみを管理し、Task は「その命令を時間でどう刻むか」という実行に集中できます。これにより、入力条件やゲームルールが変更されても、補間処理のコードを修正せずに済む堅牢な設計となっています。

12.6 どの層を使うべきか

実装において、どの層までを利用すべきかの判断基準を整理します。

- Animation のみで十分な場合 clip 全体を 1 本再生できれば十分なときです。ローダーサンプルのように、インポーターが復元した clip の見え方そのものを確認したい場面がこれに当たります。
- Action まで利用すべき場合 clip の内部キー区間を明示的に再利用したいときです。指のポーズや歩行のステップのように、短い区間を分かりやすい名前でまとめたい場合や、サンプル側で action の再生を直接制御しても複雑にならない規模の場合に適しています。

- AnimationState まで利用すべき場合状態名（State）を定義して管理したいとき、あるいは入力やゲームプレイの条件をアニメーション再生ロジックから完全に分離したいときに導入します。また、HUD や診断情報に現在の state 情報を共通形式で表示したい場合や、入力イベントを高レベルコマンドとして扱いたい場合にも有効です。

判断の目安は、「条件分岐がどれだけ増えたか」と「再生対象の選択ロジックを別層に切り出したほうがコードの可読性が向上するか」という点にあります。

12.7 よくある誤解

設計時に注意すべき点として、いくつかよくある誤解を挙げます。

まず、「AnimationState を導入すれば遷移が必ず滑らかになる」という考え方です。これは誤りです。AnimationState はあくまで再生先を決定するヘルパーであり、補間品質そのものを向上させる仕組みではありません。滑らかさは、アセット側のキー配置、Action の fromKey / toKey、entryDurationMs、および Animation.transitionTo() の設定によって決定されます。

次に、「Action があればステートマシンは不要である」という理解です。小規模なサンプルではそれで十分かもしれませんが、一般化はできません。条件分岐が増えるにつれ、サンプル側が「どの action を開始すべきか」という判断ロジックを持ちすぎることになります。その際にステートマシンを別層に設けることで、再生対象の選択ルールを一箇所にまとめて管理できるようになります。

また、「state と action は同じものである」という誤解もあります。action は再生対象のシーケンス（並び）であり、state はどの action を使用するかを決定する判断単位です。同じ action を複数の state から参照することも可能ですし、state が直接 clip を指すこともあります。

最後に、「AnimationState は遷移評価型（update による更新）でしか使えない」という点です。janken サンプルのように、入力イベントごとに setState(..., { force: true }) を呼び出すコマンド型の運用も、非常に自然で有効な使い方です。

12.8 まとめ

本章で最も重要なのは、webg のアニメーションを単一の仕組みとしてではなく、clip → pattern → action → state という階層構造で理解することです。

- clip: インポーターが取り込んだキーフレーム列。
- Action: 内部区間を切り出して再利用しやすくする層。
- AnimationState: 現在どの action または clip を使うかを決定する上位制御層。
- Animation → Schedule → Task: 実際の補間計算とボーンへの反映を担当する低レイヤー層。

この構造を把握することで、問題が発生した際にそれが「アセット側の問題」なのか、「アクション区間の設定問題」なのか、あるいは「状態遷移のロジック問題」なのかを迅速に切り分けることができます。

本章では、ローダーサンプルとポーズ切り替えサンプルの違い、fromKey / toKey と entryDurationMs の役割、エクスポート時の設定条件、そして AnimationState の 2 つの運用形態について整理しました。次章では、これらの層を実際のサンプル（animation_state、janken）でどのように活用し、アセットのキー区間をどう判断するかを実践的に見ていきます。

第 13 章

アニメーションとアセット

本章では、webg/samples/animation_state、webg/samples/janken を中心に、Animation、Action、AnimationState を実際のアプリケーションへどのように組み込むかを整理した実践ガイドを解説します。第 12 章で各層の役割について説明しましたが、本章では「実際のサンプルではどこに何を記述するのか」「アセットのキー区間をどのように判断するのか」という具体的な実装面に焦点を当てます。

本章では、実装および core の仕様に沿って、利用者がそのままアプリケーション開発に適用できる構成を提示します。主な参照先として、animation_state と janken の 2 つのサンプルを扱います。

まず、本章で扱う 2 つのサンプルの特性を整理します。animation_state は条件評価型の AnimationState を、janken は入力駆動型の AnimationState を示しています。これら 2 つのサンプルを理解することで、webg におけるアニメーションの活用方法を把握することが可能です。

実装における重要な分かれ目は、「何を再生するか」と「どう進めるか」の分離にあります。サンプル側で定義するのは、どのステートやアクションを選択するかという制御層です。一方で、実際の補間処理や bone への反映といった低レイヤー（ローレベル）な処理は、Animation → Schedule → Task → CoordinateSystem が担当します。この役割分担を明確に意識することで、入力処理とアニメーション処理の混在を防ぎ、保守性の高い実装が可能になります。

また、fromKey / toKey と entryDurationMs については、アセット側の定義とランタイム側の制御を分けて考える必要があります。fromKey / toKey は使用するキー区間を指定し、entryDurationMs は現在の姿勢からその区間へ何ミリ秒かけて遷移するかを指定します。アニメーションの挙動に違和感がある場合は、これら 2 つの設定を個別に確認してください。

なお、ハンド系の現行アセットには samples/gltf_loader/hand.glb を使用します。animation_state と janken はともにこの hand.glb を利用しており、例えば N0 は 12-13、N2 は 4-5、N5 は 10-11 といった区間対応をしています。

本章の構成として、まずは AnimationState から Task に至る実フローをサンプルに沿って追い、次に animation_state、janken のそれぞれの役割の違いを確認します。後半では、hand.glb のキー区間の見方について解説します。本章の目的は、アニメーションの概念的な説明ではなく、「現行サンプルを読み解く際に、どの層をどのように確認すべきか」を整理することにあります。

13.1 AnimationState から Task までの実フロー

AnimationState から実際の移動・回転に至るまでのフローを確認すると、各層の役割分担が明確に分かれていることがわかります。AnimationState は「何を再生するか」を決定するのみで、bone の座標値や補間式には関与しません。その下の低レイヤー（ローレベル）な層において、再生対象の選択、区間の切り出し、時間補間、そして個々の bone への命令実行が順次行われます。

具体的な処理の流れは以下の通りです。

1. AnimationState.update(context, deltaMs) が現在のステートを更新し、必要に応じて遷移を選択します。resolveTransition() が真を返した場合は setState() が呼ばれ、onExit の実行、currentState の更新、onEnter の実行、そして playStateTarget() まで処理が進みます。
2. playStateTarget() はコントローラーの型に応じて start()、startAction()、startAnimation()、restartAnimation() のいずれかを呼び出します。Action をコントローラーに渡している場合は、ここで指定した action 名の再生が開始されます。
3. Action.start(actionId, options) は action から最初の pattern を取り出し、currentPattern を決定した上で transitionToKey() を呼び出します。Action.play() は、再生中の pattern が終了すると次の pattern へ進み、必要に応じてループ処理を行います。
4. transitionToKey(entryDurationMs, fromKey, toKey) は Animation.js の startTimeFromTo() を呼び出します。ここで pattern の区間と entry duration が、実際の補間対象として設定されます。
5. Animation.startTimeFromTo(time, keyFrom, keyTo) は transitionTo() を呼び出し、keyFrom と keyTo のポーズを用いて Schedule へコマンドを送信します。putR

otTransByMatrix で開始姿勢を設定し、その後 doRotTrans で時間補間を進めます。

6. Animation.setData(skeleton, bind_shape_matrix) では、bone ごとに Task を生成し、各キー区間について [0, putRotTransByMatrix, ...] と [time, doRotTrans, [1.0]] のコマンドペアを登録します。これにより、補間の元となるコマンド列が組み立てられます。
7. Schedule.directExecution() は各 bone 用の Task に同じコマンドを配分し、time > 0 の場合は insertCurrentCommand() で実行中のコマンド列へ差し込みます。Schedule.doCommand() は delta_msec を計算し、各 Task.execute(delta_msec) を実行します。
8. Task.execute(delta_msec) が remaining_time を監視しながらコマンドを実行し、execCommand() を通じて CoordinateSystem.prototype.putRotTransByMatrix や CoordinateSystem.prototype.doRotTrans を対象 bone に対して呼び出します。ここが実際の移動・回転が適用される最終地点となります。

このように層を分離している理由は、ステートマシンと補間実行を切り離すためです。AnimationState を「今どのアニメーションを見せたいか」という決定のみに特化させることで、入力条件やゲームプレイ上の条件変更があっても、補間処理のコードに影響を与えずに済みます。また、Task は「命令を時間軸でどう刻むか」という点に集中しているため、アニメーション以外のコマンド列にも同様の仕組みを流用しやすくなります。

例えば webg/samples/janken のようなコマンド駆動型では、入力を受けた瞬間に setState() で action を出し直し、その action が Animation と Task のコマンド列へと流れます。一方、webg/samples/animation_state のような条件評価型では、毎フレームのコンテキストからステートを選択し、そのステートが下層の action を開始させます。どちらの場合も、実際に bone を駆動させているのは Task です。

デバッグを行う際は、AnimationState.getDebugInfo() でステート遷移を確認し、Action.getActionInfo() で現在の action および pattern を確認してください。さらに Animation の getClipInfo() や Task のコマンド列を追うことで、どの層で意図しない挙動が発生しているかを効率的に切り分けることができます。

13.2 animation_state の NO を使った具体例

webg/samples/animation_state は、Action と AnimationState の連携を確認できる最短の参照サンプルです。main.js では hand.glb のポーズ区間を pattern と action として登録し、AnimationState から pose0 を開始させています。

```
const pattern = { id: "N0", fromKey: 12, toKey: 13, entryDurationMs: 250 };
action.addPattern(pattern);
action.addAction(pattern.id, [pattern.id]);
animationState.setState("pose0", {
  context: {
    desiredPoseId,
    nowMs: app.space.now()
  },
  force: true
});
```

このときの内部的な呼び出し順序は以下の通りです。

1. AnimationState.setState("pose0") が pose0 ステートを currentState に設定する。
2. AnimationState.playStateTarget() が state.action === "N0" を判定し、Action.start("N0") を呼び出す。
3. Action.start("N0") が先頭 pattern として N0 を選択し、transitionToKey(250, 12, 13) を呼び出す。
4. transitionToKey(250, 12, 13) が anim.startTimeFromTo(250, 12, 13) を呼び出す。
5. Animation.startTimeFromTo(250, 12, 13) が transitionTo(250, 12, 13) を呼び出す。
6. transitionTo() は、各 bone について this.poses[i][12] を args に格納し、ones に 1.0 を設定する。
7. Schedule.directExecution(0, CoordinateSystem.prototype.putRotTransByMatrix, args) が、各 Task に開始姿勢を渡す。
8. Schedule.directExecution(250, CoordinateSystem.prototype.doRotTrans, ones, 24, 25) が、各 Task に doRotTrans(1.0) を 250 ms かけて流す。
9. Task.execute(delta_msec) が毎フレーム呼ばれ、remaining_time を減らしながら doRotTrans(1.0) を進行させる。
10. CoordinateSystem.doRotTrans(1.0) が execRotation() と execTranslation() を呼び出し、回転は slerp、移動は直線補間で反映される。

ここで重要なのは、12 -> 13 は pattern のキー範囲 であり、250 ms は その pattern を再生するために割り当てた時間 であるという点です。fromKey / toKey は clip のキーを選択

するためのインデックスであり、entryDurationMs はランタイム側で制御する補間時間です。animation_state では AnimationState.update() が内部で action.play() まで処理を進めるため、サンプル側は希望するステート（desired state）を更新するだけで済みます。webg/samples/janken と webg/samples/animation_state は同じ pattern 定義を使用しますが、AnimationState が「どの action を開始するか」を選択する役割を担い、実際の移動・回転の補間は同様に Action → Animation → Schedule → Task → CoordinateSystem の順で処理されます。

13.3 startFromTo() 系と startTimeFromTo() 系の違い

webg には、似た名称の startFromTo() と startTimeFromTo() が存在しますが、その役割は明確に異なります。簡潔に言えば、startFromTo() は clip のキー区間をそのまま再生する入口 であり、startTimeFromTo() は 現在姿勢から fromKey へ遷移するための時間を別途指定する入口 です。

Animation.startFromTo(keyFrom, keyTo) の実装は以下の通りです。

```
// キー区間を直接再生する
startFromTo(keyFrom, keyTo) {
  this.setPose(keyFrom);
  this.schedule.startFromTo(keyFrom * 2, keyTo * 2 - 1);
}
```

この動作は非常にシンプルです。setPose(keyFrom) が fromKey の姿勢を即座に適用し、schedule.startFromTo(keyFrom * 2, keyTo * 2 - 1) が命令列の該当区間を clip 側の時間差で進行させます。例えば startFromTo(12, 13) を呼び出した場合、setPose(12) の後に schedule.startFromTo(24, 25) が呼ばれます。つまり、12 の姿勢に到達した状態から、12 → 13 の区間を clip 本来の時間差で再生する 形となり、fromKey へ至るための追加補間は行われません。

対して Animation.startTimeFromTo(time, keyFrom, keyTo) は、entryDurationMs を別途指定することで、現在姿勢から fromKey へ至るための補間を先行して生成します。

```
// 指定した時間でキーへ入り込む
startTimeFromTo(time, keyFrom, keyTo) {
  this.transitionTo(time, keyFrom, keyTo);
```

```
}
```

transitionTo(250, 12, 13) では、各 bone について this.poses[i][12] を収集し、schedule.directExecution(0, CoordinateSystem.prototype.putRotTransByMatrix, args) と schedule.directExecution(250, CoordinateSystem.prototype.doRotTrans, ones, 24, 25) を順に呼び出します。その結果、現在姿勢から 12 の姿勢へ 250 ms で遷移する 処理と、その後に 12 → 13 の clip 区間へ入る 処理の 2 段構成で実行されます。

NO のような pattern がこの形式を採用しているのは、単に clip 区間をそのまま再生することではなく、「現在の姿勢から次の状態へ自然に移行するエントリー（導入）」が必要だからです。したがって、startFromTo() は clip 自体の動作確認に適しており、startTimeFromTo() は idle → walk や rock → scissors といった状態遷移に適しています。

13.4　サンプルから見た入口の整理

実際のサンプルにおけるエントリーポイントを整理すると、以下のようになります。

webg/samples/animation_state/main.js では、Action の pattern および action を組み立て、サンプル側は desiredPoseId を変更するのみとして、制御を AnimationState へ委譲しています。AnimationState.update() を通じて、内部的に Action.start() と action.play() が進行します。

webg/samples/janken/main.js では、ユーザー入力に応じて AnimationState.setState(..., { force: true }) を呼び出し、同様に Action へ処理を流します。

以上の整理からわかる通り、サンプルから startFromTo() を直接呼び出す場面はほとんどありません。startFromTo() は Animation 内部で clip 全体の再生や Action が生成したコマンド列を処理するための下位関数です。開発者が意識すべきは、主に Action.startPattern()、Action.start()、AnimationState.setState() であり、それらが最終的に startTimeFromTo() 系へと遷移すると理解しておくのが効率的です。

13.5　animation_state と janken の読み方

webg/samples/animation_state は、glTF 版の hand.glb を用いて AnimationState の

条件評価型を確認するためのサンプルです。サンプル側は desired state を更新するのみとし、どの action を開始するかという判断を AnimationState に委譲しています。

もしサンプル側で Action.start() を直接呼ぶ構成にした場合、「どの pattern / action を開始するか」というロジックをサンプルコード自体が明示的に保持することになります。対して webg/samples/animation_state では、サンプル側が desiredPoseId を更新し、AnimationState が pose0 から pose5 のステートを管理し、それぞれに対応する N0 から N5 のアクションを開始させます。この設計の差は、役割分担の考え方の違いです。前者は「再生対象の選択」までサンプル側で行い、後者は「どの状態を望むか」という意図のみをサンプル側が保持します。後者のアプローチは、状態遷移の条件を拡張したい場合に非常に整理しやすくなります。

また、本サンプルでは、1〜6 キーで desired state を変更し、/ キーで次の desired state へ進む、P キーで自動巡回させる、@ キーで current state を再始動させるといった操作が可能です。HUD には current state、target state、現在の action、現在の pattern、および直近の transition が表示されるため、「望んだステート」と「実際に動作しているアクション」の相関関係を容易に追跡できます。

一方、webg/samples/janken は AnimationState のコマンド型利用を確認するためのサンプルです。条件評価型とは異なり、毎フレーム次のステートを計算するのではなく、「入力があった瞬間にその手を出す」という動作が自然な場面に適しています。

webg/samples/animation_state が desiredPoseId を毎フレーム更新し、update(context) が transitions を評価して遷移先を決める「条件評価型」であるのに対し、webg/samples/janken は入力イベントごとに setState(choiceId, { force: true }) を呼び出し、update() はアクションの進行とステート情報の維持のみを行います。

この違いは「判断のタイミング」にあります。前者は「毎フレームの条件」に基づき、後者は「入力イベントそのもの」に基づき判断します。じゃんけんのようなアプリケーションでは、毎フレーム条件を確認して rock / scissors / paper を選ぶよりも、G、C、P の入力を受けた瞬間に即座に反応させることが重要です。また、同じ手を続けて出し直すケースもあるため、同じステートを force 付きで再始動できる仕組みが不可欠です。したがって janken は、AnimationState が条件評価型以外でも有効に機能することを示す重要な例となっています。

13.6　補間問題を切り分けるときの結論

検証の結果、アニメーションの補間品質に関する問題は、AnimationState の有無だけで説明されるものではないという結論に至りました。見え方の問題は、より低レイヤー（ローレベル）な層で発生していることが多いためです。

まず、AnimationState は補間品質を改善するための機構ではなく、あくまで「何を再生するか」を決定するためのヘルパーであることを理解してください。ステートマシンを導入したからといって、中間ポーズが自動生成されたり、クロスフェードが自動的に追加されたり、疎なキーフレームが自然なモーションに変換されたりすることはありません。これらの制御は Animation や Action、あるいはアセット側の設計に依存します。

特にポーズの見え方は、キーの配置と entryDurationMs に強く依存します。ハンドポーズのようなサンプルでは、2 -> 3 や 12 -> 13 といった短い保持区間を多用します。このとき、視覚的な品質を左右するのは「現在姿勢から次のキーポーズまでの距離」と「entryDurationMs をどれだけ確保するか」です。1 本の clip の中に確認用ポーズだけが疎に配置されたアセットをそのままステート遷移に使用すると、ポーズによって滑らかに見える場合と急激に切り替わる場合が混在します。これはステートマシンの問題ではなく、アセットまたは pattern 設計に起因するものです。したがって、挙動に違和感がある場合は、ステートではなく、まずキー配置と pattern の設定を確認することが推奨されます。

また、停止後の再開時には Schedule の基準時刻も影響します。検証の結果、アクションを再始動した際に Schedule の基準時刻更新が不十分であると、最初の補間が一気に進んで見える現象が確認されました。つまり、同じ fromKey / toKey と entryDurationMs を使用していても、時間基準がずれることで見え方が変わります。

そのため、問題の切り分けを行う際は、pattern だけでなく以下の点も確認してください。- 再始動直後の最初のフレームで、過剰な delta time が入力されていないか。- pause / resume / restart の後に時間基準が適切に更新されているか。- ステートを切り替えた直後のフレームで、再生を急激に進めていないか。

最終的な切り分けの推奨順序は、「アセットの確認」 → 「clip の直接再生での確認」 → 「Action の確認」 → 「AnimationState の確認」 → 「Schedule の確認」となります。この順序で検証することで、「アセットのキーフレーム問題」「ステートマシンの運用問題」「時間管理問題」を混同せずに特定することが可能です。

13.7 hand.glb のキー区間と姿勢の対応

ハンド系のサンプルを確認する際、fromKey / toKey がどの姿勢を指しているか、また同じ姿勢が複数フレーム続いている理由について混乱することがあります。2026-04-07 JST 時点の webg/samples/gltf_loader/hand.glb におけるポーズ保持区間の対応表を以下に示します。

key 区間	姿勢	animation_state / janken の pattern
0	レストポジション	なし
1	グー	なし
2-3	1 本伸ばす	N1: 2 -> 3
4-5	チョキ	N2: 4 -> 5
6-7	3 本伸ばす	N3: 6 -> 7
8-9	4 本伸ばす	N4: 8 -> 9
10-11	パー	N5: 10 -> 11
12-13	グー	N0: 12 -> 13

ここで定義されている fromKey / toKey は、Action がそのポーズを提示するための再生区間です。entryDurationMs: 250 は、その区間へ移行する際にランタイム側で 0.25 秒かけて姿勢を近づける指定です。

同じ姿勢が連続している区間は、単なる重複ではなく、意図的な設計です。これは、その姿勢を一定時間保持させ、次の姿勢へ移行する直前にバッファを設けることで、再生時の急激な跳ね上がりを抑え、遷移を安定させるためです。例えば 4-5: チョキ の場合、4 から 5 の間でチョキの姿勢を保持し、そのポーズをサンプル側のステートから再利用できるようにしています。これを「同じ絵が 2 回入っている」と捉えるのではなく、「その姿勢の滞在時間が表現されている」と理解してください。

この保持区間があることで、入力や desired state の変化に応じた切り替えが自然に見えるようになり、将来的に Action の別の再生方法や AnimationState の拡張へ流用しやすくなります。また、アセット側のキーフレーム構成として「どの姿勢をどのくらいの時間見せるか」を明確に定義できる利点もあります。

Blender から glTF / GLB を出力する際の注意点についても触れます。webg/samples/gltf_loader/hand.glb のように glTF / GLB を使用し、アニメーションをキーフレームベースで制御したい場合は、補間モードを CUBICSPLINE ではなく LINEAR で出力することが基本です。

現在のインポーターは CUBICSPLINE を受け取った際、前後の tangent を読み飛ばし、中央の値のみを抽出して LINEAR として再生します。つまり、CUBICSPLINE を厳密に再現するのではなく、LINEAR へ近似変換して扱う前提となっています。そのため、Blender から出力する際は Sampling を無効にし、全フレームのベイクではなくキーフレームのみを出力し、補間を LINEAR に設定することが最も安全です。また、アーマチュアにおいて必要な bone / action が適切にキーを持っていることを確認してください。

これらの条件が揃うことで、glTF / GLB も「毎フレームの近似データ」ではなく「元のキーフレーム列」を保持した状態で扱うことが可能になります。逆に Sampling が有効であったり、補間が CUBICSPLINE のままであったりする場合、gltf_loader 側で tangent が破棄され LINEAR に近似されるため、厳密な曲線再現は期待できません。現時点の webg において、ハンド系の glTF アニメーションは LINEAR 前提で考えるのが適切です。

13.8 まとめ

本章で最も重要な点は、現行サンプルを単なる「アニメーションの例」としてではなく、「どの層をどこで利用しているか」という視点で読み解くことです。animation_state は条件評価型の AnimationState、janken は入力駆動型の AnimationState を扱う実例です。

どのサンプルにおいても、サンプル側が決定するのは「何を再生するか」という制御であり、実際の補間や bone への反映は Animation → Schedule → Task → CoordinateSystem という低レイヤー（ローレベル）なフローが担当しています。この構造を理解することで、入力処理、状態選択、補間実行を適切に分離して実装することが可能になります。

また、本章では startFromTo() と startTimeFromTo() の違い、hand.glb のキー区間の読み方、保持区間の意義、glTF / GLB のエクスポート条件を整理しました。これらを把握しておくことで、アニメーション付きモデルの挙動に問題が生じた際、「キー区間の問題か」「遷移時間の問題か」「時間基準の問題か」を論理的に切り分けることができます。

次章では、これまで扱ってきた HUD、dialogue、panel といった表示要素を、UI 設計という観点から体系的に整理していきます。

第 14 章

UI 表示の設計

14.1　文字表示の設計指針

3D アプリケーションの最大の魅力は、ダイナミックな視覚体験にあります。しかし、その体験を最大限に引き出すためには、描画した 3D オブジェクトに加えて、適切な「情報の提示」を組み合わせることが重要です。

利用者に操作方法を伝え、現在の状態を明示し、読み込み時の状況や診断結果を丁寧に説明する。あるいは、物語を伝える会話やチュートリアルを表示させる。これらの要素が揃うことで、アプリケーションは初めて「使いやすい道具」や「没入感のある体験」へと進化します。

一見すると、これらはすべて「文字を画面に出す」という単一の機能に思えるかもしれません。しかし、表示したい情報の性質によって、求められる機能や更新頻度は大きく異なります。

- 毎フレーム変動するスコアや FPS
- 画面端に配置する簡潔な操作説明
- 日本語を含む複数行のヘルプ
- 読み込み失敗時のエラーメッセージ
- 詳細な診断レポート（diagnostics report）
- ボタンや選択肢を伴うブリーフィング
- 開発者や解析ツールに共有するための状態記録
- タッチ環境でキー入力を代替する画面ボタン

これらを単一の仕組みですべて管理しようとすると、ユーザーインターフェイス（UI）が煩雑になり、視認性が著しく低下します。一方で、用途ごとにクラスを増やしすぎると、「どの

API を使用すべきか」という判断に迷うことになります。

そこで本章では、webg における文字表示を「表示したい情報の性質」に基づいて整理していきます。API 名を暗記するのではなく、「その情報は短い HUD なのか」「じっくり読ませる文章なのか」「操作を伴うパネルなのか」「記録として残すべきものなのか」という判断基準を明確にすることを目指します。

14.2 表示経路の全体像

webg の UI 表示は、大きく分けて次の 3 つの系統に分類されます。

- **キャンバス HUD (@<tt>{Text}, @<tt>{Message})** 短い ASCII 文字を、3D シーンと同じキャンバスに直接描画します。
- **DOM オーバーレイ (@<tt>{OverlayPanel}, @<tt>{CommandPalette})** 日本語、長文、ボタン、選択肢、スクロール付きパネルなどをシーン上に重ねて表示します。
- **記録 / 開発補助 (@<tt>{Diagnostics}, @<tt>{DebugDock})** 内部状態をレポート化し、開発者や解析ツールに共有可能な形式で管理します。

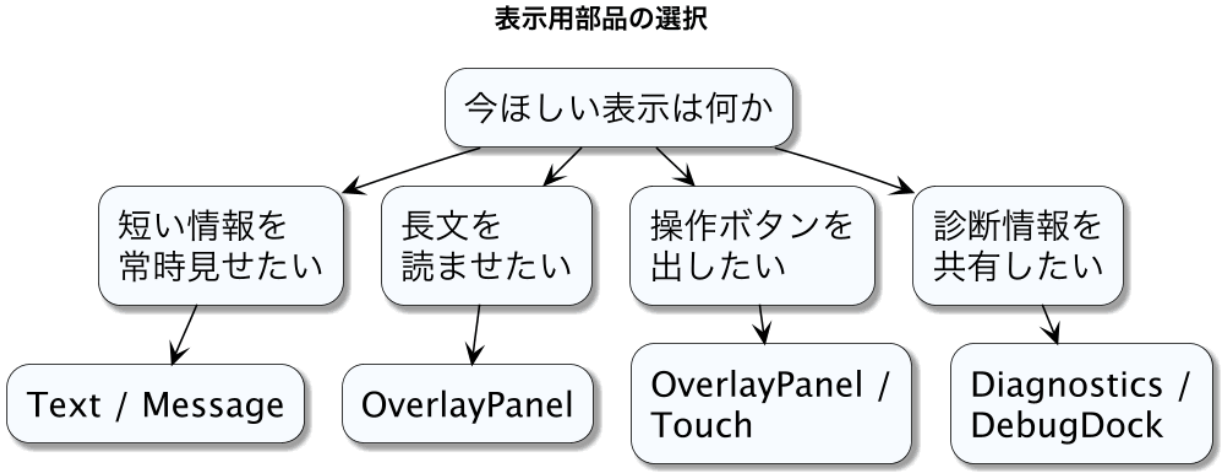


図 14.1: UI 部品選択表

これらに加え、Touch は画面上にボタンを表示しますが、これは文字表示のための API ではなく「入力 API」です。見た目は OverlayPanel や CommandPalette のボタンに似ていますが、目的はキー入力の代替であるため、詳細は後の章で詳しく解説します。

具体的な API 選択の判断基準は以下の通りです。

- **@<tt>{Message} を選択する場合**スコア、残機、FPS、簡潔な状態表示など。キャンバス HUD として軽量であり、毎フレームの更新に適しています。
- **@<tt>{OverlayPanel} を選択する場合**日本語の操作説明、ヘルプ、エラー、長文など。DOM ベースであるため、UTF-8、可変幅フォント、スクロール、ボタンを自在に扱えます。
- **@<tt>{OverlayPanel} + コントローラーを選択する場合**ボタン待機、選択肢の提示、ブリーフィングなど。表示（UI）と進行制御（ロジック）を分離して管理したい場合に適しています。
- **@<tt>{CommandPalette} を選択する場合**キャンバス上で一時的に開く操作メニューや、簡易的な設定変更など。特定の操作で呼び出し、多様な操作部品をコンパクトにまとめられます。
- **@<tt>{Diagnostics} / @<tt>{DebugDock} を選択する場合**調査ログや、解析ツールに渡す状態記録など。単なる表示ではなく、「記録」と「共有」を目的としています。
- **@<tt>{Touch} を選択する場合**タッチ操作インターフェースなど。キーボード入力の代替という「入力機能」としての役割を担います。

14.3 HUD と DOM の基礎知識

ここで、UI 設計の鍵となる「HUD」と「DOM」という 2 つの概念について整理しておきましょう。

HUD（Heads-Up Display：ヘッドアップディスプレイ）は、もともと航空機のコクピットなどで、パイロットが視線を正面から外さずに情報を確認できるよう、フロントガラスなどの視界に情報を投影するシステムのことです。コンピューターグラフィックスやゲームにおける HUD も同様の目的で利用されます。ユーザーが 3D シーンという「主役」から視線を外さず、スコアや体力、FPS（フレームレート）などの補助的な情報をリアルタイムに確認できるよう、画面の最前面に重ねて表示するインターフェースを指します。

一方、DOM（Document Object Model）は、ブラウザが HTML 文書を JavaScript から操作できるようにしたツリー構造です。HTML の `<div>` や `<button>` といった要素は、ブラウザ内部で DOM ノードとして管理されており、JavaScript を通じて内容の書き換えやスタイルの変更、クリックイベントなどの接続が可能です。

webg の 3D シーンは WebGPU の canvas に描画されます。この canvas はピクセル単位で描画する領域であり、3D モデルや背景、そして「Canvas HUD」はこの描画結果の一部として表示されます。対して、説明文やボタン、エラーパネルのような UI は、canvas の外側または上側に HTML 要素として配置できます。この経路を、本章では「DOM オーバーレイ」と呼びます。

Canvas HUD と DOM オーバーレイは、どちらもユーザーには「画面に重なっている情報」に見えますが、内部的な仕組みは大きく異なります。Canvas HUD は WebGPU の描画フローに含まれるため、3D シーンと同じタイミングで極めて軽量に描画できます。その代わり、標準の文字描画は短い ASCII 表示を前提としており、日本語の組版や可変幅フォント、テキストの選択、スクロール、ボタン操作などの機能は持ちません。

対して DOM オーバーレイはブラウザの HTML 表示そのものであるため、日本語や長文、ボタン、スクロール、コピー可能なテキスト、入力フォームなどを自然に扱うことができます。

この違いを理解すれば、「短い状態表示は Canvas HUD」「読むための説明や操作 UI は DOM」という役割分担が明確になります。HUD とオーバーレイは、単に見え方の上下関係で分けるのではなく、描画フローと利用目的の違いに基づいて選択してください。

14.4 軽量な状態表示を担う Message

Message は、キャンバス上に簡潔な文字列を重ねるための HUD 管理クラスです。文字グリッドを直接操作するローレベル（低レイヤー）な Text クラスをラップし、より使いやすく設計されたハイレベル（高レイヤー）な API となっています。id、anchor（配置基準点）、複数行ブロックといった概念を備えており、通常のサンプルでは app.message.setLines() を使用するのが基本となります。

例えば、現在のスコアと経過時間を左上に表示する場合は、次のように記述します。

```
app.message.setLines("status", [
  `score=${score}`,
  `time=${elapsedSec.toFixed(1)} sec`
], {
  anchor: "top-left",
  x: 0,
  y: 0
});
```

操作ガイドを画面下側に配置する場合は、別の id を指定して別のブロックとして作成します。

```
app.message.setLines("guide", [
  "Drag: orbit camera",
  "Wheel: zoom",
  "R: reset camera"
], {
  anchor: "bottom-left",
  x: 0,
  y: -2,
  width: 36,
  wrap: true
});
```

Message は、短く直感的に理解でき、常時表示されていても視界を妨げない情報に適しています。スコアや残機、現在のモード、デバッグモードの ON/OFF、カメラ距離、選択中の座標などが典型的な例です。

一方で、以下のような情報は Message には適していません。

- 日本語による詳細な複数行説明
- 長文のエラーメッセージ
- スタックトレースや詳細な診断レポート
- ボタンや選択肢を伴うインタラクティブな UI
- スクロールして閲覧する必要がある文章

Message は標準のフォントアトラスと文字グリッドを前提とした軽量な HUD です。日本語や一般的な UTF-8 文章を適切に表示したい場合は、DOM オーバーレイへと移行しましょう。

14.5　読ませる情報を構成する OverlayPanel

OverlayPanel は、シーン上に重ねて表示する DOM パネルです。日本語の説明、ヘルプ、エラー表示、長文レポート、ブリーフィング、ボタン、選択肢などを一元的に扱います。

ここで重要なのは、OverlayPanel が「ヘルプ専用」や「エラー専用」のクラスではないという点です。OverlayPanel はあくまで汎用的な「表示基盤」であり、用途による違いはオプ

ション（option）によって表現します。

最小限のヘルプパネルは、次のように構成します。

```
app.showOverlayPanel({
  id: "help",
  title: "Help",
  lines: [
    "Drag: orbit camera",
    "Wheel: zoom",
    "H: hide help"
  ],
  anchor: "top-left",
  format: "plain",
  collapsible: true,
  closable: false
});
```

本文の指定には、単一の文字列を渡す text と、行配列を渡す lines があります。これらを同時に指定すると表示内容が曖昧になるため、OverlayPanel ではエラーとして処理されます。

```
app.showOverlayPanel({
  id: "note",
  title: "Note",
  text: "1 つの文字列として渡す場合はこちらを使用します",
  anchor: "bottom-left"
});
```

エラーログのように、空白や改行などの整形を保持したい場合は、format: "pre" を指定します。

```
app.showOverlayPanel({
  id: "load-error",
  title: "Load Error",
  text: error.message,
  anchor: "bottom-right",
  format: "pre",
  scrollY: true,
```

```
  closable: true,
  showCloseButton: true,
  maxHeight: "40vh"
});
```

このように、ヘルプもエラーもレポートも、すべて同一の OverlayPanel で表現可能です。開発者が習得すべきはクラス名ではなく、anchor、format、scrollY、collapsible、buttons、choices といった具体的なオプションの役割です。

14.6 配置基準とドックの回避

DOM オーバーレイを設計する際、最初に検討すべきは「配置場所」です。3D シーンの中央はユーザーが最も注視する領域であるため、パネルを中央に固定すると重要な視覚情報を遮ることになります。

OverlayPanel では、以下の anchor を提供しています。

anchor	主な用途
top-left	操作説明・補助ヘルプ
top-center	一時的な案内
top-right	ステータス・補助情報
middle-left	左側メニュー
middle-center	モーダル・ダイアログ
middle-right	右側インスペクター
bottom-left	ブリーフィング・ダイアログ
bottom-center	操作待ち・プロンプト
bottom-right	ログ・エラー・レポート

さらに offsetX と offsetY を指定することで、基準点からの余白を精密に調整できます。

```
app.showOverlayPanel({
  id: "briefing",
  title: "Mission",
  lines: ["Alpha が beacon を回収します"],
  anchor: "bottom-left",
  offsetX: 16,
  offsetY: 16,
```

```
  width: 360
});
```

また、右側に DebugDock を表示している場合、右寄せパネルがドックと重なる可能性があります。そのため、OverlayPanel には avoidDebugDock: true というオプションが用意されており、これを有効にすることでドックの幅を自動的に回避して配置されます。

14.7 ヘルプ機能とプリセットの活用

多くのサンプルでは、操作説明を「折りたたみ可能なパネル」として配置したいケースがあります。本ライブラリでは、ヘルプ機能を専用 API として提供するのではなく、OverlayPanel のオプションの組み合わせで表現します。

直接記述する場合は、以下のようになります。

```
app.showOverlayPanel({
  id: "runtime-help",
  title: "Help",
  lines: [
    "Drag: orbit",
    "Wheel: zoom",
    "R: reset"
  ],
  anchor: "top-left",
  collapsible: true,
  collapsed: false,
  closable: false,
  showCollapseButton: true,
  collapseLabelExpanded: "Hide Help",
  collapseLabelCollapsed: "Show Help"
});
```

この記述を簡略化したい場合は、OverlayPanelPresets.js のヘルパー関数を利用しましょう。

```
import { buildHelpPanelOptions } from "../../webg/OverlayPanelPresets.js";

app.showOverlayPanel(buildHelpPanelOptions({
  id: "help",
  lines: [
    "Drag: orbit",
    "Wheel: zoom",
    "R: reset"
  ],
  anchor: "top-left"
}));
```

ここで重要なのは、buildHelpPanelOptions() は新しい UI クラスを生成しているのではなく、単に適切なオプションオブジェクトを組み立てるヘルパーであるということです。実際に表示を担うのはあくまで OverlayPanel です。

14.8　エラー表示と診断レポートの使い分け

エラーメッセージや診断情報の全文は、HUD に表示すべきではありません。HUD は流動的な現在値の表示には適していますが、ユーザーが精読すべき文章には不向きだからです。

エラー表示は、次のように OverlayPanel で構成します。

```
app.showOverlayPanel({
  id: "start-error",
  title: "Start Error",
  text: error.message,
  anchor: "bottom-right",
  format: "pre",
  scrollY: true,
  closable: true,
  showCloseButton: true,
  maxHeight: "40vh"
});
```

同様に、OverlayPanelPresets.js に用意されたエラー用ヘルパーを利用することも可能です。

```
import { buildErrorPanelOptions } from "../../webg/OverlayPanelPresets.js";

app.showOverlayPanel(buildErrorPanelOptions(error, {
  id: "load-error",
  title: "Load Error"
}));
```

ただし、エラーを表示するだけでなく、開発者や解析ツールに共有するための情報として残したい場合は、Diagnostics にレポートとして記録することを優先してください。OverlayPanel は「閲覧するための場所」であり、Diagnostics は「状態を記録するための場所」であるという役割分担を意識しましょう。

14.9 ブリーフィングの進行制御

会話やチュートリアル、ミッションブリーフィングのような UI は、単なる表示だけでなく「進行制御」を伴います。「次へ進む」「選択肢を選ぶ」「分岐する」「閉じた後にゲームを再開する」といったルールが必要になるためです。

OverlayPanel はボタンや選択肢のアクションを返しますが、文章のキュー（待ち行列）や分岐ルール自体は保持しません。これは意図的な設計による分離です。表示部品に進行制御まで含めてしまうと、ゲーム固有のルールが基盤側に混入し、API の汎用性が損なわれるためです。

最小限のブリーフィングコントローラーは、サンプル側で次のように実装できます。

```
const briefingEntries = [
  {
    title: "Mission",
    lines: [
      "Alpha が beacon を回収します",
      "goal が開いたら脱出してください"
    ]
  },
  {
    title: "Controls",
    lines: [
      "Arrow keys: move",
```

```
      "Enter: next"
    ]
  }
];

let briefingIndex = 0;

function showBriefing() {
  const entry = briefingEntries[briefingIndex];
  app.showOverlayPanel({
    id: "briefing",
    title: entry.title,
    lines: entry.lines,
    anchor: "bottom-left",
    width: 420,
    buttons: [
      { id: "next", label: "Next", kind: "primary" },
      { id: "close", label: "Close", kind: "secondary" }
    ],
    onAction: ({ actionId }) => {
      if (actionId === "close") {
        app.hideOverlayPanel("briefing");
        return;
      }
      briefingIndex += 1;
      if (briefingIndex >= briefingEntries.length) {
        app.hideOverlayPanel("briefing");
      } else {
        showBriefing();
      }
    }
  });
}
```

この構成により、OverlayPanel は表示のみを担当し、文章の順序や分岐ロジックはアプリケーション側で完全に制御できます。

14.10 モーダル設定とシーンの一時停止

OverlayPanel には modal と pauseScene という重要なオプションがあります。

modal は、DOM 上の操作をパネルに集中させるための設定です。背後の UI を操作させたくない確認ダイアログや、開始前のブリーフィングに使用します。

pauseScene は、アプリケーション側に「このパネルが表示されている間はシーンの進行を停止すべきである」という状態を伝えるフラグです。なお、OverlayPanel 自体が自動的にゲームループを停止させるわけではありません。

```
app.showOverlayPanel({
  id: "confirm",
  title: "Start?",
  lines: ["この設定で開始しますか"],
  anchor: "middle-center",
  modal: true,
  pauseScene: true,
  buttons: [
    { id: "start", label: "Start", kind: "primary" },
    { id: "cancel", label: "Cancel", kind: "secondary" }
  ],
  onAction: ({ actionId }) => {
    if (actionId === "start") {
      app.hideOverlayPanel("confirm");
      startGame();
    }
  }
});
```

更新ループ側では、必要に応じてこの状態を参照して処理をスキップします。

```
app.start({
  onUpdate: ({ deltaSec }) => {
    const confirm = app.getOverlayPanel("confirm");
    const state = confirm?.getState?.();
    if (state?.visible && state.pauseScene) {
      return false;
```

```
    }

    updateGame(deltaSec);
    return false;
  }
});
```

この設計により、UI 部品が勝手にゲーム時間を止めることを避け、シーンを停止させるかどうかの判断をアプリケーション側の設計に委ねることができます。

14.11 診断情報の記録と共有フロー

3D アプリケーションでは、画面上の表示だけでは把握できない内部状態が多く存在します。カメラの正確な向き、読み込まれた形状の総数、現在のシェーダー設定、入力の反応状況などは、単に HUD に表示させるだけでは後からの共有や再現が困難です。

そのため、webg では Diagnostics を「正本」として扱います。まずレポートを作成し、そのレポートを必要に応じて DebugDock、コンソール、コピー用テキスト、JSON、OverlayPanel などへ出力するフローを推奨しています。

DebugDock は PC 向けの開発補助表示であり、右側にコントロール、診断レポート、プローブ状態などを集約することで、キャンバス HUD を情報で埋め尽くすことなく詳細な状態を監視できます。

情報を整理すると、配置は以下のようになります。

- **利用者が瞬時に把握すべき現在値** → Message
- **利用者が精読すべき説明文** → OverlayPanel
- **開発者や解析ツールに共有する状態** → Diagnostics
- **開発中に常時監視したい詳細情報** → DebugDock

この分離は、特に AI を活用したデバッグにおいて重要です。画面上の文字を増やすのではなく、再現可能なレポートとして状態を渡せるようにしておくことで、問題の切り分けが飛躍的に容易になります。

14.12 入力インターフェースとしての Touch

Touch は DOM にボタンを表示しますが、その役割は文字表示ではありません。Touch は、キーボードの代わりに arrowleft、space、enter、r などのキー状態やアクションを発生させる「入力 API」です。

したがって、「ボタンが必要だから」という理由で全てを Touch に寄せるのではなく、役割で明確に使い分けてください。

- **タッチ操作で移動、決定、リセットを入力したい** → Touch
- **シーン上に説明や選択肢を表示したい** → OverlayPanel
- **キャンバス上で一時的に開くコマンドや軽い設定を置きたい** → CommandPalette
- **現在値を簡潔に表示したい** → Message

この区別を明確にすることで、入力処理と表示処理が混在することを防げます。

14.13 UI 実装時の判断フロー

新しいサンプルを構築する際は、以下の手順で検討すると整理しやすくなります。

1. **情報の性質を判断する**: 短い現在値か、読ませる文章か。
2. **短い ASCII の現在値なら**: Message に配置する。
3. **日本語、長文、ボタン、選択肢が必要なら**: OverlayPanel に配置する。
4. **進行制御（分岐や順序）が必要なら**: サンプル側にコントローラーを実装する。
5. **キャンバス上で低頻度の操作や軽い設定を出したいなら**: CommandPalette に配置する。
6. **調査や共有が必要な情報なら**: Diagnostics にレポートとして記録する。
7. **タッチ操作が必要なら**: Touch を入力 API として追加する。

例えば、ローダー系のサンプルでは、「読み込み中の短い状態 → Message」、「読み込み失敗の全文 → OverlayPanel」、「読み込み対象や統計値 → Diagnostics」と分けることで、非常に見通しの良い UI になります。

14.14 まとめ

本章では、webg における文字表示の選択基準を整理しました。

- 短い ASCII HUD → `Message`
- 日本語、長文、ヘルプ、エラー、ボタン付き説明 → `OverlayPanel`
- 定型オプションの簡略化 → `OverlayPanelPresets`
- キャンバス上の一時的なコマンド、toggle、stepper、select → `CommandPalette`
- 会話やブリーフィングの進行 → サンプル側コントローラー
- 調査記録 → `Diagnostics` / `DebugDock`
- タッチ操作 → `Touch`

この分類により、UI 表示を「どのクラスを使うか」という視点ではなく、「どの情報をどの表示面に配置するか」という設計視点で考えることができます。次章では、この分担が内部的にどのように実装されているかを、キャンバス HUD と DOM オーバーレイの構造から詳しく見ていきましょう。

第 15 章

HUD とオーバーレイの設計

前章では、webg における文字表示の使い分けについて、利用者視点から整理しました。簡潔な状態表示は Message、精読させる文章は OverlayPanel、調査記録は Diagnostics / DebugDock、そしてタッチ入力は Touch という役割分担です。

本章では、この分担が内部的にどのような構造で実現されているかを解説します。単なる API のリファレンスではなく、なぜ Message と OverlayPanel を分ける必要があるのか、WebgApp がどこまで責任を持つべきか、そして OverlayPanel のオプションがどのように実際の表示構造に反映されるのかを確認していきます。

具体的には、まずキャンバス HUD と DOM オーバーレイの技術的な差異を整理し、Text から Message へ至る描画フローを確認します。続いて、CommandPalette の構築方法や状態管理、ページ構成について詳しく見ていきましょう。さらに、OverlayPanel の DOM 構造やオプションの反映仕組み、WebgApp によるオーバーレイ管理の範囲、そしてテーマや埋め込みレイアウトへの対応について解説します。最後に、実装時の確認手順とユニットテストの活用方法についてまとめます。

15.1 表示面の技術的差異と設計思想

webg の UI 表示は、描画される「面」の違いから理解すると明確になります。

キャンバス HUD (@<tt>{Text}, @<tt>{Message}) - **実体**: WebGPU で描画する文字クアッド（四角形ポリゴン）- **特性**: 軽量な更新、短い ASCII 文字、シーンとの一体的な描画が可能- **制約**: 日本語、長文、ボタン、スクロールには向かない

DOM オーバーレイ (@<tt>{OverlayPanel}, @<tt>{CommandPalette}, @<tt>{DebugDock}, @<tt>{Touch}) - **実体**: HTML 要素（DOM）- **特性**: UTF-8、可変幅フォント、ボタン、スクロール、フォーカス管理が可能- **制約**: 3D 深度との直接的な合成、過剰な DOM 更新による負荷に注意が必要

キャンバス HUD は 3D シーンと同じ描画パイプラインに乗るため、極めて軽量に動作します。毎フレーム変動するスコアやステータス表示に最適です。一方で、表示できる文字はフォントアトラスの範囲に依存し、標準では ASCII を前提としているため、日本語の表示はできません。また、長文の扱いにも向きません。

対して DOM オーバーレイは、ブラウザが持つ高度なテキストレイアウト機能をそのまま活用できます。日本語、可変幅フォント、ボタン、スクロール、アクセシビリティなどをブラウザに委ねることが可能です。ただし、これは 3D シーンの一部ではなく、キャンバスの上に重なる別レイヤーであるため、配置や重なり順を適切に管理しなければ、シーンの重要な部分を遮ることになります。

この技術的な特性の違いが、`Message` と `OverlayPanel` を明確に分ける設計思想の根拠となっています。なお、`CommandPalette` も DOM オーバーレイですが、`OverlayPanel` とは用途が異なります。`OverlayPanel` はヘルプやエラーレポートのように、ユーザーが情報を読むためのパネルです。対して `CommandPalette` は、必要な時だけ開き、低頻度のコマンドや設定値を素早く変更するための小さな操作面として設計されています。

15.2 CommandPalette の構築と状態管理

全画面キャンバスを使う編集アプリやビューワでは、常時表示のメニューバーが操作領域を狭めてしまうことがあります。そのような場合に有効なのが、ダブルクリックやキー操作で一時的に開くパレットです。`CommandPalette` は DOM の生成と入力の検出を担当しますが、シーンやエフェクトの状態自体は保持しません。状態はアプリケーション側に置き、パレットは現在値を読み取って操作結果を通知する「入口」として機能します。

生成とキャンバスへの接続

最小構成では、`document`、DOM の配置先となる `container`、操作対象の `viewport`、そして `commands` をコンストラクタへ渡します。コンストラクタは DOM と既定の CSS を準備しますが、この段階ではキャンバスの入力は奪いません。`attachToCanvas()` を呼ぶことで、ダブルクリック、ダブルタップ、指定キーによる開閉イベントが登録されます。

```
import CommandPalette, {
  getDefaultCommandPaletteCss
} from "../../webg/CommandPalette.js";

const canvas = document.getElementById("canvas");
const state = {
  shadow: true,
  radius: 18,
  brush: "Draw"
};

const palette = new CommandPalette({
  document,
  container: document.body,
  viewport: canvas,
  title: "Tools",
  closeOnCommand: false,
  onCommand: (id) => {
    if (id === "reset-camera") {
      resetCamera();
    }
  },
  onChange: (id, value) => {
    if (id === "shadow") state.shadow = value;
    if (id === "radius") state.radius = value;
    if (id === "brush") state.brush = value;
  },
  commands: [
    { id: "reset-camera", label: "Reset", detail: "camera" },
    { type: "toggle", id: "shadow", label: "Shadow", detail: "toggle",
      value: () => state.shadow },
    { type: "stepper", id: "radius", label: "Radius",
      value: () => state.radius, min: 8, max: 64, step: 2, input: true },
    {
      type: "select",
      id: "brush",
      label: "Brush",
      value: () => state.brush,
      options: [
        { value: "Draw", label: "Draw" },
        { value: "Blur", label: "Blur" },
        { value: "Grab", label: "Grab" }
```

```
      ]
    }
  ]
});

palette.attachToCanvas(canvas, {
  key: "/"
});
```

この例では、ダブル操作で指定位置の近くへ開き、/ キーで開く場合はビューポートの中央へ配置されます。開いた後はタイトル行をドラッグして、パレットを見やすい位置へ移動できます。ドラッグハンドルはタイトル行だけなので、button、toggle、stepper、select の操作とパレット移動が衝突しにくい構成になっています。アプリ側で位置移動を禁止したい場合は、コンストラクタに `draggable: false` を指定します。

起動方法を個別に制限する場合は、`doubleClick: false`、`doubleTap: false`、または `key: null` を指定します。また、`open()`、`close()`、`toggle()` メソッドを使って手動で制御することも可能です。

操作部品の選定と状態管理

`commands` の各要素では、まず操作を識別する `id` を決めます。`label` はボタンや行の主要な表示、`detail` はその下に表示する短い補足です。その上で、利用者に何をさせたいかに応じて `type` を選択します。

- **ボタン（@<tt>{button}）**：カメラのリセットやファイルの保存など、押した時点で 1 回の処理を実行する操作に適しています。ユーザーが押すと、パレット全体の `onCommand` と、定義している場合は個別の `onSelect` が呼ばれます。
- **切り替え（@<tt>{toggle}）**：グリッドの表示 ON/OFF などの boolean 状態を反転させる操作に使用します。`value` には現在の状態を返す関数を渡し、操作後は反転した値が `onChange` へ通知されます。
- **数値調整（@<tt>{stepper}）**：ブラシの半径やエフェクトの強度など、段階的に数値を増減させる操作に適しています。`min` / `max` に許容範囲、`step` に増減量を指定します。`input: true` を追加すると、数値入力欄が表示され、キーボードによる直接入力も可能になります。

- **選択（@<tt>{select}）**：ブラシの種類や表示モードなど、あらかじめ決めた候補から 1 つを切り替える操作に使用します。value に現在値を返す関数を渡し、options に { value, label } の配列を指定します。ボタンを押すたびに次の候補へ進み、選ばれた値が onChange へ通知されます。

toggle、stepper、select の value には通常、アプリケーションが持つ現在値を返す関数を指定します。操作時は共通の onChange または個別の onChange で状態を更新します。

また、複数の button から 1 つのモードを選ぶ場合は、modeSwitch: true と getCommandState() を組み合わせます。modeSwitch で色の種類を指定し、getCommandState() でどの button を active にするかを決定します。

```
const state = { mode: "object" };

const palette = new CommandPalette({
  document,
  container: document.body,
  viewport: canvas,
  closeOnCommand: false,
  getCommandState: (id) => ({
    active: id === `mode-${state.mode}`
  }),
  onCommand: (id) => {
    if (id === "mode-object") state.mode = "object";
    if (id === "mode-edit") state.mode = "edit";
  },
  commands: [
    { id: "mode-object", label: "Obj", detail: "mode", modeSwitch: true },
    { id: "mode-edit", label: "Edit", detail: "mode", modeSwitch: true }
  ]
});
```

closeOnCommand の既定値は true です。単発コマンドであれば操作後に閉じるのが自然ですが、複数の設定を続けて変更させるパレットでは false を指定します。

ページ構成とスタイルのカスタマイズ

pageRows は 1 ページの高さを行数で指定します。button と toggle は 1 セル、stepper と

select は 1 行全体を使用します。pageRowsByPage を指定すると、ページ番号ごとに行数を上書きできます。ページは表示枚数に応じて自動的に作成されますが、切り替えは自動では行われません。利用者が次ページへ進めるよう、id: "palette-next" の button を各ページのコマンド定義に含める必要があります。

```
const palette = new CommandPalette({
  document,
  container: document.body,
  viewport: canvas,
  pageRows: 2,
  pageRowsByPage: [2, 1],
  commands: [
    { id: "select", label: "Sel", detail: "tool" },
    { id: "move", label: "Move", detail: "tool" },
    { id: "rotate", label: "Rot", detail: "tool" },
    { id: "scale", label: "Scale", detail: "tool" },
    { id: "brush", label: "Brush", detail: "tool" },
    { id: "erase", label: "Erase", detail: "tool" },
    { id: "snap", label: "Snap", detail: "toggle" },
    { id: "palette-next", label: "Next", detail: "page", pageSwitch: true },

    { id: "view-front", label: "Front", detail: "view" },
    { id: "view-top", label: "Top", detail: "view" },
    { id: "view-side", label: "Side", detail: "view" },
    { id: "palette-next", label: "Next", detail: "page", pageSwitch: true }
  ]
});
```

palette-next は nextPage() を呼ぶ特別な ID で、最終ページの次は先頭に戻ります。pageSwitch: true はページ切り替え用ボタンの配色を適用する指定です。

CommandPalette は既定のスタイルを自動的に注入しますが、setStyle() で CSS 全体を差し替えたり、setTheme() で色を変更したりできます。既定のレイアウトを維持しつつ一部だけ変更したい場合は、getDefaultCommandPaletteCss() の後ろに独自の規則を連結してください。

```
const compactPaletteCss = `${getDefaultCommandPaletteCss()}
.command-palette {
  width: 300px;
```

```
}

.palette-button,
.palette-control-button,
.palette-select-button {
  border-radius: 6px;
}

.palette-button {
  height: 42px;
}
`;

palette.setStyle(compactPaletteCss);
```

パレットが不要になった場合は、イベントの解除と DOM の削除をまとめて行う destroy() を呼び出して適切に後始末を行ってください。

```
window.addEventListener("pagehide", () => {
  palette.destroy();
}, { once: true });
```

15.3　Text と Message の階層構造と描画

Text は、文字グリッドを GPU 上に描画するためのローレベル（低レイヤー）なクラスです。文字位置、文字コード、スケール、色、フォントテクスチャなどを管理し、最終的に文字ごとのクアッド（四角形ポリゴン）として画面に描画します。

Message は、この Text クラスをラップして HUD として使いやすくしたハイレベル（高レイヤー）な管理層です。文字列を id 付きのブロックとして登録し、anchor や width を指定することで容易に配置できます。

```
app.message.setLines("status", [
  "mode=orbit",
  "debug=off"
```

```
], {
  anchor: "top-left",
  x: 0,
  y: 0
});
```

Message は setLine()（単一行）、setLines()（複数行）、setBlock()（明示的なブロック管理）というインターフェースを提供しています。最大の利点は、表示内容を id で更新できる点にあります。毎フレームすべてを再構築するのではなく、特定の id を持つブロックだけを更新することで、スコアやガイドなどの異なる情報を効率的に管理できます。

HUD の描画タイミング

WebgApp を使用する場合、通常は app.start() の描画ループ内でシーンの描画と HUD の描画がまとめて処理されます。アプリケーション側は onUpdate ハンドラ内で app.message.setLines() を呼び出し、最新の状態を登録します。

```
app.start({
  onUpdate: ({ deltaSec }) => {
    elapsedSec += deltaSec;

    app.message.setLines("status", [
      `time=${elapsedSec.toFixed(1)}`,
      `camera=${orbit.orbit.distance.toFixed(1)}`
    ], {
      anchor: "top-left",
      x: 0,
      y: 0
    });

    return false;
  }
});
```

この時点では内部的なデータとして登録しているだけであり、実際の描画は WebgApp 内部の drawMessages() メソッドを通じて、シーン描画後の最終工程として重ねられます。「HUD

はキャンバスの一部として描画される」という特性を理解しておくことで、DOM オーバーレイとの使い分けがより明確になります。

15.4 OverlayPanel の構造と動作

OverlayPanel は DOM 要素で構成されており、概念的には以下のような階層構造を持っています。

```
root（オーバーレイ全体の基準）
  backdrop（背景遮蔽層）
  shell（配置制御層：anchor に従って配置）
    panel（パネル本体）
      header（ヘッダー：タイトル、閉じる/畳むボタン）
      body（本文：text または lines）
      choices（選択肢領域）
      buttons（操作ボタン領域）
```

通常は WebgApp の管理メソッドを介して操作します。同一の id で showOverlayPanel() を呼び出すと既存のパネルが更新され、一部のみを更新したい場合は updateOverlayPanel() を使用します。

本文の指定とフォーマット

本文は単一文字列の text または文字列配列の lines で指定します。これらを同時に指定することはできません。これは、更新時の意図しないデータの持ち越しを防ぎ、コードの可読性と安全性を高めるための設計です。表示形式は format オプションで切り替えます。

- format: "plain": 一般的な説明文向け。改行を保持しつつ適切に折り返します。
- format: "pre": ログやエラーメッセージなど、空白や改行の書式を厳密に保持したい文章に使用します。

長文を扱う場合は scrollY: true と maxHeight を組み合わせることで、本文領域のみをスクロールさせ、画面全体を覆い尽くさないように制御できます。

配置制御とレイアウトモード

OverlayPanel は anchor（基準点）に基づいて配置されます。アンカーは 9 方向から選択可能です。また、配置モードとして positioningMode が存在します。通常のフルスクリーンアプリでは fixed が使用されますが、教材ページなどにキャンバスを埋め込む layoutMode : "embedded" では、オーバーレイコンテナをキャンバスホストに関連付けるため absolute が使用されます。WebgApp を介して呼び出すことで、現在のレイアウト設定に応じた最適なモードが自動的に適用されます。

パネルの操作性とインタラクション

ヘルプパネルのように、完全に消去せず本文だけを畳んで最小限のボタンとして残したい場合は collapsible オプションを使用します。一方で、閲覧を終了してパネルを完全に破棄したい場合は closable と showCloseButton を使用します。

また、パネル下部には操作ボタン (buttons) や選択肢 (choices) を配置できます。これらが押下されると、onAction ハンドラに actionId が返されます。

```
app.showOverlayPanel({
  id: "choice",
  title: "Route",
  lines: ["どちらへ進みますか"],
  choices: [
    { id: "left", label: "Left" },
    { id: "right", label: "Right" }
  ],
  buttons: [
    { id: "cancel", label: "Cancel", kind: "secondary" }
  ],
  onAction: ({ panelId, actionId }) => {
    console.log(panelId, actionId);
  }
});
```

buttons はパネル全体に対する操作（保存・キャンセルなど）、choices は本文に対する選択肢（ルート選択など）という意味づけで使い分けるのが適切です。実装上はいずれも actio

nId を返すため、アプリケーション側では同一の入口で処理可能です。

モーダル設定とシーン停止の連携

modal: true を設定すると、背後の DOM 操作を受け付けず、ユーザーの意識をパネルに集中させることができます。また、pauseScene: true は、アプリケーション側に「このパネルが表示されている間はシーンの進行を停止すべきである」という状態を伝えるフラグです。重要な点として、OverlayPanel 自体が自動的にゲームループを停止させることはありません。

```
app.showOverlayPanel({
  id: "pause-menu",
  title: "Paused",
  lines: ["Resume or restart?"],
  modal: true,
  pauseScene: true,
  anchor: "middle-center",
  buttons: [
    { id: "resume", label: "Resume", kind: "primary" },
    { id: "restart", label: "Restart", kind: "secondary" }
  ],
  onAction: ({ actionId }) => {
    if (actionId === "resume") {
      app.hideOverlayPanel("pause-menu");
    }
  }
});
```

更新ループ側では、以下のように状態を参照して処理を制御します。

```
const pausePanel = app.getOverlayPanel("pause-menu");
const pauseState = pausePanel?.getState?.();
if (pauseState?.visible && pauseState.pauseScene) {
  return false; // 更新をスキップ
}
```

この分離により、UI 部品が勝手にゲーム時間を止めるという副作用を避け、停止のタイミ

ングをアプリケーション側の設計判断に委ねることができます。

15.5 WebgApp による統合管理

WebgApp は、OverlayPanel のインスタンス管理と配置制御を担います。

```
app.showOverlayPanel(options);
app.updateOverlayPanel(id, patch);
app.hideOverlayPanel(id);
app.removeOverlayPanel(id);
app.clearOverlayPanels();
app.getOverlayPanel(id);
app.hasOverlayPanel(id);
app.listOverlayPanels();
```

WebgApp はあえて「ヘルプ専用」などの個別 API を持たず、汎用的な管理機能に特化しています。これにより、個別のアプリケーションが持つ多様な進行 UI を柔軟に実装することが可能です。定型的なオプションが必要な場合は、OverlayPanelPresets.js を活用してください。

テーマ適用とデバッグドックの回避

DOM オーバーレイの視覚スタイルは WebgUiTheme で管理されます。WebgApp.setUiTheme() を呼び出すことで、DebugDock およびすべての OverlayPanel にテーマが反映されます。

また、右側に固定表示される DebugDock とパネルが重ならないよう、avoidDebugDock オプションが用意されています。これを有効にすると、現在のドックオフセットを自動的に取得し、右寄せパネルを適切に内側へ寄せます。

```
app.showOverlayPanel({
  id: "runtime-log",
  title: "Runtime Log",
  text: logText,
  format: "pre",
  anchor: "bottom-right",
```

```
  avoidDebugDock: true
});
```

DebugDock はあくまで開発者向けの調査領域です。一般利用者が閲覧する説明は OverlayPanel、開発中の内部状態確認は DebugDock と明確に使い分けてください。

埋め込みレイアウトへの対応

WebgApp は、フルスクリーンアプリだけでなく、教材ページ等にキャンバスを埋め込む layoutMode: "embedded" にも対応しています。埋め込みモードでは、キャンバスが HTML 文書の中間に配置されるため、ページ全体をスクロールすると、ビューポート固定のオーバーレイはキャンバスから離れて見えてしまいます。

これを解決するため、WebgApp は埋め込みモード時にオーバーレイコンテナをキャンバスホストに同期させます。OverlayPanel はこのコンテナを基準に absolute 配置されるため、ページをスクロールしても常にキャンバスと共に移動します。

この挙動は unittest/embedded で確認でき、ホスト、キャンバス、各種パネル、タッチコントロールの表示領域（矩形）に差異がないかを毎フレーム検証しています。

15.6 実装例：複合的な UI 構成

ここまでの要素を組み合わせた、典型的な実装例を示します。

```
import WebgApp from "../../webg/WebgApp.js";
import { buildHelpPanelOptions } from "../../webg/OverlayPanelPresets.js";

const app = new WebgApp({
  document,
  messageFontTexture: "../../webg/font512.png"
});

await app.init();

// 1. 操作説明（プリセット利用）
app.showOverlayPanel(buildHelpPanelOptions({
```

```
  id: "help",
  lines: [
    "Drag: orbit",
    "R: reset",
    "B: briefing"
  ],
  anchor: "top-left"
}));

// 2. 実行時レポート（整形済みテキスト、スクロールあり）
app.showOverlayPanel({
  id: "runtime-report",
  title: "Runtime Report",
  text: "ready",
  format: "pre",
  scrollY: true,
  anchor: "bottom-right",
  maxHeight: "32vh"
});

// 3. ブリーフィング（ボタン付き、進行制御は関数で管理）
function showBriefing() {
  app.showOverlayPanel({
    id: "briefing",
    title: "Briefing",
    lines: [
      "このパネルはボタン付き説明です",
      "進行制御はサンプル側で持ちます"
    ],
    anchor: "bottom-left",
    buttons: [
      { id: "close", label: "Close", kind: "primary" }
    ],
    onAction: ({ actionId }) => {
      if (actionId === "close") {
        app.hideOverlayPanel("briefing");
      }
    }
  });
}

app.attachInput({
```

```
  onKeyDown: (key, ev) => {
    if (ev.repeat) return;
    if (key === "b") {
      showBriefing();
    }
  }
});

app.start({
  onUpdate: ({ deltaSec }) => {
    // 4. 短い状態表示（キャンバス HUD）
    app.message.setLines("status", [
      "status: running",
      `dt=${deltaSec.toFixed(3)}`
    ], {
      anchor: "top-left",
      x: 0,
      y: 0
    });

    return false;
  }
});
```

情報の性質に応じて「Message（状態）」「OverlayPanel プリセット（ヘルプ）」「format: "pre"（レポート）」「カスタム関数（ブリーフィング）」と使い分けています。表示経路を分離することで、規模が拡大しても管理しやすい構成となります。

15.7 検証と実装チェックリスト

UI 系の変更を行った際は、以下のユニットテストとサンプルを確認することで意図した動作を検証できます。

- **OverlayPanel の基本動作**: unittest/overlay_panel でアンカー、pre 形式、ボタン、モーダル等の動作を確認。
- **埋め込みレイアウト**: unittest/embedded で埋め込み時の配置追従を確認。
- **テーマ適用**: unittest/theme で setUiTheme() による切り替えを確認。

- **Message の基本動作**: unittest/message で確認。
- **タッチ入力**: unittest/touch で Touch による入力代替動作を確認。
- **CommandPalette の詳細**: samples/com_palette で行数に応じたページ分割、Mode Switch の active 表示、数値の直接入力、ダブルクリックとダブルタップによる起動を確認。

特に unittest/overlay_panel は、本章で解説した UI 設計の核心部分を網羅しており、9 方向のアンカーやモーダル動作、アクション返却などを一画面で確認できるため、開発時の参照を推奨します。

DOM オーバーレイを実装する際は、以下のチェックリストを活用してください。

- **ID の一意性**: パネル ID やアクション ID が重複していないか。
- **指定の排他性**: text と lines を同時に指定していないか。
- **表示面の選択**: 日本語や長文を Message に入れていないか。
- **可読性の確保**: 長文の場合、scrollY と maxHeight を適切に指定しているか。
- **視認性の配慮**: シーン中央を遮らない適切な anchor を選択しているか。
- **ドック回避**: 右寄せパネルで avoidDebugDock を考慮しているか。
- **アクション ID の一意性**: ボタンや選択肢の id が重複していないか。
- **進行停止の必要性**: modal: true のパネルで、アプリ側が pauseScene を参照して更新を止める必要があるか。
- **責務の分離**: 会話やブリーフィングの進行ロジックを OverlayPanel 自体に組み込んでいないか。
- **記録の優先**: 開発者や解析ツールに渡す情報は Diagnostics レポートとして保存しているか。
- **状態の所有**: CommandPalette の value がアプリケーションの現在値を読み、callback がその状態を更新する構成になっているか。
- **操作の選択**: 単発操作、ON/OFF、数値変更、候補切り替えに、button、toggle、stepper、select を正しく使い分けているか。
- **ページ移動**: 複数ページが作られる場合、利用者が到達できる palette-next が各ページにあるか。
- **後始末**: 画面を破棄するときに CommandPalette.destroy() を呼び、canvas と document のイベントを解除しているか。

15.8 まとめ

本章では、webg における HUD とオーバーレイの内部構造について解説しました。

Message は Text クラスを基盤としたキャンバス HUD であり、軽量な ASCII 状態表示に適しています。一方、OverlayPanel は DOM ベースのオーバーレイであり、日本語、長文、ボタン、選択肢、スクロールなどを柔軟に扱います。CommandPalette も DOM オーバーレイですが、読ませる情報ではなく、必要な時だけ開くコマンドと軽い設定の操作面を担当します。

この分担により、WebgApp の汎用性を維持しつつ、開発者には「短い HUD は Message、読むパネルは OverlayPanel、一時的な操作は CommandPalette、記録は Diagnostics / DebugDock」という一貫した判断軸を提供しています。

次章では、UI の中でも特に入力に関わる Touch と InputController を扱い、画面上のボタンをどのようにキー入力へ接続し、アプリケーションを操作させるかについて詳しく見ていきましょう。

第 16 章

タッチ機能と入力

16.1　タッチ入力の設計コンセプト

webg におけるタッチ入力の設計は、単に画面上にボタンを配置することではなく、得られた入力をどのようにキー入力のステート（状態）やアクションへと流すかという「入力経路」の設計に主眼を置いています。本章では、webg/Touch.js と webg/InputController.js を中心に、直感的でストレスのない操作系を構築する方法を解説します。

特にモバイル端末やタッチデバイスのように、指先で操作するため細かな位置指定が難しい入力環境（精緻な操作が難しいポインター：粗いポインター/coarse pointer）では、押しやすさと反応の良さがユーザー体験に直結します。そのため、UI の見た目だけでなく、入力がシステムに伝達されるまでの経路を適切に設計することが不可欠です。

なお、UI 全体の使い分けや文字表示の指針については「UI 表示の設計」で述べており、入力が 3D 空間のどの地点に到達したかを判定する仕組みについては、次章の「衝突判定」で詳しく解説します。

16.2　入力インターフェースの役割分担

効率的な実装を行うためには、Touch、OverlayPanel、CommandPalette、そして InputController がどのような役割を担っているかを整理しておく必要があります。

Touch：ローレベル（低レイヤー）な入力インターフェース

Touch は、キャンバス外の DOM に仮想ボタンを生成するローレベル（低レイヤー）な機能を担います。その本質的な役割は、onPress、onRelease、onAction といったコールバックを通じて、物理的なタッチ操作をアプリケーション側のキー入力ステートやアクションへと仲介することです。つまり、仮想ボタンは「キーボード入力の代替手段」として機能します。

OverlayPanel：シーン操作のための UI

OverlayPanel もボタンや選択肢を配置できますが、その用途は説明パネル、メニュー、会話ウィンドウといった「シーン上の操作 UI」です。対して Touch は、ArrowLeft や R といったキーボード入力を代替する「固定ボタン群」を指します。このように、シーンに付随する UI は OverlayPanel、システム的な入力代替は Touch と使い分けることで、役割を明確に分離できます。

CommandPalette：一時的な操作 UI

CommandPalette は、キャンバスを中心に据えたアプリで活用する小さなコマンド UI です。OverlayPanel のように文章を読ませるパネルではなく、Touch のように常時表示される仮想キーでもありません。ユーザーが必要な時だけダブルクリック、ダブルタップ、またはキー操作で開き、設定値を素早く変更します。

CommandPalette.attachToCanvas() は、PC のダブルクリック、タッチデバイスのダブルタップ、指定キーによる開閉をまとめて登録します。この起動経路は Touch クラスや InputController.installTouchControls() の有無に依存しません。したがって、PC ブラウザでもモバイルと同じ一時 UI を使いたい場合は、固定タッチボタンとは別に CommandPalette を追加して運用することが可能です。

```
import CommandPalette from "../../webg/CommandPalette.js";

const canvas = document.getElementById("canvas");
const state = {
  grid: true
};
```

```
const palette = new CommandPalette({
  document,
  container: document.body,
  viewport: canvas,
  commands: [
    { id: "reset", label: "Reset", detail: "camera" },
    { type: "toggle", id: "grid", label: "Grid", detail: "toggle", value: () => state.grid }
  ],
  onCommand: (id) => {
    if (id === "reset") resetCamera();
  },
  onChange: (id, value) => {
    if (id === "grid") state.grid = value;
  }
});

palette.attachToCanvas(canvas, {
  key: "/"
});
```

InputController：ハイレベル（高レイヤー）な統合管理

InputController は、キーボード、タッチボタン、そしてジェスチャーといった多様な入力ソースを統合して管理するハイレベル（高レイヤー）な層です。

標準的な実装では、InputController.installTouchControls() を利用してこれらを一括管理します。Touch を直接制御することも可能ですが、InputController を介してホールド入力を keyState へ、アクション入力を pulseAction() へ流すことで、アプリケーション側は入力ソースが何かを意識せず、単一の判定ロジックで処理を行うことが可能になります。

ここで登場する pulseAction() は、リセットや決定、モード切り替えのような「1 回の操作で 1 回だけ反応してほしい処理」に適した一時的なアクション状態です。押しっぱなしによる連続実行を避けたい操作に使用します。

16.3 入力設計の基本原則

実用的な操作系を構築するために、以下の 3 点を設計原則とします。

1. ホールド入力とアクション入力の分離

「押している間だけ有効なホールド入力（移動、回転、視点操作など）」と、「1 回の操作で 1 回だけ反応するアクション入力（リセット、決定、ポーズなど）」を明確に分離します。この整理を行うことで、ボタン数が増加しても処理フローをシンプルに保つことができます。

2. キー名称の統一

キーボード入力では、生の `event.key` をそのまま扱うのではなく、`InputController` 内部の `normalizeKey()` によって正規化されたキー名を使用します。通常の英字キーは小文字化され、`Space` や `Escape` などの特殊キーは `space`、`escape` のような比較しやすい名称に変換されます。タッチボタン側でも同じ正規化済みキー名を使用することで、キーボード入力とタッチ入力を共通の判定ロジックで扱うことができます。

3. 入力経路の抽象化

アプリケーション側は「どの物理キーが押されたか」ではなく、「どのアクションが発生したか」を参照するように設計します。これにより、将来的な入力デバイスの変更や操作割り当ての変更に強い構成になります。

16.4 標準的な導入手順

キーボード入力の接続と正規化

まず `InputController.attach()` を使用してキーボード入力を有効にします。キー名は内部で `normalizeKey()` を通り、比較しやすい名称に正規化されます。そのため、比較時は生の `event.key` ではなく、正規化後のキー名を使用してください。

例えば Space は " " ではなく space、Esc は escape として処理されます。タッチボタンと共通の条件で扱いたい場合は、以下の名称を使用してください。

特殊キーの名称一覧

キー	比較に使用する名称
Space	space
Escape / Esc	escape
Enter	enter
Tab	tab
Backspace	backspace
Delete	delete
Shift	shift
Control / Ctrl	ctrl
Alt / Option	alt
Meta / Command / Cmd	meta
Home / End	home / end
PageUp / PageDown	pageup / pagedown
Arrow (L/R/U/D)	arrowleft / right / up / down
F1 〜 F12	f1 〜 f12
Numpad0 〜 9	numpad0 〜 numpad9

これらの修飾キー（modifier key）名は、InputController.has() での比較だけでなく、EyeRig などの視点制御における panModifierKey に指定する際にも使用されます。

Touch コントロールの組み込み

次に installTouchControls() を呼び出し、Touch 機能を組み込みます。ホールド入力用ボタンは press() / release() を通じてステートに反映され、アクション入力用ボタンは pulseAction() を通じて一時的なアクションステートとして反映されます。

```
import InputController from "./webg/InputController.js";

const input = new InputController(document);

input.attach({
  onKeyDown: (key) => {
    if (key === "escape") {
      pauseGame();
```

```
    }
  }
});

input.registerActionMap({
  reset: "r"
});

input.installTouchControls({
  touchDeviceOnly: false,
  autoSpread: true,
  groups: [
    {
      id: "move",
      buttons: [
        { key: "arrowleft", label: "←", kind: "hold", ariaLabel: "move left" },
        { key: "arrowright", label: "→", kind: "hold", ariaLabel: "move right" }
      ]
    },
    {
      id: "turn",
      buttons: [
        { key: "a", label: "A", kind: "hold", ariaLabel: "turn left" },
        { key: "d", label: "D", kind: "hold", ariaLabel: "turn right" }
      ]
    },
    {
      id: "action",
      buttons: [
        { key: "r", label: "R", kind: "action", ariaLabel: "reset" }
      ]
    }
  ]
});

function update() {
  if (input.has("arrowleft")) moveLeft();
  if (input.has("arrowright")) moveRight();
  if (input.has("a")) rotateLeft();
  if (input.has("d")) rotateRight();

  if (input.wasActionPressed("reset")) {
```

```
    resetPlayer();
  }
}
```

この構成の最大の利点は、キーボードとタッチ入力のどちらが操作されていても、アプリケーション側は同一のステートとアクションを参照するだけで済む点にあります。連続的な操作は input.has() で、単発操作は wasActionPressed() で判定します。

また、InputController.registerActionMap() を活用してキー名をアクション名に紐付けることで、デバイスを問わず共通のアクション体系で制御することが可能になります。

16.5　キャンバス上での空間操作とジェスチャー

仮想ボタンによる入力とは別に、モバイル環境でマウスの代わりにシーンを操作したい場合は、キャンバス上のポインターイベントを利用した「ジェスチャー操作」を実装します。

ここで重要なのは、仮想ボタンが「特定のコマンド（キー）の代替」であるのに対し、キャンバスジェスチャーは「空間的な操作（回転・ズーム・平行移動）」であるという点です。役割が根本的に異なるため、実装を分けて管理することが適切です。

空間操作の実装ステップ

1. canvas.style.touchAction = "none" を設定し、ブラウザ標準のスクロールやピンチ操作による干渉を防ぎます。
2. pointerdown / pointermove / pointerup / pointercancel イベントでポインターを追跡し、単指操作と多指操作を区別します。
3. 単指ドラッグでは、移動量（dx / dy）をカメラの回転（yaw / pitch）に反映させます。
4. 二本指ドラッグでは、中心位置の移動量をカメラターゲットの平行移動へ、指間距離の変化をズームへ変換します。
5. タップとドラッグを区別する場合、移動量のしきい値（threshold）を設け、それを超えなかった pointerup のみをタップ（選択処理）として判定します。

これらの連続的な空間操作はアプリケーションごとのカメラ制御に強く依存するため、個別に実装します。一方で、タップやフリックのような離散的なジェスチャーは、Touch.attach

Surface() を使って共通的に扱うことができます。

タッチジェスチャーとマウス操作の同期

webg の Touch は、スマートフォンのタッチ操作だけを特別扱いするための機能ではありません。タッチ、マウス、ペンをブラウザ標準の Pointer Events として同一の入口に集め、共通のジェスチャー情報としてアプリケーションへ渡せる点に真価があります。

従来の DOM 入力では touchstart や mousedown を個別に扱う実装になりがちですが、それではデバイス間で判定条件やタイミングの不一致が生じます。Touch.attachSurface() はこれらを共通のポインターイベントとして扱うため、タップ、ダブルタップ、長押し、フリックの判定ロジックを一つにまとめることができます。

これにより、スマートフォンでの短いタップは PC ではクリック、ダブルタップはダブルクリック、長押しはマウスボタンの押し続けとして同一のロジックで検証でき、「デバイスが何か」ではなく「どのジェスチャーが発生したか」に集中して開発を進められます。

効率的なデバッグ手法

モバイル向けの操作を実機だけで調整すると、開発サイクルが著しく低下します。PC ブラウザ上でスマートフォンと同じジェスチャー判定フローを通せるようにすることで、開発者ツールやブレークポイントを最大限に活用した高速な調整が可能になります。

そのため Touch では、端末判定とポインター種別判定を別々のオプションとして提供しています。

- touchDeviceOnly: 粗いポインター（coarse pointer）を持つ端末でのみ機能を有効にするかどうかを決定します。
- touchOnly: 登録後に pointerType === "touch" 以外を無視するかどうかを決定します。

PC ブラウザでスマートフォン向けジェスチャーを検証したい場合は、Touch の生成時と attachSurface() の両方で touchDeviceOnly: false を指定し、さらに attachSurface() 側で touchOnly: false を指定してください。

```
const touch = new Touch(document, {
  touchDeviceOnly: false
});

touch.attachSurface(canvas, {
  touchDeviceOnly: false,
  touchOnly: false,
  onGesture: (gesture) => {
    console.log(
      gesture.type,
      gesture.pointerType,
      gesture.dx,
      gesture.dy
    );
  }
});
```

この設定により、PC のマウス入力でも tap、doubletap、longpress、flick が発火し、実機で起こりうる問題を PC 上で迅速に再現・修正しやすくなります。

16.6 attachSurface() による高度なジェスチャー実装

スマートフォンの限られた画面領域では、仮想ボタンを多数並べると操作領域が不足します。特にモデラーや編集ツールのように機能分岐が多いアプリケーションでは、キャンバス上のジェスチャーをアクションへ変換する構成が非常に有効です。

Touch.attachSurface() は、指定した要素上の Pointer Events から、タップ、ダブルタップ、長押し、フリックを検出します。window 全体ではなく対象要素に限定してリスナーを張るため、ページ全体のスクロールや他の UI を巻き込みにくい設計となっています。

```
import Touch from "./webg/Touch.js";

const touch = new Touch(document, {
  touchDeviceOnly: false
});

touch.attachSurface(canvas, {
```

```
  touchDeviceOnly: false,
  touchOnly: false,
  minDistance: 50,
  longPressTime: 500,
  longPressMoveTolerance: 10,
  doubleTapTime: 320,
  onGesture: (gesture) => {
    if (gesture.type === "flick") {
      console.log(`flick ${gesture.direction}`);
    }
    if (gesture.type === "longpress") {
      console.log("open command palette");
    }
    if (gesture.type === "doubletap") {
      console.log("toggle mode");
    }
  }
});
```

実装上の注意点

- **オプション名の正確な指定**: longPressTime や doubleTapTime など、API で定義された名称を正確に使用してください。誤った名称を指定しても既定値で動作するため、意図したしきい値が反映されていないことに気づきにくい点に注意が必要です。
- **イベントの優先順位**: キャンバス上で通常の pointerdown / pointerup による選択処理も行う場合は、サーフェスジェスチャーのリスナーを先に登録してください。通常処理が先に実行されると、ダブルタップの判定待ち時間中に選択解除などが走り、ジェスチャーが正しく機能しない場合があります。
- **標準操作の抑止**: setTouchActionNone が既定で true になっているため、対象要素ではブラウザ標準のスクロールやピンチが抑止されます。ページ全体のスクロールを維持したい場合は、ジェスチャーを受ける領域を適切に限定してください。

ジェスチャー情報の詳細

onGesture に渡される主な情報は以下の通りです。

プロパティ	内容
type	tap / doubletap / longpress / flick
direction	flick 時のみ left / right / up / down
x, y	pointerup または検出時の client 座標
startX, startY	pointerdown 時の client 座標
dx, dy	開始位置からの移動量
distance	移動距離
elapsedMs	入力開始からの経過時間
pointerType	touch / mouse / pen などの pointer 種別

実践例：mmodeler における入力設計

samples/mmodeler は、モバイルプロフィールを主対象としながら、PC ブラウザ上でも同一の操作体系を検証できるように設計された好例です。このサンプルでは、単なるタップだけでなく、以下のような複雑な状態遷移を同一キャンバス上で実現しています。

- オブジェクト、頂点、面の選択
- ダブルタップによるコマンドパレットの表示
- 長押しによる Object / Edit モードの切り替え
- G/R/S/E キーによるトランスフォームプレビュー
- ダブルタップ直後のドラッグによるボックス選択

このようなアプリケーションでは、入力タイミングのわずかな差が挙動を大きく変えます。例えば、1 回目のタップを即座に選択として確定させると、ダブルタップの判定が完了する前に選択対象が変わってしまいます。

こうした繊細な調整を実機のみで行うのは困難であるため、mmodeler では PC のマウス入力も Touch.attachSurface() の共通フローへ通しています。これにより、開発者ツールで gesture.pointerType や elapsedMs を監視しながら、スマートフォン向け UI の判断条件を PC 上で迅速に追い込むことが可能になります。

モバイル向け編集アプリでの入力レイヤー設計

機能の多いアプリケーションでは、入力を以下の 4 つの層に分けて整理することを推奨します。

1. **常時表示ボタン**: 最重要かつ高頻度の操作に絞り、迷わず押せる配置にする。

2. **補助 UI**: 低頻度の操作は OverlayPanel や CommandPalette へ集約する。
3. **ジェスチャーショートカット**: フリックやダブルタップを、熟練者向けの効率的なショートカットとして提供する。
4. **文脈メニュー**: 長押しを、その地点での追加操作や詳細設定の入口にする。

重要なのは、ジェスチャーを唯一の操作経路にせず、必ずボタンやメニューからも同一の操作を実行できるようにすることです。例えば unittest/flick では、タップで頂点選択、ダブルタップでモード切り替え、長押しでコマンドパレット、フリック左右でツール切り替え、フリック上下で複製・削除を行う検証例を用意しています。

16.7　Touch クラスの直接利用と詳細仕様

機能の個別検証や、最小限の構成で動作を確認したい場合は、InputController を介さず Touch クラスを直接利用します。

```
import Touch from "./webg/Touch.js";

const keyState = new Set();

const press = (key) => {
  keyState.add(String(key).toLowerCase());
};

const release = (key) => {
  keyState.delete(String(key).toLowerCase());
};

const touch = new Touch(document, {
  touchDeviceOnly: false
});

touch.create({
  autoSpread: true,
  groups: [
    {
      id: "move",
      buttons: [
        { key: "arrowleft", label: "←", kind: "hold" },
        { key: "arrowright", label: "→", kind: "hold" }
```

```
      ]
    },
    {
      id: "action",
      buttons: [
        { key: "r", label: "R", kind: "action" }
      ]
    }
  ],
  onPress: ({ key }) => press(key),
  onRelease: ({ key }) => release(key),
  onAction: ({ key }) => {
    if (key === "r") {
      resetPlayer();
    }
  }
});
```

この形式では InputController の normalizeKey() やアクションマップが介在しないため、キー名の統一を手動で行う必要があります。実際のアプリケーションでは、InputController.installTouchControls() を使用することがより安全です。

また、アクションボタンのみを配置したい場合は createActionButtons() メソッドが利用可能です。これはすべてのボタンを kind: "action" として生成する簡易メソッドであり、単発コマンドを並べるメニューの実装に便利です。

レイアウトと表示仕様

Touch のルート要素は画面下部に固定されます。標準 CSS では left: 0、right: 0、bottom: 0 が設定され、デバイスの safe-area-inset も考慮されています。

- **ボタン密度（Density）**：ボタンの総数に応じて、視認性と押しやすさを維持するためにボタンサイズが自動的に調整されます（applyDensitySize() が担当）。
- **レイアウトモード**: 画面幅とボタン数に基づいて「単一行」「複数行」「展開配置」を自動的に切り替えます。autoSpread: true が設定されている場合は、利用可能な余白に応じて左右に広がった配置が選択されます。

主要なオプション設定

Touch インスタンスオプション

- touchDeviceOnly: 既定値は true です。粗いポインターを持つ端末でのみ表示させたい場合に利用します。PC 環境で動作確認を行いたい場合は false に設定してください。
- force: true を指定すると、デバイス判定に関わらず強制的に UI を表示します。
- className: ルート要素に独自のクラス名を付与し、CSS でレイアウトを微修正したい場合に有効です。
- onAnyPress: いずれかのボタンが押された瞬間に実行したい処理（初回入力の検知など）がある場合に利用します。

attachSurface() オプション

- touchDeviceOnly: 粗いポインターの端末でのみサーフェスジェスチャーを有効にします。
- touchOnly: true なら touch pointer のみを扱い、mouse / pen を無視します。
- minDistance: フリックと判定する最小距離です。
- longPressTime: 長押しと判定する時間です。
- longPressMoveTolerance: 長押し中に許容する移動距離です。
- tapMoveTolerance: タップとして許容する移動距離です。
- doubleTapTime: ダブルタップと判定する最大間隔です。
- doubleTapDistance: ダブルタップと判定する 2 回のタップ間の最大距離です。
- preventDefault: 対象要素上のポインターイベントで preventDefault() を呼び出します。
- setTouchActionNone: 対象要素の touch-action を none にします。
- onGesture: すべてのジェスチャーを受け取ります。
- onFlick / onLongPress / onDoubleTap / onTap: 種類別コールバックです。

preventDefault や setTouchActionNone は非常に強力ですが、適用範囲を canvas や専用の操作領域に限定してください。通常のフォーム部品やスクロール領域に適用すると、ブラウザ標準の操作性を損なう恐れがあります。

また、detachSurface() または destroy() を呼ぶことで、サーフェスジェスチャーのリスナーと長押しタイマーは適切に解除されます。

16.8 実装時の指針と注意点

実装の際は、以下の使い分けを徹底してください。

- **共通ロジックで扱いたい** → InputController を使用。
- **ボタンの最小挙動を個別に検証したい** → Touch を直接使用。
- **シーン上の操作パネルやメニューを構築したい** → OverlayPanel を使用。
- **キャンバス上で一時的なコマンドや軽い設定を操作したい** → CommandPalette を使用。
- **会話やチュートリアルを提示したい** → OverlayPanel とサンプル側コントローラーを組み合わせる。
- **ボタンの意味を画面上で説明したい** → Message や WebgApp のガイドヘルパーを併用。

よくあるミスと対策

- **キー名の比較**: 生の event.key をそのまま比較せず、必ず InputController が正規化したキー名で比較してください。Space のように、単純な小文字化だけでは期待する名称にならないキーが存在します。
- **入力種類の混同**: 移動などの連続的な操作は「ホールド入力」、リセットや決定などの単発操作は「アクション入力」に明確に分けてください。
- **アクションの直接実行への依存**: タッチボタンの onAction だけで処理を完結させると、キーボード入力との統合が崩れやすくなります。共通化したい操作は、registerActionMap()、pulseAction()、wasActionPressed() の経路に乗せて扱うことが肝要です。
- **ジェスチャーの排他利用**: 重要な操作をフリックや長押しだけにせず、必ずボタンやメニューからも実行できるようにしてください。
- **入力範囲の不適切な設定**: ジェスチャーリスナーを window 全体へ張らず、canvas や専用 surface のような明確な要素に限定してください。
- **UI の目的外利用**: 入力代替は Touch、読むパネルは OverlayPanel、一時的な command と軽い設定は CommandPalette と使い分けてください。

- **ユーザーへのガイド**: タッチボタンを配置する際は、その機能が何であるかを Message 等で提示し、ユーザーが迷わないよう配慮してください。

第 17 章

衝突判定

ユーザーからの入力を受け取れるようになると、次に必要になるのは「その入力が 3D 空間の何に当たったのか」、あるいは「シーン内の何と何が重なったのか」を判断することです。

クリックしてオブジェクトを選択したいのか、オブジェクト同士の接触を調べたいのかによって、使用する API は異なります。webg では、これらの役割を Space.raycast()、Space.checkCollisions()、および Space.checkCollisionsDetailed() が分担して担っています。

ここで重要なのは、これらがすべて「判定」に関する API であっても、判定の起点が異なるという点です。raycast() は 1 本のレイ（光線）を飛ばして何に当たるかを調べる API であり、checkCollisions 関連の API はシーン内のシェイプ同士が重なっているかを調べるものです。

本章では、単に API 名を覚えることではなく、「どのような判定を行いたいときに、どの API を選択すべきか」という基準を整理して解説します。

ここで扱うのは主に「当たっているかを調べる」ための問い合わせです。接触した後にどう押し戻すか、反発や摩擦で速度をどう変えるか、回転付きの box をどう扱うかといった物理応答は、第 26 章「物理エンジン」で扱います。

なお、現在の webg の実装では、raycast() と checkCollisions() はまず AABB（軸並行境界ボックス）ベースで動作します。そのため、非常に細い棒や斜めの円柱のような形状の場合、見た目よりも少し大きめの範囲でヒット判定が出る場合があります。より厳密な判定が必要な場合には、広域判定の後に三角形レベルでの絞り込みを行う checkCollisionsDetailed() を使用します。

また、どの API を使用する場合でも、filter を適切に設定することで、カメラ自身や補助用のシェイプを判定候補から除外でき、結果をより正確に得ることができます。

17.1 クリック位置から Space.raycast() へつなぐ

Space.raycast() は、レイの起点（origin）と方向（direction）を受け取り、ヒットしたシェイプを返します。この API はマウスやタップの画面座標を直接受け取るものではありません。そのため、まずは画面座標を正規化デバイス座標（NDC）に変換し、投影行列とビュー行列を用いてワールド空間のレイを生成する工程が必要です。unittest/raycast にはこの一連の流れが実装されており、実装例として非常に参考になります。

```
import Matrix from "./webg/Matrix.js";

// 画面上の座標（CSS ピクセル）を正規化デバイス座標（NDC）に変換する
const cssToNdc = (canvas, clientX, clientY) => {
  const rect = canvas.getBoundingClientRect();
  const x = ((clientX - rect.left) / rect.width) * 2.0 - 1.0;
  const y = 1.0 - ((clientY - rect.top) / rect.height) * 2.0;
  return [x, y];
};

// マウス位置から 3D 空間へのレイを生成する
const makeRayFromMouse = (canvas, clientX, clientY, eyeNode, proj, view) => {
  const [nx, ny] = cssToNdc(canvas, clientX, clientY);
  const invVp = proj.clone();
  invVp.mul(view);
  invVp.inverse_strict();

  const far = invVp.mulVector([nx, ny, 1.0]);
  const eyePos = eyeNode.getWorldPosition();
  const dir = [
    far[0] - eyePos[0],
    far[1] - eyePos[1],
    far[2] - eyePos[2]
  ];
  return { origin: eyePos, dir };
};

canvas.addEventListener("click", (ev) => {
  // カメラのワールド行列を最新の状態に更新する
```

```
  app.eye.setWorldMatrix();
  const view = new Matrix();
  view.makeView(app.eye.worldMatrix);

  const ray = makeRayFromMouse(
    app.screen.canvas,
    ev.clientX,
    ev.clientY,
    app.eye,
    app.projectionMatrix,
    view
  );

  // レイキャストを実行し、最初に当たったオブジェクトを取得する
  const hit = app.space.raycast(ray.origin, ray.dir, {
    firstHit: true,
    // カメラ自身や非表示のシェイプを判定から除外するフィルタ
    filter: ({ node, shape }) => node !== app.eye && !shape.isHidden
  });

  if (!hit) {
    return;
  }
  console.log(hit.node.name, hit.point, hit.t, hit.boundsOnly);
});
```

この実装で重要なポイントは、raycast() を呼び出す前に app.eye.setWorldMatrix() と view.makeView(app.eye.worldMatrix) を実行している点です。レイを生成する基準となる視点行列が古いままでは、画面上のクリック位置と 3D 空間での方向がずれてしまいます。また、filter を活用することで、不要なオブジェクトを候補から排除し、効率的な判定が可能になります。

raycast() は内部的に方向ベクトルを正規化し、各シェイプのローカル境界ボックスをワールド AABB へ変換して判定を行います。戻り値の point はレイ上のワールド座標であり、boundsOnly: true は「メッシュ表面そのものではなく、境界ボックスベースでのヒットであること」を示しています。クリックによるオブジェクト選択や大まかなヒットテストには十分な精度ですが、厳密な表面判定とは異なります。現在の webg における raycast() は、「精密な表面判定」というよりも「選択 UI のための入り口」として活用するのが適切です。

17.2 シーン内の重なりを調べる

シーン内のオブジェクト同士が一括して重なっているかを調べたい場合は、レイキャストではなく衝突判定（Collision）関連の API を使用します。samples/collisions では、移動するオブジェクトと複数のシェイプを組み合わせ、この判定の違いを確認できるようになっています。

衝突判定の基本的な考え方は、「まず広域判定で候補を絞り込み、必要に応じて詳細判定へ進む」という 2 段構えの構成です。

checkCollisions() による広域判定

まず使用するのは checkCollisions() です。

```
const collisions = app.space.checkCollisions({
  firstHit: false,
  filter: ({ node, shape }) => {
    if (!shape || shape.isHidden) return false;
    // 地面 (ground) は判定対象から除外する
    return node.name !== "ground";
  }
});

for (let i = 0; i < collisions.length; i++) {
  const pair = collisions[i];
  console.log(pair.nodeA.name, pair.nodeB.name, pair.boundsOnly);
}
```

この API は、各シェイプのワールド AABB を生成し、それらが重なっているペアを列挙します。firstHit: true に設定すれば最初に見つかった 1 件のみを返しますが、挙動の確認やデバッグにおいては、全件を配列で取得したほうが全体の流れを把握しやすくなります。ここでも filter を用いて地面や非表示シェイプを除外しておくことで、「意図しないオブジェクトに当たっている」という混乱を防ぐことができます。

checkCollisionsDetailed() による詳細判定

AABB による判定は広域的なものであるため、斜めに配置された細長い形状などでは、見た目以上に多くの候補が検出されることがあります。より精緻な判定が必要な場合に checkCollisionsDetailed() を使用します。

```
const detailed = app.space.checkCollisionsDetailed({
  firstHit: false,
  maxTrianglePairs: 80000,
  filter: ({ node, shape }) => {
    if (!shape || shape.isHidden) return false;
    return node.name !== "ground";
  }
});

for (let i = 0; i < detailed.length; i++) {
  const pair = detailed[i];
  console.log(
    pair.nodeA.name,
    pair.nodeB.name,
    pair.boundsOnly,
    pair.detailedSkipped === true
  );
}
```

checkCollisionsDetailed() は、広域判定を通過したペアに対して、三角形同士の交差判定を追加で行います。ただし、三角形の数が極端に多い場合に計算量が急増するため、maxTrianglePairs を超えたペアについては詳細判定をスキップし、boundsOnly: true および detailedSkipped: true というフラグを付けて返します。

したがって、常に詳細判定版を呼び出すのではなく、まずは checkCollisions() で候補の量を確認し、必要な場面に限定して詳細判定を行う運用が効率的です。

17.3 判定 API の使い分けまとめ

どの API を選択すべきかは、目的とする操作によって決まります。

- マウスやタップでオブジェクトを選択したい場合 → Space.raycast() クリック位置から 3D レイを生成し、最初にヒットした要素を取得することで、シンプルな選択 UI を実装できます。
- シーン内のオブジェクト同士の重なりを判定したい場合 → checkCollisions() ゲームロジックにおける基本的な当たり判定や、衝突候補の抽出に適しています。
- 境界ボックス（AABB）では判定が粗すぎる場合 → checkCollisionsDetailed() checkCollisions() で絞り込んだ後、さらに三角形レベルでの厳密な判定を行いたい場合に使用します。

17.4 注意点

まず、raycast() および衝突判定系 API は、すべて最新のワールド行列を前提として動作します。オブジェクトの移動や回転を更新した直後に判定を行う場合は、フレーム内の更新順序を正しく管理してください。特にカメラ側で app.eye.setWorldMatrix() を忘れると、生成されるレイが古い姿勢を向いてしまい、判定位置がずれる原因となります。

次に、現在の raycast() と checkCollisions() が AABB ベースであるという点を意識してください。戻り値の boundsOnly フラグは、その判定が「メッシュ表面の厳密なヒット」ではなく「境界ボックスのヒット」であることを示しています。細い棒状のオブジェクトや、斜めに配置された円柱、あるいは中空の形状の付近では、見た目と判定位置にわずかな乖離が生じることがあります。この特性を理解しておくことで、「当たっているように見えるのに反応しない」あるいはその逆といった混乱を避けることができます。

また、filter を設定せずに使用すると、カメラ自身や補助用シェイプ、非表示オブジェクトまで判定候補に含まれてしまい、意図した結果が得られにくくなります。開発の初期段階から node !== app.eye や !shape.isHidden、あるいは特定の地面オブジェクトを除外するといった条件を組み込んでおくことで、サンプルコードを実用的なアプリケーションへと発展させやすくなります。

17.5 関連サンプル

本章の内容を具体的に確認するには、以下のサンプルおよびユニットテストが適しています。

- レイキャストの最小構成例: unittest/raycast

- 衝突判定の比較: `samples/collisions`

ユーザー入力との連携については、前章の「タッチ機能と入力」 を合わせて参照してください。タップとドラッグを同一キャンバスで共存させる実装の流れを把握できます。

第 18 章

サウンドの設計

webg では、ブラウザの標準機能である Web Audio API を用いてサウンドを制御します。まず、Web Audio API を用いてサイン波の単音を 1 秒間鳴らす最もシンプルな実装例を見てみましょう。

```
<head>
  <meta charset="UTF-8">
  <title>単音を鳴らす</title>
</head>
<body>
  <button id="playButton">単音を鳴らす</button>
  <script>
    const playButton = document.getElementById("playButton");
    let audioContext = null;
    let osc = null;

    const startAudio = () => {
      audioContext = new AudioContext();           // AudioContext を作成
      osc = audioContext.createOscillator();       // 音源 OscillatorNode を作成
      const gainNode = audioContext.createGain();  // 音量調整用 GainNode を作成
      osc.type = "sine";                           // 正弦波を指定
      osc.frequency.value = 220;                   // 周波数を指定 A4 = 220Hz
      gainNode.gain.value = 0.2;                   // 音量を指定
      osc.connect(gainNode);                       // 接続する
      gainNode.connect(audioContext.destination);
    }

    const playTone = () => {
```

```
      osc.start();                                    // 音を鳴らす
      osc.stop(audioContext.currentTime + 1);         // 1 秒後に止める
    }

    playButton.addEventListener("click", () => {
      startAudio();
      playTone();
    });
  </script>
</body>
</html>
```

このように、単に音を鳴らすだけであれば Web Audio API を直接操作することで比較的簡単に実現できます。しかし、実際のアプリケーション開発において、単発の効果音を鳴らすだけで十分なケースは稀です。効果音（SE）とバックグラウンドミュージック（BGM）では、音響的な目的が根本的に異なります。また、音量、ディレイ、リバーブ、そしてメロディの管理をそれぞれ独立して制御したいという要求が必ず発生します。

そこで webg では、これらの要求を効率的に満たすため、機能を `ToneSynth`、`AudioSynth`、`GameAudioSynth` の 3 つの階層に分けて設計しています。`ToneSynth` は単音の再生・停止およびエフェクトを担う最小の基盤であり、`AudioSynth` はその上に SE と BGM のバス分離およびシーケンサ機能を追加します。そして `GameAudioSynth` は、これらを統合し、プリセット名で直感的に操作できるインターフェースを提供します。

この設計の核心は、「何を鳴らすか」というコンテンツ側の視点と、「どう聞かせるか」という音響設計側の視点を明確に分離している点にあります。また、Web Audio API 特有の制約である `AudioContext` の遅延初期化への対応や、系統別のエフェクト調整などを一貫したフローで管理することで、開発者が配線処理に煩わされることなく音響演出に集中できる環境を構築しています。

18.1　音響設計の思想とオーディオグラフ

Web Audio API を理解する上で最も重要な概念が、音の流れを「オーディオグラフ」として組み立てるという考え方です。これは、音源から最終出力までを、ノード（Node）と呼ばれる小さな部品の接続として表現するものです。物理的な機材で例えるなら、ギターからエフェクターを経てミキサーに接続し、最終的にスピーカーから出力させる構成に似ています。

基本的には、[音源] → [加工] → [出力] という一本の流れになります。例えば、オシレーターで音を作り、音量を調整し、リバーブを通して出力する場合、概念的には次のような接続になります。

[OscillatorNode] → [GainNode] → [ConvolverNode] → [AudioDestinationNode]

ここで、OscillatorNode は波形を発生させるソースノードであり、GainNode や ConvolverNode は音量や残響などの性質を変える加工ノード、AudioDestinationNode は最終的な出力先となります。

しかし、実際のアプリケーションでは「SE だけ音量を上げたい」「BGM だけディレイを変えたい」といった個別の制御が求められます。これを個々の音源に対して行うと管理が極めて複雑になるため、ここで「バス（Bus）」という概念を導入します。バスとは、複数の音をいったんまとめて流すための共通経路のことです。SE 用の音を SE バスへ、BGM 用の音を BGM バスへ集約させれば、経路上のノードを一つ調整するだけで、その系統全体の音量や空間特性を一括して制御できます。

webg のサウンド基盤はこのオーディオグラフの構築を自動化しています。ToneSynth.ensureContext() で AudioContext とマスターバスを生成し、buildToneFxChain() でディレイやリバーブを含むエフェクトチェーンを構築します。AudioSynth はこの基盤を継承し、さらに BGM 専用のバスを追加することで、SE と BGM の完全な経路分離を実現しています。

このように、ToneSynth は低レイヤー（ローレベル）な「単音用の配線ボード」に相当し、AudioSynth はそれを統合する「ミキシングコンソール」、そして GameAudioSynth はそれらを論理的な名前で操作する「高レイヤー（ハイレベル）」なインターフェースとして機能します。

18.2 サウンド機能の役割分担

サウンド機能の階層構造とそれぞれの責任範囲は以下の通りです。

1. **@<tt>{ToneSynth} (単音基盤層)** 単音の再生・停止、ADSR エンベロープ制御、ディレイ・リバーブ、およびインパルスレスポンスの動的生成を担当します。「1 つの音をどのように鳴らし、どのように止めるか」という最小単位を管理する低レイヤー（ローレベル）な層です。
2. **@<tt>{AudioSynth} (SE/BGM 基盤層)** ToneSynth を継承し、SE バスと BGM バスの分離、BGM シーケンサ、メロディ登録、BGM 専用エンベロープを管理します。単音基盤をアプリケーション全体の音構成へと拡張する役割を担います。

3. **@<tt>{GameAudioSynth} (高レイヤー/インターフェース層)** AudioSynth の上に構築され、ゲーム向けのメロディプリセットや SE カタログ、便利な発音ヘルパーを提供します。「どの名前の音を鳴らすか」という論理的な管理を行う高レイヤー（ハイレベル）な層です。
4. **@<tt>{webg/samples/sound} (検証層)** これらの機能を実際の UI を通じて調整し、音響特性を確認するためのサンプル実装です。

この階層構造により、単発の SE と連続的な BGM という異なる性質の音を適切に分離しつつ、開発者はコアな配線処理を意識せずに、プリセットの切り替えやパラメータ調整だけで演出を行えるようになっています。

18.3 標準的な利用フロー

オーディオコンテキストの有効化

Web Audio API を利用する際、最も注意すべき点は、音声再生は必ず「ユーザーの操作後」に行わなければならないというブラウザのオートプレイポリシーです。

このため、ToneSynth および AudioSynth はコンストラクタで即座に AudioContext を生成しません。ユーザーがボタンを押すなどの操作を行い、resume() が呼ばれたタイミングで初めて ensureContext() が動作し、マスターバスや各系統のルーティングが構築されます。webg/samples/sound に「Audio Start」ボタンが用意されているのは、この制約を適切に処理するためです。

SE と BGM の使い分けと空間調整

音声機能が有効化した後は、SE と BGM を別々のバスとして制御します。特に BGM は単純なファイルの再生ではなく、scheduleBgm() による先読みスケジューリングで進行するため、BPM やメロディの変更が楽曲全体の流れに動的に反映されます。

空間演出におけるディレイとリバーブの調整では、「量」と「特性」を明確に区別して制御します。- **量（Amount）：** setSeReverb() や setBgmReverb() などで調整します。これはセンド/リターン量、つまり「どれだけリバーブ成分を混ぜるか」を決定します。- **特性（Characteristic）：** setSeReverbImpulse() や setBgmReverbImpulse() で調整します。

これはコンボリューションリバーブに用いるインパルスレスポンス（IR）を変更し、「部屋（room）」「ホール（hall）」「プレート（plate）」といった空間の質や残響時間を決定します。

量と特性を分けて制御することで、音量バランスを維持したまま空間の広がりだけを変えるといった、精密な音響設計が可能になります。

18.4 使い方の基本例

効果音（SE）の再生

通常は GameAudioSynth を使用します。オーディオを有効化し、プリセット名で SE を再生する最小の実装例は以下の通りです。

```
import GameAudioSynth from "./webg/GameAudioSynth.js";

const synth = new GameAudioSynth();

// ユーザー操作（クリックなど）をきっかけにオーディオを有効化して再生する
button.addEventListener("click", async () => {
  // AudioContext を有効化し、ルーティングを構築する
  await synth.resume();

  // 全体の音量を設定
  synth.setMasterVolume(0.25);

  // プリセット名で SE を再生
  synth.playSe("poyoon");
});
```

音量バランスの調整

マスターボリュームを基準とし、SE と BGM の相対的なバランスを個別に設定します。

```
synth.setMasterVolume(0.25); // 全体の基準音量
synth.setSeVolume(0.90);     // SE の相対音量
synth.setBgmVolume(0.75);    // BGM の相対音量
```

空間エフェクト（ディレイ・リバーブ）の調整

ディレイとリバーブの「量」を調整し、音の奥行きや空間の広がりを制御します。

```
// SE ディレイ: 時間, フィードバック, ウェット量
synth.setSeDelay(0.11, 0.26, 0.22);
// BGM ディレイ: 時間, フィードバック, ウェット量
synth.setBgmDelay(0.18, 0.22, 0.18);

// SE リバーブ: センド量, リターン量
synth.setSeReverb(0.32, 0.55);
// BGM リバーブ: センド量, リターン量
synth.setBgmReverb(0.28, 0.48);
```

リバーブ特性の変更

インパルスレスポンス（IR）を変更して、空間の質（部屋の大きさや材質など）を切り替えます。

```
// SE のリバーブ特性を「ルーム」に設定
synth.setSeReverbImpulse({
  kind: "room",
  durationSec: 2.2,
  decay: 1.6
});

// BGM のリバーブ特性を「ホール」に設定
synth.setBgmReverbImpulse({
  kind: "hall",
  durationSec: 4.0,
  decay: 1.9
});
```

エンベロープの調整

音の立ち上がり（アタック）や減衰（ディケイ）を調整します。SE は輪郭をはっきりさせ、BGM は持続感を出すといった使い分けが可能です。

```
// SE 用のエンベローププリセットを適用
synth.setSeEnvelopePreset("piano", {
  attack: 0.005,
  decay: 0.05,
  sustain: 0.55,
  release: 0.08
});

// BGM 用のエンベロープを個別に設定
synth.setBgmEnvelope({
  attack: 0.03,
  decay: 0.20,
  sustain: 0.60,
  release: 0.40
});
```

BGM プリセットの切り替え

GameAudioSynth に内蔵されているメロディプリセットを setMelody() で切り替えます。BPM や root、BGM envelope も同時に変更したい場合は、以下のようにプロファイルテーブルを用いて管理すると効率的です。

```
const bgmProfiles = {
  boss: {
    melody: "boss_alert",
    bpm: 144,
    rootHz: 196.0,
    envelope: { attack: 0.02, decay: 0.16, sustain: 0.56, release: 0.32 }
  }
};
```

```
const applyBgmProfile = (synth, name) => {
  const profile = bgmProfiles[name];
  synth.setMelody(profile.melody);
  synth.setBpm(profile.bpm);
  synth.setRootHz(profile.rootHz);
  synth.setBgmEnvelope(profile.envelope);
};

// メロディ preset を直接切り替えて開始
synth.setMelody("minor_drive");
synth.startBgm();

// 場面に応じた profile をアプリ側で適用
applyBgmProfile(synth, "boss");
```

18.5 独自 BGM をアプリ側で追加する

GameAudioSynth は既存のプリセットを切り替えるだけでなく、registerMelody() を用いてアプリケーション側から独自の BGM を追加することが可能です。これにより、ライブラリ本体を書き換えることなく、ステージ曲やボス戦専用のループ曲などを定義できます。

webg の BGM は、16 ステップを 1 小節とする簡易シーケンサで構成されています。各ステップは 8 分音符単位で進行し、ベースとリードが個別に予約再生されます。

以下は、アプリ側で story_field という新しい BGM を追加して利用する例です。

```
import GameAudioSynth from "./webg/GameAudioSynth.js";

const synth = new GameAudioSynth();

async function startStoryFieldBgm() {
  await synth.resume();

  synth.registerMelody("story_field", {
    scale: [0, 2, 4, 5, 7, 9, 11],
    bassPattern: [0, -5, -3, -2, 0, -4, -2, -5],
    bassOctave: -12,
    leadDegrees: [
```

```
      0, 2, 4, 5, 7, 5, 4, 2,
      0, 2, 4, 7, 9, 7, 5, 4,
      0, 2, 4, 5, 7, 9, 7, 5,
      4, 5, 7, 9, 11, 9, 7, 5
    ],
    leadHoldSteps: [
      1.0, 1.0, 1.0, 1.6, 1.0, 1.0, 1.0, 2.0,
      1.0, 1.0, 1.0, 1.7, 1.0, 1.0, 1.0, 2.4,
      1.0, 1.0, 1.0, 1.6, 1.0, 1.0, 1.0, 2.0,
      1.0, 1.0, 1.0, 1.8, 1.0, 1.0, 1.0, 2.8
    ],
    leadOctave: 12,
    leadType: "triangle",
    leadDur: 0.11,
    leadGain: 0.060,
    bassType: "triangle",
    bassDur: 0.16,
    bassGain: 0.074,
    rhythmDropWeakProb: 0.28,
    rhythmDropTailProb: 0.52
  });

  synth.setBpm(116);
  synth.setRootHz(220.0);
  synth.setBgmEnvelope({
    attack: 0.03,
    decay: 0.18,
    sustain: 0.60,
    release: 0.36
  });
  synth.setMelody("story_field");
  synth.startBgm();
}
```

leadDegrees の要素数を増やすことで、小節をまたいだ長いフレーズを作成できます。また、bassPattern を 8 要素に伸ばせば、2 小節周期の低音進行を構築することが可能です。

BGM パラメータの定義

独自 BGM を作成する際の主要なパラメータの役割は以下の通りです。

- scale: 度数を半音へ変換する基準です。[0, 2, 3, 5, 7, 8, 10] ならマイナー系、[0, 2, 4, 5, 7, 9, 11] ならメジャー系に近い響きになります。
- bassPattern: ベースの進行です。1 要素が 4 ステップ分割り当てられるため、4 要素で 1 小節、8 要素で 2 小節分の進行になります。
- bassOctave: ベース全体のピッチを調整します。-12 は 1 オクターブ下を指定します。
- leadDegrees: リード旋律を定義する配列です。null を指定したステップは休符となります。
- leadGate と leadDegreeStep: leadDegrees を記述せず、一定の規則で旋律を生成したい場合の簡易指定です。
- leadHoldSteps: 各ステップの音価倍率です。フレーズの終端や強拍で音を伸ばす際に使用します。
- leadOctave: リード全体の高さを指定します。
- leadType と bassType: オシレーターの波形です。音色の印象を決定づけます。
- leadDur と bassDur: 各音の基本持続時間です。
- leadGain と bassGain: 各パートの音量バランスを調整します。
- rhythmDropWeakProb: 弱拍を確率的に休符にする設定です。値を高くすると軽快なグルーヴになります。
- rhythmDropTailProb: 小節末尾付近を確率的に休符にする設定です。値を高くすると「息継ぎ」のような間が生まれます。

なお、registerMelody() では leadHoldSteps、rhythmDropWeakProb、rhythmDropTailProb の指定が必須です。不完全なデータによる曖昧な再生を避けるため、不足している場合は例外が発生します。

18.6 波形の選択と音色の調整

OscillatorNode を用いたサウンド生成では、波形の選択が音色の印象を決定します。

- sine (正弦波): 倍音が少なく、丸く柔らかい音です。静かな BGM や背景的な SE に適しています。
- triangle (三角波): sine より芯がありつつも柔らかい音です。BGM のリードや汎用的な UI 音に適しています。
- square (矩形波): 輪郭が強く、レトロゲームらしい主張のある音です。メロディを強調したい場合や通知音に適しています。
- sawtooth (鋸歯波): 倍音が非常に多く、強く荒い印象の音です。警告音やボス戦 BGM、

派手な SE に適しています。

波形の変更は、BGM であれば leadType や bassType、SE であれば playTone() や playGameTone() の options.type で指定します。

18.7 エンベロープと主要パラメータの制御

音の立ち上がりや余韻などの時間的変化は、エンベロープと各種パラメータで制御します。

- attack: 音量が 0 からピークに達するまでの時間です。
- decay: ピークから持続音量（Sustain）まで減衰する時間です。
- sustain: 減衰後に維持される音量比です。
- release: ノート終了後に音が消えるまでの時間です。
- gain: 個別の音量です。ミックス全体の音量とは別に設定します。
- dur: ノートの基本持続時間です。
- pan: 左右の定位（パンニング）です。-1（左）から 1（右）で指定します。
- detune: セント単位の微調整です。わずかにずらすことで、音の厚みや揺らぎを表現できます。
- when: AudioContext 上の絶対時刻を指定します。複数の音を精密にずらして重ねる際に使用します。

18.8 独自の効果音をアプリ側で構築する

独自の SE を作成する場合、必ずしも GameAudioSynth のプリセットに追加する必要はありません。アプリ側で関数を定義し、playTone() または playGameTone() を組み合わせることで、柔軟に音を構築できます。

playTone() は波形やエンベロープをすべて個別に指定できる低レイヤー（ローレベル）なメソッドです。一方、playGameTone() は percussion や piano といった楽器カテゴリのプリセットを選択し、必要な項目だけを上書きできる高レイヤー（ハイレベル）なヘルパーです。

以下は、宝箱が開いた際の演出音を独自に構築する例です。

```
import GameAudioSynth from "./webg/GameAudioSynth.js";

const synth = new GameAudioSynth();

function playTreasureOpen() {
  const t0 = synth.ctx.currentTime;

  // 最初の明るい立ち上がり
  synth.playGameTone(523.25, 0.08, "guitar", {
    when: t0,
    type: "triangle",
    gain: 0.07,
    release: 0.06,
    pan: -0.08
  });

  // 少し遅れて和音感を足す
  synth.playGameTone(783.99, 0.12, "piano", {
    when: t0 + 0.040,
    type: "triangle",
    gain: 0.06,
    release: 0.10,
    pan: 0.06
  });

  // 高域のきらめき
  synth.playGameTone(1174.66, 0.16, "woodwind", {
    when: t0 + 0.090,
    type: "sine",
    gain: 0.04,
    release: 0.16,
    pan: 0.20
  });
}

button.addEventListener("click", async () => {
  await synth.resume();
  playTreasureOpen();
});
```

単一の音ではなく、数十ミリ秒の差を設けて複数の音を重ね合わせることで、単純なビープ

音を避け、奥行きのあるサウンドを構築できます。

名前付き SE カタログによる管理

アプリケーションの規模が大きくなった場合は、SE の対応表をアプリ側で管理することを推奨します。これにより、ゲームロジックと具体的な音色定義を分離できます。

```
const customSe = {
  treasure_open: () => playTreasureOpen(),
  portal_start: () => playPortalStart(),
  dialogue_tick: () => playDialogueTick()
};

function playAppSe(name) {
  const se = customSe[name];
  if (!se) {
    throw new Error(‘Unknown app sound effect: ${name}‘);
  }
  se();
}
```

18.9　音声ファイル（AudioBuffer）の利用

合成音ではなく、mp3、wav、ogg などの音声ファイルを使用したい場合は、AudioSynth の AudioBuffer 管理機能を利用します。

AudioSynth.loadAudioBuffer(name, url) を用いてファイルを読み込み、playAudioBuffer(name, options) で再生します。このとき、options.bus に "se" または "bgm" を指定することで、それぞれのバスに設定された音量やエフェクト（ディレイ・リバーブ）を適用させることができます。

```
import AudioSynth from "./webg/AudioSynth.js";

const synth = new AudioSynth();

button.addEventListener("click", async () => {
```

```
  await synth.resume();

  // mp3 を読み込み、SE bus へ流して再生
  await synth.loadAudioBuffer("door", "./audio/door.mp3");
  synth.playAudioBuffer("door", {
    bus: "se",
    gain: 0.8,
    pan: -0.15
  });
});
```

環境音などのループ素材を扱う場合は、bus: "bgm" と loop: true を指定します。再生中の voice ハンドルを保持していれば、voice.stop() や stopAudioBuffer(voice) を呼び出して停止させることができ、fadeSec を指定することで自然なフェードアウトが可能です。

```
await synth.loadAudioBuffer("forest", "./audio/forest_loop.ogg");
const voice = synth.playAudioBuffer("forest", {
  bus: "bgm",
  loop: true,
  gain: 0.45
});

// シーン終了時に短くフェードアウト
voice.stop(synth.ctx.currentTime, { fadeSec: 0.4 });
```

素材の読み込み失敗や未登録名の再生要求は、設定ミスを早期に発見するため、あえて例外として処理されます。

18.10 AudioSynth の内部構造

バス構成とルーティング

AudioSynth は音響基盤としての役割を担います。親クラスである ToneSynth の ensureContext() および resume() による遅延初期化を起点とし、マスターバス → SE バス → 各エフェクトチェーンというルーティングを構築します。

ここに BGM バスと BGM シーケンサを追加することで、SE と BGM の完全な分離を実現しています。単発音の生成と停止は ToneSynth が受け持ち、AudioSynth は playSe()、AudioBuffer の再生、メロディデータに基づく先読みスケジューリング、および BGM 専用エンベロープの管理を担当します。

インパルスレスポンスの動的生成

ToneSynth.createImpulseResponse() は、外部ファイルを読み込まずに数学的な計算によってステレオの AudioBuffer を生成します。

1. **プロファイルの取得:** getImpulseProfile(kind) により、「room」「hall」「plate」それぞれの反射密度や減衰率を決定します。
2. **テールの書き込み:** writeImpulseTail() で、時間経過とともに高域が減衰するノイズ列を生成し、残響の尾（テール）を作成します。
3. **初期反射の追加:** addEarlyReflections() で、左右チャンネルに時間差を持たせた初期反射を加えます。これにより、空間の「広さ」や「方向感」をシミュレートします。
4. **正規化:** normalizeImpulse() でピーク音量を揃え、特性を切り替えても極端な音量差が出ないように調整します。

BGM シーケンサの仕組み

webg の BGM は、スコア（楽譜）に基づいた予約再生方式を採用しています。startBgm() が処理を開始し、scheduleBgm(lookAheadSec) が少し先の未来までのノートを予約し、scheduleBgmStep(step, when) がベースとリードの音を組み立てます。

メロディプリセットは、単なる音階の並びではなく、スケール、ベースパターン、リードのゲート時間、音色タイプなどのパラメータセットとして定義されています。リードの音域を広く取り、ベースパターンを多ステップにすることで、単調な反復を避け、音楽的な展開を実現しています。

18.11 GameAudioSynth の役割

GameAudioSynth は AudioSynth の高レイヤーラッパーであり、ゲーム開発者が即座に利用できるメロディプリセットと SE カタログを提供します。

アプリケーション開発においては、このクラスをエントリーポイントとして利用し、「どの場面でどのメロディや BPM を使用するか」といった対応表をアプリ側に持たせることで、ゲームロジックと音響設定をシンプルに接続できます。

18.12 webg/samples/sound による検証

webg/samples/sound は、音響設計の調整を効率的に行うための検証ツールです。GameAudioSynth および AudioSynth のパラメータをリアルタイムに操作し、その結果を即座に確認できるインターフェースを提供します。

UI は「System（全体設定）」「Sound Effects（SE 調整）」「Background Music（BGM 調整）」の 3 つのブロックで構成されており、音量、ディレイ、リバーブ、エンベロープの聞き比べが可能です。

特に tail_probe という検証用 SE は、リバーブとエンベロープの差を明確に聞き分けるために設計されています。冒頭の短いアタック音で立ち上がりを確認し、その後の持続音でディケイやリバーブのテールを追跡できるため、音響調整の基準音として活用してください。

18.13 サウンド機能の拡張とカスタマイズ

サウンド機能を拡張、または内部構造を変更する場合は、以下の関連箇所を確認してください。

- **ルーティングの変更:** ToneSynth.ensureContext()、buildToneFxChain()、AudioSynth.buildBgmFxChain()。
- **リバーブ特性の変更:** getImpulseProfile()、writeImpulseTail()、addEarlyReflections()。
- **BGM の挙動変更:** startBgm()、scheduleBgm()、scheduleBgmStep()。
- **プリセットの追加:** 共通のメロディや SE は GameAudioSynth へ追加し、アプリ固有のものはアプリ側の対応表で管理することを推奨します。
- **サンプルの UI 修正:** webg/samples/sound/main.js および関連ドキュメント。

特にインパルスレスポンスをカスタマイズする際は、「エフェクトの量（センド量）」と「空間の特性（IR）」の制御を混同させないことで、直感的な調整環境を維持できます。

第 19 章

診断情報の共有

単にデバッグ用の文字列を画面端に表示させるだけでは、現在の 3D アプリケーションの状態を記録したり、コーディング AI に正確に伝えることは困難です。真に必要とされるのは、その瞬間の状態を「レポート」としてまとめ、画面上で確認し、必要に応じてそのまま共有できる仕組みです。

本章では、DebugDock、Diagnostics、DebugProbe、および WebgApp を通じて、診断情報を単に「見るもの」ではなく「共有するもの」として扱う設計思想について解説します。webg の診断システムは、アプリケーションの現在状態を画面で読み取り、そのままクリップボード経由で共有することを目的として構成されています。

例えば、描画された画面の状態をコーディング AI に伝えて解析を依頼したい場合、Debug Dock の Copy Summary または Copy JSON を利用します。これにより、その瞬間のカメラ位置、視野角、シーン規模、警告、およびサンプル固有の状態がクリップボードにコピーされます。利用者はそれを貼り付けるだけで、コーディング AI に対して「現在見えているはずの状態」を数値やテキストとして正確に渡すことができます。つまり、診断情報は単なる補助的な表示ではなく、アプリケーションの内部状態を外部へ出力するための正式なインターフェースなのです。

このシステムは、役割の異なる以下のコンポーネントの組み合わせで構成されています。

- DebugConfig.js：デバッグモードとリリースモードの切り替え、および診断機能の有効/無効を管理します。
- DebugDock.js：現在状態の表示と、レポートのコピー操作を担当します。
- Diagnostics.js：レポートオブジェクトを生成し、要約テキストや JSON 形式に整形します。

- DebugProbe.js：特定の瞬間における状態採取を支援します。

このように、「診断情報の正本（マスターデータ）を生成する層」と、「それを表示またはコピーする層」が明確に分離されています。本章では、まず DebugDock とクリップボード共有を中心とした情報の流れを整理し、レポートの構成について詳しく見ていきます。

19.1　診断情報を共有するという設計思想

webg の診断システムは、ブラウザのコンソール（console）を主役にした仕組みではありません。もちろん開発者ツールを用いてコンソールを確認することは可能ですが、標準的な共有経路として Copy Summary と Copy JSON を提供しています。navigator.clipboard.writeText() を用いてレポートを直接クリップボードに書き込むことで、利用者がチャットツールやバグ報告フォームへ即座に貼り付けられる形式を実現しています。

この設計を採用している理由は、画面を見ている本人にしか分からない情報を、再現性のある形で外部へ伝達するためです。Diagnostics 自体は表示部品ではなく「レポート生成層」であり、第 14 章で整理した表示経路においても、DebugDock や OverlayPanel に情報を渡す前の「正本」を作成する役割を担っています。表示と共有の両方が、常に同一のレポートを基準に動作することがこの設計の肝となります。

19.2　DebugDock による状態の可視化と操作

DebugDock は、診断情報の中心となる開発者向けドックです。ここには Current State の表示領域と、Copy Summary、Copy JSON の 2 つのボタンが配置されています。画面上で現在状態を確認し、そのまま同じ内容をコピーできるため、確認した内容と共有した内容に齟齬が生じません。

WebgApp の標準設定では、デバッグ用のキー入力にプレフィックス方式を採用しています。ただし、初期状態は release モードであり、通常の利用者に開発者向けの DebugDock を突然表示させないようになっています。診断機能を使うときは debugTools.mode: "debug" を指定するか、app.attachInput() で入力を接続したうえで、「F9」キーの後に「M」キーを押下する順次入力によってデバッグモードへ切り替えます。同様に、「F9」の後に「C」で要約を、「F9」の後に「V」で JSON をコピーできます。

リリースモードでは enableDiagnostics と enableProbe が無効になり、デバッグ用のドックや補助表示も非表示となります。アプリケーション側の処理から明示的に切り替えたい

場合は、app.setDebugMode("debug") または app.setDebugMode("release") を使用してください。このメソッドは、診断フラグを管理する DebugConfig、DebugDock の表示状態、およびキャンバスレイアウトをまとめて更新します。

19.3 レポートの構成と自動収集メカニズム

診断レポートは、人間が把握するための「要約（Summary）」と、コーディング AI が処理するための「JSON」の 2 形式で提供されます。

要約（Summary）の構成

Copy Summary は Diagnostics.toSummaryText() によって生成され、以下の 5 つのセクションで構成されています。

1. Overview: source（ソースファイル）、system（システム名）、stage（現在の段階）、ok（正常判定）、timestamp（時刻）が含まれます。
2. Latest Issue: 最新のエラー情報を表示します。エラーがない場合は最新の警告（warning）を、それもない場合は（none）と表示されます。
3. Key Stats: カメラの位置・姿勢・ターゲット、距離、視野角、シェーダークラス、シーン規模、キャンバスおよびディスプレイサイズ、フレームカウント、稼働時間などが並びます。
4. Warnings: 警告配列の末尾から最大 4 件を表示します。
5. Recent Details: 詳細配列の末尾から最大 8 件を表示します。

JSON 形式による詳細情報

Copy JSON は、より詳細な構造情報を含みます。特に context.webgAuto セクションには、カメラの正確な視点位置（eye position）や姿勢（eye attitude）、シーン内で使用されているシェーダークラスの一覧とその使用数、マテリアル ID の使用数などが格納されており、レンダリング構成を精査する場合に適しています。

WebgApp による自動統計情報の収集

WebgApp は、開発者が明示的に指定しなくても、起動直後の環境情報と現在のシーン規模を自動的に収集します。app.init() の最終段階で checkEnvironment({ stage: "ready" }) が呼び出され、WebGPU のサポート状況や基本整合性がレポートに記録されます。

さらに、collectCurrentSceneStats() によって、以下の統計情報が自動的に収集されます。

- シーン規模: nodeCount、shapeCount、meshCount、vertexCount、triangleCount、boneCount など。
- 描画条件: eyeX/Y/Z、eyeYaw/Pitch/Roll、cameraTargetX/Y/Z、fovX/Y、canvasWidth/Height など。

この自動収集機能により、サンプルごとにカメラ位置やシーン規模を手書きで記録する必要はありません。診断機能は付随的なオプションではなく、WebgApp の標準構成の一部として組み込まれています。

19.4 アプリケーション固有の診断情報の追加

自動収集される情報だけでは、アプリケーション固有の文脈を十分に伝えることはできません。そこで、開発者は以下のインターフェースを用いて情報を追加します。

実行段階（stage）の記録

現在アプリケーションがどの処理段階にあるかを記録します。これにより、レポートを読んだ人が「この値がリソース取得中（fetch）のものか、実行中（runtime）のものか」を判別できるようになります。

```
app.setDiagnosticsStage("fetch");
// ... リソース読み込み...
app.setDiagnosticsStage("runtime");
```

統計情報（stats）と詳細情報（details）の追加

サンプル固有の重要な数値は mergeDiagnosticsStats() を用い、補足的な情報は addDiagnosticsDetail() で追加します。

```
// 重要な数値（Key Stats セクションへ）
app.mergeDiagnosticsStats({
  score: currentScore,
  focusDistance: dof.focusDistance.toFixed(1)
});

// 補足情報（Recent Details セクションへ）
app.addDiagnosticsDetail(`selectedClip=${selectedClipName}`);
```

警告情報（warnings）の追加

優先的に確認してほしい内容は addDiagnosticsWarning() で追加します。警告情報は Latest Issue と Warnings の両方に表示されるため、非常に注目度が高くなります。

また、checkEnvironment() の結果を診断情報に統合することで、要約を見ただけで「そのシーンが基本条件を満たしているか」を即座に判断させることが可能です。

19.5　特殊な採取手法：ワンショットプローブとエラーレポート

ワンショットプローブ（One-shot Probe）

連続的に更新される状態ではなく、「設定を切り替えた直後の 1 回だけ」の状態を固定して採取したい場合は、configureDiagnosticsCapture() と createProbeReport() を使用します。

```
app.configureDiagnosticsCapture({
  labelPrefix: "eye_rig",
  collect: () => {
```

```
    const report = app.createProbeReport("runtime-probe");
    Diagnostics.mergeStats(report, { frameCount: app.screen.getFrameCount() });
    return report;
  }
});

// 指定したフレーム後に状態を採取
app.captureDiagnosticsSummary({ afterFrames: 1 });
```

起動失敗時のレポート処理

アプリケーションの起動に失敗した場合、エラーパネルを表示すると同時に、診断レポートをエラー版に差し替えておくことで、問題の共有が容易になります。

```
start().catch((error) => {
  app?.setDiagnosticsReport?.(Diagnostics.createErrorReport(error, {
    system: "eye_rig",
    source: "samples/eye_rig/main.js"
  }));
  app?.showOverlayPanel?.(buildErrorPanelOptions(error, { id: "start-error" }));
});
```

成功時だけでなく、失敗時にも共有経路を維持することが、webg の診断設計における大きな利点です。

19.6 診断機能の個別実装経路

通常は WebgApp を利用しますが、samples/sound のように独自の UI を構築し、Diagnostics.js を直接利用する方法もあります。これは 3D シーンの統計情報よりも、UI の調整値の方が重要な場合に適しています。

```
const report = Diagnostics.createSuccessReport({
  system: "sound",
  source: "samples/sound/main.js",
```

```
  stage: "runtime"
});

Diagnostics.mergeStats(report, { masterVol: ui.masterVol.value });
```

このように、診断機能は WebgApp 専用ではなく、レポート生成層として単独で利用可能です。

19.7 まとめ

本章で最も重要な点は、診断情報を単なる「画面上の補助表示」ではなく、「現在の状態を共有するための正式な出力口」として扱うことです。システムの中心にあるのは「現在のレポート」であり、それを Copy Summary と Copy JSON によって外部へ持ち出せる点に価値があります。

WebgApp は共通情報を自動的に収集しますが、アプリケーション固有の意味を伝えるには、stage、stats、details、warnings を適切に使い分ける必要があります。

第 14 章と第 15 章で整理した表示経路において、診断機能は「レポートを生成する層」という特別な位置にあります。その結果を DebugDock や OverlayPanel が表示・共有へと接続することで、開発効率を最大化させる設計となっています。

第 20 章

ポストプロセスの実装

Bloom（ブルーム）や被写界深度（Depth of Field)、曇りガラスのような視覚効果は、単に個別のオブジェクトにシェーダーを追加するだけでは実現できません。一度シーン全体をオフスクリーン（画面外）のバッファへ描き出し、その結果を後段のフルスクリーンパスで加工してから最終的にキャンバスへ出力するという、描画フローの再構築が必要になります。

本章では、`BloomPass`、`DofPass`、`FrostedGlassPass`、`VignettePass` の導入手順を軸に、ポストプロセスの設計思想と実装方法を解説します。

20.1　ポストプロセスの概念と描画フロー

ポストプロセスとは、シーン内の個々のオブジェクトのシェーダーを書き換える手法ではなく、描画が完了した後の「画面全体」に対して後処理をかけるアプローチです。まず 3D シーンをオフスクリーンターゲットへ描き出し、得られたカラーテクスチャや深度テクスチャをフルスクリーンパスで読み取って加工します。

このフローを理解せずに導入すると、「Bloom を実装したが画面に変化がない」「DOF を導入したが深度情報が読み取れない」といった問題に直面しやすくなります。そのため、まずは共通の描画フローを確立させましょう。

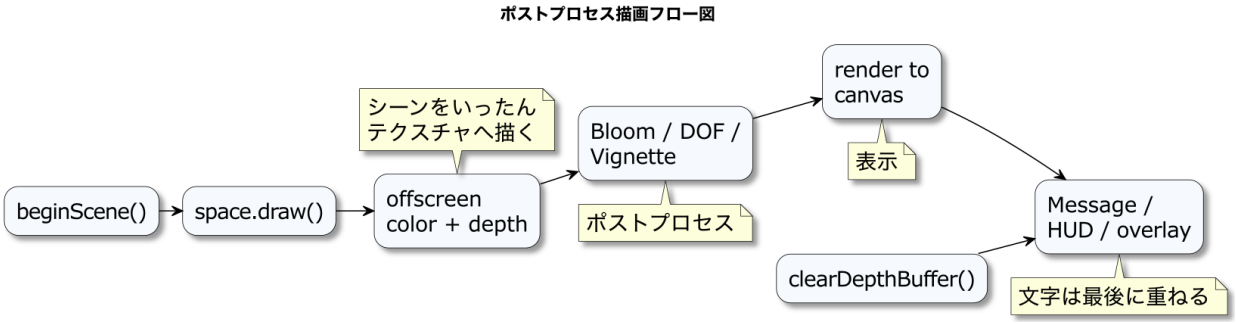


図 20.1: ポストプロセス描画フロー図

ポストプロセスでは、シーンを一度オフスクリーンへ描き出し、その結果に後処理を適用した後で *HUD* を重ねる。

描画フローの制御：`autoDrawScene: false`

`WebgApp` の標準設定では、シーンが直接キャンバスへ出力されます。しかし、ポストプロセスを適用するには、キャンバスに描画される前に処理を挟む必要があるため、`WebgApp` の設定で `autoDrawScene: false` に指定します。これにより、描画のタイミングを完全に制御できるようになります。

基本的な実装サイクル

ポストプロセスの多くは、以下の 3 つのステップで構成されるサイクルを 1 フレームの中で完結させる必要があります。

1. `beginScene()`: 描画先をオフスクリーンターゲットに切り替え、シーン描画の準備を行う。
2. `app.space.draw(app.eye)`: 通常通り 3D シーンを描画する。
3. `render()`: 描き出されたテクスチャを加工し、最終的な結果をキャンバスへ出力する。

この流れを具体的に実装すると、以下のようになります。

```
import WebgApp from "./webg/WebgApp.js";
import FullscreenPass from "./webg/FullscreenPass.js";
```

```
import BloomPass from "./webg/BloomPass.js";

const app = new WebgApp({
  document,
  messageFontTexture: "./webg/font512.png",
  autoDrawScene: false
});
await app.init();

const bloom = new BloomPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await bloom.ready;

const debugPass = new FullscreenPass(app.getGPU(), {
  targetFormat: app.getGPU().format
});
await debugPass.init();

app.start({
  onUpdate: ({ screen }) => {
    bloom.resizeToScreen(screen);
  },
  onBeforeDraw: () => {
    // ステップ 1: 描画の入口
    bloom.beginScene(app.screen, app.clearColor);
    // ステップ 2: 3D シーンの描画
    app.space.draw(app.eye);
  },
  onAfterDraw3d: () => {
    // ステップ 3: ポストプロセス処理の適用と出力
    bloom.render(app.screen, {
      clearColor: app.clearColor
    });

    // HUD を正しく重ねるための深度バッファクリア
    app.screen.clearDepthBuffer();
  }
});
```

ここで重要なのは、3D シーンを描画する責任は引き続き app.space.draw(app.eye) が担っており、BloomPass などのパス・クラスはその前後を繋ぐ役割を果たしている点です。

HUD の重ね合わせ

ポストプロセス適用後も Message やデバッグオーバーレイを正しく表示させたい場合は、最終パスの後に app.screen.clearDepthBuffer() を呼び出す必要があります。これは、ポストプロセス処理で書き込まれた深度情報が残っていると、後から描画する HUD が深度テストによって棄却されてしまうためです。これはエフェクト自体の機能ではなく、描画フローを適切に制御するための必須処理となります。

20.2 BloomPass の実装と活用

役割と仕組み

BloomPass は、シーン内の輝度が高い部分のみを抽出し、それをぼかして元のシーンに加算することで、光がにじみ出すような視覚効果を演出します。カラー情報のみを利用して動作するため、ポストプロセスの基本を理解するのに最適なコンポーネントです。

内部では sceneTarget（元のシーン）、extractTarget（輝度抽出）、extractHeatTarget（しきい値処理後）、および SeparableBlurPass が管理するぼかし用ターゲットを使用します。利用者は new BloomPass(gpu, options) でインスタンス化し、await bloom.ready で準備完了を待機した後、前述の基本サイクルに従って制御します。

実装例

```
import BloomPass from "./webg/BloomPass.js";

const bloom = new BloomPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight(),
  threshold: 0.68,
  extractIntensity: 1.0,
  softKnee: 0.35,
  bloomStrength: 1.15,
```

```
  exposure: 1.0,
  toneMapMode: 0,
  blurScale: 1.0,
  blurIterations: 2,
  blurRadius: 1.0
});
await bloom.ready;

app.start({
  onUpdate: ({ screen }) => {
    bloom.resizeToScreen(screen);
  },
  onBeforeDraw: () => {
    bloom.beginScene(app.screen, app.clearColor);
    app.space.draw(app.eye);
  },
  onAfterDraw3d: () => {
    bloom.render(app.screen, {
      clearColor: app.clearColor
    });
    app.screen.clearDepthBuffer();
  }
});
```

デバッグビューによる調整

ポストプロセスの調整において、中間結果を可視化することは極めて重要です。BloomPass は内部ターゲットへのアクセス用ゲッターを提供しているため、FullscreenPass を利用して現在の状態を確認できます。

```
import FullscreenPass from "./webg/FullscreenPass.js";

const debugPass = new FullscreenPass(app.getGPU(), {
  targetFormat: app.getGPU().format
});
await debugPass.init();

let bloomView = "composite";

app.start({
```

```
  onUpdate: ({ screen }) => {
    bloom.resizeToScreen(screen);
  },
  onBeforeDraw: () => {
    bloom.beginScene(app.screen, app.clearColor);
    app.space.draw(app.eye);
  },
  onAfterDraw3d: () => {
    bloom.render(app.screen, {
      clearColor: app.clearColor
    });

    if (bloomView !== "composite") {
      const debugSource = bloomView === "scene"
        ? bloom.getSceneTarget()
        : bloomView === "extract"
          ? bloom.getExtractTarget()
          : bloomView === "extractHeat"
            ? bloom.getExtractHeatTarget()
            : bloomView === "blurA"
              ? bloom.getBlurTargetA()
              : bloomView === "blurTargetB";

      app.screen.beginPass({
        clearColor: app.clearColor,
        colorLoadOp: "clear",
        depthView: null
      });
      debugPass.draw(debugSource);
    }

    app.screen.clearDepthBuffer();
  }
});
```

特に extract（抽出）と extractHeat（熱量抽出）を比較することで、しきい値（threshold）や softKnee がどのように効いているかを視覚的に把握でき、効率的なパラメータ調整が可能になります。

20.3 DofPass の実装と活用

役割と仕組み

DofPass は、シーンのカラー情報に加えて「深度情報」を参照し、合焦面（フォーカス面）だけを鮮明に残し、それ以外をぼかすエフェクトです。Bloom との最大の違いは、後段の処理で深度テクスチャを利用する点にあります。

現在の DofPass は、多段階ぼかし（staged blur）として、1 枚の強いぼかし画像へ急に切り替えるのではなく、small、medium、large の 3 段階のぼかし画像を作り、深度差に応じて scene -> small -> medium -> large の順に合成します。これにより、合焦面から外れた直後に強いぼかしへ飛ぶ見え方を抑えやすくなります。内部では sceneTarget、depthDebugTarget、focusDebugTarget、stageDebugTarget、および 3 段階のぼかし用ターゲットを保持しています。また、深度を正しく計算するために projectionNear や projectionFar といった投影行列の設定が必要になります。

focusRange は、最大ぼかしへ到達するまでの全体距離ではなく、1 つの blur stage が進む距離幅として扱います。焦点からの距離差が focusRange の 1 倍までで scene -> small、2 倍までで small -> medium、3 倍までで medium -> large へ移ります。この考え方にしておくと、焦点面から外れた直後の見え方、stageMask の色分け、Compute Shader 版の samples/compute_dof を同じ感覚で比較できます。

実装例

```
import DofPass from "./webg/DofPass.js";

const dof = new DofPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight(),
  projectionNear: app.projectionNear,
  projectionFar: app.projectionFar,
  focusDistance: 34.0,
  focusRange: 6.0,
  maxBlurMix: 1.0,
  blurScale: 0.5,
```

```
  stageBlurIterations: {
    small: 1,
    medium: 2,
    large: 4
  },
  stagedStageCount: 3,
  blurRadius: 2.4
});
await dof.ready;

app.start({
  onUpdate: ({ screen }) => {
    dof.resizeToScreen(screen);
  },
  onBeforeDraw: () => {
    dof.beginScene(app.screen, app.clearColor);
    app.space.draw(app.eye);
  },
  onAfterDraw3d: () => {
    dof.render(app.screen, {
      clearColor: app.clearColor
    });
    app.screen.clearDepthBuffer();
  }
});
```

DOF を導入する際、最も注意すべき点は sceneTarget がカラー情報だけでなく、サンプリング可能な深度情報を保持している必要があることです。focusRange は DofPass 内部でも stage 1 つ分の距離幅として扱われるため、アプリ側で 3 倍する必要はありません。blurIterations は後方互換のために残っていますが、指定すると small / medium / large の 3 段階すべてに同じ反復回数が適用されます。現在の標準的な使い方では、stageBlurIterations で段階ごとに反復回数を分けます。小さいぼかしは画面解像度に近いターゲットで実行されるため、反復を増やすと負荷が増えやすい一方、見た目の改善は限定的です。そのため、small: 1、medium: 2、large: 4 のように、必要な段階へだけ反復を割り当てる方が扱いやすくなります。

深度デバッグとフォーカスマスク

DOF の調整は、深度条件と合焦範囲が視覚的に分からないと困難です。そのため、内部的に深度（depth）、合焦範囲（focusMask）、段階選択（stageMask）をカラーターゲットへ出力するヘルパーを活用します。`stageMask` では、現在の画素が scene、small、medium、large のどの段階として合成されるかを確認できます。

```
import FullscreenPass from "./webg/FullscreenPass.js";

const debugPass = new FullscreenPass(app.getGPU(), {
  targetFormat: app.getGPU().format
});
await debugPass.init();

let dofView = "composite";

app.start({
  onUpdate: ({ screen }) => {
    dof.resizeToScreen(screen);
  },
  onBeforeDraw: () => {
    dof.beginScene(app.screen, app.clearColor);
    app.space.draw(app.eye);
  },
  onAfterDraw3d: () => {
    dof.render(app.screen, {
      clearColor: app.clearColor
    });

    if (dofView !== "composite") {
      const debugSource = dofView === "scene"
        ? dof.getSceneTarget()
        : dofView === "depth"
          ? dof.getDepthDebugTarget()
          : dofView === "focusMask"
            ? dof.getFocusDebugTarget()
            : dofView === "stageMask"
              ? dof.getStageDebugTarget()
              : dofView === "blurSmall"
                ? dof.getSmallBlurTarget()
```

```
                : dofView === "blurMedium"
                  ? dof.getMediumBlurTarget()
                  : dofView === "blurLarge"
                    ? dof.getLargeBlurTarget()
                    : dof.getSceneTarget();

      app.screen.beginPass({
        clearColor: app.clearColor,
        colorLoadOp: "clear",
        depthView: null
      });
      debugPass.draw(debugSource);
    }

    app.screen.clearDepthBuffer();
  }
});
```

depth と focusMask を切り替えて確認することで、focusDistance（合焦距離）が投影設定（near/far）と整合しているか、どの領域がシャープに維持されているかを即座に判断できます。さらに stageMask と blurSmall、blurMedium、blurLarge を確認すると、ぼかし画像そのものが破綻しているのか、深度に応じた段階選択が意図と違うのかを切り分けられます。

20.4　FrostedGlassPass の実装と活用

半透明の表現を考えるとき、単純にマテリアルの alpha 値を下げてアルファブレンドで描画すれば十分なケースもあるでしょう。しかし、曇りガラスのように「背後のシーンがぼけて見える」表現では、半透明 Shape 自体の色を出すだけでは不十分です。背後のシーンを一度テクスチャとして取り出し、それをぼかした結果を、ガラス面が重なる領域だけに合成する必要があります。

FrostedGlassPass は、この処理をポストプロセスとして実装したクラスです。実際に行っているのは、「不透明シーン」「ぼかしたシーン」「ガラス領域を示すマスク」の 3 つを合成する処理です。ガラス Shape は通常の描画ではなく、GlassMaskShader を用いてマスクターゲットへ描き出されます。このマスクの alpha 値は「ぼかしを混ぜる強度」を、RGB 値は最終的な色に加える「ティント（色付け）」を表します。

この構成の利点は、手前にある不透明オブジェクトとの関係を自然に扱える点です。FrostedGlassPass.beginMask() は、不透明シーンの深度バッファを保持した状態でマスクを描画します。そのため、ガラス板より手前にオブジェクトがある場合、その部分はマスクから除外され、正しく遮蔽されます。

実装の考え方とフロー

FrostedGlassPass の 1 フレームは、通常のポストプロセスより複雑な手順を踏みます。

1. beginScene(): ガラス Shape を除外して、不透明なシーンだけを描画する。
2. beginMask(): ガラス Shape だけを GlassMaskShader でマスクターゲットへ描画する。
3. render(): シーンをぼかし、元のシーン、ぼけたシーン、マスクを合成する。

同じシーングラフ（Space）を使いながら、描画する Shape をパスごとに分けるため、Space.draw(eye, { filter }) を利用して、ガラス Shape の抽出と除外を制御します。

実装例

以下は unittest/translucent の構成をベースにした実装例です。

```
import WebgApp from "./webg/WebgApp.js";
import Shape from "./webg/Shape.js";
import Primitive from "./webg/Primitive.js";
import FrostedGlassPass from "./webg/FrostedGlassPass.js";
import GlassMaskShader from "./webg/GlassMaskShader.js";

const GLASS_MATERIAL_ID = "frosted-glass";

const app = new WebgApp({
  document,
  messageFontTexture: "./webg/font512.png",
  autoDrawScene: false
});
await app.init();

const glassMaskShader = new GlassMaskShader(app.getGPU(), {
```

```
  targetFormat: app.getGPU().format,
  cullMode: "none"
});
await glassMaskShader.init();
glassMaskShader.setProjectionMatrix(app.projectionMatrix);

const frosted = new FrostedGlassPass(app.getGPU(), {
  sceneFormat: app.getGPU().format,
  canvasFormat: app.getGPU().format,
  blurRadius: 2.8,
  blurScale: 0.5,
  blurIterations: 2,
  blurStrength: 0.9,
  tintStrength: 0.28,
  maskPower: 0.9
});
await frosted.ready;

const isGlassShape = (shape) => {
  return shape?.materialId === GLASS_MATERIAL_ID ||
    shape?.shaderParam?.frosted_glass === true ||
    shape?.shaderParam?.frosted_glass === 1;
};

const glassShape = new Shape(app.getGPU());
glassShape.applyPrimitiveAsset(Primitive.cuboid(60.0, 38.0, 0.8));
glassShape.endShape();
glassShape.setShader(glassMaskShader);
glassShape.setMaterial(GLASS_MATERIAL_ID, {
  frosted_glass: true,
  color: [0.78, 0.76, 0.68, 0.74]
});

const glassNode = app.space.addNode(null, "glass");
glassNode.setPosition(0.0, 2.0, -24.0);
glassNode.addShape(glassShape);

app.start({
  onUpdate: ({ screen }) => {
    glassMaskShader.setProjectionMatrix(app.projectionMatrix);
    frosted.resizeToScreen(screen);
  },
```

```
  onBeforeDraw: () => {
    // 1. 不透明シーンの描画（ガラスを除外）
    frosted.beginScene(app.screen, app.clearColor);
    app.space.draw(app.eye, {
      filter: ({ shape }) => !isGlassShape(shape)
    });

    // 2. ガラスマスクの描画（ガラスのみを抽出）
    frosted.beginMask(app.screen);
    app.space.draw(app.eye, {
      filter: ({ shape }) => isGlassShape(shape)
    });
  },
  onAfterDraw3d: () => {
    // 3. 合成と出力
    frosted.render(app.screen, {
      clearColor: app.clearColor
    });
    app.screen.clearDepthBuffer();
  }
});
```

ここで `glassShape.setShader(glassMaskShader)` を呼んでいる点に注目してください。ガラス Shape は通常の質感を描くのではなく、マスクターゲットへ「この領域にガラスが存在する」という情報を書き込みます。`setMaterial()` で設定した `color` の RGB はティント色として、alpha はぼかしを混ぜる強度として利用されます。

パラメータの調整

- `blurRadius`: ぼかしサンプルの間隔を制御します。値を大きくすると背後の形状がより曖昧になります。
- `blurIterations`: ぼかし処理の繰り返し回数です。増やすほど滑らかになりますが、描画負荷が増加します。
- `blurScale`: ぼかし用ターゲットの解像度倍率です。`0.5`（半解像度）に設定すると、処理が軽量化されるだけでなく、より柔らかい表現になる場合があります。
- `blurStrength`: マスク領域において、元のシーンとぼけたシーンをどれだけ混ぜるかを決定します。

- tintStrength: マスクの RGB 色を最終的な出力にどれだけ反映させるかを制御します。
- maskPower: マスクの alpha 効き方を指数的に調整します。

デバッグと注意点

曇りガラスの調整では、最終結果だけでなく中間ターゲット（scene, mask, blur）を切り替えて確認することが不可欠です。これにより、フィルタリングが正しく機能しているか、深度関係が維持されているかを切り分けることができます。

なお、FrostedGlassPass は「全ての半透明ポリゴンを深度順に描画する」ための汎用的な仕組みではなく、あくまで「背後をぼかして見せる」ための合成パスです。水や煙、複数の透明な板を重ねてアルファブレンドしたい場合は、別途描画順の制御が必要になります。

20.5 VignettePass の実装と活用

役割と仕組み

VignettePass は、画面の周辺部の輝度を減衰させる、非常に軽量なエフェクトです。Bloom や DOF のような複雑な中間ターゲットを必要とせず、「1 枚のカラーテクスチャを読み込み、周辺部を暗くする」というシンプルな処理を行います。

実装例

VignettePass 自体は内部でオフスクリーンターゲットを保持しないため、ソースとなるターゲットを別途用意して渡す必要があります。

```
import FullscreenPass from "./webg/FullscreenPass.js";
import VignettePass from "./webg/VignettePass.js";

const sceneTarget = app.screen.createRenderTarget({
  label: "my-scene",
  format: app.getGPU().format,
  hasDepth: true
```

```
});
await sceneTarget.ready;

const rawScenePass = new FullscreenPass(app.getGPU());
await rawScenePass.init();

const vignettePass = new VignettePass(app.getGPU(), {
  centerX: 0.5,
  centerY: 0.5,
  radius: 0.74,
  softness: 0.42,
  strength: 0.92,
  tint: [0.0, 0.0, 0.0, 1.0]
});
await vignettePass.init();

app.start({
  onUpdate: ({ screen }) => {
    sceneTarget.resizeToScreen(screen);
  },
  onBeforeDraw: () => {
    app.screen.clear(sceneTarget);
    app.space.draw(app.eye);
  },
  onAfterDraw3d: () => {
    vignettePass.render(app.screen, {
      source: sceneTarget,
      clearColor: app.clearColor
    });
    app.screen.clearDepthBuffer();
  }
});
```

20.6 ポストプロセスを支える共通部品

各エフェクトの内部で利用されている共通コンポーネントについて解説します。

- RenderTarget: オフスクリーン描画の先となるターゲットです。ポストプロセスの土台となる「描き出し先」を制御します。

- FullscreenPass: 1 枚のテクスチャを現在のパスへ出力するための最小限のヘルパーです。デバッグビューの表示や、エフェクト適用前のシーンとの比較に多用されます。
- SeparableBlurPass: 横方向と縦方向のぼかし処理を交互に繰り返す（ピンポン処理）ことで、効率的にガウスぼかしなどを実現する共有ヘルパーです。BloomPass、DofPass、FrostedGlassPass の内部で利用されています。

20.7 導入パターンと使い分け

初めてポストプロセスを導入する場合は、VignettePass または BloomPass から始めることを推奨します。これらは「オフスクリーン → フルスクリーン → キャンバス」という基本フローを追いやすく、深度情報の参照という複雑な工程を含まないためです。

DofPass は投影行列の near / far やカメラ設定と密接に結びついているため、シーンのカメラ設定が確定した段階で導入してください。FrostedGlassPass の場合は、どの Shape をガラスとして扱うかを material id やシェーダーパラメータで明確に判定できるように設計しておくことが安定運用の鍵となります。

20.8 トラブルシューティング

画面に変化が見られない場合: autoDrawScene: false に設定されているか確認してください。設定されていない場合、シーンが直接キャンバスへ描画され、ポストプロセス処理がスキップされます。

リサイズ後に解像度がずれる場合: onUpdate() 内で resizeToScreen(screen) を呼び出しているか確認してください。オフスクリーンターゲットは、キャンバスのサイズ変更に合わせて適切にリサイズする必要があります。

被写界深度（DOF）が正しく効かない場合: 投影設定の near / far がカメラの設定と不整合を起こしている可能性があります。まずはデバッグビューで depth と focusMask を表示し、意図した領域が合焦しているかを確認してください。

HUD が表示されなくなった場合: 最終パスの後に app.screen.clearDepthBuffer() を呼び出しているか確認してください。これを忘れると、ポストプロセスで書き込まれた深度情報が残り、後から描画する HUD が深度テストで棄却されます。

VignettePass が動作しない場合: VignettePass は内部でソーステクスチャを生成しませ

ん。シーンをあらかじめ別の RenderTarget へ描画し、そのテクスチャを render({ source }) に渡しているか確認してください。

FrostedGlassPass でガラスが見えない場合: 不透明シーンパスとマスクパスのフィルタ（filter）が逆になっていないか確認してください。また、GlassMaskShader の投影行列が通常シェーダーと同じ値に更新されていないと、マスクの位置がずれて表示されます。

20.9 関連サンプルとテスト

- Bloom の標準サンプル (webg/samples/bloom): パラメータ調整 UI、デバッグドック、extractHeat の可視化など、包括的な実装を確認できます。
- DOF の標準サンプル (webg/samples/dof): フォーカスガイド、深度デバッグ、フォーカスマスク、stage mask、3 段階 blur の差異を確認できます。
- 曇りガラスの最小テスト (webg/unittest/translucent): 不透明シーン、ぼかし、マスク、合成を切り替えながら、FrostedGlassPass と GlassMaskShader の関係を確認できます。
- 最小構成のテスト (webg/unittest/vignette): RenderTarget、FullscreenPass、VignettePass の関係性を最短ルートで追うことができます。

第 21 章

パーティクルと軽量エフェクト

画面全体の見え方を制御するポストプロセスに対し、火花や煙のような短命な演出は、シーンの中へ局所的に実装するほうが効果的です。こうした表現を、ボリュームレンダリング（Volume Rendering）のような重いモデルや複雑なアニメーションに頼らずに軽量に実現するために、webg では ParticleEmitter を提供しています。本章では、ビルボード方式のパーティクルをどのように発生させ、WebgApp のフローへどのように統合するかを解説します。

ここで重要なのは、ParticleEmitter が単一のメッシュを制御するクラスではないという点です。これは多数のビルボード型パーティクルを生成し、寿命、速度、重力、減衰を毎フレーム更新しながら描画するヘルパーであり、短命な演出をまとめて管理することが役割です。そのため、本章では「どのようなテクスチャを使うか」という見た目の議論よりも、「どのような条件で、どのタイミングで、どの程度の数を発生させるか」という制御面に焦点を当てます。

また、preset は emit() の省略記法ではない点に注意してください。webg の emit() は、position、positionSpread、velocity、velocitySpread、gravity、color、colorSpread などのパラメータを明示的に受け取ります。preset はテクスチャの見た目と初期設定のセットを定義するものですが、不足した値を発生器側が自動的に補完する設計ではありません。必要なベクトルや色は、実装側ではっきりと指定することが前提となっています。これにより、「何がどの方向へ飛び、どの程度散らばり、どのような色で表示されるか」という演出意図を、コードから直接読み取ることが可能になります。

21.1　パーティクルの活用場面と仕組み

ゲームやデモでは、当たり判定のヒット、アイテムの取得、着地、破壊、煙、きらめきといった短時間の演出が頻繁に必要になります。これらの演出を毎回個別のメッシュやアニメーションで構築すると、負荷が高くなりがちです。そこで有効なのが、常にカメラに向き合う「ビルボード」を多数使い、短い寿命の中で色や大きさを変化させるパーティクルです。局所的で短命な効果を、シーン全体の進行を妨げずに差し込めるのが最大の強みです。

webg の ParticleEmitter は、この用途に特化して設計されています。内部的にパーティクル用の空き枠を一定数保持し、emit() で空き枠に発生情報を書き込み、update() で位置と寿命を更新し、draw() でビルボードとして描画します。必要に応じて影用のビルボードも同時に描画できるため、地面に接する火花や煙も簡潔に表現可能です。単にスプライトを並べるのではなく、「短時間だけ存在する局所エフェクト」を効率的に運用するためのコンポーネントとして活用してください。

21.2　ParticleEmitter の登録と基本設定

通常、パーティクルを利用する場合は、まず WebgApp.createParticleEmitter() を呼び出します。このヘルパーメソッドは ParticleEmitter を生成して GPU の初期化を行い、WebgApp の発生器リストへ登録します。開発者は、パーティクル本体の詳細な初期化よりも、「どの発生器を、どの程度の容量で、どの preset を用いて運用するか」を定義することに集中できます。

```
import WebgApp from "./webg/WebgApp.js";

const app = new WebgApp({
  document,
  messageFontTexture: "./webg/font512.png",
  clearColor: [0.08, 0.10, 0.15, 1.0]
});
await app.init();

const sparkEmitter = await app.createParticleEmitter({
  name: "sparkEmitter",
  maxParticles: 320,
```

```
  useShadow: true,
  groundY: -7.5,
  preset: "spark",
  seed: 31
});
```

上記の例において、maxParticles は同時に生存させることができるパーティクルの最大数です。この値を小さくしすぎると、長寿命のパーティクルが残っている間に新しいエフェクトを発生させることができず、演出が欠落する原因となります。逆に大きくしすぎると不要なメモリを消費するため、演出の同時発生数を見積もって適切な値を設定してください。

useShadow は影用ビルボードを使用するかどうかを決定し、groundY は影を配置する高さを指定します。seed を固定することで、乱数によるばらつきを再現可能にできます。特に spark や dust のように床との接触感が重要な演出では、影用ビルボードがあるだけでシーンのリアルさが大きく向上します。

WebgApp.start() を使用する標準的な構成であれば、ここで発生器を作成しておけば、フレームループ内で updateParticleEmitters() と drawParticleEmitters() が自動的に呼び出されます。これにより、実装側では「いつ発生させるか」というロジックに専念でき、更新や描画のタイミングを個別に管理する必要はありません。

21.3 emit() による発生条件の明示

emit() メソッドは、発生に必要なベクトルや色を明示的に受け取ります。前述の通り、ParticleEmitter は内部で安易な補完を行わないため、演出意図を正確にコードに反映させることができます。

```
const emitSparkBurst = (x, y, z) => {
  sparkEmitter.emit(24, {
    position: [x, y, z],
    positionSpread: [0.3, 0.3, 0.3],
    velocity: [0.0, 12.0, 0.0],
    velocitySpread: [8.0, 4.0, 8.0],
    gravity: [0.0, -18.0, 0.0],
    drag: 0.06,
    life: 0.9,
```

```
    lifeSpread: 0.2,
    size: 1.6,
    sizeSpread: 0.5,
    color: [1.0, 0.85, 0.40, 1.0],
    colorSpread: [0.0, 0.08, 0.10, 0.0],
    shadowAlpha: 0.55,
    shadowScale: 1.1,
    shadowY: -7.5
  });
};
```

各パラメータの役割は以下の通りです。- positionSpread: 発生位置のばらつき- velocitySpread: 飛び出し方向のばらつき- gravity: 時間経過による加速度- drag: 速度の減衰率- life: 寿命（秒数）- size: ビルボードの大きさ

また、shadowAlpha、shadowScale、shadowY で影の見え方を制御します。特に shadowY はシーンの床の高さと一致させる必要があります。ここがずれていると、パーティクル本体が自然であっても影だけが浮いて見えてしまうため、samples/circular_breaker のように定数で管理することを推奨します。

このように、position、velocity、gravity、color およびそれぞれの広がり量を明示することで、「火花は上へ強く飛ぶ」「煙は速度を抑えて寿命を長くする」といった演出の違いを明確に記述でき、バグの発見や調整も容易になります。

21.4 preset の役割と切り替え

ParticleEmitter は spark、smoke、debris、pickup といった preset を備えています。preset の役割は、パーティクル用テクスチャの外見や、演出の初期傾向を定義することです。運用中にこれらを切り替えたい場合は setPreset() を呼び出します。

```
sparkEmitter.setPreset("smoke", {
  texture: {
    width: 96,
    height: 96,
    style: "smoke",
    noiseScale: 5.8,
    noiseAmount: 0.68
```

```
  },
  defaults: {
    life: 1.4,
    lifeSpread: 0.3,
    size: 2.8,
    sizeSpread: 1.0
  }
});
```

setPreset() は必要に応じて手続き生成のビルボード用テクスチャを再生成します。ここでの defaults はあくまで「この演出向けの設定セット」であり、実際の emit() 呼び出しでは依然として明示的なパラメータを使用することが推奨されます。preset は「見た目の系統」を決定し、emit() は「個別の発生条件」を決定するという役割分担になっています。

21.5　描画ループの統合と個別制御

標準的な実装では WebgApp.start() を利用するため、開発者は発生条件を記述するだけで十分です。WebgApp が内部で更新と描画を適切に処理するため、パーティクルの時間更新や描画順を意識することなく、シーンの通常フローに統合されます。

```
app.start({
  onUpdate: ({ deltaSec }) => {
    actor.update(deltaSec);
    if (actor.justHit) {
      emitSparkBurst(actor.hitX, actor.hitY, actor.hitZ);
    }
  }
});
```

一方で、描画順を厳密に制御したい場合や、独自のループ構造を持つ場合は、draw() メソッドを直接呼び出す個別制御が可能です。samples/circular_breaker/particleEffects.js では、この構成を小さなヘルパー関数にまとめて実装しています。

```
const alive = sparkEmitter.draw(app.eye, app.projectionMatrix);
```

この手法は、WebgApp.start() を使わずに独自のレンダリングループを構築する場合に適しています。ただし、WebgApp.start() を使いながら個別に emitter.draw() を呼び出すと、二重に描画されてしまうため注意してください。標準的なフローに乗せるか、個別制御を行うか、どちらか一方に統一して実装してください。

21.6　軽量エフェクトとしての使い分け

パーティクルが最も威力を発揮するのは、短時間で消失する視覚効果です。具体的には、ヒット時の spark、着地時の dust、破壊時の debris、アイテム取得時の pickup glow などが挙げられます。これらは形状よりも時間的な変化が主役となるため、ビルボードによる実装が軽量かつ調整しやすくなります。

逆に、以下のようなケースにはパーティクルは不向きです。- 長時間シーンに残り続けるオブジェクト- 精密な当たり判定を持つオブジェクト- 視点移動に伴い、立体的な形状を見せたいオブジェクト

これらの場合は、Shape やモデルアセットを使用し、必要に応じてアニメーションやマテリアルで演出を行うのが自然です。「局所的で短命な効果」には ParticleEmitter を、「存在感のあるオブジェクト」にはモデルアセットを、という使い分けを意識してください。

21.7　学習のためのサンプルコード

本機能の理解を深めるには、まず unittest/particle_emitter を参照することをお勧めします。ここでは擬似レンダラーによる自動検証と、実画面での視覚確認用サンプルが同一ファイルにまとめられており、preset → emit → update → draw → clear というライフサイクルを追いやすくなっています。

次に、実際のゲームへの組み込み例として samples/circular_breaker を確認してください。ここでは、ヒット時の spark 発生ロジックをゲームプレイ側のコードから切り離して管理する構成が採用されており、「ゲームロジックのどのタイミングでパーティクルを発生させるか」という実践的な実装パターンを学ぶことができます。

このように、まずは unittest/particle_emitter で基本仕様を理解し、その後に samples/circular_breaker で応用例を確認するという段階的な学習が効率的です。

本章で解説したパーティクルによる局所演出を、第 20 章で解説した画面全体を制御するポ

ストプロセスと組み合わせることで、より密度が高く、視覚的に豊かなシーンを構築することが可能になります。

第 22 章

ローレベル API の基礎

ここまでの章では、WebgApp、Scene JSON、ModelAsset など、比較的ハイレベル（高レイヤー）の構成を中心に扱ってきました。第 22 章以降では、その下で実際に描画・変換・メッシュ構築を支えている低レイヤー API を改めて整理します。ハイレベルな WebgApp を利用すると、描画先の初期化やシーンの組み立てを非常に簡潔に記述できます。しかし、その背後では Screen、Shader、CoordinateSystem、Node、Matrix、Quat といったローレベル（低レイヤー）API が、それぞれ独立した役割を担っています。本章では、webg の土台となっているこの中間層を、「GPU 側の入口」「シーン変換の基礎」「数理の土台」という 3 つの観点から整理して解説します。

ここで重要なのは、webg のローレベル API は生の WebGPU をそのまま露出させるのではなく、「3D アプリケーションを構築するための単位」へと抽象化した中間層であるという点です。ブラウザで 3D 描画を行う場合、本来はキャンバスと GPU デバイス、レンダーパイプライン、ユニフォームバッファ、頂点・インデックスバッファ、カメラ行列、モデル変換などを個別に連携させる必要があります。webg では、これらを一つの巨大な API にまとめるのではなく、役割ごとにクラスを分けることで管理しやすくしています。

具体的には、Screen と Shader が GPU に近い「入口」となり、キャンバスやデバイス、パイプライン、ユニフォームの初期化を担います。一方で CoordinateSystem と Node はシーン上の「姿勢と配置」を管理し、位置、回転、親子関係、そして Shape を配置する単位としての役割を分担します。さらに Matrix と Quat は、これらの上位レイヤーが利用する「数理の土台」として、投影行列の計算や行列積、クォータニオン補間などの変換計算を基盤から支えています。

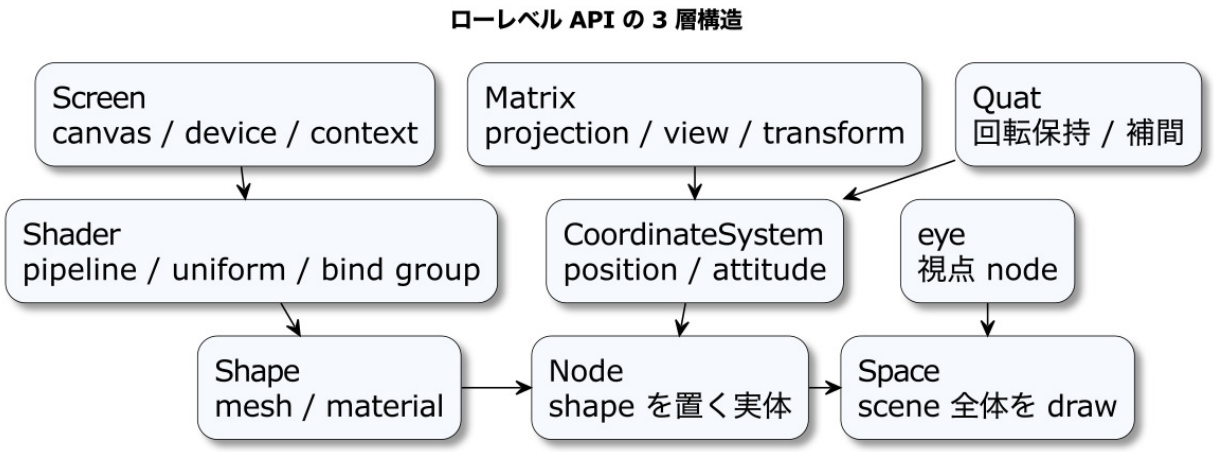


図 22.1: ローレベル API の 3 層構造

本章では、*Screen / Shader* を「*GPU* 側の入口」、*CoordinateSystem / Node / Shape / Space* を「シーン変換と配置」、*Matrix / Quat* を「数理の土台」として捉えることで、各 *API* の役割を体系的に理解することができます。

22.1　GPU 側の入口：Screen と Shader

Screen

`Screen` は、HTML のキャンバス（canvas）と WebGPU の橋渡しを行うクラスです。内部では `navigator.gpu` からアダプター（adapter）とデバイス（device）を取得し、キャンバスの `webgpu` コンテキストを `configure` し、デプス（depth）テクスチャを管理します。つまり `Screen` は、「描画先の窓」と「その窓へ描画するための WebGPU コンテキスト」を統合して管理する、GPU 側入口の最外殻に位置するクラスです。

`Screen` の主な役割は、キャンバスの特定、WebGPU デバイスの確保、コンテキストの構成、リサイズ時のデプステクスチャの再構築、および `clear()` と `present()` のインターフェース提供です。ここで注目すべきは、`Screen` 自体はシェーダーやジオメトリ（geometry）の情報を保持しない点です。役割はあくまで「どこに描くか」という描画先の整備に特化しており、「どう描くか」という描画内容の制御は `Shader` 側に委ねられています。

最小のコード例を以下に示します。

```
import Screen from "./webg/Screen.js";

// Screen が canvas と WebGPU の初期化を担当する
const screen = new Screen(document);

// device と context の準備が終わるまで待つ
await screen.ready;

// 背景色を決める
screen.setClearColor([0.1, 0.15, 0.1, 1.0]);

// Screen から GPU ラッパーを取り出す
const gpu = screen.getGPU();
console.log(gpu.device, gpu.context);
```

この例で重要なのは、await screen.ready を待機するまで、WebGPU デバイスやコンテキストが利用可能な状態ではないということです。ローレベル API を利用する際、デバイスの準備が完了する前にシェーダーやバッファを作成しようとしてエラーになるケースが多く見られます。Screen は非同期初期化を持つクラスであると認識しておく必要があります。

また、resize() メソッドは単にキャンバスのサイズを変更するだけではありません。内部でデプステクスチャの再構築を行うため、画面サイズを変更した後は投影行列（projection matrix）も合わせて更新するという一連の流れで処理を行うのが一般的です。

```
screen.resize(
  Math.max(1, Math.floor(window.innerWidth)),
  Math.max(1, Math.floor(window.innerHeight))
);
```

Shader

WebGPU では、描画のたびにレンダーパイプライン、シェーダーモジュール、ユニフォームバッファ、バインドグループ（bind group）が必要になります。Shader はこれらをクラスとしてまとめる基底層です。Screen が「描画先の窓」を作るのに対し、Shader は「描画手法」を定義します。

webg では Shader 基底クラスを直接使うよりも、通常は SmoothShader のような派生クラスを利用することが一般的です。教材や比較用の book/examples/Phong.js のような補助実装を参照する場面はありますが、現行のコア側で標準入口になっているのは SmoothShader です。Shader が担う具体的な処理は、ユニフォームバッファの作成、WGSL からのシェーダーモジュール生成、バインドグループレイアウトの構築、テクスチャとサンプラーのバインドグループ管理、およびパスエンコーダーへのパイプライン設定です。

Shader は、CPU 側の描画パラメータを行列や光源値などの「GPU が解釈可能な形式」に整えて渡す中継点としての役割を果たします。以下に、派生クラスである SmoothShader を用いた最小のコード例を示します。

```
import Screen from "./webg/Screen.js";
import SmoothShader from "./webg/SmoothShader.js";
import Matrix from "./webg/Matrix.js";

const screen = new Screen(document);
await screen.ready;

// SmoothShader は Shader の派生クラス
const shader = new SmoothShader(screen.getGPU());
await shader.init();

// projection 行列を shader へ渡す
const projection = new Matrix();
projection.makeProjectionMatrix(
  0.1,
  1000.0,
  screen.getRecommendedFov(55.0),
  screen.getAspect()
);
shader.setProjectionMatrix(projection);

// 光源位置も shader 側へ渡す
shader.setLightPosition([120.0, 180.0, 140.0, 1.0]);
```

この例のように、投影行列や光源位置は Matrix や配列として CPU 側で計算し、それを Shader が GPU へ転送します。Shader 自体が数学的な計算を行うのではなく、計算済みの結果を描画設定に反映させる役割を担っています。

22.2　シーン変換と配置：CoordinateSystem と Node

CoordinateSystem

CoordinateSystem は、シーングラフにおける変換の基底となるクラスです。位置、回転、スケール、および親子関係を保持し、必要に応じてローカル行列とワールド行列を再構築します。ここで重要なのは、CoordinateSystem 自体は描画機能を持たないということです。見えるオブジェクトではなく、空間上の「姿勢」を定義するための基底として機能します。

CoordinateSystem が提供する機能は、座標（position）の保持、クォータニオンによる回転の管理、一様スケールの設定、親子関係の構築、およびローカル/ワールド行列の計算です。このクラスは、シェーダーや形状（shape）、GPU の情報を一切持たず、純粋に「3D 空間上のどこに、どちらを向いて、どの親の下に配置されるか」という姿勢情報のみを管理します。

最小のコード例は次のとおりです。

```
import CoordinateSystem from "./webg/CoordinateSystem.js";

const cs = new CoordinateSystem(null, "marker");

// 位置を決める
cs.setPosition(2.0, 1.0, 0.0);

// yaw(Y), pitch(X), roll(Z) の順で回転を決める
cs.setAttitude(45.0, 15.0, 0.0);

// uniform scale を設定する
cs.setScale(1.5);

// 行列を更新する
cs.setWorldMatrix();

console.log(cs.getWorldPosition());
console.log(cs.getWorldAttitude());
```

特に注意すべき点は、回転の適用順が yaw（Y軸）→ pitch（X軸）→ roll（Z軸）であることです。これは webg 全体における共通規約となっており、後の Matrix や Quat の処理に

おいても一貫して適用されます。

Node

Node は、CoordinateSystem を継承し、そこに Shape の保持と描画機能を追加したクラスです。シーンにオブジェクトを配置する実体と言えます。CoordinateSystem が「変換のみの基底」であるのに対し、Node はその変換の上に「何を描画するか」という実体を載せたものです。

ローレベルな実装では、Space.addNode() を通じて Node を生成し、そこに Shape を割り当てます。視点（カメラ）も通常は Node として配置されます。つまり Node は、可視オブジェクトに限らず、カメラリグやピボット、オフセット用の親ノードなど、「シーンに存在するあらゆる要素」を表す汎用的なクラスです。

Node が提供する主な機能は、Shape の保持、CoordinateSystem と同様の変換 API の利用、親子階層の構築、および自身と子孫ノードの再帰的な描画処理です。

最小のコード例は次のとおりです。

```
import Space from "./webg/Space.js";

const space = new Space();

// 視点も Node として置く
const eye = space.addNode(null, "eye");
eye.setPosition(0.0, 0.0, 10.0);
space.setEye(eye);

// 描画オブジェクト用 Node
const box = space.addNode(null, "box");
box.setPosition(0.0, 0.0, 0.0);
box.setAttitude(0.0, 20.0, 0.0);
box.addShape(shape);
```

ここで、Shape はジオメトリとマテリアルを持つ「部品」であり、それをシーンに配置して位置や親子関係を与える実体が Node であるという関係性を理解してください。Shape 単体ではシーンに属さず、Node に紐付いて初めて空間上の位置を持ちます。

22.3 数理の土台：Matrix と Quat

Matrix

3D グラフィックスでは、回転、移動、投影、ビュー変換のすべてを行列で処理します。webg の Matrix は、列優先（column-major）の 4×4 行列です。CoordinateSystem、Node、Shader、Space の内部ではすべてこの Matrix が利用されており、あらゆる変換処理の最終的な集約点となっています。

Matrix が担う機能は、4×4 行列の保持、オイラー角やクォータニオンからの回転行列生成、平行移動の適用、投影行列およびビュー行列の作成、ベクトルへの行列適用などです。特筆すべきは、同一の Matrix クラスが「カメラ用の投影行列」としても「オブジェクト用の変換行列」としても利用される点です。用途に応じて使い分けるため、Shader や CoordinateSystem の処理と密接に連携します。

オブジェクト変換の最小例を以下に示します。

```
import Matrix from "./webg/Matrix.js";

const m = new Matrix();

// yaw, pitch, roll から回転を作る
m.setByEuler(30.0, 10.0, 0.0);

// 最後の列へ平行移動を入れる
m.position([2.0, 1.0, 0.0]);

// 点 [0, 0, 1] をこの行列で動かしてみる
const moved = m.mulVector([0.0, 0.0, 1.0]);
console.log(moved);
```

投影行列を作成する例は次のとおりです。

```
const projection = new Matrix();
projection.makeProjectionMatrix(
  0.1,
```

```
  1000.0,
  screen.getRecommendedFov(55.0),
  screen.getAspect()
);
```

このように、行列を生成する操作自体は共通ですが、それをどの API（Shader や Space など）に渡すかによって、その意味（投影なのかモデル変換なのか）が決まります。

Quat

Quat は、回転を表現するためのクォータニオン（四元数）です。webg では [w, x, y, z] の順で値を保持します。CoordinateSystem は内部的にクォータニオンを用いて姿勢を保持し、描画時にそれを Matrix へ変換します。つまり Quat は、姿勢の保持やアニメーションの補間を効率的に行うための内部的な部品としての役割を担っています。

Quat が提供する主な機能は、各軸の回転クォータニオン生成、オイラー角からの変換、クォータニオン同士の積、slerp による球面線形補間、および行列との相互変換です。

ここで重要なのは、Quat は「向きを保持するための形式」であり、最終的な描画には Matrix への変換が必要であるという点です。保持と補間は Quat が担い、最終的な変換適用は Matrix が担うという役割分担になっています。

最小のコード例を以下に示します。

```
import Quat from "./webg/Quat.js";
import Matrix from "./webg/Matrix.js";

const q = new Quat();

// yaw(Y), pitch(X), roll(Z) から quaternion を作る
q.eulerToQuat(45.0, 10.0, 0.0);

// 行列へ変換して確認する
const m = new Matrix();
m.setByQuat(q);

console.log(q.q);            // [w, x, y, z]
console.log(m.mat.slice(0)); // 4x4 matrix
```

この例から分かるように、クォータニオンを生成しただけでは描画に利用できず、最終的に行列へと落とし込む必要があります。これがアニメーションや姿勢補間を実装する際の重要なポイントとなります。

22.4 全体をつないだ最小例

ここでは、これまで解説した Screen、Shader、Shape、CoordinateSystem、Node、Matrix、Quat のすべての役割がどのように連携するかを、一つのコード例で示します。

```
import Screen from "./webg/Screen.js";
import Space from "./webg/Space.js";
import Shape from "./webg/Shape.js";
import Primitive from "./webg/Primitive.js";
import Matrix from "./webg/Matrix.js";
import Quat from "./webg/Quat.js";
import CoordinateSystem from "./webg/CoordinateSystem.js";
import SmoothShader from "./webg/SmoothShader.js";

const screen = new Screen(document);
await screen.ready;
screen.setClearColor([0.1, 0.15, 0.1, 1.0]);

// Shader は WebGPU の pipeline と uniform をまとめる
const shader = new SmoothShader(screen.getGPU());
await shader.init();

// projection は Matrix で作る
const projection = new Matrix();
const updateProjection = () => {
  screen.resize(
    Math.max(1, Math.floor(window.innerWidth)),
    Math.max(1, Math.floor(window.innerHeight))
  );
  projection.makeProjectionMatrix(
    0.1,
    1000.0,
    screen.getRecommendedFov(55.0),
    screen.getAspect()
  );
  shader.setProjectionMatrix(projection);
```

```
};
updateProjection();

// シーン全体を管理する
const space = new Space();

// 視点 Node を置く
const eye = space.addNode(null, "eye");
eye.setPosition(0.0, 0.0, 12.0);
space.setEye(eye);

// CoordinateSystem は変換の下書きに使える
const pose = new CoordinateSystem(null, "cubePose");
pose.setPosition(0.0, 0.0, 0.0);
pose.setAttitude(20.0, 10.0, 0.0);

// Quat を直接作って回転を確認する例
const spin = new Quat();
spin.setRotateY(1.0);

// Shape は geometry と material を持つ
const shape = new Shape(screen.getGPU());
shape.setShader(shader);
shape.applyPrimitiveAsset(Primitive.cube(3.0, shape.getPrimitiveOptions()));
shape.endShape();
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [1.0, 0.5, 0.3, 1.0]
});

// Node は Shape をシーンに置く
const cube = space.addNode(null, "cube");
cube.setPosition(...pose.getPosition());
cube.setAttitude(...pose.getLocalAttitude());
cube.addShape(shape);

const loop = () => {
  // Node 側の API で回転を足していく
  cube.rotateY(0.8);
  cube.rotateX(0.3);
```

```
  screen.clear();
  space.draw(eye);
  screen.present();
  requestAnimationFrame(loop);
};
loop();
```

このコードの流れを整理すると、Screen が WebGPU の窓を用意し、Shader が描画手法を定義し、Matrix が投影行列を計算し、Quat が回転の土台となり、CoordinateSystem が位置と向きを定義し、Shape が形状を定義し、最後に Node がそれらをシーンに配置して実体化させています。このように、ローレベル API は「GPU 側の入口」から始まり、「変換の土台」を経て、「シーンに配置する実体」へと繋がる構造になっています。

22.5　ローレベル API を利用する判断基準

ローレベル API を直接利用するのが適しているのは、以下のようなケースです。- 描画の基盤となる仕組みを深く理解したいとき- 独自のレンダリングパスやカスタムシェーダーを構築する前段階であるとき- WebgApp が内部でどのような自動化を行っているかを確認したいとき

つまり、「素早く何かを構築すること」よりも、「どのレイヤーで何が起きているかを正確に把握すること」を優先する場合に有効です。ローレベル API を学ぶ価値は、すべてを手書きすることではなく、ハイレベル API の内部挙動を見通せるようになることにあります。

一方で、サンプルアプリケーションを迅速に作成したい場合や、HUD、ユーザー入力、デバッグ機能、カメラリグなどの便利な機能を統合して利用したい場合は、WebgApp を利用するのが自然です。ローレベル API は学習と拡張のための土台であり、日常的なアプリケーション開発においては必ずしも最短経路ではありません。WebgApp による自動化の恩恵を受けつつ、必要なときだけローレベルな制御に降りるという使い方が最も効率的です。

22.6　特に注意すべき重要なポイント

ローレベル API を扱う際に、特に混同しやすく注意が必要なポイントを以下にまとめます。

1. 非同期初期化の待機: `await screen.ready` および `await shader.init()` を忘れないこと。

2. Shape の確定: `shape.endShape()` を呼び出して形状を確定させること。
3. Node による実体化: `Shape` 単体では表示されず、必ず `Node` に追加してシーンに配置すること。
4. 描画の実行: `Node` を作成しても、`Space.draw(eye)` を呼び出さなければ描画されないこと。
5. 行列の形式: `Matrix` は列優先（column-major）であること。
6. クォータニオンの順序: `Quat` は `[w, x, y, z]` の順で保持すること。
7. 回転の適用順: 回転順は `yaw (Y)` → `pitch (X)` → `roll (Z)` であること。

これらの基本仕様をあらかじめ固定して理解しておくことで、実装時の混乱を避けることができます。

本章で `Screen`、`Shader`、`CoordinateSystem`、`Node`、`Matrix`、`Quat` の骨格を把握したことで、次章の `Shape`（ジオメトリとマテリアル）の詳細な解説に進む準備が整いました。本章をローレベル API 全体の地図として活用し、各機能のつながりを意識しながら学習を進めてください。

第 23 章

Shape によるメッシュ構築

Shape は、単に 3D の形状を格納するコンテナではありません。CPU 側で頂点や面を編集する「構築段階」と、GPU 側で描画可能なメッシュとして確定させる「描画段階」の境界を司るクラスです。本章では、高水準なプリミティブの利用から、低水準な頂点登録までを通して、Shape がどのようにメッシュ構築の中心として機能しているかを見ていきます。

まず押さえておきたいのは、webg において Shape を利用する経路には 2 つのアプローチがあることです。1 つは Primitive.cube() のように、あらかじめ定義された形状を流し込んで素早く構築する「高水準（高レイヤー）」の経路。もう 1 つは、頂点と面を一つずつ登録して組み立てる「低水準（低レイヤー）」の経路です。

どちらの経路を選択したとしても、最終的に endShape() を呼び出すことで、CPU 側の配列として保持していたメッシュデータが GPU 側の描画可能なメッシュへと切り替わります。つまり、本章の核心は Shape というクラスそのものの仕様よりも、「どのようにメッシュを組み立て、どのタイミングで描画フェーズへと移行させるか」というフローを理解することにあります。

また、メッシュ構築において極めて重要なのが「頂点共有」の設計です。滑らかな曲面を表現したい場合は頂点を共有し、エッジの立った鋭い形状を表現したい場合は、角で頂点を複製して分離させます。法線は単に面ごとに固定されるのではなく、頂点に集まる各面の法線寄与を加算し、最終的に endShape() で正規化されるため、頂点を共有するかどうかがそのまま陰影の見た目に直結します。つまり、メッシュ構築とは単に「形を作る」作業ではなく、「どのような陰影で表現したいか」という視覚的な設計を行う工程でもあるのです。

23.1 Shape の役割と基本構造

Shape は、頂点、インデックス、法線、UV、マテリアル、およびシェーダーを統合的に管理するクラスです。WebGPU の視点で見れば、頂点バッファとインデックスバッファを生成する直前のデータ集合体といえます。概念的に捉えると、「何を描画するかを定義する 3D 形状そのもの」と考えると分かりやすいでしょう。

ここで注目すべきは、Shape がジオメトリ（形状）だけでなく、マテリアルやシェーダーとの接点も持っている点です。ただし、本章ではメッシュ構築に焦点を当てるため、マテリアルや描画の詳細については必要最小限の記述に留めます。

以下に、高水準な経路を用いた最小の実装例を示します。

```
import Shape from "./webg/Shape.js";
import Primitive from "./webg/Primitive.js";

// Shape は GPU 参照を受けて生成する
const shape = new Shape(screen.getGPU());

// Primitive から cube のデータを流し込む
shape.applyPrimitiveAsset(
  Primitive.cube(2.0, shape.getPrimitiveOptions())
);

// ここで GPU バッファを確定させる
shape.endShape();

// マテリアルを指定する
shape.setMaterial("smooth-shader", {
  use_texture: 0,
  color: [1.0, 0.5, 0.3, 1.0]
});
```

この例では、利用者が頂点や面の構成を記述することなく、Primitive が生成したデータを Shape へ流し込んでいます。それでも最後に endShape() の呼び出しが必要なのは、高水準・低水準どちらの経路であっても、GPU バッファを確定させるための共通の関門を通る必要があるためです。

23.2　メッシュ構築の 2 つのアプローチ

高水準（高レイヤー）な作成手法

高水準な作成手法は、すでに形状データがまとまっている場合に適しています。既製のプリミティブやモデルアセットを利用する場合、この経路が最も自然です。

```
const shape = new Shape(screen.getGPU());
shape.applyPrimitiveAsset(
  Primitive.cube(2.0, shape.getPrimitiveOptions())
);
shape.endShape();
```

この手法では、頂点や面の構成は Primitive 側で完結しています。利用者は「どのプリミティブを使用するか」を選択するだけであり、形状の編集よりも「適切な既製メッシュの選択」が中心となります。

低水準（低レイヤー）な作成手法

一方で、プロシージャルジオメトリの生成や、特殊な形状を構築する場合は、低水準な作成手法を用います。ここでは利用者が自らメッシュを組み立てることになります。

```
const shape = new Shape(screen.getGPU());

const p0 = shape.addVertexUV(-1.0, -1.0, 0.0, 0.0, 0.0) - 1;
const p1 = shape.addVertexUV( 1.0, -1.0, 0.0, 1.0, 0.0) - 1;
const p2 = shape.addVertexUV( 1.0,  1.0, 0.0, 1.0, 1.0) - 1;
const p3 = shape.addVertexUV(-1.0,  1.0, 0.0, 0.0, 1.0) - 1;

shape.addTriangle(p0, p1, p2);
shape.addTriangle(p0, p2, p3);

shape.endShape();
```

この手法では、頂点と面の定義に関するすべての責任を利用者が担います。webg は登録されたデータを保持し、法線計算や GPU への転送処理を担当します。つまり、低水準な手法を用いる場合、Shape はジオメトリを定義する設計図そのものとして機能します。

23.3　低水準 API による詳細なメッシュ構築

低水準な構築では、まず頂点を登録し、次にそれらを結んで面を構成するという手順を踏みます。

頂点の登録

Shape は頂点ごとにポジション（座標）、法線、UV を保持します。内部では positionArray、normalArray、texCoordsArray といった配列にデータが蓄積されていきます。ここで重要なのは、「頂点の登録」と「法線の確定」は別段階であるということです。法線は面を貼った後に計算され、最終的に endShape() で正規化されて初めて正しい描画方向として確定します。

頂点を登録するためのメソッドには、用途に応じていくつかの選択肢があります。

- setVertex(x, y, z): ポジションを登録し、同時に法線用の初期値 [0, 0, 0] を追加します。返り値は「追加後の頂点数」であるため、0 始まりのインデックスとして利用する場合は - 1 する必要があります。
- addVertex(x, y, z): setVertex() に加え、現在のテクスチャマッピング設定から UV を自動計算して追加します。
- addVertexUV(x, y, z, u, v): ポジションと UV を明示的に指定して登録します。単純な四角形や教材用のメッシュを作成する際に最も安全で分かりやすいメソッドです。
- addVertexPosUV([x, y, z], [u, v]): 配列形式でデータを渡します。ループ処理を用いて大量の頂点を生成するプロシージャルジオメトリの構築において、コードの可読性を高めることができます。

面の登録

頂点だけでは単なる「点の集合」に過ぎません。これらを三角形として描画するには、どの頂点 3 つで 1 枚の面を作るかを定義する必要があります。

- `addTriangle(p0, p1, p2)`: 内部の `indicesArray` にインデックスを追加します。単なる登録に留まらず、`autoCalcNormals` が有効な場合は、外積を用いて面法線を算出し、その寄与分を 3 つの頂点それぞれの法線へ加算します。この段階ではまだ正規化は行わず、複数の面からの寄与を蓄積させます。これがスムーズシェーディングを実現する仕組みの本質です。
- `addPlane(indices)`: 先頭の頂点を基準に扇状に三角形分割を行う補助メソッドです。四角形だけでなく、凸多角形を簡易的に三角形化したい場合に利用できます。

実装例：四角形を構築する

以下に、プリミティブを使用せず、4 つの頂点と 2 つの三角形で平面を構築する最小例を示します。

```
const shape = new Shape(screen.getGPU());
shape.setShader(shader);

const p0 = shape.addVertexUV(-1.0, -1.0, 0.0, 0.0, 0.0) - 1;
const p1 = shape.addVertexUV( 1.0, -1.0, 0.0, 1.0, 0.0) - 1;
const p2 = shape.addVertexUV( 1.0,  1.0, 0.0, 1.0, 1.0) - 1;
const p3 = shape.addVertexUV(-1.0,  1.0, 0.0, 0.0, 1.0) - 1;

// 面を 2 枚の三角形として登録
shape.addTriangle(p0, p1, p2);
shape.addTriangle(p0, p2, p3);

shape.endShape();
shape.setMaterial("smooth-shader", {
  use_texture: 0,
  color: [0.90, 0.72, 0.32, 1.0]
});
```

このコードでは、座標、UV、面の張り方、そして endShape() を呼ぶタイミングまで、すべてを制御しています。

立方体への応用と設計思想

立方体を構築する場合、一度に全てを作るのではなく、「1 面ずつ追加する」アプローチが理解を助けます。前面、背面、左右面、上下面の順に構築していく流れです。

ここで設計上の判断が必要になります。各面の法線を明確に分けて「角を立たせたい」場合は、面ごとに頂点を複製して共有させない設計にします。逆に、球体やチューブのように「滑らかに見せたい」場合は、面同士で頂点を共有させ、法線寄与を混ぜ合わせます。つまり、メッシュ構築とは「形状の定義」と同時に「頂点共有による陰影の設計」を行う作業なのです。

23.4 endShape() による GPU への転送処理

endShape() は、CPU 上で編集していたジオメトリを GPU へ渡し、描画可能な形式に変換する最終確定ステップです。このメソッドが呼ばれることで、Shape は「編集対象」から「描画対象」へとその役割を変えます。内部では以下のような高度な処理が実行されています。

法線の正規化とシーム処理

addTriangle() で加算された法線は、面数に応じて長さが異なります。そのため、endShape() の段階で全ての頂点法線を正規化し、単位ベクトルへと揃えます。

また、Shape は altVertices という補助データを保持しています。これは UV シーム（テクスチャの継ぎ目）などで、「位置は同じだが属性が異なるため別頂点として扱う」ペアを管理するものです。endShape() や syncAltVertexNormals() はこの対応関係を利用して法線寄与を同期させ、シーム部分でライティングが不自然に途切れないように制御しています。

パッキングされた頂点バッファの生成

GPU が効率的に読み込めるよう、endShape() はバラバラに保持していた配列を 1 本のパッキングされた頂点列（vObj）に詰め直します。通常のメッシュでは、1 頂点あたり以下の 8 つの float 値が並ぶ構成となります。

[pos.x, pos.y, pos.z, normal.x, normal.y, normal.z, u, v]

このデータのストライドは 8 * Float32Array.BYTES_PER_ELEMENT となります。このパッキングされた配列を用いて GPUBufferUsage.VERTEX バッファが作成され、queue.writeBuffer() を通じて GPU へ転送されます。なお、スキンメッシュ（hasSkeleton が有効な場合）では、ボーンインデックスやウェイトを別スロットに分けた専用のレイアウトが構築されます。

インデックスバッファとワイヤーフレームの構築

インデックスバッファの生成においては、メモリ効率と互換性を考慮し、インデックス値に応じて uint16 と uint32 を自動的に切り替えます。全てのインデックスが 65535 以下であれば uint16 を使用し、それを超える場合や 4 バイトアライメントが必要な場合は uint32 を選択します。

さらに、endShape() は内部で _buildWireIndexBuffer() を呼び出し、三角形のインデックスから重複するエッジを除去して、ワイヤーフレーム表示専用のインデックスバッファを別途構築します。これにより、setWireframe(true) を呼び出すだけで、同一のメッシュを用いて面描画と線描画を切り替えることが可能になります。

23.5 endShape() の後に残るものと destroy()

endShape() を呼んだ時点で、Shape は単なる編集中の配列ではなく、描画に必要な GPU バッファを持つ runtime resource へ移行します。ここで重要になるのは、「表示される shape」と「その背後にある shared resource」を分けて考えることです。webg の Shape は、見た目の差分や表示状態を持つインスタンス層として振る舞いながら、ジオメトリや GPUBuffer は ShapeResource として共有できる構造になっています。

この構造を理解すると、破棄の考え方も整理しやすくなります。ある Shape インスタンスが不要になったとしても、同じ shared resource を参照している別の Shape がまだ存在しているなら、resource 自体は残っているべきです。逆に、どこからも参照されなくなった shared resource は、その時点で明示的に寿命を終えられる方が安全です。

```
const sourceShape = new Shape(app.getGPU());
sourceShape.applyPrimitiveAsset(
  Primitive.cube(2.0, sourceShape.getPrimitiveOptions())
```

```
);
sourceShape.endShape();

const shape = sourceShape.createInstance();
const node = app.space.addNode(null, "box");
node.addShape(shape);

// 使い終わったら instance を終了する
shape.destroy();

// 元の shared resource も今後使わないなら source 側も終了する
sourceShape.destroy();
```

shape.destroy() は、この shape instance がもう不要であることを webg に伝える API です。これにより、node に載っていた instance が取り外され、shape 側の material、animation、texture 参照も整理されます。shared resource を参照している instance が他に残っていなければ、対応する GPUBuffer も破棄対象になります。つまり destroy() は「表示を消す」ための API ではなく、「この shape の寿命を終える」ための API です。

ここで注意したいのは、scene から見えなくすることと resource を破棄することは同じではない、という点です。たとえば hide(true) は表示を止めるだけであり、shape と GPUBuffer は残っています。また、node の親子関係を変えるだけでも、メッシュ resource 自体は残ったままです。メモリや GPU resource まで含めて後始末したいときは、destroy() を使って寿命を明示する必要があります。

一方で、今後も再利用する予定の shape まで不用意に destroy() する必要はありません。ステージ中ずっと使い回す地形、何度も出現する同一形状の弾や障害物、複数 node で共有するベースメッシュなどは、shared resource として保持した方が自然です。つまり destroy() は毎回の儀式ではなく、「この shape はもう不要である」と判断できたときに使う終了操作と捉えてください。

また、JavaScript の参照が切れれば最終的に GC の対象になる可能性はありますが、いつ GPU 側 resource が解放されるかは保証されません。長時間動作するアプリや、shape を大量に生成・破棄するツールでは、その不確定さがそのままメモリ使用量の増加として現れやすくなります。そのため webg では、不要になった Shape は destroy() で明示的に終了させる運用を推奨します。

endShape() が生成完了を意味するのに対し、destroy() は寿命の終了を意味します。こ

の 2 つを対として理解しておくと、低レイヤーで shape を構築する場合でも、描画の開始と後始末の両方を見通しよく設計できるようになります。

23.6 学習のステップ

低レイヤーの仕組みをより深く理解するためには、以下の順序で小さな課題に取り組むことを推奨します。

1. 三角形 1 枚を構築する
2. 四角形を 2 つの三角形で構築する
3. 立方体の 6 面を個別に構築する
4. 円周上に頂点を配置し、円板を構築する
5. シームを持つ形状（円筒やトーラス）を構築する

このステップを踏むことで、頂点インデックスの概念、法線の挙動、UV シームの必要性、そして `endShape()` が果たす役割を自然に理解できるはずです。

23.7 まとめ

`Shape` は単なる図形クラスではなく、メッシュの「編集フェーズ」と「描画フェーズ」を繋ぐ境界線としての役割を担っています。

プリミティブを利用する高水準（高レイヤー）な経路と、頂点から積み上げる低水準（低レイヤー）な経路のどちらを通ったとしても、最終的に `endShape()` という関門を通ることで、CPU 上のデータが GPU のバッファへと変換されます。

特に低水準な構築においては、「頂点をどこで共有し、どこで分離させるか」という設計が、そのまま最終的な陰影の表現に直結します。`addTriangle()` による法線寄与の蓄積から、`endShape()` による正規化、パッキング、そして GPU 転送に至る一連の流れを理解することで、より高度なプロシージャルジオメトリの構築が可能になります。

第 24 章

プロシージャル形状の作り方

ローレベル（低レイヤー）API を学ぶ価値は、組み込みのプリミティブを再実装することにあるのではなく、既製の形状では扱いにくい複雑な構造を自ら設計し、メッシュとして成立させられる点にあります。本章では、「icosphere（正二十面体ベースの球）」「メビウス帯」「シェルピンスキー四面体」「フラクタル地形」の 4 つを題材に、プロシージャル形状を構築する際に習得すべき要点を整理します。

ここでまず押さえたいのは、低レイヤーの API を触る目的は、単に「頂点を手動で定義する練習」をすることではないという点です。組み込み形状を少し組み替えただけでは作りにくい形を題材にしてはじめて、Shape クラスの低レイヤーな経路が何のために存在するのかが明確になります。つまり、本章の主眼は API の暗記ではなく、「既製形状では実現困難な構造を、どのように組み立てるか」という設計思想を学ぶことにあります。

また、扱う題材によって得られる知見はそれぞれ異なります。icosphere ではサブディビジョン（細分割）と頂点共有を、メビウス帯ではパラメトリック曲面とシーム（継ぎ目）の処理を、シェルピンスキー四面体では再帰処理とシャープな面構成を、そしてフラクタル地形ではハイトフィールドと頂点グリッドの扱いを学びます。すべては「プロシージャル形状」という共通点を持っていますが、注目すべき論点は異なります。この違いを意識することで、学習内容をより体系的に整理できるはずです。

学習にあたっては、本文を読むだけでなく、提供しているサンプルコードと往復しながら進めてください。コード例は実際のサンプルと対応しており、「コードを読む → サンプルで見た目を確認する → 再びコードに戻る」というサイクルを繰り返すことで、画面上で何が起きているのかを直感的に理解できるようになります。

24.1　学習のステップ：icosphere とメビウス帯

円筒や UV シームを持つ球の構造を理解した後は、「組み込み形状の変形だけでは作りにくい形」を題材にすることで学習効果が高まります。候補としては、icosphere、メビウス帯、シェルピンスキー四面体、経路に沿ったチューブ、ノイズベースのハイトフィールド地形、星形多角形の押し出しなどが挙げられます。

この中でも、特に最初に取り組むべきは icosphere とメビウス帯です。この 2 つは、「頂点共有とトポロジー」および「パラメトリック曲面とシーム」という、低レイヤーの形状構築における 2 大基本要素を明確に分けて学べるため、非常にバランスの良い導入となります。

icosphere：サブディビジョンと共有トポロジー

icosphere は、正二十面体から始めて各三角形を 4 分割し、頂点を球面へと押し戻すことで作成するメッシュです。通常の緯度経度ベースの球体よりも三角形の密度が均一であるため、惑星のメッシュや滑らかな岩のような形状に適しています。

この題材で重要なのは、「頂点を共有しながら面を増やすこと」「同じエッジの中点を二重に生成しないこと」、そして「法線を位置から直接算出すること」の 3 点です。単に球を作るのではなく、「トポロジーの整合性を保ったままサブディビジョンを進める」プロセスに注目してください。

```
import Shape from "./webg/Shape.js";

function buildIcosphere(shape, radius = 1.0, subdivisions = 2) {
  // 球では法線が「中心から外へ向く方向」と一致するため、
  // addTriangle() の面法線加算ではなく、位置から直接 normal を与える
  shape.setAutoCalcNormals(false);

  const vertices = [];
  let faces = [];

  const addSphereVertex = (x, y, z) => {
    const len = Math.hypot(x, y, z);
    const nx = x / len;
    const ny = y / len;
```

```
  const nz = z / len;

  // addVertex() は球面 UV を既定設定から自動計算してくれる
  const index = shape.addVertex(nx * radius, ny * radius, nz * radius) - 1;
  shape.setVertNormal(index, nx, ny, nz);
  vertices.push([nx * radius, ny * radius, nz * radius]);
  return index;
};

const t = (1.0 + Math.sqrt(5.0)) * 0.5;

// 正二十面体の 12 頂点
addSphereVertex(-1,  t,  0);
addSphereVertex( 1,  t,  0);
addSphereVertex(-1, -t,  0);
addSphereVertex( 1, -t,  0);

addSphereVertex( 0, -1,  t);
addSphereVertex( 0,  1,  t);
addSphereVertex( 0, -1, -t);
addSphereVertex( 0,  1, -t);

addSphereVertex( t,  0, -1);
addSphereVertex( t,  0,  1);
addSphereVertex(-t,  0, -1);
addSphereVertex(-t,  0,  1);

// 正二十面体の 20 面
faces = [
  [0, 11, 5], [0, 5, 1], [0, 1, 7], [0, 7, 10], [0, 10, 11],
  [1, 5, 9], [5, 11, 4], [11, 10, 2], [10, 7, 6], [7, 1, 8],
  [3, 9, 4], [3, 4, 2], [3, 2, 6], [3, 6, 8], [3, 8, 9],
  [4, 9, 5], [2, 4, 11], [6, 2, 10], [8, 6, 7], [9, 8, 1]
];

const getMidpoint = (cache, ia, ib) => {
  const key = ia < ib ? `${ia}:${ib}` : `${ib}:${ia}`;
  if (cache.has(key)) {
    return cache.get(key);
  }

  const a = vertices[ia];
```

```
    const b = vertices[ib];
    const mx = (a[0] + b[0]) * 0.5;
    const my = (a[1] + b[1]) * 0.5;
    const mz = (a[2] + b[2]) * 0.5;
    const index = addSphereVertex(mx, my, mz);
    cache.set(key, index);
    return index;
  };

  for (let level = 0; level < subdivisions; level++) {
    const nextFaces = [];
    const midpointCache = new Map();

    for (let i = 0; i < faces.length; i++) {
      const [a, b, c] = faces[i];
      const ab = getMidpoint(midpointCache, a, b);
      const bc = getMidpoint(midpointCache, b, c);
      const ca = getMidpoint(midpointCache, c, a);

      // 1 三角形を 4 三角形へ分割する
      nextFaces.push([a, ab, ca]);
      nextFaces.push([b, bc, ab]);
      nextFaces.push([c, ca, bc]);
      nextFaces.push([ab, bc, ca]);
    }

    faces = nextFaces;
  }

  for (let i = 0; i < faces.length; i++) {
    const [a, b, c] = faces[i];
    shape.addTriangle(a, b, c);
  }

  return {
    vertexCount: vertices.length,
    faceCount: faces.length
  };
}

const shape = new Shape(screen.getGPU());
shape.setShader(shader);
```

```
const info = buildIcosphere(shape, 2.0, 2);
shape.endShape();
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [0.52, 0.78, 0.96, 1.0]
});

console.log(info); // { vertexCount, faceCount }
```

このコードのポイントは、addSphereVertex() で位置を正規化してから半径を乗算している点、法線に球の中心から外側へ向かう方向を直接使用している点、そして midpointCache を用いて同じエッジの中点が重複して生成されるのを防いでいる点です。もしこのキャッシュがなければ、頂点共有が失われ、メッシュのトポロジーが崩れてしまいます。icosphere を学ぶ最大の意義は、まさにこの「共有すべき頂点を適切に管理しなければメッシュが壊れる」という基本を理解することにあります。

メビウス帯：パラメトリック曲面とシーム

メビウス帯は、帯を 1 回ひねって端同士をつなげた曲面です。これは、パラメトリック曲面を学ぶための非常に優れた題材となります。

ここでは、周方向のパラメータ u と幅方向のパラメータ v を用いて面を構成する方法、頂点グリッドを先に構築してから面を張る手法、そしてシームを閉じるために最後のリングを複製する理由などを学びます。また、法線計算を「面法線の加算」に任せる設計の自然さについても確認してください。

```
import Shape from "./webg/Shape.js";

function buildMobiusStrip(shape, options = {}) {
  const radius = Number(options.radius ?? 2.2);
  const halfWidth = Number(options.halfWidth ?? 0.45);
  const uSegments = Math.max(3, Math.floor(options.uSegments ?? 96));
  const vSegments = Math.max(1, Math.floor(options.vSegments ?? 12));

  // 面法線加算 -> endShape() 正規化という標準経路を使う
```

```
shape.setAutoCalcNormals(true);

const grid = [];

for (let iu = 0; iu <= uSegments; iu++) {
  const row = [];
  const u01 = iu / uSegments;
  const theta = u01 * Math.PI * 2.0;
  const halfTheta = theta * 0.5;

  for (let iv = 0; iv <= vSegments; iv++) {
    const v01 = iv / vSegments;
    const offset = (v01 - 0.5) * 2.0 * halfWidth;

    // webg では Y 軸が上方向であるため、輪の平面を XZ に置き、ひねり分だけ Y へ逃がす
    const x = (radius + offset * Math.cos(halfTheta)) * Math.cos(theta);
    const z = (radius + offset * Math.cos(halfTheta)) * Math.sin(theta);
    const y = offset * Math.sin(halfTheta);

    const index = shape.addVertexUV(x, y, z, u01, v01) - 1;
    row.push(index);
  }

  grid.push(row);
}

for (let iu = 0; iu < uSegments; iu++) {
  for (let iv = 0; iv < vSegments; iv++) {
    const p00 = grid[iu][iv];
    const p10 = grid[iu + 1][iv];
    const p11 = grid[iu + 1][iv + 1];
    const p01 = grid[iu][iv + 1];

    shape.addTriangle(p00, p10, p11);
    shape.addTriangle(p00, p11, p01);
  }
}

return {
  rows: grid.length,
  cols: grid[0]?.length ?? 0
};
```

```
}

const shape = new Shape(screen.getGPU());
shape.setShader(shader);

const info = buildMobiusStrip(shape, {
  radius: 2.4,
  halfWidth: 0.50,
  uSegments: 96,
  vSegments: 12
});

shape.endShape();
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [0.94, 0.64, 0.42, 1.0]
});

console.log(info); // { rows, cols }
```

このコードの注目点は、頂点を「周方向の行」として管理することで面張りのロジックを簡潔にしている点です。また、ループ条件を `iu <= uSegments` とすることで、最後のリングを複製してシームを閉じています。`u01` と `v01` をそのまま UV 座標に割り当てることで、曲面パラメータとテクスチャ座標の対応が明確になります。ここで重要なのは、最後のリングを単に「つなぐ」のではなく「複製する」という手法です。これはメビウス帯に限らず、多くのパラメトリック曲面でシームを扱う際の基本となります。

両面表示とシェーダーの局所調整

メビウス帯は「表と裏が連続する」という特殊な性質を持つため、描画においては陰影の扱いが重要になります。メビウス帯は非向き付け可能な曲面であるため、全周にわたって法線を完全に連続させることは数学的に不可能です。

しかし、学習用としては、裏面のライティングが不自然に反転せず、帯全体を連続した面として観察できる方が望ましいでしょう。このような場合、`SmoothShader` 本体を書き換えるのではなく、サンプル専用の派生クラスを作成してシェーダーを局所的に調整するのが効率的です。つまり、低レイヤーの学習では「形状の構築」だけでなく、「その形状の性質に合わせて

シェーダーを最適化する」ことも重要な要素となります。

book/examples/24_02.html では、以下のような MobiusSmoothShader を使用しています。

```
import SmoothShader from "../../webg/SmoothShader.js";

class MobiusSmoothShader extends SmoothShader {
  constructor(gpu) {
    super(gpu, {
      cullMode: "none"
    });

    // メビウス帯では裏面も同じ帯の続きとして見せたいので、
    // frontFacing に応じて法線向きをそろえる
    this.wgslSrc = this.wgslSrc.replace(
      "var nnormal = normalize(input.vNormal);",
      `let facing = select(-1.0, 1.0, input.frontFacing);
        var nnormal = normalize(input.vNormal) * facing;`
    );
  }
}
```

ここでのポイントは 2 点です。第一に、低レイヤーで Shape を組み立てる際でも、見た目の補正をシェーダーの派生クラスとして局所化できること。第二に、共通基盤である SmoothShader を維持したまま、「この題材だけ両面の見え方を揃えたい」という個別の要求をサンプル側で完結できることです。

24.2　再帰とハイトフィールドへの応用

次に、より複雑な構造である「シェルピンスキー四面体」と「フラクタル地形」について見ていきましょう。

シェルピンスキー四面体：再帰とシャープな面構成

シェルピンスキー四面体は、正四面体を 4 つの小さな四面体に分割し、その操作を再帰的に繰り返すことで自己相似な 3D フラクタルを構築する題材です。

ここでの学習ポイントは、再帰によるメッシュの増殖、および「あえて頂点を複製して面を分ける」ことで、フラクタルの稜線をシャープに見せる手法です。icosphere では頂点共有による滑らかさが重要でしたが、ここでは逆に「共有を避けて折れ目を強調する」ことが意図となります。

```
import Shape from "./webg/Shape.js";

function buildSierpinskiTetrahedron(shape, options = {}) {
  const radius = Number(options.radius ?? 2.2);
  const depth = Math.max(0, Math.floor(options.depth ?? 3));

  // 面ごとに頂点を分けてフラットな見え方を保ちたいので、
  // addTriangle() の面法線加算 -> endShape() 正規化経路を使う
  shape.setAutoCalcNormals(true);

  const scale = radius / Math.sqrt(3.0);
  const v0 = [ scale,  scale,  scale];
  const v1 = [-scale, -scale,  scale];
  const v2 = [-scale,  scale, -scale];
  const v3 = [ scale, -scale, -scale];

  let triangleCount = 0;

  const midpoint = (a, b) => ([
    (a[0] + b[0]) * 0.5,
    (a[1] + b[1]) * 0.5,
    (a[2] + b[2]) * 0.5
  ]);

  const pushFace = (a, b, c) => {
    // フラクタルの角をはっきり見せたいので、面ごとに頂点を複製する
    const p0 = shape.addVertexUV(a[0], a[1], a[2], 0.0, 0.0) - 1;
    const p1 = shape.addVertexUV(b[0], b[1], b[2], 1.0, 0.0) - 1;
    const p2 = shape.addVertexUV(c[0], c[1], c[2], 0.5, 1.0) - 1;
    shape.addTriangle(p0, p1, p2);
    triangleCount++;
  };

  const addTetraFaces = (a, b, c, d) => {
    pushFace(a, c, b);
    pushFace(a, b, d);
    pushFace(a, d, c);
```

```
    pushFace(b, c, d);
  };

  const recurse = (a, b, c, d, level) => {
    if (level <= 0) {
      addTetraFaces(a, b, c, d);
      return;
    }

    const ab = midpoint(a, b);
    const ac = midpoint(a, c);
    const ad = midpoint(a, d);
    const bc = midpoint(b, c);
    const bd = midpoint(b, d);
    const cd = midpoint(c, d);

    // 中央の tetra は抜き、4 つの corner tetra だけを再帰する
    recurse(a,  ab, ac, ad, level - 1);
    recurse(ab, b,  bc, bd, level - 1);
    recurse(ac, bc, c,  cd, level - 1);
    recurse(ad, bd, cd, d,  level - 1);
  };

  recurse(v0, v1, v2, v3, depth);

  return {
    depth,
    triangleCount
  };
}

const shape = new Shape(screen.getGPU());
shape.setShader(shader);

const info = buildSierpinskiTetrahedron(shape, {
  radius: 2.6,
  depth: 3
});

shape.endShape();
shape.setMaterial("smooth-shader", {
  has_bone: 0,
```

```
  use_texture: 0,
  color: [0.96, 0.78, 0.42, 1.0],
  ambient: 0.16,
  specular: 0.90,
  power: 36.0
});

console.log(info); // { depth, triangleCount }
```

このコードでは、recurse() 関数によって 1 つの四面体を 4 つの小さな四面体に分割し、midpoint() を用いてフラクタルの階層を構築しています。面ごとに頂点を複製して addVertexUV() を呼び出すことで、稜線が丸まらず、フラクタル特有のシャープな外見を維持しています。プロシージャル形状において「頂点を共有するか否か」は、常に見た目の目的と密接に結びついていることが分かります。

フラクタル地形：ハイトフィールドの構築

フラクタル地形のハイトフィールドは、2D の高さ配列から 3D 地形を生成する手法です。セルごとに独立したブロックを並べるのではなく、1 枚の連続したメッシュ曲面を構築する低レイヤーの学習として非常に有効です。

ここでのポイントは、「高さ関数の定義」と「メッシュ化」を明確に分離して考えることです。まず数学的・ノイズ的に高さを決定し、その後に Shape として変換するという 2 段構成で実装します。

離散的な盤面表現との使い分け

セル単位の盤面データを主役にする設計では、移動判定や地形種別の管理が中心になります。対して、本例のような Shape による構築は「見た目の連続曲面メッシュ」を生成することが主目的です。ゲームロジックの基準データとして格子を持つか、描画用の地形メッシュとして連続面を作るかを先に分けて考えると、必要なデータ構造を選びやすくなります。

```
import Shape from "./webg/Shape.js";

function buildFractalTerrain(shape, options = {}) {
```

```
const cols = Math.max(2, Math.floor(options.cols ?? 48));
const rows = Math.max(2, Math.floor(options.rows ?? 48));
const sizeX = Number(options.sizeX ?? 10.0);
const sizeZ = Number(options.sizeZ ?? 10.0);
const heightScale = Number(options.heightScale ?? 2.4);
const octaves = Math.max(1, Math.floor(options.octaves ?? 5));
const persistence = Number(options.persistence ?? 0.52);
const lacunarity = Number(options.lacunarity ?? 2.0);
const seed = Number(options.seed ?? 1337);

// 地形は頂点共有で滑らかに見せたいので、自動法線計算を使う
shape.setAutoCalcNormals(true);

const vertexGrid = Array.from({ length: rows + 1 }, () => new Array(cols + 1).fill(0));

const hashNoise = (ix, iz, localSeed = seed) => {
  let h = localSeed | 0;
  h ^= Math.imul(ix + 1, 374761393);
  h ^= Math.imul(iz + 1, 668265263);
  h = Math.imul(h ^ (h >>> 13), 1274126177);
  h ^= h >>> 16;
  return (h >>> 0) / 4294967295;
};

const lerp = (a, b, t) => a + (b - a) * t;
const fade = (t) => t * t * (3.0 - 2.0 * t);

const valueNoise2d = (x, z) => {
  const x0 = Math.floor(x);
  const z0 = Math.floor(z);
  const x1 = x0 + 1;
  const z1 = z0 + 1;
  const tx = x - x0;
  const tz = z - z0;

  const n00 = hashNoise(x0, z0);
  const n10 = hashNoise(x1, z0);
  const n01 = hashNoise(x0, z1);
  const n11 = hashNoise(x1, z1);

  const sx = fade(tx);
  const sz = fade(tz);
```

```
  const nx0 = lerp(n00, n10, sx);
  const nx1 = lerp(n01, n11, sx);
  return lerp(nx0, nx1, sz);
};

const fbm = (x, z) => {
  let amplitude = 1.0;
  let frequency = 1.0;
  let sum = 0.0;
  let norm = 0.0;

  for (let octave = 0; octave < octaves; octave++) {
    sum += valueNoise2d(x * frequency, z * frequency) * amplitude;
    norm += amplitude;
    amplitude *= persistence;
    frequency *= lacunarity;
  }

  return norm > 0.0 ? sum / norm : 0.0;
};

const radialFalloff = (u, v) => {
  const dx = u - 0.5;
  const dz = v - 0.5;
  const d = Math.sqrt(dx * dx + dz * dz) * 2.0;
  return Math.max(0.0, 1.0 - d * d * 0.55);
};

for (let row = 0; row <= rows; row++) {
  const v = row / rows;
  const z = (v - 0.5) * sizeZ;

  for (let col = 0; col <= cols; col++) {
    const u = col / cols;
    const x = (u - 0.5) * sizeX;

    // 低周波と高周波を混ぜて地形の大きな起伏と細部を両立させる
    const coarse = fbm(u * 3.0 + 1.7, v * 3.0 + 2.9);
    const detail = fbm(u * 10.0 + 7.1, v * 10.0 + 5.3);
    const falloff = radialFalloff(u, v);

    // 島状の地形に寄せるため、中心ほど高く、外周は下げる
```

```
      const h01 = Math.max(0.0, coarse * 0.78 + detail * 0.22);
      const lifted = Math.pow(h01, 1.55);
      const y = (lifted * falloff - 0.12) * heightScale;

      const index = shape.addVertexUV(x, y, z, u, v) - 1;
      vertexGrid[row][col] = index;
    }
  }

  for (let row = 0; row < rows; row++) {
    for (let col = 0; col < cols; col++) {
      const p00 = vertexGrid[row][col];
      const p10 = vertexGrid[row][col + 1];
      const p11 = vertexGrid[row + 1][col + 1];
      const p01 = vertexGrid[row + 1][col];

      // 四角形を 2 三角形へ分けるヘルパーを使う
      shape.addPlane([p00, p01, p11, p10]);
    }
  }

  return {
    rows,
    cols,
    vertexCount: (rows + 1) * (cols + 1),
    quadCount: rows * cols
  };
}

const shape = new Shape(screen.getGPU());
shape.setShader(shader);

const info = buildFractalTerrain(shape, {
  cols: 56,
  rows: 56,
  sizeX: 12.0,
  sizeZ: 12.0,
  heightScale: 2.8,
  octaves: 5,
  persistence: 0.54,
  lacunarity: 2.0,
  seed: 2026
```

```
});

shape.endShape();
shape.setMaterial("smooth-shader", {
  has_bone: 0,
  use_texture: 0,
  color: [0.40, 0.72, 0.38, 1.0],
  ambient: 0.20,
  specular: 0.42,
  power: 18.0
});

console.log(info); // { rows, cols, vertexCount, quadCount }
```

このコードの見どころは、valueNoise2d() による格子点補間から、fbm() によるオクターブの重ね合わせまで、フラクタルらしい粗密を段階的に構築している点です。また、頂点を vertexGrid[row][col] に保持することで、後段の面張りのロジックを非常に読みやすくしています。addPlane() を利用して四角形単位で曲面を構成し、頂点を共有したまま面法線の加算を行うことで、地形表面に自然な滑らかさを実現しています。

24.3 学習の進め方と注意点

本章の題材に取り組む際は、まず icosphere から始めることを推奨します。そこで「頂点共有が崩れるとメッシュが壊れる」という感覚を掴み、次にメビウス帯で「頂点グリッドを先に構築して面を張る」という整理術を学んでください。この 2 つの基礎が身につけば、スプラインに沿ったチューブや回転体曲面の一般化といった、より高度な題材へスムーズに移行できるはずです。

実装上の注意点として、endShape() の呼び出しを忘れないでください。これを忘れると描画が行われません。また、Shape クラスはあくまで形状定義であり、シーン内での表示位置はそれを保持する Node 側で制御することを忘れないでください。さらに、setVertex() や addVertex*() の戻り値は「現在の頂点数」であるため、インデックスとして利用する場合は通常 -1 する必要があります。

プロシージャルメッシュにおける最大のポイントは、「頂点を共有して滑らかに見せるか、面ごとに分けてシャープに見せるか」という選択です。icosphere では共有が不可欠であり、シェルピンスキー四面体では独立性が重要でした。どのような複雑な形状であっても、突き詰

めれば「頂点をどう配置し、面をどう張り、共有するか分けるか」という基本原則に集約されます。迷ったときは、第 23 章で学んだ基本原則に立ち返ってください。

24.4 まとめ

本章で最も重要なのは、低レイヤー API を触る目的を見失わないことです。ここで扱った 4 つの題材は、単に見た目が面白いだけではなく、それぞれ重要な設計概念を学ぶためのものです。

- icosphere: サブディビジョンと共有トポロジーの管理
- メビウス帯: パラメトリック曲面とシームの処理
- シェルピンスキー四面体: 再帰的な構造と独立した面構成
- フラクタル地形: ハイトフィールドと頂点グリッドの活用

「何を作るか」以上に、「その題材を通じて何を学ぶか」を意識して取り組んでください。また、コードとサンプルを往復し、法線の見え方やシェーダーの局所調整の効果を実機で確認することで、理解はより深まります。低レイヤー API を使いこなす価値は、既製形状の枠を超え、自らの設計によって独自の構造をメッシュとして成立させる経験にあります。

第 25 章

スキニングの仕組み

普段の利用では、armature（骨格）付きの glb ファイルを読み込んでアニメーションを再生すれば十分です。しかし、モデルの見た目が崩れた際に「何が原因で壊れているのか」を正確に判断するには、スキニングの原理そのものを理解しておく必要があります。本章では、1 つの頂点を複数のボーンでどう変形させて混ぜ合わせるのかという最小原理から出発し、`Skeleton`、`SmoothShader`、そして 2 本のボーンで構成される円柱の教材コードを通して、スキニングの仕組みをローレベル（低レイヤー）な視点から確認していきます。

ここでまず押さえておきたいのは、実務における入り口はあくまで glb のインポートであるという点です。通常のアプリ開発では、ボーンとウェイトを手で入力するよりも、Blender などで作成した glb を `loadModel()` で読み込む方が自然です。ただし、そのハイレベル（高レイヤー）な経路の背景で何が起きているかを知らなければ、変形が不自然なときに原因を切り分けることが困難になります。つまり、本章は「ハイレベルな利用を捨ててローレベルな手法だけを使う」ためのものではなく、「ハイレベルな利用の背景にある仕組みを理解するために、一度最小構成で原理を確かめる」ための章です。

スキニングの核心は、「1 つの頂点を複数のボーン座標系で動かし、その結果を混ぜ合わせる」ことにあります。1 つの頂点が必ずしも 1 本のボーンだけで動くとは限りません。関節の境界付近にある頂点は、複数のボーンによる変形結果を、重み（ウェイト）に基づいて合成した位置へ移動します。`webg` では、CPU 側で `Skeleton.updateMatrixPalette()` が各ボーンの変形行列を並べ、GPU 側で `SmoothShader` の頂点シェーダーがボーンインデックスとウェイトを使用してブレンドを行います。つまり、CPU は「どのボーンが今どのような姿勢か」を行列パレットとして整え、GPU は「各頂点がそのパレットのどこをどれだけ利用するか」を実行するという役割分担になっています。

25.1　ハイレベル経路とローレベル経路の使い分け

普段のアプリ開発では、次のようなハイレベル（高レイヤー）な経路を利用することが多くなります。

```
const runtime = await app.loadModel("./character.glb");
runtime.instantiate(app.space);
```

この経路では、ジョイントの階層構造、逆バインド行列、頂点ごとのウェイト、アニメーションクリップといった情報は、すでに glb ファイルに含まれています。開発者は Action や AnimationState を組み合わせて再生を制御すれば十分であり、「ボーンをどう作るか」よりも「読み込まれた runtime をどう使うか」が主題となります。

しかし、以下のような場面では、ハイレベルな経路だけでは原因の判断がつきません。- どのボーンがどの頂点を引っ張っているのかを正確に把握したいとき- 曲がり方が不自然な原因が、ウェイトの設定にあるのか、ボーンの配置にあるのかを切り分けたいとき- SmoothShader の内部的なスキニング処理を深く理解したいとき- インポーターが出力したスケルトン情報を疑う前に、最小原理を自力で確認したいとき

そのため、本章ではあえてローレベル（低レイヤー）な構成から作ってみるアプローチを取ります。ハイレベルな経路とローレベルな経路は対立するものではなく、理解の階層が異なるだけです。

25.2　1 頂点におけるスキニングの原理

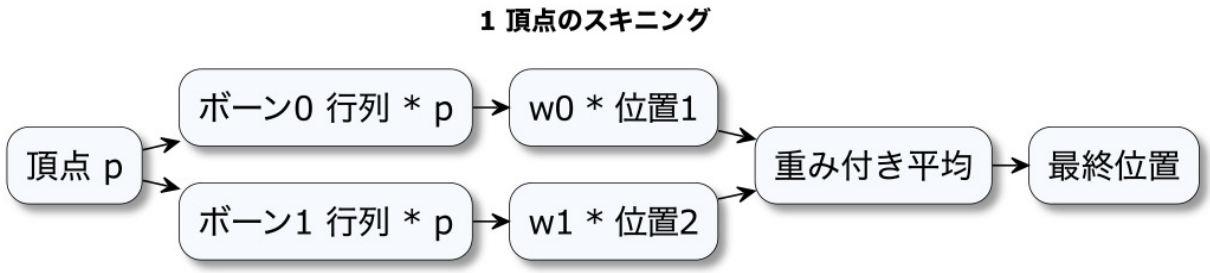


図 25.1: 1 頂点スキニング図

スキニングは、*1* つの頂点を複数のボーン行列で変換し、その結果を重み付きで合成する処理として理解すると本質をつかみやすくなります。

まずは、円柱全体や複雑なモデルではなく、1 つの頂点だけに注目して考えます。原理を最小単位で理解しておくことで、後で多数の頂点へ拡張したときに見通しがよくなります。

レスト姿勢（初期状態）の頂点位置を `p_rest` とします。bone0 と bone1 それぞれについて、スキニングされたメッシュのローカル座標系で見た現在の姿勢行列を `L0`、`L1` とし、レスト姿勢から算出された逆バインド行列（bone offset inverse）を `M0^-1`、`M1^-1` とすると、各ボーンによって変形された頂点位置は次のように計算できます。

```
p0_local = L0 * M0^-1 * p_rest
p1_local = L1 * M1^-1 * p_rest
```

ここで、ウェイトを `w0`、`w1` とし、`w0 + w1 = 1` となるように設定すると、最終的な頂点位置は次の重み付き平均となります。

```
p_final_local = p0_local * w0 + p1_local * w1
```

重要なのは、この段階ではシーン全体のワールド座標ではなく、スキニングされるメッシュ自身のローカル座標系で計算している点です。オブジェクトノード自体の移動や回転がある場合は、後からメッシュ全体に共通の変換が一括して適用されます。webg では、この `L * M^-1` をボーンごとに並べた配列が「行列パレット」であり、`SmoothShader` はこれを頂点シェーダーでブレンドします。

CPU 側で動作を確認する最小例

以下のコードは、`Skeleton.updateMatrixPalette()` の実行後に `bone.worldMatrix` を現在のボーン行列として読み込み、1 つの頂点だけを CPU 側でブレンドする最小の実装例です。

```
function buildSkinMatrix(bone) {
  const skin = bone.worldMatrix.clone();
  skin.mul(bone.getBofMatrix());
  return skin;
```

```
}

function transformVertexByBone(restPosition, bone) {
  const skin = buildSkinMatrix(bone);
  return skin.mulVector(restPosition);
}

function blendVertex(restPosition, influences) {
  const out = [0.0, 0.0, 0.0];
  for (let i = 0; i < influences.length; i++) {
    const moved = transformVertexByBone(restPosition, influences[i].bone);
    out[0] += moved[0] * influences[i].weight;
    out[1] += moved[1] * influences[i].weight;
    out[2] += moved[2] * influences[i].weight;
  }
  return out;
}
```

このヘルパー関数は描画に必須ではありません。実際の描画では GPU 側で同様の処理が行われますが、教材として「GPU が行っている処理を CPU 側で再現できる」ことを確認することは非常に重要です。実装上は Skeleton.updateMatrixPalette() が bone.worldMatrix をメッシュローカル座標系の現在のポーズへ更新し、SmoothShader がその情報を元に行列パレットを用いてブレンドします。

実際の検証ページでは、以下の順序で呼び出すことで本文の数式と対応させることができます。

```
skeleton.updateMatrixPalette();

const rootOnlyLocal = transformVertexByBone(restPosition, rootBone);
const childOnlyLocal = transformVertexByBone(restPosition, childBone);
const blendedLocal = blendVertex(restPosition, [
  { bone: rootBone, weight: w0 },
  { bone: childBone, weight: w1 }
]);
```

ここで得られるのはオブジェクトローカル座標です。画面上にマーカーを置いて確認したい場合は、最後にオブジェクトノードのワールド行列を掛け合わせます。

```
const rootOnlyWorld = objectNode.getWorldMatrix().mulVector(rootOnlyLocal);
const childOnlyWorld = objectNode.getWorldMatrix().mulVector(childOnlyLocal);
const blendedWorld = objectNode.getWorldMatrix().mulVector(blendedLocal);
```

25.3 2 ボーン円柱の構築

仕組みを理解したところで、これを頂点群へ広げて適用します。題材として、下から上へ細かく分割した「円柱の側面」を作成します。下部ほど rootBone、上部ほど childBone の影響度（influence）を強くすることで、ウェイトの意味を視覚的に追いやすくします。

構築の手順は以下の通りです。1. Shape を使用して円柱側面の頂点を定義する。2. Skeleton に root ボーンと child ボーンを作成する。3. 各頂点に addVertexWeight() を使い、2 本分のウェイトを設定する。4. shape.endShape() を呼び出し、GPU 用バッファへ変換する。5. SmoothShader で has_bone: 1 を有効にして描画する。

以下にその実装コードを示します。

```
import Shape from "./webg/Shape.js";
import Skeleton from "./webg/Skeleton.js";

function createTwoBoneCylinder(gpu, options = {}) {
  const height = Number(options.height ?? 8.0);
  const radius = Number(options.radius ?? 0.45);
  const rings = Math.max(2, Math.floor(options.rings ?? 14));
  const segments = Math.max(3, Math.floor(options.segments ?? 24));

  const shape = new Shape(gpu);
  shape.setAutoCalcNormals(true);

  const skeleton = new Skeleton();
  shape.setSkeleton(skeleton);

  const rootBone = skeleton.addBone(null, "rootBone");
  const childBone = skeleton.addBone(rootBone, "childBone");

  rootBone.setRestPosition(0.0, 0.0, 0.0);
  childBone.setRestPosition(0.0, height * 0.5, 0.0);
```

```
  skeleton.bindRestPose();
  skeleton.setBoneOrder(["rootBone", "childBone"]);

  for (let ring = 0; ring <= rings; ring++) {
    const v = ring / rings;
    const y = height * v;
    for (let segment = 0; segment < segments; segment++) {
      const u = segment / segments;
      const angle = u * Math.PI * 2.0;
      const x = Math.cos(angle) * radius;
      const z = -Math.sin(angle) * radius;

      const vIndex = shape.addVertexUV(x, y, z, u, v) - 1;

      // 下ほど rootBone、上ほど childBone の影響度を強くする
      shape.addVertexWeight(vIndex, 0, 1.0 - v);
      shape.addVertexWeight(vIndex, 1, v);
    }
  }

  for (let ring = 0; ring < rings; ring++) {
    const row0 = ring * segments;
    const row1 = (ring + 1) * segments;
    for (let segment = 0; segment < segments; segment++) {
      const next = (segment + 1) % segments;
      const a = row0 + segment;
      const b = row0 + next;
      const c = row1 + segment;
      const d = row1 + next;
      shape.addTriangle(a, c, b);
      shape.addTriangle(b, c, d);
    }
  }

  shape.endShape();
  return { shape, skeleton, rootBone, childBone };
}
```

この実装における重要なポイントは 3 点あります。第一に、Skeleton は単なる親子関係の定義だけでなく、setRestPosition() でレスト姿勢を決定し、bindRestPose() で逆バイン

ド行列を再計算し、setBoneOrder() でシェーダーが参照するパレットの順序を確定させる必要がある点です。第二に、addVertexWeight() によって「どの頂点がどのボーンにどれだけ影響を受けるか」を定義している点です。第三に、Shape.endShape() の実行時に、スキニングメッシュの場合はボーンインデックスとウェイトが専用の頂点属性バッファに格納される点です。SmoothShader のスキニング処理はこのデータ構造を前提に動作します。

25.4 GPU 側での処理フロー

CPU 側で Skeleton.updateMatrixPalette() が呼ばれると、「現在のボーン行列 × 逆バインド行列」の結果が、12 個の float 単位で配列に並びます。これにより、GPU へ渡すための準備が整います。

SmoothShader の頂点シェーダーは、各頂点のインデックスとウェイトを読み取り、最大 4 本までのボーン行列を重み付きで合成して、最終的なスキニング行列を算出します。この行列を頂点位置と法線に適用した後、通常の静的メッシュと同様にビュー変換および投影変換へと流します。つまり、前述の「1 頂点に対する数式」が、数万個の頂点に対して GPU 上で並列に実行されていることになります。

ここで重要なのは、「ボーンごとに頂点を描き直している」のではなく、「1 つの頂点に対して最大 4 本のボーンの影響を 1 回の頂点シェーダー実行でブレンドしている」という点です。トラブルシューティングを行う際は、以下の順序で確認すると効率的です。

1. Shape にボーンインデックスとウェイトが正しく格納されているか。
2. Skeleton のレストポーズとボーン順序（Bone Order）が正しいか。
3. Skeleton.updateMatrixPalette() が現在のポーズを正しく反映しているか。
4. shape.setMaterial("smooth-shader", { has_bone: 1 }) が有効になっているか。
5. weight_debug モードを使用して、影響度の分布が意図通りか。

問題をシェーダーやインポーターだけのせいにせず、「頂点属性 → スケルトン → パレット → シェーダー設定」の順に切り分けることが重要です。

25.5 サンプルコードによる検証

本章の理解を深めるため、検証目的を分けた 2 つのサンプルページを用意しています。「数式としての正しさ」と「メッシュとしての見え方」を切り分けて確認することで、どこに問題があるのかを明確にします。

25_01: 1 頂点の原理検証

25_01 では、ボーンとマーカーのみを表示し、代表的な 1 つの頂点について以下の 4 点を比較します。- レスト位置（初期位置）- bone0 のみで変形させた位置- bone1 のみで変形させた位置- ウェイトによる平均をとった最終位置

ここでは円柱メッシュを表示せず、数式通りにマーカーが移動するかを最優先で確認します。検証には円柱軸から少し離したプローブポイント（観測点）を使用しますが、適用されている原理は同一です。このページは、本文の数式が画面上でどのように実現されているかを視覚的に確認するためのものです。

25_02: メッシュ全体の挙動検証

25_02 では、円柱メッシュ全体の変形を確認します。ここではマーカーは表示せず、SmoothShader による曲がり方と、weight_debug を切り替えた際の影響度分布を確認します。25_01 が「個別の頂点における数式の正しさ」を検証するのに対し、25_02 は「GPU によるスキニング結果がメッシュとしてどう見えるか」を検証します。この役割分担により、計算ロジックの問題なのか、描画結果の問題なのかを容易に切り分けることができます。

25.6 Blender / glb のハイレベル経路への回帰

ここまでのローレベルな実装を確認したあとで、再び glb のハイレベルな経路に目を向けると、Blender で行っている作業の意味がより明確になります。ハイレベルな経路は全く別の仕組みではなく、これまで手動で行っていた設定をインポーターが自動的に肩代わりしているに過ぎません。

Blender で行っている作業の多くは、以下に集約されます。- ボーンの階層構造を構築すること。- 各ボーンのレスト姿勢を決定すること。- ウェイトペイントにより、頂点ごとのボーン影響度を決定すること。- アニメーションクリップとして、時系列に沿ったボーンの姿勢を定義すること。

glb をインポートすると、webg はこれらを ModelAsset、Skeleton、Animation として取り込みます。したがって、ハイレベルな経路を利用することは「低レイヤーの仕組みを回避すること」ではなく、「低レイヤーで定義すべき情報をインポーター経由で効率的に設定すること」であると言えます。

25.7 よくあるつまずきと解決策

has_bone: 1 を設定したにもかかわらずモデルが曲がらない場合は、ボーンインデックスやウェイトが Shape に正しく格納されていないか、スケルトンのパレット順と頂点側のインデックスが一致していない可能性があります。まずはシェーダーよりも「頂点属性」と「ボーン順」の整合性を疑ってください。

ボーンは動いているのにメッシュがレスト姿勢のままの場合は、スキニング非対応のシェーダー設定になっている可能性があります。スキニングメッシュでは、スキニング経路が有効な SmoothShader を使用し、has_bone: 1 が有効である必要があります。

曲がり方はしているが、関節の境界が折れ曲がりすぎる場合は、ウェイトの分布が極端である可能性があります。weight_debug を有効にして分布を確認し、境界付近でウェイトが 0 または 1 に急激に切り替わっていないかを確認してください。見え方の問題をボーンの回転だけで解決しようとせず、ウェイト分布を同時に検証することが大切です。

複雑な glb モデルで問題が発生したときは、アニメーション、スケルトン名、マテリアル、テクスチャ、ノードのスケールなど、原因の切り分けが困難になります。そのようなときこそ、この 2 ボーン円柱の最小例に戻り、どのレイヤーに問題があるのかを再確認してください。この最小構成の知識が、複雑なモデルのトラブルシューティングにおける基準となります。

25.8 まとめ

本章で最も重要なのは、スキニングを「ボーンがメッシュを動かす魔法の仕組み」としてではなく、「1 つの頂点を複数のボーン座標系で変形し、その結果を重み付きで合成する処理」と

して理解することです。CPU 側では Skeleton がそのための行列パレットを作成し、GPU 側では SmoothShader が頂点ごとにブレンドを行います。

2 ボーン円柱という最小教材を通して、「レスト姿勢」「逆バインド行列」「ウェイト」「パレット順」「シェーダー設定」という切り分け軸を明確にしました。複雑な glb モデルが意図通りに動作しないときでも、この最小原理に立ち返ることで、問題の所在を論理的に判断できるようになります。本章の内容を、ハイレベルなインポート結果を正しく読み解き、制御するための基礎知識として活用してください。

第 26 章

物理エンジン

これまでの章では、Node、Space、Shape、Scene JSON、そしてレイキャストや衝突判定（collision）など、3D シーンを構成するための基礎的な部品を扱ってきました。これらを組み合わせることで、物体を表示し、ユーザー入力で動かし、境界ボックスによって重なりを調べる基礎的な機能は実装可能です。

しかし、より本格的なゲームやインタラクティブなシーンを構築しようとすると、単なる座標指定による移動ではなく、「重力で落下し、床に当たって跳ね返り、摩擦で停止し、他の物体と押し合う」といった自然な物理挙動へのニーズが生じます。こうした挙動をアプリケーション側のコードで個別に実装しようとすると、落下処理や衝突応答といった同様のロジックを何度も記述することになり、効率的ではありません。

webg の物理エンジンは、こうした物理挙動を効率的に実現するための軽量な基盤です。本エンジンは、外部の巨大な汎用物理エンジンを完全に置き換えることを目的としたものではなく、webg のシーングラフ（scene graph）と座標管理の上に自然に統合され、学習しやすく拡張しやすい剛体物理を提供することを目的として設計されています。

本章では、物理エンジンが提供する価値から始まり、PhysicsNode、PhysicsSpace、Collider の役割と基本的な使い方、Scene JSON による宣言方法、そして内部的なブロードフェーズ（広域判定）、ナローフェーズ（詳細判定）、ソルバー（解決器）に至る処理フローまでを詳しく解説します。

26.1　物理エンジンが提供する価値

物理エンジンが必要になるのは、「視覚的な挙動」と「当たり判定の結果」を継続的に一致させたいときです。

たとえば、単純に箱を落下させるだけであれば、`Node.move()` や `setPosition()` を毎フレーム呼び出すだけで実装できます。しかし、「床に当たった瞬間に停止させる」「反発係数に応じて跳ね返す」「斜め方向の速度を摩擦で減衰させる」「複数の箱が同時に接触した際に互いに押し戻す」といった処理を追加し始めると、単純な座標更新だけでは管理しきれなくなります。

また、通常の衝突判定（collision）やレイキャスト（raycast）は、「当たっているかどうか」を調べるためのクエリ API です。第 17 章で扱った `Space.raycast()` や `Space.checkCollisions()` は、物体の選択や重なり判定には適していますが、「当たった物体をどう押し戻すか」「速度をどう変化させるか」という物理的な応答（レスポンス）までは担当しません。

物理エンジンは、以下のような一連の処理を統合的に管理します。

1. 重力や速度に基づいた物体の位置更新（積分）。
2. 接触候補となる物体ペアの抽出（ブロードフェーズ：広域判定）。
3. 実際の接触法線（normal）とめり込み量（penetration）の算出（ナローフェーズ：詳細判定）。
4. めり込みを解消するための位置補正。
5. 反発と摩擦による速度の変化（インパルス：衝撃量による応答）。
6. 静止した物体をスリープ（sleep）させ、必要に応じてウェイク（wake）させる最適化。
7. 接触イベントや問い合わせ API を通じたアプリケーション側への通知。

`webg` の物理機能は、これらを `PhysicsSpace` と `PhysicsNode` を中心に構築しています。

26.2　物理エンジンの全体構成

物理機能の中核となるクラスは、次の 3 つの系統で構成されています。

- `PhysicsNode`: 物理シミュレーションに参加する個々の物体。

- PhysicsSpace: 複数の PhysicsNode をまとめて管理し、物理更新を行う空間。
- Collider: 接触判定や問い合わせを担当する形状定義。

PhysicsNode は Node を継承しています。これは webg における重要な設計上の特徴です。物理オブジェクトをシーングラフの外側に置くのではなく、通常の Node と同様に Space へ配置し、Shape を付与できるようにしています。これにより、見た目の位置と物理的な位置を同一のオブジェクト上で管理でき、同期の手間を省いています。

ただし、物理更新中のダイナミック（dynamic）なボディに対して、スクリプトから自由に setPosition() を行うと、物理計算の結果と手動の座標更新が衝突します。そのため、dynamic ボディでは通常のトランスフォーム書き換えを制限しています。強制的に移動させたい場合は teleport() を、一時的に手動配置したい場合は pauseDynamic() / resumeDynamic() や setBodyType() を使用します。

PhysicsSpace は、描画のための Space とは異なる役割を持ちます。Space がシーングラフと描画を管理するのに対し、PhysicsSpace は固定タイムステップ（fixed timestep）、重力、接触解決、スリープ管理、クエリ処理を扱う物理専用の空間です。両者は分離されていますが、PhysicsNode が Node を継承していることで密接に連携しています。

Collider は、見た目の Shape とは独立して設定されます。たとえば、見た目は球体として描画していても、物理判定は箱形にすることが可能です。また、床の見た目は薄い箱であっても、物理判定は無限平面に設定することで、計算の安定性を高めることができます。

現在、以下の 4 種類のコリジョン形状（collider）が用意されています。

- BoxCollider: ボディのクォータニオン姿勢を反映した OBB（有向境界ボックス）として扱います。
- PlaneCollider: 床や壁のような無限平面に適しています。
- SphereCollider: 跳ねる球体や単純な範囲判定に適しています。
- CapsuleCollider: y 軸方向のカプセル形状として扱い、球体よりも背の高い形状を簡潔に定義できます。

BoxCollider の場合、ブロードフェーズでは計算負荷を抑えるためにその OBB を包むワールド AABB（軸平行境界ボックス）を使用し、ナローフェーズやクエリ、レイキャストでは向きを持つボックスとして厳密に判定します。

26.3 実装例：床と箱を落下させる

まずは、床を設置し、その上に箱を落下させる最小構成の実装例を見てみます。

```
import Space from "./webg/Space.js";
import PhysicsSpace from "./webg/PhysicsSpace.js";
import BoxCollider from "./webg/BoxCollider.js";
import PlaneCollider from "./webg/PlaneCollider.js";

const space = new Space();

const physics = new PhysicsSpace({
  gravity: [0.0, -9.8, 0.0],
  fixedTimeStepMs: 1000.0 / 120.0,
  solverIterations: 4
});

const floor = space.addPhysicsNode(null, "floor", {
  bodyType: "static"
});
floor.setPosition(0.0, 0.0, 0.0);
floor.setCollider(new PlaneCollider([0.0, 1.0, 0.0]));
floor.setPhysicsMaterial({
  restitution: 0.0,
  friction: 0.7
});
physics.addBody(floor);

const box = space.addPhysicsNode(null, "box", {
  bodyType: "kinematic",
  mass: 1.0,
  linearDamping: 0.2
});
box.setPosition(0.0, 12.0, 0.0);
box.setCollider(new BoxCollider([2.0, 2.0, 2.0]));
box.setPhysicsMaterial({
  restitution: 0.1,
  friction: 0.5
});
physics.addBody(box);
```

```
box.setBodyType("dynamic", {
  clearVelocity: false,
  restoreVelocity: false
});
```

ここで重要なポイントは、位置やコリジョンを設定した後に dynamic へ切り替えている点です。dynamic に設定された後は、物理エンジン側がボディのトランスフォームを制御します。初期配置を行う際は kinematic に設定しておくことで、通常の setPosition() を安全に使用できます。

毎フレームの更新処理では、描画の前に physics.step(deltaMs) を呼び出します。

```
let previousTimeMs = null;

const frame = (timeMs) => {
  if (previousTimeMs === null) {
    previousTimeMs = timeMs;
  }
  const deltaMs = Math.min(timeMs - previousTimeMs, 80.0);
  previousTimeMs = timeMs;

  physics.step(deltaMs);

  screen.clear();
  space.draw(eye);
  screen.present();

  requestAnimationFrame(frame);
};

requestAnimationFrame(frame);
```

PhysicsSpace.step(deltaMs) は、内部で固定タイムステップに分割して stepFixed(dtSec) を呼び出します。これにより、ブラウザのフレームレートが変動しても、物理挙動が不安定になるのを防いでいます。

26.4 PhysicsNode の詳細仕様

PhysicsNode には、用途に応じて 3 種類の bodyType が用意されています。

- static: 全く動かない物体。床、壁、地形などに使用します。
- kinematic: スクリプトや Tween で動かす物体。物理的な押し戻し（位置補正）は受けませんが、他の物体への接触判定には参加します。
- dynamic: 重力、衝突、反発、摩擦など、物理エンジンの計算によって動く物体。

これらは単なる ON/OFF の切り替えではなく、ソルバー（解決器）内部での処理方法が異なります。

以下は、演出中だけ物理挙動を停止させ、位置を調整する例です。

```
box.pauseDynamic({
  clearVelocity: true
});

box.setPosition(0.0, 16.0, 0.0);
box.setLinearVelocity(0.0, 0.0, 0.0);

box.resumeDynamic({
  clearVelocity: false,
  restoreVelocity: false
});
```

瞬時に再配置したい場合は、teleport() メソッドを使用します。

```
box.teleport([0.0, 20.0, 0.0], {
  keepVelocity: false,
  wakeUp: true
});
```

また、物体に対して力（Force）やインパルス（Impulse：衝撃量）を与えることも可能です。

```
box.applyForce([0.0, 30.0, 0.0]);
box.applyImpulse([4.0, 10.0, 0.0]);
box.applyTorque([0.0, 0.0, 120.0]);
```

applyForce() は次の固定ステップの積分で速度に反映されます。applyImpulse() は瞬間的に速度を変化させ、applyTorque() は角速度を変化させる力となります。接触解決においては、接触点と局所的な対角慣性（local diagonal inertia）を用いることで、中心から外れた位置への衝撃が自然な回転へと変換されます。

角速度の単位と内部処理

PhysicsNode が公開 API として保持する角速度は、setAngularVelocity(x, y, z)、getAngularVelocity()、Scene JSON の angularVelocity のいずれも **degree/sec**（度/秒）です。つまり、[0.0, 90.0, 0.0] は「1 秒間に Y 軸まわりへ 90 度回る」角速度を意味します。

これは、Node の姿勢指定やサンプルコードで角度を度数法（degree）として扱ってきた流れに合わせた仕様です。利用者側では、トルクや回転速度を見た目と対応させやすくなります。

一方で、物理計算の低レイヤー（ローレベル）な内部処理では、外積 omega x r や慣性テンソルの式を扱います。このような計算では、角速度は通常 rad/sec（ラジアン/秒）として扱う必要があります。そのため PhysicsSpace は、外部状態としての degree/sec と、内部計算としての rad/sec を明示的に変換しています。

- PhysicsNode.angularVelocity: degree/sec で保存。
- 姿勢積分 _buildAngularStepQuat(): degree/sec を使い、dtSec を掛けて回転角を作る。
- 接触点速度 v + omega x r: omega を rad/sec へ変換して計算。
- 接触インパルスから得た角速度変化: 慣性計算で得た rad/sec を degree/sec へ戻して保存。
- トルク積分: rad/sec^2 として求めた角加速度を degree/sec^2 へ戻して保存。

この単位変換を混同すると、見た目の回転速度が極端に大きくなったり、反対にスリープ判定が厳しすぎたりします。たとえば sleepAngularThreshold: 0.5 は 0.5 degree/sec であり、0.5 rad/sec ではありません。

26.5 コリジョンと物理材質

Collider は物理判定用の形状であり、見た目の Shape とは別に設定します。

```
body.setCollider(new SphereCollider(2.0));
```

各コリジョンの基本的な指定方法は以下の通りです。

```
new BoxCollider([width, height, depth]);
new PlaneCollider([0.0, 1.0, 0.0]);
new SphereCollider(radius);
new CapsuleCollider(radius, segmentLength);
```

BoxCollider、SphereCollider、CapsuleCollider は有限な形状であるため、AABB を返すことができます。一方、PlaneCollider は無限平面であるため AABB を持たず、ブロードフェーズ（広域判定）では特殊形状として処理されます。

物理材質（physics material）は、見た目のマテリアルとは別に設定します。

```
body.setPhysicsMaterial({
  restitution: 0.7,
  friction: 0.05
});
```

- restitution（反発係数）：0.0 に近いほど跳ねず、1.0 に近いほど強く跳ね返ります。
- friction（摩擦係数）：接触面に沿った速度をどれだけ減衰させるかに影響します。

接触する 2 つのボディの材質は、ソルバー（解決器）内部で組み合わせて使用されます。基本的には、反発は大きい方の値を採用し、摩擦は平方根で組み合わせた値を採用します。材質が設定されていない場合は、PhysicsSpace の defaultRestitution と defaultFriction が適用されます。

PlaneCollider と BoxCollider の特殊な扱い

PlaneCollider と BoxCollider の組み合わせには特別な扱いがあります。静的な PlaneCollider の restitution が 0.0 の場合、その平面は「跳ねない床」として扱われ、箱側の restitution は床反発として使われません。

これは長い箱や梁（beam）の安定性のためです。箱の端が床に当たると、上向きの反発インパルスは中心から大きく外れた位置に作用します。その結果、線形の跳ね返りではなく大きなトルクになり、床が跳ねない材質であるにもかかわらず、梁が立ち上がって暴れ続けることがあります。webg では、床側が明示的に restitution: 0.0 の plane-box 接触では、床の「跳ねない」という指定を優先します。

さらに plane-box では、箱の 1 頂点だけを即座に接触点とみなすのではなく、床面に十分近い複数頂点を support patch としてまとめる処理を行います。これは、細長い箱や薄い板が床へ倒れ込む途中で、一時的に角 1 点だけがめり込んだように見える状況でも、実際には面接触へ育つ途中の patch として扱いやすくするためです。support patch が作れた場合は、その極値点から複数接点を構成し、単一点の支点で立ち上がる挙動を減らしています。

26.6 物理エンジンの内部動作原理

ここからは、webg がどのようにして回転を含む複雑な接触を計算しているか、その内部設計について解説します。

回転を伴うボックス判定（OBB）の実現

物理エンジンの実装において、ボックスを「軸に平行な箱（AABB)」として扱うか、「回転した箱（OBB)」として扱うかは非常に重要な分岐点となります。AABB のみの実装は軽量ですが、見た目のボックスが回転しているにもかかわらず当たり判定がワールド軸に固定されていると、接触位置や押し戻し方向が直感とずれてしまいます。

そこで webg では、BoxCollider をボディのクォータニオン姿勢を反映した OBB（Oriented Bounding Box）として扱う設計を採用しています。一方で、毎フレームすべてのボックス同士に重い判定を行うと負荷が高くなるため、ブロードフェーズ（広域判定）では外接 AABB を用いて候補を絞り込む最適化を行っています。

具体的な処理は webg/BoxCollider.js の getWorldInfo() と getAabb() で実装されています。

```
// webg/BoxCollider.js
getWorldInfo(position, quat = null) {
  return {
    center: this.getWorldPosition(position, quat),
    half: this.getHalfExtents(),
    axes: this._getOrientationAxes(quat)
  };
}

getAabb(position, quat = null) {
  const info = this.getWorldInfo(position, quat);
  const extent = [0.0, 0.0, 0.0];
  for (let worldAxis = 0; worldAxis < 3; worldAxis++) {
    extent[worldAxis] =
      Math.abs(info.axes[0][worldAxis]) * info.half[0] +
      Math.abs(info.axes[1][worldAxis]) * info.half[1] +
      Math.abs(info.axes[2][worldAxis]) * info.half[2];
  }
  return {
    min: [info.center[0] - extent[0], info.center[1] - extent[1],
          info.center[2] - extent[2]],
    max: [info.center[0] + extent[0], info.center[1] + extent[1],
          info.center[2] + extent[2]]
  };
}
```

getWorldInfo() は、現在のボディの位置と向きに基づいて、OBB の中心、半サイズ、およびワールド空間での方向軸（axes）を算出します。getAabb() では、これらの情報を基に OBB を完全に包み込む最小のワールド AABB を算出しています。この手法により、ブロードフェーズでは高速な AABB 比較で候補を絞り込み、候補に残ったペアのみを詳細な OBB 判定（ナローフェーズ）に回すことができます。

OBB 同士の接触判定（SAT）

候補に残ったボックス同士の接触は、SAT（分離軸定理：Separating Axis Theorem）を用いて判定します。これは、「ある軸に投影したときに隙間があれば接触していない」という原

理に基づいています。判定に使用する軸は、各 OBB の 3 本の方向軸と、それらの外積から得られる 9 本の軸、計 15 本です。

実装は webg/BoxCollider.js の _buildObbContact() にあります。

```
// webg/BoxCollider.js
for (let i = 0; i < 3; i++) {
  if (!testAxis(boxA.axes[i])) return null;
  if (!testAxis(boxB.axes[i])) return null;
}
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    const axis = [
      boxA.axes[i][1] * boxB.axes[j][2] - boxA.axes[i][2] * boxB.axes[j][1],
      boxA.axes[i][2] * boxB.axes[j][0] - boxA.axes[i][0] * boxB.axes[j][2],
      boxA.axes[i][0] * boxB.axes[j][1] - boxA.axes[i][1] * boxB.axes[j][0]
    ];
    if (!testAxis(axis)) return null;
  }
}
```

testAxis() 関数は、指定された軸へ両方の OBB を投影し、その区間が重なっているかを確認します。すべての軸で重なりがあった場合にのみ「接触している」と判断し、最も重なりが小さい軸を接触法線（normal）およびめり込み量（penetration）の候補として採用します。

また、接触点（contact point）については、常に 1 点だけを返すわけではありません。webg では、SAT で選ばれた軸が face 軸に近い場合は、face 側の頂点群から接触多様体（contact manifold）を構成します。edge 軸がわずかに有利なだけであれば face manifold を優先するため、梁や板が box や床に触れたときでも、早い段階で面接触へ育ちやすくなっています。一方で、明確に edge 接触である場合は代表点 1 点の接触として扱い、必要最小限の情報で回転応答を解いています。

慣性テンソルの近似的な扱い

回転を伴う接触応答では、質量だけでなく「物体の形状による回りやすさ」を考慮する必要があります。これを表すのが慣性テンソルです。本来は 3x3 行列で管理し、姿勢ごとに更新する必要がありますが、計算負荷を抑えるため webg では局所軸に沿った対角成分（diagonal inertia）のみを保持する近似手法を採用しています。

この近似計算は webg/PhysicsNode.js で行われています。

```
// webg/PhysicsNode.js
if (collider?.type === "box" && Array.isArray(collider.size)) {
  const sx = collider.size[0];
  const sy = collider.size[1];
  const sz = collider.size[2];
  return [
    this.mass * (sy * sy + sz * sz) / 12.0,
    this.mass * (sx * sx + sz * sz) / 12.0,
    this.mass * (sx * sx + sy * sy) / 12.0
  ];
}
```

ボックス形状では直方体の慣性モーメントの公式を用い、球体やカプセル形状でもそれぞれの特性に応じた対角成分を算出します。これにより、「細長い物体は特定の軸まわりには回りやすいが、別の軸には回りにくい」といった基本的な物理特性を表現できます。

回転コンタクトソルバーの動作

接触応答では、中心速度だけでなく「接触点における速度」を評価します。回転している物体では、中心速度がゼロであっても接触点は動いているためです。webg では v + omega x r（線形速度 + 角速度 × 半径ベクトル）を接触点速度として計算し、そこにインパルス（衝撃量）を適用します。

実装は webg/PhysicsSpace.js の _getContactPointVelocity()、_getImpulseDenominator()、_applyContactImpulse() にあります。

```
// webg/PhysicsSpace.js
_getContactPointVelocity(state, r) {
  return this._addVec3(
    state.velocity,
    this._crossVec3(this._degVec3ToRad(state.angularVelocity), r)
  );
}

_getImpulseDenominator(bodyA, stateA, rA, bodyB, stateB,
                       rB, direction, invMassA, invMassB) {
```

```
  const angularA = this._crossVec3(
    this._applyWorldInverseInertia(bodyA, stateA.quat, this._crossVec3(rA, direction)),
    rA
  );
  const angularB = this._crossVec3(
    this._applyWorldInverseInertia(bodyB, stateB.quat, this._crossVec3(rB, direction)),
    rB
  );
  return invMassA + invMassB
    + this._dotVec3(direction, angularA)
    + this._dotVec3(direction, angularB);
}
```

rA や rB はボディの中心から接触点までのベクトルです。中心から離れた位置で衝撃を受けたほど、角速度への影響が大きくなります。_getContactPointVelocity() では、保存されている角速度を degree/sec から rad/sec へ変換してから omega x r を計算します。ここで degree/sec のまま外積に渡すと、接触点速度が約 57 倍ずれてしまいます。

_getImpulseDenominator() では、線形的な動きやすさと回転による逃げやすさを合算し、有効質量の分母を算出しています。_applyWorldInverseInertia() は、ワールド空間のトルクや角インパルスを、ボディの現在姿勢に応じた局所対角逆慣性へ通します。戻り値は rad/sec 系の角速度変化として扱われます。

実際にインパルスを反映させる処理は以下の通りです。

```
// webg/PhysicsSpace.js
_applyContactImpulse(body, state, r, impulse, sign, invMass) {
  state.velocity[0] += impulse[0] * sign * invMass;
  state.velocity[1] += impulse[1] * sign * invMass;
  state.velocity[2] += impulse[2] * sign * invMass;
  if (body?.getFixedRotation?.() === true) {
    return;
  }
  const angularImpulse = this._crossVec3(r, impulse);
  const angularDeltaRad =
    this._applyWorldInverseInertia(body, state.quat, angularImpulse);
  const angularDeltaDeg = this._radVec3ToDeg(angularDeltaRad);
  state.angularVelocity[0] += angularDeltaDeg[0] * sign;
  state.angularVelocity[1] += angularDeltaDeg[1] * sign;
```

```
  state.angularVelocity[2] += angularDeltaDeg[2] * sign;
}
```

この処理により、接触点が中心からずれていれば、線形的な反発だけでなく回転運動も誘発されます。また、反発インパルスの後に摩擦インパルスも同様の枠組みで適用されるため、接触面に沿った滑りが回転へと変換される挙動も実現しています。

トルク積分でも同じ考え方を使います。`applyTorque()` で蓄積されたトルクは、`_applyWorldInverseInertia()` によって rad/sec^2 系の角加速度へ変換されます。その後、`_radVec3ToDeg()` で degree/sec^2 へ戻してから `state.angularVelocity` へ加算します。つまり、公開状態は degree/sec に統一し、物理式の内部だけ rad を使う設計です。

床上での姿勢安定化と support chain

軽量な反復ソルバーでは、床に箱が落ち着く直前に、角や辺の 1 点接触だけが残ることがあります。完全な接触多様体（contact manifold）を厳密に解く大型物理エンジンであれば自然に面接触へ収束しやすいですが、`webg` の軽量実装では小さな角速度が残り、箱がいつまでも微振動することがあります。

このため `PhysicsSpace` には `_stabilizeRestingBoxes()` という静止安定化処理（resting stabilizer）があります。これは高速衝突を無理に止める処理ではなく、すでに support（支持）を得て低速になった `BoxCollider` を、自然に静止姿勢へ寄せる補助です。最初の対象は床 `PlaneCollider` 上の箱でしたが、現在は「床などの静止 support を持つ下側 box の上に乗った box」に対しても、限定的に support chain を伝播させます。

安定化では、まず接触法線が support とみなせる contact を集めます。床 plane に対してはその法線を直接使い、box-box では「どちらが上に乗っているか」を contact 法線と相対位置から判定したうえで、下側 box がすでに静止 support を持ち、接触点での相対速度も十分小さい場合だけ support chain へ参加させます。その後、箱の局所軸を現在のクォータニオンでワールド空間へ回転し、そのうち最小 half extent の軸を support 法線へ向ける候補として選びます。

```
const axes = [
  this._rotateVec3ByQuat([1.0, 0.0, 0.0], state.quat),
  this._rotateVec3ByQuat([0.0, 1.0, 0.0], state.quat),
```

```
  this._rotateVec3ByQuat([0.0, 0.0, 1.0], state.quat)
];
const halfExtents = collider?.getHalfExtents?.() ?? [1.0, 1.0, 1.0];
const minHalfExtent = Math.min(halfExtents[0], halfExtents[1], halfExtents[2]);
```

最小 half extent の軸を優先するのは、箱が最も薄い方向を support 法線へ向けた姿勢が、重心が低く安定しやすいためです。立方体のように複数の軸が同じ half extent を持つ場合は、現在もっとも support 法線に近い軸を選びます。これにより、立方体を不必要に別の軸へ回そうとせず、梁や板では寝やすい姿勢へ寄せられます。

角速度の分解と制御

安定化で特に重要なのが、角速度を次の 2 つに分けて扱う点です。

- **support 法線方向の角速度**: support 面の上でコマのように回る成分。
- **support 面へ倒れ込む方向の角速度**: 箱の姿勢を傾け、角や辺を support 面へ落とす成分。

実装では、床法線 `normal` への射影を使って分解しています。

```
const angularNormalSpeed = Math.abs(this._dotVec3(state.angularVelocity, normal));
const angularTangentVelocity = this._subVec3(
  state.angularVelocity,
  this._scaleVec3(normal, this._dotVec3(state.angularVelocity, normal))
);
const angularTangentSpeed = this._lengthVec3(angularTangentVelocity);
```

support 法線方向の回転は、箱が support 面の上で姿勢を保ったまま回る成分です。これは見た目にも物理的にも「まだ回っている」状態なので、resting stabilizer が勝手に消してしまうと不自然です。実際、`sleepAngularThreshold` より大きい support 法線方向の角速度が残っているボディは、sleep へ入れてはいけません。

一方、support 面に倒れ込む方向の回転は、箱が微妙に傾き続けたり、角接触や辺接触のまま揺れたりする原因になります。この成分は support を持って落ち着きかけている場合にだけ、安定化処理で減衰させます。

```
const angularNormalVelocity = this._scaleVec3(
  normal,
  this._dotVec3(state.angularVelocity, normal)
);
state.angularVelocity[0] = angularNormalVelocity[0]
                         + angularTangentVelocity[0] * angularScale;
state.angularVelocity[1] = angularNormalVelocity[1]
                         + angularTangentVelocity[1] * angularScale;
state.angularVelocity[2] = angularNormalVelocity[2]
                         + angularTangentVelocity[2] * angularScale;
```

この処理では、support 法線方向の角速度はそのまま残し、support 面に倒れ込む方向だけを angularScale で弱めます。さらに完全に停止姿勢へスナップする条件でも、angularTangentSpeed だけでなく angularNormalSpeed <= sleepAngularThreshold を要求します。つまり、箱が support 面の上でまだ回っているなら、姿勢が安定していても sleep させません。

この分解は、回転の慣性とスリープ判定を両立させるための重要な実装です。単に角速度の長さだけを見て全成分を減衰すると、回っている物体が突然止まります。逆に全成分を残すと、床上や box stack 上で寝るべき箱がいつまでも角接触で揺れます。webg では support 法線方向と倒れ込み方向を分けることで、「回っているものは回し続ける」「寝るべきものは寝かせる」という挙動を両立しています。

26.7 接触イベントとクエリ API

PhysicsSpace は、直近のステップで発生した接触情報を保持しており、これをアプリケーション側で利用できます。

```
physics.step(deltaMs);

for (const contact of physics.getLastContacts()) {
  console.log(
    contact.bodyA.getName(),
    contact.bodyB.getName(),
    contact.normal,
    contact.penetration
  );
}
```

また、接触の開始、継続、終了をリスナーで受け取ることも可能です。

```
const offBegin = physics.onBeginContact((event) => {
  console.log("begin", event.bodyA.getName(), event.bodyB.getName());
});

const offEnd = physics.onEndContact((event) => {
  console.log("end", event.bodyA.getName(), event.bodyB.getName());
});

// 不要になったら解除する
offBegin();
offEnd();
```

なお、トリガーボディ（isTrigger: true）は接触イベントには現れますが、物理的な押し戻しや反発は行いません。ゴール判定や範囲侵入判定、センサーなどの用途に適しています。

```
const sensor = space.addPhysicsNode(null, "sensor", {
  bodyType: "static",
  isTrigger: true
});
sensor.setCollider(new BoxCollider([6.0, 3.0, 6.0]));
physics.addBody(sensor);
```

物理空間への問い合わせ（Query API）

接触解決とは別に、物理空間の状態を調べるための問い合わせ API が提供されています。

raycast() と raycastAll()

```
const hit = physics.raycast(
  [0.0, 10.0, 20.0],
  [0.0, -0.2, -1.0],
  {
    maxDistance: 100.0,
    includeTriggers: false,
```

```
    layerMask: 0xffffffff
  }
);

if (hit) {
  console.log(hit.body.getName(), hit.position, hit.normal, hit.distance);
}
```

raycast() は最初にヒットした物体を返し、raycastAll() は条件に合うすべてのヒットを距離順に返します。

queryAabb()

```
const bodies = physics.queryAabb(
  [-5.0, 0.0, -5.0],
  [ 5.0, 8.0,  5.0],
  {
    includeTriggers: true
  }
);
```

queryAabb() は、指定した AABB 領域と重なるボディを返します。

overlapSphere()

```
const overlaps = physics.overlapSphere(
  [0.0, 4.0, 0.0],
  6.0,
  {
    triggerOnly: true
  }
);
```

overlapSphere() は、球状の範囲内に重なるボディを返します。範囲攻撃や近接判定に有用です。

各クエリでは、includeTriggers、triggerOnly、layerMask、filter などのオプションを併用して対象を絞り込むことができます。

26.8 コリジョンレイヤーとマスク

物理接触やクエリの対象を効率的に制御するために、コリジョンレイヤー（collision layer）とマスク（mask）を使用します。

```
const PLAYER = 1 << 0;
const ENEMY = 1 << 1;
const SENSOR = 1 << 2;

player.setCollisionLayer(PLAYER);
player.setCollisionMask(ENEMY | SENSOR);

enemy.setCollisionLayer(ENEMY);
enemy.setCollisionMask(PLAYER);
```

2 つのボディが接触候補になるには、双方のマスクが相手のレイヤーを含んでいる必要があります。

クエリ側では、layerMask オプションを使用して対象レイヤーを限定できます。

```
const enemyHits = physics.overlapSphere(player.getPosition(), 12.0, {
  layerMask: ENEMY
});
```

26.9 Scene JSON による物理設定の宣言

Scene JSON を使用して、physicsSpace の設定や各オブジェクトの physics プロパティを宣言できます。

```
{
  "version": "1.0",
```

```
"type": "webg-scene",
"physicsSpace": {
  "gravity": [0.0, -42.0, 0.0],
  "fixedTimeStepMs": 8.3333333333,
  "solverIterations": 6,
  "broadphaseMode": "sweepAabb",
  "defaultRestitution": 0.0,
  "defaultFriction": 0.4
},
"primitives": [
  {
    "id": "floor",
    "type": "cube",
    "args": [20.0],
    "transform": {
      "translation": [0.0, -1.0, 0.0],
      "scale": [1.0, 0.05, 1.0]
    },
    "physics": {
      "bodyType": "static",
      "collider": {
        "type": "plane",
        "normal": [0.0, 1.0, 0.0]
      },
      "material": {
        "restitution": 0.0,
        "friction": 0.7
      }
    }
  },
  {
    "id": "ball",
    "type": "sphere",
    "args": [1.0, 20, 16],
    "transform": {
      "translation": [0.0, 8.0, 0.0]
    },
    "physics": {
      "bodyType": "dynamic",
      "mass": 1.0,
      "velocity": [2.0, 0.0, 0.0],
      "angularVelocity": [0.0, 90.0, 0.0],
```

```
        "collider": {
          "type": "sphere",
          "radius": 1.0
        },
        "material": {
          "restitution": 0.7,
          "friction": 0.05
        }
      }
    }
  ]
}
```

`angularVelocity` は JavaScript API と同じく degree/sec です。上の例では、球体に Y 軸まわり毎秒 90 度の初期角速度を与えています。Scene JSON で `angularVelocity` を省略した場合は `[0.0, 0.0, 0.0]` として扱われます。

JavaScript 側では、シーンランタイムの `stepPhysics(deltaMs)` を呼び出して物理更新を行います。

```
const sceneRuntime = await app.loadScene(sceneObject);

app.start({
  onUpdate({ deltaMs }) {
    sceneRuntime.stepPhysics(deltaMs);
    sceneRuntime.update();
  }
});
```

26.10 物理更新のパイプライン

`PhysicsSpace.step(deltaMs)` の内部で実行される処理の流れを解説します。

固定タイムステップの管理

ブラウザの requestAnimationFrame は実行間隔が一定ではないため、PhysicsSpace は deltaMs をアキュムレータ（蓄積変数）に溜め、fixedTimeStepMs ごとに stepFixed(dtSec) を実行します。これにより、フレームレートの変動に関わらず物理挙動の安定性を維持しています。

ブロードフェーズ（広域判定）

ブロードフェーズは、「衝突する可能性があるペア」を粗く抽出する段階です。

既定では sweepAabb モードが採用されており、有限なコリジョンを x 軸方向にソートしてスキャンすることで、効率的に候補を絞り込みます。検証用に全組み合わせを確認する bruteForce モードも提供されています。PlaneCollider のような無限平面は特殊形状として扱い、常に候補に残るように制御されています。

ナローフェーズ（詳細判定）

ナローフェーズでは、ブロードフェーズで抽出された候補に対し、実際の接触法線（normal）とめり込み量（penetration）を算出します。

webg では、形状ごとの接触計算ロジックをコリジョンクラス側（Collider.buildContactWith()）に委譲しています。これにより、新しい形状を追加する場合でも PhysicsSpace 本体のロジックを変更することなく、形状固有の数式を追加するだけで拡張が可能です。

ソルバー（解決器）

ソルバーは、ナローフェーズで得られた接触情報を基に、位置と速度を補正します。

1. トリガーボディを物理応答から除外。
2. 必要に応じてスリープ状態のボディをウェイク（wake）させる。
3. めり込み量に応じて位置を押し戻す（位置補正）。
4. 接触点速度 v + omega x r を用いて、法線方向の反発インパルスを適用。

5. 接線方向の速度に対し、摩擦インパルスを適用。
6. 床上で落ち着きつつある box を、resting stabilizer で面接触側へ寄せる。
7. 線形速度と角速度の静止条件に基づき、スリープ候補を更新。

`solverIterations`（反復回数）を増やすことで、複数の物体が積み重なった際の安定性が向上しますが、計算負荷は増加します。

スリープとウェイク

ボディが静的物体に接触し、線形速度と角速度が一定しきい値以下になった状態が `sleepStepsThreshold` 回続くと、そのボディはスリープ状態に入ります。`sleepAngularThreshold` は degree/sec 単位です。スリープ中のボディは積分処理から除外されるため、計算コストを大幅に削減できます。アクティブなダイナミックボディやキネマティックボディと接触した際に、再びウェイクして物理計算に復帰します。

床上の `BoxCollider` については、sleep 判定の前に resting stabilizer が働く場合があります。この処理は、床面に倒れ込む方向の角速度だけを減衰し、床法線まわりの回転は sleep threshold を下回るまで残します。そのため、床上でまだ回転している箱は勝手には眠らず、角速度が十分に小さくなってから sleep します。

26.11 現状の機能範囲と制限事項

本物理エンジンで効率的に実現できるのは、以下のような用途です。

- 床へ落下する箱や球体。
- 低い壁に囲まれた範囲で跳ねる球体。
- 反発係数や摩擦による挙動の検証。
- トリガー領域によるイベント判定。
- レイキャストや AABB/球体オーバーラップによる空間問い合わせ。
- Scene JSON による基本的な物理ボディの宣言。

一方で、以下の項目は現在の実装範囲外であり、将来的な拡張候補として検討しています。

- `BoxCollider` のブロードフェーズにおける完全な OBB 最適化（現在は外接 AABB を使用）。

- CapsuleCollider の任意方向への対応（現在は y 軸方向固定）。
- 複数接点（contact manifold）の厳密な解決。
- 非対角慣性テンソルの完全なサポート。
- 連続衝突判定（Continuous Collision Detection: CCD）。
- ジョイント（Joint）やラグドール（Ragdoll）機能。

26.12 確認用ユニットテスト

物理挙動の正しさを検証するため、役割別に分かれたユニットテストを提供しています。

- unittest/physics_node_contracts: PhysicsNode 単体 API の規約確認。
- unittest/physics_space_contracts: 重力、固定タイムステップ、接触、クエリ、スリープ、レイヤーマスクなどの機能確認。
- unittest/physics_node_fall: ボックスが落下して床で停止する最小構成の視覚確認。
- unittest/physics_node_rotate: 角速度、トルク、回転固定（fixedRotation）の視覚確認。
- samples/physics_bounce: 複数の球体が床や壁、球体同士で跳ねる公開 sample。反発と相互作用を直感的に確認でき、?count=1000 で負荷検証も可能です。
- samples/compute_physics_bounce: 球体の線形運動と相互衝突を Compute Shader で処理する比較用 sample。GPU の ping-pong storage buffer から直接 instance 描画し、?count=512 まで負荷を確認できます。
- unittest/scene_loader_contracts: Scene JSON の設定が正しく読み込まれ、ランタイムに反映されるかの確認。

特に samples/physics_bounce は、物理エンジンらしい挙動を網羅的に確認できるため、動作検証の基準として活用してください。

Compute Shader 版との比較

samples/compute_physics_bounce は、PhysicsSpace を GPU へそのまま移植したものではありません。球体だけに形状を限定し、回転、スリープ、接触イベント、レイヤー、クエリ API を省略する代わりに、全球体の状態更新と描画を GPU 内で完結させています。

Compute Shader では 1 invocation が 1 球体を担当します。すべての invocation が同じ前状態 buffer を読み、自分の次状態だけを別 buffer へ書きます。この ping-pong 方式により、複数 invocation が同じ球体へ同時に速度を書き込む競合を避けています。

球体同士の判定は理解しやすさを優先した全組み合わせの走査であり、計算量は球数を N とすると $O(N^2)$ です。そのため、大量球体向けの完成された GPU 物理エンジンではなく、次の項目を確認するための応用例です。

- 固定上限を持つ Storage Buffer 上の剛体状態。
- 重力、減衰、位置積分。
- 球体と平面・壁との衝突。
- 等質量球体間の反発インパルスと簡易摩擦。
- Ping-pong Buffer を使った競合回避。
- Compute 結果を CPU へ戻さない instance 描画。

CPU 版 `physics_bounce` は接触イベントやスリープを含むアプリケーション向けの物理機能を確認し、Compute Shader 版は多数の独立状態を GPU で並列更新するデータフローを確認する、という役割の違いがあります。

26.13 まとめ

`webg` の物理エンジンは、既存のシーングラフと分断された別システムではなく、`PhysicsNode extends Node` という構成により、通常の `Space` 上に統合されています。これにより、見た目の `Shape`、配置の `Node`、物理判定の `Collider`、更新の `PhysicsSpace` を適切に分離しながら、一つのシーンとして一貫して扱うことができます。

利用者が意識すべき基本フローは以下の 4 点です。

1. 物理挙動をさせたい物体は `PhysicsNode` として生成する。
2. 見た目の `Shape` と物理判定の `Collider` は独立して設計する。
3. `dynamic` ボディの位置を直接変更せず、初期配置後に `dynamic` へ切り替える。
4. 毎フレーム `PhysicsSpace.step(deltaMs)` または `sceneRuntime.stepPhysics(deltaMs)` を呼び出す。

この基本フローに従うことで、重力、反発、摩擦、接触イベント、物理クエリといった機能を最小限のコードで実装できます。今後のアップデートにより、回転応答の厳密化やジョイン

ト機能の追加などが計画されており、より高度な物理シミュレーションへの拡張が可能です。

第 27 章

コンピュートシェーダーの基礎

27.1　本章の目的

本章では、これまで個別に学んできたシェーダー、画面効果、GPU リソース、アニメーション、シミュレーションの知識を統合し、コンピュートシェーダーによる並列計算へと発展させます。WGSL の構文や WebGPU の基本操作について、改めて詳しく繰り返すのではなく、必要に応じて前後の章を参照しながら読み進めてください。

本章では、特に以下の知識を拡張・統合します。

- **WGSL の拡張**: 第 8 章と第 9 章で学んだ構成を、コンピュートシェーダーのエントリポイント、ストレージバッファ、ストレージテクスチャ、ワークグループへと拡張します。
- **パス処理の発展**: 第 20 章のポストプロセスの考え方を、コンピュートパス（Compute Pass）による画像処理へと発展させます。
- **GPU 内完結の更新**: 第 21 章では CPU で粒子状態を更新していましたが、本章ではストレージバッファを用いて、更新から描画までを GPU 内で完結させる手法を扱います。
- **リソース管理の深化**: 第 22 章で学んだバッファやパイプラインの接続関係を基礎とし、パスの実行順序、明示的なバインディング、ディスパッチ、GPU タイムスタンプなどの詳細を確認します。
- **データ構造の継承**: 第 25 章のスキニングデータ構造を、SSAO やシャドウマップなどのコンピュート処理に接続して利用します。
- **シミュレーション設計**: 第 26 章の物理エンジンの考え方を、大量の粒子や頂点を GPU で並列更新する設計に適用します。

本章で解説する主要な処理フローは、以下のサンプルファイルで動作を確認できます。

- **book/examples/27_01.html**：`ComputePass` による画像コンピュート処理
- **book/examples/27_02.html**：ストレージバッファの 1 次元計算とリードバック（readback）
- **book/examples/27_03.html**：G-buffer 生成から SSAO 表示まで

これまで主に扱ってきたのは、頂点シェーダーとフラグメントシェーダーによる描画フローでした。頂点シェーダーで位置を決定し、フラグメントシェーダーで色を決定するという流れは、3D グラフィックスにおける標準的な手法です。

対してコンピュートシェーダー（Compute Shader）は、描画という制約に縛られず、GPU 上で「多数の小さな計算」を並列に実行するためのモデルです。1 ピクセルを単位とした画面加工はもちろん、1 粒子や 1 頂点、あるいは 1 つの物理オブジェクトを単位として、大量の状態を同時に更新することが可能です。

ただし、コンピュートシェーダーは単に処理が高速な万能機能ではありません。その本質的な価値は、画像、深度、法線、あるいは構造化された配列データを GPU 上に保持したまま、複数の処理段階へ順番に接続できる点にあります。本章では、ローレベル（低レイヤー）な実行モデルとリソースの仕様を中心に解説します。これらの機能を統合して高度な画面効果を構築するハイレベル（高レイヤー）な使い方は、次章で詳しく扱います。

27.2　描画とコンピュートのパイプライン比較

頂点・フラグメントシェーダーとコンピュートシェーダーは、いずれも GPU 上で動作するプログラムですが、その「起動される単位」が根本的に異なります。

- **頂点シェーダー**: 頂点バッファを入力とし、頂点ごとに起動し、クリップ空間の頂点位置を出力します。
- **フラグメントシェーダー**: ラスタライズ結果を入力とし、ピクセル候補ごとに起動し、Color Attachment の色を出力します。
- **コンピュートシェーダー**: 任意の Buffer や Texture を入力とし、`dispatchWorkgroups()` で指定した単位で起動し、Storage Buffer や Storage Texture へ出力します。

頂点・フラグメントシェーダーは「レンダーパイプライン（Render Pipeline）」の中で動作し、三角形の描画や深度テストといった標準的な描画フローに組み込まれています。

一方、コンピュートシェーダーは「コンピュートパイプライン（Compute Pipeline）」として独立して実行されます。三角形を描く必要はなく、JavaScript 側から「この範囲のデータを処理してほしい」と明示的に実行指示（ディスパッチ）を出します。例えば、画面効果であれば画面サイズと同じ数のピクセルを、粒子シミュレーションであれば粒子数と同じ数の要素を処理対象とします。

27.3　コンピュートシェーダーの実行モデル

コンピュートシェーダーを制御するために不可欠な概念として、「インボケーション（Invocation）」「ワークグループ（Workgroup）」「ディスパッチ（Dispatch）」の 3 つがあります。

- **インボケーション**: シェーダー関数の 1 回の実行単位です。画面処理では 1 ピクセル、粒子処理では 1 粒子に対応させることが一般的です。
- **ワークグループ**: 複数のインボケーションをまとめた実行グループです。GPU 内部で効率的に管理される単位となります。
- **ディスパッチ**: 必要な数のワークグループを GPU へ発行し、計算を開始させる操作です。

画面全体を処理する基本的な実装例を以下に示します。

```
@compute @workgroup_size(8, 8, 1)
fn main(@builtin(global_invocation_id) id : vec3<u32>) {
  let coord = vec2<i32>(id.xy);
  // coord がこのインボケーションの担当ピクセル座標となる
}
```

`@workgroup_size(8, 8, 1)` は、1 つのワークグループに $8 \times 8 \times 1 = 64$ 個のインボケーションを含める指定です。画面サイズが 1280×720 の場合、JavaScript 側では次のようにワークグループ数を計算し、切り上げてディスパッチします。

```
pass.dispatchWorkgroups(
  Math.ceil(width / 8),
  Math.ceil(height / 8),
  1
```

```
);
```

ここで注意が必要なのは、画面の幅や高さが必ずしもワークグループサイズの倍数とは限らない点です。切り上げてディスパッチすると、画面の外側を担当するインボケーションが発生します。そのため、WGSL の先頭で必ず範囲外アクセスを防止するガード処理を記述してください。

```
let size = textureDimensions(outputTexture);
if (id.x >= size.x || id.y >= size.y) {
  return;
}
```

この範囲確認を怠ると、ストレージテクスチャやストレージバッファの範囲外へアクセスし、WebGPU のバリデーションエラーや不定な計算結果を招く原因となります。

27.4　ストレージテクスチャとストレージバッファ

コンピュートシェーダーは、出力先として「ストレージテクスチャ（Storage Texture）」または「ストレージバッファ（Storage Buffer）」を使用します。

ストレージテクスチャは、コンピュートシェーダーから画像データとして直接書き込めるテクスチャです。

```
@group(0) @binding(1)
var outputTexture : texture_storage_2d<rgba8unorm, write>;
```

書き込みには `textureStore()` を使用します。

```
textureStore(outputTexture, coord, vec4f(color, 1.0));
```

ストレージバッファは、構造体や配列などの汎用データを読み書きするためのバッファです。粒子や物理オブジェクトの状態のように、画像形式ではなく「要素の配列」としてデータ

を更新したい場合に適しています。

```
struct Particle {
  positionLife : vec4f,
  velocitySize : vec4f,
};

@group(0) @binding(0)
var<storage, read_write> particles : array<Particle>;
```

なお、1 つのディスパッチ内で同じリソースを入力と出力の両方に使用する設計には注意してください。あるインボケーションが読み取ろうとしている値を、別のインボケーションが先に書き換えてしまうと、結果が GPU の実行順序に依存する「データ競合」が発生します。ブラー（ぼかし）や布のシミュレーションのように、前状態を読み取って次状態を書き込む処理では、2 つのリソースを交互に使用する「ピンポン（Ping-pong）」構成を採用します。

27.5　GPU 計算の 2 つの処理フロー

webg におけるコンピュートシェーダーの利用パターンは、大きく分けて「Render-first（描画後処理）」と「Compute-first（計算先行）」の 2 つのフローに分類されます。

Render-first（描画後処理）

先に 3D シーンをレンダーターゲット（RenderTarget）へ描き、その画像や深度情報をコンピュートシェーダーで後加工するフローです。Bloom、DoF、SSAO、ディファードライティング、SSR などのポストプロセスはこの系統に属します。

```
3D シーン Render Pass
  -> Compute Pass（エフェクト適用）
  -> FullscreenPass（画面へのコピー）
  -> canvas
```

このフローでは、標準の WebgApp.frame() を維持しながら、onBeforeDraw と onAfterDraw3d を使用して処理を挿入します。

Render-first の処理では、GPU 内部で扱う色と、canvas へ表示する色を分けて考えることが重要です。照明、影、SSAO、SSR、Bloom などの計算は、基本的に「値が物理量に近い意味を持つリニアカラー」として進めます。一方、画面に出す直前には、人間の目とディスプレイで自然に見えるように、明るすぎる値を圧縮するトーンマッピングと、表示用のガンマ変換を行います。

webg の `ComputeEffectToneMapPass` では、トーンマッピング方式として `reinhard` と `linear` を選択できます。`reinhard` は `color / (color + 1)` に近い考え方で、強い光をなだらかに圧縮するため、反射や Bloom を含むシーンで白飛びを抑えやすい方式です。`linear` は値を大きく変形せずに表示範囲へ収めるため、元の色の鮮やかさを確認したい調整用や、すでに明るさを制御できているシーンに向きます。

表示変換に関係する代表的なパラメータは、`exposure`、`saturation`、`gamma` です。`exposure` はトーンマッピング前の明るさを増減させ、写真の露出補正のように全体の明暗を調整します。`saturation` は輝度を保ちながら色の鮮やかさを調整し、SSR や影で色が眠く見える場合の確認に役立ちます。`gamma` は最終的なディスプレイ表示へ合わせる変換で、通常は `2.2` を基準にします。これはトゥーン表現の段階境界を調整する `gamma` とは目的が異なり、最終表示の明るさカーブを整えるための値です。

Render-first の負荷を下げるときは、必ずしもシーン全体を低解像度にする必要はありません。たとえば SSR は、G-buffer を full 解像度のまま保持し、反射を計算する storage texture だけを低解像度にできます。`ComputeSsrPass` の `resolutionScale` はこのための設定で、既定値は `0.7`、指定範囲は `0.5` から `1.0` です。反射は元のシーン色に混ぜられる補助成分なので、出力だけを少し低解像度にしても、通常の色や深度の読み取り精度は保てます。

もう一つ重要なのは、計算しない pixel を明示的に決めることです。SSR では G-buffer の albedo alpha を反射率として使うため、`albedo.a * intensity` が十分小さい pixel は、反射レイを探索しても最終結果にほとんど寄与しません。`ComputeSsrPass` の `reflectivityThreshold` はその判定値で、既定値は `0.05` です。`enabled: false` や `intensity: 0` の場合も shader 内で ray marching を省略します。このように「出力解像度を下げる」方法と「寄与しない pixel を早期に終える」方法を組み合わせると、見た目の情報量を大きく落とさずに GPU Compute 時間を下げやすくなります。

Compute-first（計算先行）

先に GPU 上の状態（粒子位置や物理状態など）を更新し、その最新結果をそのまま描画するフローです。GPU 粒子や布シミュレーション、GPU 物理などがこれに当たります。

```
Compute Pass（状態更新）
  -> Render Pass（最新状態の描画）
  -> submit
```

このフローでは、WebgApp に computeFrame: true を指定し、onComputeFrame ハンドラの中で 1 フレーム分の GPU コマンド発行から queue.submit() までの全行程を制御します。

27.6 ComputePass による処理の構築

ComputePass は、WGSL コード、バインディング定義、コンピュートパイプライン、ユニフォームバッファ、ディスパッチ計算を一つにまとめたローレベル（低レイヤー）なコアクラスです。

このクラスの目的は、内部処理をブラックボックス化することではなく、WGSL が要求するリソースを JavaScript 側でも「名前」「バインディング番号」「種類」として明示し、実装上の不整合を早期に検出することにあります。

以下に、入力画像を少し暗くしてストレージテクスチャへ書き込むシンプルな構成例を示します。

```
import ComputePass from "./webg/ComputePass.js";

const code = ‘
struct Params {
  values : vec4f,
};

@group(0) @binding(0) var<uniform> params : Params;
@group(0) @binding(1) var sourceTexture : texture_2d<f32>;
@group(0) @binding(2) var outputTexture : texture_storage_2d<rgba8unorm, write>;

@compute @workgroup_size(8, 8, 1)
fn main(@builtin(global_invocation_id) id : vec3<u32>) {
  let size = textureDimensions(outputTexture);
  if (id.x >= size.x || id.y >= size.y) {
    return;
  }
```

```
  let coord = vec2<i32>(id.xy);
  let color = textureLoad(sourceTexture, coord, 0);
  let gain = params.values.x;
  textureStore(outputTexture, coord, vec4f(color.rgb * gain, color.a));
}
`;

const darkenPass = new ComputePass(app.getGPU(), {
  label: "darken",
  code,
  workgroupSize: [8, 8, 1],
  uniformFloats: 4,
  bindings: [
    { binding: 0, name: "params", type: "uniform-buffer" },
    { binding: 1, name: "source", type: "sampled-texture" },
    {
      binding: 2,
      name: "output",
      type: "storage-texture",
      dispatchSize: true
    }
  ]
});

darkenPass.setUniforms(new Float32Array([0.82, 0.0, 0.0, 0.0]));
```

実行時は、現在動作しているレンダーパス（Render Pass）を閉じてから、同じコマンドエンコーダーへコンピュートパスを追加します。

```
app.getGPU().endPass();

darkenPass.encode(
  app.getGPU().commandEncoder,
  {
    source: sceneTarget,
    output: outputTarget
  },
  {
    timestampWrites: app.getGpuTimestampWrites(true, true)
  }
```

```
);
```

ComputePass 自体はコマンドエンコーダーを所有せず、queue.submit() も行いません。これにより、「シーン描画 → コンピュート効果 → フルスクリーンコピー」という 1 フレーム内の実行順序を、アプリケーション側で完全にコントロールできます。

最小の画像コンピュート処理をフレームへ組み込む

ここまでの断片を実際のアプリケーションへ組み込むには、入力となるシーン用レンダーターゲット（RenderTarget）、出力となるストレージテクスチャ、最終結果を canvas へ描く FullscreenPass が必要です。次の例は、前節の darkenPass を Render-first のフレームへ接続する全体の構成を示します。

シーンの Shape と Node は第 23 章までと同じ方法で app.space へ登録済みであるものとします。

```
import WebgApp from "./webg/WebgApp.js";
import ComputePass from "./webg/ComputePass.js";
import FullscreenPass from "./webg/FullscreenPass.js";
import StorageTargetFactory, {
  resizeTarget
} from "./webg/StorageTargetFactory.js";

const app = new WebgApp({
  document,
  autoDrawScene: false,
  clearColor: [0.04, 0.05, 0.07, 1.0]
});
await app.init();

// createScene() 内で Shape と Node を app.space へ追加する
createScene(app);

const sceneTarget = app.screen.createRenderTarget({
  label: "darken:scene",
  format: app.getGPU().format,
  hasDepth: true
});
```

```
const targetFactory = new StorageTargetFactory(app.getGPU(), {
  label: "darken:storage"
});
const outputTarget = targetFactory.create({
  label: "darken:output",
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});

const darkenPass = new ComputePass(app.getGPU(), {
  label: "darken",
  code,
  workgroupSize: [8, 8, 1],
  uniformFloats: 4,
  bindings: [
    { binding: 0, name: "params", type: "uniform-buffer" },
    { binding: 1, name: "source", type: "sampled-texture" },
    {
      binding: 2,
      name: "output",
      type: "storage-texture",
      format: "rgba8unorm",
      dispatchSize: true
    }
  ]
});

const copyPass = new FullscreenPass(app.getGPU(), {
  targetFormat: app.getGPU().format
});

await Promise.all([
  sceneTarget.ready,
  outputTarget.ready,
  copyPass.init()
]);

app.start({
  onUpdate: ({ screen }) => {
    // 入力と出力の pixel 対応を保つため、両方を同じ寸法へ変更する
    resizeTarget(sceneTarget, screen.getWidth(), screen.getHeight());
    resizeTarget(outputTarget, screen.getWidth(), screen.getHeight());
```

```
  },

  onBeforeDraw: () => {
    // 第 1 段階: 通常の 3D scene を offscreen target へ描く
    app.screen.beginPass({
      target: sceneTarget,
      clearColor: app.clearColor,
      colorLoadOp: "clear",
      depthClear: true
    });
    app.space.draw(app.eye);
  },

  onAfterDraw3d: () => {
    // 第 2 段階: scene Render Pass を閉じ、同じ command encoder へ Compute Pass を追加す
る
    darkenPass.setUniforms(
      new Float32Array([0.82, 0.0, 0.0, 0.0])
    );
    app.getGPU().endPass();
    darkenPass.encode(
      app.getGPU().commandEncoder,
      {
        source: sceneTarget,
        output: outputTarget
      }
    );

    // 第 3 段階: Storage Texture を canvas 用 Render Pass へコピーする
    app.screen.beginPass({
      clearColor: app.clearColor,
      colorLoadOp: "clear",
      depthView: null
    });
    copyPass.draw(outputTarget);
    app.screen.clearDepthBuffer();
  }
});
```

この例では、WebgApp がフレームのコマンドエンコーダーと最終的なサブミット（submit）を管理します。ストレージテクスチャはスワップチェインへ直接表示せず、最後に Fullscre

enPass から読み取って canvas へ描画します。

アプリケーションを終了してこれらのリソースが不要になった場合は、所有者が明示的に破棄します。

```
darkenPass.destroy();
outputTarget.destroy();
sceneTarget.destroy();
```

27.7　ストレージテクスチャとピンポン構成

コンピュートシェーダーの出力先となるテクスチャには、通常のレンダーターゲットとは異なる「Usage（用途）」設定が必要です。StorageTargetFactory は、コンピュートシェーダーから書き込み可能（STORAGE_BINDING）であり、かつ後段の処理でサンプリング読み取り可能（TEXTURE_BINDING）なレンダーターゲットを一貫した条件で生成するためのファクトリクラスです。

```
import StorageTargetFactory from "./webg/StorageTargetFactory.js";

const targetFactory = new StorageTargetFactory(app.getGPU(), {
  label: "main-storage",
  format: "rgba8unorm"
});

const outputTarget = targetFactory.create({
  label: "compute-output",
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await outputTarget.ready;
```

また、反復計算やフィードバックループを行う処理では、入力と出力を交互に入れ替える「ピンポン構成」が不可欠です。webg では用途に応じて 3 種類のピンポン管理クラスを提供しています。

- **@<tt>{PingPongBuffer}**: 2 本の GPUBuffer を管理。布シミュレーションや物理

状態の更新などに使用。

- **@<tt>{PingPongTexture}**: 2 枚の GPUTexture を管理。動的テクスチャ生成やセルオートマトンなどに使用。
- **@<tt>{PingPongTarget}**: 2 個の RenderTarget を管理。ブラーの反復や画像処理の多段適用などに使用。

PingPongTarget は StorageTargetFactory と組み合わせて利用することで、リソースの生成から切り替えまでを簡潔に記述できます。

```
const pingPong = targetFactory.createPingPong({
  label: "blur-pair",
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await pingPong.ready;

const source = pingPong.getCurrent();
const destination = pingPong.getNext();

// source から読み取り destination へ書き込む処理を記録した後、役割を交換する
pingPong.swap();
```

同じテクスチャを同時に読み書きすることを避ける設計は、コンピュートシェーダーにおける不可欠な指針です。これを怠ると、結果が GPU の内部的な実行順序に依存し、環境によって再現性のない不安定な挙動を招くことになります。

27.8 ストレージバッファによる 1 次元コンピュート処理

画像処理では 1 つのインボケーションを 1 ピクセルへ対応させましたが、粒子、頂点、物理状態などの配列を扱う場合は、global_invocation_id.x を配列のインデックス（index）として使います。

次の例は、入力配列の各値を 2 倍し、1 を加えた結果を別のストレージバッファへ書き込む処理です。

```
import ComputePass from "./webg/ComputePass.js";

const inputValues = new Float32Array([
  1.0, 2.0, 3.0, 4.0,
  5.0, 6.0, 7.0, 8.0
]);
const count = inputValues.length;

const inputBuffer = app.getGPU().device.createBuffer({
  label: "scale-bias:input",
  size: inputValues.byteLength,
  usage:
    GPUBufferUsage.STORAGE |
    GPUBufferUsage.COPY_DST
});
app.getGPU().queue.writeBuffer(inputBuffer, 0, inputValues);

const outputBuffer = app.getGPU().device.createBuffer({
  label: "scale-bias:output",
  size: inputValues.byteLength,
  usage:
    GPUBufferUsage.STORAGE |
    GPUBufferUsage.COPY_SRC
});

const readbackBuffer = app.getGPU().device.createBuffer({
  label: "scale-bias:readback",
  size: inputValues.byteLength,
  usage:
    GPUBufferUsage.COPY_DST |
    GPUBufferUsage.MAP_READ
});
```

WGSL では、入力を read、出力を read_write として区別します。params.x は要素数、params.y と params.z はスケール（scale）とバイアス（bias）です。

```
const scaleBiasWgsl = `
struct Params {
  values : vec4f,
};
```

```
@group(0) @binding(0)
var<uniform> params : Params;

@group(0) @binding(1)
var<storage, read> source : array<f32>;

@group(0) @binding(2)
var<storage, read_write> destination : array<f32>;

@compute @workgroup_size(64, 1, 1)
fn main(@builtin(global_invocation_id) id : vec3<u32>) {
  let index = id.x;
  let count = u32(params.values.x);
  if (index >= count) {
    return;
  }

  destination[index] =
    source[index] * params.values.y + params.values.z;
}
`;
```

JavaScript 側ではストレージバッファの種類をバインディング定義へ明示し、処理する要素数を dispatchSize へ渡します。

```
const scaleBiasPass = new ComputePass(app.getGPU(), {
  label: "scale-bias",
  code: scaleBiasWgsl,
  workgroupSize: [64, 1, 1],
  uniformFloats: 4,
  bindings: [
    { binding: 0, name: "params", type: "uniform-buffer" },
    {
      binding: 1,
      name: "source",
      type: "read-only-storage-buffer"
    },
    {
      binding: 2,
      name: "destination",
```

```
      type: "storage-buffer"
    }
  ]
});

scaleBiasPass.setUniforms(
  new Float32Array([count, 2.0, 1.0, 0.0])
);

const commandEncoder =
  app.getGPU().device.createCommandEncoder();

scaleBiasPass.encode(
  commandEncoder,
  {
    source: inputBuffer,
    destination: outputBuffer
  },
  {
    dispatchSize: [count, 1, 1]
  }
);

commandEncoder.copyBufferToBuffer(
  outputBuffer,
  0,
  readbackBuffer,
  0,
  inputValues.byteLength
);

app.getGPU().queue.submit([commandEncoder.finish()]);
```

結果を JavaScript から確認する場合は、GPU 処理の完了後にリードバックバッファをマップ（map）します。

```
await readbackBuffer.mapAsync(GPUMapMode.READ);

const result = new Float32Array(
  readbackBuffer.getMappedRange().slice(0)
```

```
);
readbackBuffer.unmap();

console.log(result);
// [3, 5, 7, 9, 11, 13, 15, 17]
```

リードバックは GPU と CPU の同期を必要とするため、デバッグや最終結果の取得には有効ですが、毎フレームの描画処理では避けます。粒子描画などでは、コンピュートシェーダーが更新したストレージバッファを頂点シェーダーから直接読み取り、GPU 内で処理を完結させます。

不要になったバッファとパスは所有者が破棄します。

```
scaleBiasPass.destroy();
readbackBuffer.destroy();
outputBuffer.destroy();
inputBuffer.destroy();
```

27.9 WGSL と JavaScript のメモリレイアウトの一致

ストレージバッファやユニフォームバッファへ構造体を渡す場合、JavaScript 側の配列順だけでなく、WGSL のアラインメント（alignment）、サイズ（size）、ストライド（stride）を一致させる必要があります。

代表的な型の基本的な配置は次の通りです。- `f32` / `u32` / `i32`: アラインメント 4byte / サイズ 4byte - `vec2f`: アラインメント 8byte / サイズ 8byte - `vec3f`: アラインメント 16byte / サイズ 12byte - `vec4f`: アラインメント 16byte / サイズ 16byte - `mat4x4f`: アラインメント 16byte / サイズ 64byte

特に `vec3f` はサイズが 12byte であっても、次のメンバーを置く位置は 16byte 境界の影響を受けます。JavaScript 側で 3 個の float だけを詰め、その直後へ別の値を置くと、WGSL が期待するオフセット（offset）と一致しない場合があります。

配列要素や頻繁に CPU から更新する構造体では、関連する値を `vec4f` 単位へまとめると配置を追跡しやすくなります。

```
struct Particle {
  positionLife : vec4f,
  velocityMass : vec4f,
  colorSize : vec4f,
};
```

この Particle は 3 個の vec4f で構成され、1 要素のストライドは 48byte です。JavaScript では 1 要素を 12 個の f32 として配置できます。

```
const FLOATS_PER_PARTICLE = 12;
const particleData = new Float32Array(
  particleCount * FLOATS_PER_PARTICLE
);

const offset = particleIndex * FLOATS_PER_PARTICLE;
particleData.set([
  px, py, pz, life,
  vx, vy, vz, mass,
  r, g, b, size
], offset);
```

ComputePass.setUniforms() が指定 float 数との完全一致を要求するのは、この不整合を GPU 実行前に検出するためです。構造体を変更するときは、「WGSL のメンバー順と型」「JavaScript 側のオフセットと要素数」「バッファ全体のバイトサイズと配列ストライド」の 3 点を同時に更新してください。

27.10 データ競合と同期

GPU では多数のインボケーションが並列に実行されます。シェーダーのコードが上から順に書かれていても、異なるインボケーションがその順序で実行される保証はありません。

インデックスごとに独立した更新

各インボケーションが自分のインデックス（index）だけを読み書きする処理は、同じストレージバッファを read_write で更新できます。

```
let index = id.x;
particles[index].position +=
  particles[index].velocity * deltaTime;
```

別のインボケーションが同じ particles[index] へ書き込まないことが条件です。GPU 粒子のように、1 インボケーションと 1 粒子を一対一に対応させる処理がこの形式です。

近傍を読む処理

布、ブラー、流体、セルオートマトンのように、現在要素だけでなく隣接要素も読む場合は注意が必要です。

```
let left = source[index - 1];
let center = source[index];
let right = source[index + 1];
destination[index] =
  (left + center + right) / 3.0;
```

この処理で source と destination を同じバッファにすると、あるインボケーションが書き換えた値を別のインボケーションが途中で読む可能性があります。前状態をすべて読み終えるまで保持する必要があるため、入力と出力を分け、ディスパッチ後にピンポンリソースの役割を交換します。

アトミック（Atomic）操作

複数のインボケーションが同じカウンターへ加算する場合、通常の read-modify-write では更新が失われます。この用途では atomic 型とアトミック関数を使います。

```
struct Counters {
  activeCount : atomic<u32>,
};

@group(0) @binding(0)
var<storage, read_write> counters : Counters;
```

```
if (particleIsActive) {
  atomicAdd(&counters.activeCount, 1u);
}
```

アトミック操作は更新を壊さずに行えますが、すべてのインボケーションが 1 つのカウンターへ集中すると並列性が低下します。ワークグループごとに部分集計し、別のパスで全体を合成するなど、競合箇所を減らす設計が必要です。

ワークグループメモリとバリア

var<workgroup> は、同じワークグループに属するインボケーションだけが共有できる高速な一時領域です。入力を一度ワークグループメモリへ読み込み、複数回再利用するブラーや集計処理に向いています。

```
var<workgroup> tile : array<f32, 64>;

@compute @workgroup_size(64, 1, 1)
fn main(
  @builtin(local_invocation_id) localId : vec3<u32>,
  @builtin(global_invocation_id) globalId : vec3<u32>
) {
  tile[localId.x] = source[globalId.x];

  // 全員が tile への書き込みを終えるまで待つ
  workgroupBarrier();

  let value = tile[localId.x];
  destination[globalId.x] = value;
}
```

workgroupBarrier() は同じワークグループ内の同期です。ストレージバッファへの書き込みを同じワークグループ内で同期する必要がある場合は storageBarrier() を使います。

バリアは、ワークグループ内の全インボケーションが同じ制御フローで到達する位置へ置く必要があります。インボケーションごとに条件が異なる if の内側へ置くと、一部のインボケーションがバリアに到達せず、正しい同期になりません。

ディスパッチ全体を一度停止し、すべてのワークグループの結果を次の計算へ渡す必要がある場合は、1 つのコンピュートパス内で解決しようとせず、パスを分けて実行順序を明示します。

```
Compute Pass A
  -> 中間 Buffer へ書く
Compute Pass B
  -> A の完成結果を読む
```

同じコマンドエンコーダーに順番に記録したパス間では、WebGPU がリソース利用の依存関係を管理します。

27.11　ワークグループサイズの決定

`@workgroup_size()` は、大きければ速いという値ではありません。1 ワークグループに含めるインボケーション数、使用するレジスタ、ワークグループメモリ、分岐、GPU アーキテクチャによって適切な値は変わります。

最初の候補として、以下の値から比較して判断してください。- **2 次元画像処理**: `8 x 8` または `16 x 16` - **1 次元配列処理**: 64、128、または 256

どの値を使う場合も、WGSL の `@workgroup_size()` と `ComputePass` へ渡す `workgroupSize` を一致させます。

```
@compute @workgroup_size(16, 16, 1)
```

```
const pass = new ComputePass(app.getGPU(), {
  // ...
  workgroupSize: [16, 16, 1]
});
```

デバイスが許容する上限は `device.limits` で確認できます。ただし、上限内であることが最適であることを意味しません。実際の候補を切り替え、同じシーン、解像度、要素数で `FrameTimer` の GPU コンピュート時間を比較してください。

比較では、初回パイプライン生成やブラウザの一時的な負荷を避け、一定フレーム数の移動平均を使います。また、ワークグループサイズを変更するとパイプラインが変わるため、同一実行中に数値を書き換えるのではなく、別パイプラインとして比較を行います。

画像処理では、ワークグループの形状も重要です。例えば 8 x 8 と 4 x 16 はどちらも 64 インボケーションですが、隣接ピクセルの読み方やメモリへのタイル形状が異なります。まずは処理対象の次元に合う正方形または連続した 1 次元構成を使い、計測結果に基づいて調整してください。

27.12 GPU 機能指定と FrameTimer

WebGPU の一部の高度な機能は、デバイス作成前に明示的に要求する必要があります。例えば、GPU 上の処理時間を精密に計測する timestamp-query はオプション機能（Optional Feature）として提供されています。

```
import WebgApp from "./webg/WebgApp.js";

const app = new WebgApp({
  document,
  gpu: {
    requiredFeatures: [],
    optionalFeatures: ["timestamp-query"]
  },
  frameTiming: true
});

await app.init();

if (!app.screen.hasGPUFeature("timestamp-query")) {
  console.log("GPU timestamp query is unavailable");
}
```

- **requiredFeatures**: 指定した機能がサポートされていない場合、初期化を停止してエラーを投げます。
- **optionalFeatures**: サポートされている場合のみ有効化し、未対応であっても動作を継続します。

また、frameTiming: true を指定すると内部的に FrameTimer が作動し、フレーム間隔、JavaScript の処理時間、GPU のコンピュート時間、GPU のレンダー時間を計測します。これにより、ボトルネックが CPU 側にあるのか、GPU の計算にあるのかを客観的に分析できます。

```
app.start({
  onUpdate: () => {
    app.message.setLines("timing", app.getFrameTimingLines());
  }
});
```

27.13 本章のまとめ

コンピュートシェーダーは、単にフラグメントシェーダーを置き換えるためのものではありません。その真価は、画像、深度、法線、そして構造化された配列データを GPU 上に保持したまま、自由自在な計算パイプラインを構築できる点にあります。

webg では、この強力な機能を安全かつ効率的に利用するための基盤を提供しています。ComputePass による厳密なリソース管理から、StorageTargetFactory による効率的なテクスチャ生成まで、段階的に機能を組み合わせてください。

ストレージバッファを扱う場合は、WGSL と JavaScript のメモリレイアウト、インボケーション間のデータ競合、ワークグループサイズ、そして CPU と GPU の同期範囲も処理設計の一部となります。次章では、これらの基礎知識を用いて、具体的なポストプロセスや高度な視覚効果を構築する方法を学びます。

第 28 章

コンピュートパスによる高度な表現

前章で学んだコンピュートシェーダーの基礎を踏まえ、本章ではそれらを活用した具体的な視覚効果の実装へと進みます。

ここでは、単一の計算処理を組み合わせ、シーン全体の質感を向上させる「ポストプロセス・パイプライン」の構築方法を詳しく解説します。

本章で扱う G-buffer 生成から SSAO 表示までの基本的な処理フローは、**book/examples/28_01.html** で確認できます。

28.1　個別のコンピュートポストプロセス

`ComputePass` は汎用的なローレベル（低レイヤー）クラスですが、実際のアプリケーションでは、ブラー、Bloom、DoF、トゥーン、輪郭抽出のように、特定の画像処理を一つの機能単位へまとめた個別パスを利用するのが効率的です。個別パスは、WGSL、バインディング、内部ストレージテクスチャを自前で所有しており、利用者はエフェクトのパラメータ設定と実行順序に集中できます。

分離可能ブラーによる高速化

`ComputeBlurPass` は、2 次元のブラーを「水平方向」と「垂直方向」の 2 回に分けて実行するコアクラスです。単純な 2 次元ブラーはピクセルあたり $N \times N$ 回のサンプリングが必要ですが、この分離（Separable Blur）により $N + N$ 回にまで計算コストを削減できます。

```
import ComputeBlurPass from "./webg/ComputeBlurPass.js";

const blurPass = new ComputeBlurPass(app.getGPU(), {
  label: "main-blur",
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await blurPass.ready;

app.getGPU().endPass();
const blurredTarget = blurPass.encode(
  app.getGPU().commandEncoder,
  sceneTarget,
  {
    radius: 3,
    iterations: 2
  }
);
```

Bloom による高輝度のにじみ

ComputeBloomPass は、「高輝度領域の抽出 → ブラー適用 → 元シーンとの合成」という一連のフローを一手に管理します。これにより、強い光が周囲ににじみ出す幻想的な表現を実現します。

```
import FullscreenPass from "./webg/FullscreenPass.js";
import ComputeBloomPass from "./webg/ComputeBloomPass.js";

const sceneTarget = app.screen.createRenderTarget({
  label: "bloom:scene",
  format: app.getGPU().format,
  hasDepth: true,
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await sceneTarget.ready;

const bloomPass = new ComputeBloomPass(app.getGPU(), {
```

```
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await bloomPass.ready;

const fullscreen = new FullscreenPass(app.getGPU(), {
  targetFormat: app.getGPU().format
});
await fullscreen.init();

app.start({
  onUpdate: () => {
    const width = app.screen.getWidth();
    const height = app.screen.getHeight();
    if (
      sceneTarget.getWidth() !== width ||
      sceneTarget.getHeight() !== height
    ) {
      sceneTarget.resize(width, height);
      bloomPass.resize(width, height);
    }
  },
  onBeforeDraw: () => {
    app.screen.beginPass({
      target: sceneTarget,
      clearColor: app.clearColor,
      depthClear: true
    });
    app.space.draw(app.eye);
  },
  onAfterDraw3d: () => {
    app.getGPU().endPass();

    const output = bloomPass.encode(
      app.getGPU().commandEncoder,
      sceneTarget,
      {
        threshold: 0.58,
        intensity: 1.2,
        softKnee: 0.35,
        strength: 1.4,
        exposure: 1.0,
```

```
        blurRadius: 3,
        blurIterations: 2,
        enabled: true
      }
    );

    app.screen.beginPass({
      clearColor: app.clearColor,
      colorLoadOp: "clear",
      depthView: null
    });
    fullscreen.draw(output);
    app.screen.clearDepthBuffer();
  }
});
```

被写界深度（DoF）の表現

ComputeDofPass は、シーンのカラー、3 段階のぼかし画像、そして深度情報を使い、焦点距離に応じた被写界深度（Depth of Field）を合成します。scene color を small / medium / large の 3 つの低解像度 target へ downsample してから各段階をぼかすため、表示品質と GPU 負荷を調整しやすい構成です。最終合成では、焦点距離からの差を focusRange で割って stage 位置を求め、scene -> small -> medium -> large の順に補間します。

```
import ComputeDofPass from "./webg/ComputeDofPass.js";

const dofPass = new ComputeDofPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await dofPass.ready;

app.getGPU().endPass();
const output = dofPass.encode(
  app.getGPU().commandEncoder,
  sceneTarget,
  {
    focusDistance: 36.0,
```

```
    focusRange: 7.0,
    maxBlurMix: 0.9,
    projectionNear: app.projectionNear,
    projectionFar: app.projectionFar,
    debugView: "composite",
    sharpnessWidth: 0.2,
    sharpnessPower: 8.0,
    blurRadius: 3,
    blurIterations: 2,
    sampleStep: 1,
    stageSmallScale: 0.7,
    stageMediumScale: 0.5,
    stageLargeScale: 0.3,
    enabled: true
  }
);
```

28.2 トゥーン表現の構築

トゥーン表現（セルルック）は、「色の段階化（ポスタリゼーション）」と「輪郭線の抽出」の 2 つの処理を組み合わせて構築します。

色の段階化による塗り表現

ComputeToonPass は、照明済みのシーンカラーを入力として受け取り、明るさを少数の明暗帯に分けることで、アニメ調の塗り表現を実現します。

明るさの量子化アルゴリズム

RGB の各チャンネルを個別に丸めると、色相が崩れやすくなります。そのため、ComputeToonPass では RGB の最大値をピクセルの強度（intensity）として取り出し、これを量子化します。

```
let intensity = max(max(source.r, source.g), source.b);
```

次に、強度を levels で指定した段階数へ量子化します。ここで gamma パラメータを用いることで、量子化の境界を暗部寄り、あるいは明部寄りへ調整できます。

```
fn quantizeIntensity(
  intensity : f32,
  levels : f32,
  gammaValue : f32
) -> f32 {
  let safeGamma = max(gammaValue, 0.0001);
  let encoded = pow(clamp(intensity, 0.0, 1.0), safeGamma);
  let stepCount = max(levels - 1.0, 1.0);
  let band =
    floor(encoded * stepCount + 0.5) / stepCount;
  return pow(
    clamp(band, 0.0, 1.0),
    1.0 / safeGamma
  );
}
```

量子化した値は、floor （最暗部の明るさ）から 1.0 までの範囲へ写し、元の RGB 全体へ同じ倍率を掛け合わせます。これにより、色相を維持したまま明るさだけを置き換えることができます。

ComputeToonPass の利用方法

```
import ComputeToonPass from "./webg/ComputeToonPass.js";

const toonPass = new ComputeToonPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await toonPass.ready;

app.getGPU().endPass();

const toonColor = toonPass.encode(
  app.getGPU().commandEncoder,
  sceneTarget,
  {
```

```
    levels: 4,
    strength: 1.0,
    gamma: 1.0,
    floor: 0.28,
    enabled: true
  }
);
```

パラメータ	意味	有効範囲
levels	明暗の段階数	2 〜 16
strength	元の色とトゥーン色の混合比	0.0 〜 1.0
gamma	強度分布の調整	$0.0 < \gamma \leq 4.0$
floor	最も暗い帯の明るさ	0.0 〜 1.0

輪郭線の抽出と合成

ComputeEdgePass は、シーンカラーの輝度差、または G-buffer の法線差と深度差から輪郭を検出し、入力カラーへ線を合成します。

2 種類の輪郭検出

1. **Color Edge**: シーンカラーを輝度に変換し、Sobel フィルタで勾配を検出します。テクスチャの模様や照明の境界も拾いやすいため、漫画的な詳細表現に向いています。
2. **Geometry Edge**: 法線の変化（面の折れ目）と深度の不連続（物体外形）から検出します。純粋な形状としての輪郭を抽出したい場合に有効です。

線の太さと合成モード

検出した輪郭は、周囲の最大値を取るダイレーション処理によって太くします。また、blendMode によって線の合成方法を選択できます。

- "black-multiply": 元の色を乗算して暗くする（自然な黒線）。
- "black-subtract": 元の色から減算する（より強い黒線）。
- "white-add": 元の色に加算する（発光線）。

ComputeEdgePass の利用例

```
import ComputeEdgePass from "./webg/ComputeEdgePass.js";

const edgePass = new ComputeEdgePass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await edgePass.ready;

const outlined = edgePass.encode(
  app.getGPU().commandEncoder,
  displayColor,
  {
    strength: 1.0,
    threshold: 0.16,
    mix: 1.0,
    thickness: 2,
    blendMode: "black-multiply",
    colorEnabled: true,
    geometryEnabled: false,
    enabled: true
  }
});
```

Geometry Edge を使用する場合は、G-buffer から法線、深度、および投影パラメータを渡す必要があります。

```
const resources = gbuffer.getBindingResources();

const outlined = edgePass.encode(
  app.getGPU().commandEncoder,
  displayColor,
  {
    normal: resources.normal,
    depth: resources.depth,
    projection,
    strength: 1.0,
    threshold: 0.16,
```

```
    mix: 1.0,
    thickness: 2,
    blendMode: "black-multiply",
    colorEnabled: false,
    geometryEnabled: true,
    normalWeight: 1.0,
    depthWeight: 1.0,
    enabled: true
  }
);
```

トゥーン表現の統合フロー

トゥーン色と輪郭線を組み合わせる際は、以下の順序で処理を接続します。

```
linear scene color
  -> ComputeToonPass（色段階化）
  -> DoF / Bloom（リニア空間でのエフェクト）
  -> tone mapping と表示用ガンマ変換
  -> ComputeEdgePass（最終的な輪郭合成）
  -> canvas
```

ComputeToonPass は照明結果の意味が残っているリニア空間で適用し、ComputeEdgePass は表示変換後の最終的な色に対して線を加えます。これにより、色段階と輪郭の太さを独立して調整でき、破綻のないトゥーン表現が可能です。

28.3 G-buffer を用いた高度なコンピュート描画

単なるカラー画像ではなく、法線や深度といった幾何学情報を併せて利用することで、遮蔽、反射、影などの形状に基づく高度な表現が可能になります。これらの情報をまとめて保存する仕組みを「G-buffer（ジオメトリバッファ）」と呼びます。

GeometryBufferPass は、シーンから不透明な形状を収集し、カラー、ビュー空間法線、深度を同時に生成します。

```
import {
  GeometryBufferPass,
  createGBufferProjectionParams,
  createViewMatrix
} from "./webg/GeometryBufferPass.js";

const gbuffer = new GeometryBufferPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight(),
  colorMode: "material",
  normalSpace: "view"
});
await gbuffer.ready;

const vfov =
  app.screen.getRecommendedFov(55.0) * Math.PI / 180.0;
const projection = createGBufferProjectionParams(
  app.projectionNear,
  app.projectionFar,
  vfov,
  app.screen.getAspect()
);

gbuffer.renderSpace(app.space, app.projectionMatrix, app.eye, app.clearColor);
```

環境遮蔽（SSAO）

SsaoPass は、G-buffer の法線と深度から、物体同士の隙間や隅などの「遮蔽（Occlusion）」を近似的に計算し、柔らかい影を付け加えます。

```
import SsaoPass from "./webg/SsaoPass.js";

const ssaoPass = new SsaoPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight(),
  radius: 18,
  strength: 1.15,
  bias: 0.08,
```

```
  samples: 16,
  resolutionScale: 0.7
});
await ssaoPass.ready;

gbuffer.renderSpace(app.space, app.projectionMatrix, app.eye, app.clearColor);
app.getGPU().endPass();

const output = ssaoPass.encode(
  app.getGPU().commandEncoder,
  gbuffer.getBindingResources(),
  {
    projection,
    enabled: true,
    radius: 18,
    strength: 1.15,
    bias: 0.08,
    samples: 16,
    resolutionScale: 0.7,
    view: "composite"
  }
);
```

resolutionScale は、SSAO の raw AO target だけを低解像度化する倍率です。既定値は 0.7 で、0.5 から 1.0 の範囲を指定できます。G-buffer の color、normal、depth と最終的な合成結果はフル解像度のまま保たれるため、画面全体を縮小するのではなく、AO の生成にかかる画素数を減らす設定です。フル解像度へ合成する段階では、depth と normal による重み付けを使い、物体境界を越えた AO のにじみを抑えます。

ディファード・ライティング

DeferredLightingPass は、G-buffer と点光源の配列を読み、ピクセルごとに効率的にライティングを評価します。光源数が増えても、ジオメトリの描画回数が変わらないため、非常に効率的な手法です。

```
import DeferredLightingPass from "./webg/DeferredLightingPass.js";

const deferredPass = new DeferredLightingPass(app.getGPU(), {
```

```
  width: app.screen.getWidth(),
  height: app.screen.getHeight(),
  maxLights: 128
});
await deferredPass.ready;

const lights = [{
  position: [0.0, 2.5, -4.0],
  color: [1.0, 0.55, 0.18],
  radius: 9.0,
  intensity: 3.2
}];

gbuffer.renderSpace(app.space, app.projectionMatrix, app.eye, app.clearColor);
app.getGPU().endPass();

const output = deferredPass.encode(
  app.getGPU().commandEncoder,
  gbuffer.getBindingResources(),
  {
    projection,
    viewMatrix: createViewMatrix(app.eye),
    lights,
    lightCount: lights.length,
    view: "lighting"
  }
);
```

スクリーンスペース反射（SSR）

ComputeSsrPass は、G-buffer の情報を使い、画面内の情報を鏡のように反射させる SSR を実現します。

```
import ComputeSsrPass from "./webg/ComputeSsrPass.js";

const ssrPass = new ComputeSsrPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
```

```
});
await ssrPass.ready;

gbuffer.renderSpace(app.space, app.projectionMatrix, app.eye, app.clearColor);
app.getGPU().endPass();

const output = ssrPass.encode(
  app.getGPU().commandEncoder,
  gbuffer.getBindingResources(),
  {
    projection,
    enabled: true,
    intensity: 0.82,
    distance: 42.0,
    thickness: 0.42,
    steps: 48,
    resolutionScale: 0.7,
    reflectivityThreshold: 0.05,
    view: "composite"
  }
);
```

`ComputeSsrPass` の出力は、反射色そのものを RGB へ、反射の有効度を alpha へ格納します。反射の有効度は、マテリアルの反射率、レイ探索が交差したかどうか、距離や画面端での減衰を含む重みです。

SSR は各 pixel で反射レイを探索するため、画面解像度と反射率の分布がそのまま負荷に影響します。`resolutionScale` は SSR の出力 target だけを低解像度化する設定で、既定値は `0.7`、指定範囲は `0.5` から `1.0` です。G-buffer は full 解像度のまま読み、低解像度の SSR 出力だけを後段で拡大して合成します。`reflectivityThreshold` は、`albedo.a * intensity` が指定値以下の pixel で ray marching を省略するための閾値です。既定値は `0.05` で、低反射の壁や物体が多い scene では GPU Compute 時間を抑えやすくなります。

この構成にしておくと、後段の `ComputeEffectComposer` で合成方法を選びやすくなります。`add` 合成では反射色を元の色へ足し込み、濡れた床や強い光沢のような明るい反射を作ります。`mix` 合成では反射重みに応じて元の色と反射色を混ぜるため、反射を強くしても画面全体が白っぽくなりにくく、色の鮮やかさを残したい材質調整に向きます。

SSR 交差探索の最適化

ComputeSsrPass では、反射レイの交差探索に「動的 Coarse Step + 二分探索」という手法を採用しています。

固定間隔でレイを進める方式では、カメラ距離や FOV によって探索密度が変わり、曲面に帯（バンディング）が出やすくなります。これを解決するため、画面上のレイ移動量に応じてステップ数を動的に決定し、候補区間が見つかったときだけ二分探索で精密化します。

このアプローチにより、計算負荷を抑えつつ、物体寸法やカメラ条件に依存しにくい安定した反射結果を得ることができます。

シャドウマップによる影の評価

影の描画は、光源視点の深度を保存する「シャドウマップ生成」と、カメラ視点での「影判定」の 2 段階で行います。

Directional Light（平行光源）

ShadowMapPass で深度マップを作成し、ComputeShadowPass で G-buffer のワールド位置をライト空間へ投影して比較します。

```
import ShadowMapPass, {
  createDirectionalLightMatrices
} from "./webg/ShadowMapPass.js";
import ComputeShadowPass from "./webg/ComputeShadowPass.js";

const shadowMap = new ShadowMapPass(app.getGPU(), {
  width: 1024,
  height: 1024
});
await shadowMap.ready;

const shadowPass = new ComputeShadowPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
```

```
await shadowPass.ready;

const light = createDirectionalLightMatrices({
  direction: [0.55, -1.0, 0.38],
  target: [0.0, -0.5, -1.0],
  distance: 35.0,
  halfWidth: 21.0,
  halfHeight: 18.0,
  near: 1.0,
  far: 75.0
});

shadowMap.renderSpace(app.space, light.viewProjection);
gbuffer.renderSpace(app.space, app.projectionMatrix, app.eye, app.clearColor);

app.getGPU().endPass();
const output = shadowPass.encode(
  app.getGPU().commandEncoder,
  {
    ...gbuffer.getBindingResources(),
    ...shadowMap.getBindingResources()
  },
  {
    projection,
    cameraWorld: app.eye.getWorldMatrix(),
    lightViewProjection: light.viewProjection,
    lightDirection: light.direction,
    bias: 0.0015,
    normalBias: 0.003,
    ambient: 0.18,
    directIntensity: 1.0,
    pcfRadius: 1,
    view: "composite"
  }
);
```

ambient は、影の中でも最低限残す明るさを指定します。0.0 に近づけるほど影側は黒くなり、1.0 に近づけるほど影の有無が弱くなります。室内や模型のようなシーンでは、影の形を見たいときは低め、影で形状が潰れるときは少し高めに設定します。

directIntensity は、光が当たっている面の直接光の強さを調整します。ambient が影側

の底上げであるのに対し、directIntensity は照明側の明るさとコントラストを変えるための値です。反射やトーンマッピングと組み合わせる場合、照明側が白っぽく感じるときは directIntensity を下げ、光の向きを強く見せたいときは上げます。

Spot Light（スポットライト）

SpotShadowMapPass を使い、円錐状の照射範囲を考慮して深度を作成します。判定側では ComputeSpotShadowPass を用い、深度比較に加えて円錐の内外判定を行います。

```
import SpotShadowMapPass, {
  createSpotLightMatrices
} from "./webg/SpotShadowMapPass.js";
import ComputeSpotShadowPass from "./webg/ComputeSpotShadowPass.js";

const spotShadowMap = new SpotShadowMapPass(app.getGPU(), {
  width: 1024,
  height: 1024
});
await spotShadowMap.ready;

const spotShadowPass = new ComputeSpotShadowPass(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight()
});
await spotShadowPass.ready;

const spotLight = createSpotLightMatrices({
  position: [0.4, 1.8, 2.8],
  direction: [0.0, -0.22, -1.0],
  fov: 70.0,
  near: 0.05,
  far: 42.0
});

spotShadowMap.renderSpace(app.space, spotLight.viewProjection);
gbuffer.renderSpace(app.space, app.projectionMatrix, app.eye, app.clearColor);

app.getGPU().endPass();
const spotOutput = spotShadowPass.encode(
  app.getGPU().commandEncoder,
  {
```

```
    ...gbuffer.getBindingResources(),
    ...spotShadowMap.getBindingResources()
  },
  {
    projection,
    cameraWorld: app.eye.getWorldMatrix(),
    lightViewProjection: spotLight.viewProjection,
    lightPosition: spotLight.position,
    lightDirection: spotLight.direction,
    innerCos: Math.cos(40.0 * Math.PI / 180.0),
    outerCos: Math.cos(50.0 * Math.PI / 180.0),
    bias: 0.0015,
    normalBias: 0.003,
    ambient: 0.10,
    directIntensity: 1.0,
    pcfRadius: 1,
    view: "composite"
  }
);
```

スポットライトでも、ambient と directIntensity の意味は平行光源と同じです。違いは、直接光がライトの円錐範囲内でだけ有効になる点です。innerCos と outerCos の間では明るさがなだらかに減衰するため、directIntensity を上げると照射範囲の中心が強く見え、ambient を上げると円錐の外側や影側が見やすくなります。

28.4 シミュレーションと描画の統合：GpuParticleEmitter

画像処理ではなく、GPU 上のデータを更新してそのまま描画に利用する場合、前章で触れた「Compute-first（計算先行）」の構成をとります。WebgApp に computeFrame: true を指定すると、onComputeFrame ハンドラの中で、計算から描画までの全行程をコントロールできます。

この構成で最も強力なのが GpuParticleEmitter です。これは、粒子の状態を保持するストレージバッファ、計算パイプライン、描画パイプラインを統合的に管理するクラスです。

```
import GpuParticleEmitter from "./webg/GpuParticleEmitter.js";
```

```
const emitter = new GpuParticleEmitter(app.getGPU(), {
  label: "main-particles",
  particleCount: 32768,
  floatsPerParticle: 12,
  workgroupSize: 128,
  paramFloats: 20,
  targetFormat: app.getGPU().format,
  initialData: createInitialParticleData(),
  computeCode: createParticleComputeWGSL(),
  renderCode: createParticleRenderWGSL()
});

app.start({
  onComputeFrame: (ctx) => {
    const gpu = app.getGPU();
    const commandEncoder = gpu.device.createCommandEncoder();
    app.beginGpuTiming();

    emitter.writeParams(buildParticleParams(ctx.timeSec, ctx.deltaSec));
    emitter.encodeCompute(commandEncoder, {
      timestampWrites: app.getGpuTimestampWrites(true, true)
    });

    const colorView = gpu.context.getCurrentTexture().createView();
    emitter.encodeRender(commandEncoder, {
      timestampWrites: app.getGpuRenderTimestampWrites(),
      colorView,
      depthView: gpu.depthView,
      clearColor: app.clearColor
    });

    app.endGpuTiming(commandEncoder);
    gpu.queue.submit([commandEncoder.finish()]);
    app.afterGpuSubmit();
  }
});
```

GpuParticleEmitter の最大の利点は、コンピュートシェーダーで更新したストレージバッファを、そのまま頂点シェーダーから読み取れる点です。データを CPU へ戻す（リードバック）必要がなく、GPU 内部で完結するため、数万個の粒子を極めて高速に描画できます。

28.5　パイプラインの統合と運用

リサイズとリソースの整合性

コンピュート処理では、入力と出力のリソース寸法が一致していることが不可欠です。キャンバスがリサイズされた際は、G-buffer やエフェクトパス、ストレージテクスチャをすべて同じ寸法に更新してください。

```
const resizeIfNeeded = (target, width, height) => {
  if (target.getWidth() === width &&
      target.getHeight() === height) {
    return false;
  }
  target.resize(width, height);
  return true;
};

const width = app.screen.getWidth();
const height = app.screen.getHeight();

if (resizeIfNeeded(sceneTarget, width, height)) {
  gbuffer.resize(width, height);
  ssaoPass.resize(width, height);
  deferredPass.resize(width, height);
  ssrPass.resize(width, height);
  shadowPass.resize(width, height);
  bloomPass.resize(width, height);
  dofPass.resize(width, height);
}
```

デバッグビューによる切り分け

多段に繋げたコンピュート処理では、最終結果だけを見て不具合を探すのは困難です。そのため、webg の各パスには専用のデバッグビューが搭載されています。

パス	代表的なビュー	確認内容
SsaoPass	ao, normal	遮蔽係数のみの抽出、入力法線の正当性
DeferredLightingPass	albedo, normal, depth	G-buffer 各成分の正当性
ComputeSsrPass	reflection, normal	反射ヒット状況、入力情報の正当性
ComputeShadowPass	shadow, albedo	影マスク領域、入力情報の正当性
ComputeDofPass	depth, focus	深度の線形化、焦点面と多段階ぼかしの範囲

不具合発生時は、「入力リソース（G-buffer）」 → 「中間結果（AO 係数など）」 → 「合成結果」の順で確認してください。

28.6　トラブルシューティング

コンピュートシェーダー実装における頻出のトラブルと解決策をまとめます。

- **画面が真っ黒になる**: レンダーパスを閉じずにコンピュートパスを開始していないか、ストレージテクスチャの Usage に STORAGE_BINDING が設定されているか、最終的に FullscreenPass でキャンバスへ戻しているかを確認してください。
- **結果が更新されない**: queue.writeBuffer() によるユニフォーム更新漏れや、dispatchWorkgroups() の回数が不足して全ピクセルを処理できていない可能性があります。
- **数値が崩れる**: WGSL の構造体と JavaScript 側のメモリレイアウト（アラインメント、ストライド）が一致しているか確認してください。特に vec3f のパディングに注意してください。
- **ちらつきが発生する**: 同一リソースの読み書き競合が起きている可能性があります。入力と出力をピンポンリソースへ分離してください。
- **ノイズや不自然な帯が出る**: 投影パラメータ（Near, Far, FOV, Aspect）がシーン描画時と一致しているか確認してください。シャドウマップの場合は bias を調整してセルフシャドウ（Shadow Acne）を抑制します。
- **トゥーン表現が不自然**: トーンマッピングやガンマ変換の「後」に段階化を行うと、明暗帯が潰れやすくなります。必ず表示変換の「前」に ComputeToonPass を配置してください。

28.7 まとめ

本章では、コンピュートシェーダーを用いて、単純な画面加工から G-buffer を利用した高度なポストプロセス、そして GPU シミュレーションまでを実装する方法を学びました。

コンピュートシェーダーの真価は、画像、深度、法線、構造化配列といった多様なデータを GPU 上に保持し、それらを自由な順序で加工・合成できるパイプラインを構築できる点にあります。

まずは個別のパスによる視覚効果の実装から始め、徐々にそれらを統合した複雑なパイプラインへと発展させてください。これにより、WebGPU が提供する圧倒的な計算能力を最大限に引き出し、高品質なグラフィックス表現を実現することができます。

第 29 章

リアルタイム 3D 表現の統合

29.1　本章の目的

3D アプリケーションの表現力を高める方法として、物理ベースレンダリング（Physically Based Rendering, PBR）は極めて重要です。しかし、表面の材質を物理量として精密に記述することだけが、画面の視覚的な説得力を高める唯一の方法ではありません。

物体が床に接していること、光を遮った物体が別の物体へ影を落とすこと、周囲の物体が光沢面へ映り込むこと。これら「空間的な関係」を視覚化することが、空間の理解を助ける決定的な情報となります。

- **SSAO (Screen Space Ambient Occlusion)**: 接地部、隅、近接面へ局所的な遮蔽（暗がり）を加えます。
- **シャドウマップ**: 光源、遮蔽物、影を受ける面の三者の関係を正確に表します。
- **SSR (Screen Space Reflection)**: 視点から見えている周囲の形状を、鏡面や光沢面へ映し出します。

本章では、前章（第 28 章）で解説した個別パスを、標準の `WebgApp`、`Space`、`Shape` へ統合する方法を詳しく解説します。本書の「高水準（ハイレベル）API を優先する」方針に合わせ、まずは `ComputeEffectPipeline` による統合的な利用方法を示し、その後で G-buffer と個別パスへ分解して、具体的な処理順と入力仕様を確認します。

通常のアプリケーション開発では、まず高水準 API から始めてください。中間リソースの確認、独自エフェクトの研究、あるいは不具合の切り分けが必要な場合にのみ、ローレベル（低レイヤー）API へ降りて詳細を制御する構成としています。`ComputePass`、ストレージテ

クスチャ、ワークグループについては第 27 章を、個別パスの内部仕様については第 28 章を参照してください。

29.2 比較の基準となるシーン

まず、床、壁、立方体、球、柱を標準の Shape と Primitive で作成したシーンを用意します。特殊なレンダラーは使用せず、WebgApp がシーングラフを通常どおり描画する構成とします。

実行例は **book/examples/29_01.html** で確認できます。SSAO、Shadow Map、SSR を同じシーンへ統合した結果は、**book/examples/29_02.html** で確認できます。

Shape を作成する処理は、これまでの章と同様です。

```
function createPrimitiveShape(gpu, primitiveFactory, material) {
  const shape = new Shape(gpu);
  shape.applyPrimitiveAsset(
    primitiveFactory(shape.getPrimitiveOptions())
  );
  shape.endShape();
  shape.setMaterial("smooth-shader", {
    has_bone: 0,
    use_texture: 0,
    ...material
  });
  return shape;
}
```

WebgApp には autoDrawScene を指定しません。既定値の通常描画が使われるため、アプリケーションは Shape を Space へ登録し、onUpdate でカメラとアニメーションを更新するだけで動作します。

```
const app = new WebgApp({
  document,
  renderMode: "continuous",
  clearColor: [0.045, 0.065, 0.09, 1],
  camera: {
    target: [0, -0.7, -4.0],
```

```
    distance: 27,
    yaw: 24,
    pitch: -13
  }
});
await app.init();

createScene(app);

app.start({
  onUpdate: ({ deltaSec }) => {
    movingCube.rotateY(18 * deltaSec);
  }
});
```

この簡潔な実装が、本章における比較の基準となります。SSAO などの高度な効果を追加する際、シーングラフや Shape の作り直しを強いるのではなく、既存の構成を維持したまま統合できることが重要です。

29.3　PBR とは異なる「空間的な」表現力

本章で扱う例は、金属度、粗さ、IBL（Image Based Lighting）を備えた厳密な PBR レンダラーではありません。しかし、通常描画と比較すると、次のような決定的な情報が増加します。

- **SSAO の導入**: 物体が床から浮いているのか接しているのか、壁の隅がどちらへ折れているのかという「接地感」が読み取りやすくなります。
- **シャドウマップの導入**: 光がどちらから差し込み、どの物体が別の物体を遮っているかという「光の方向性と遮蔽」が明確になります。
- **SSR の導入**: 光沢面が単なる色面ではなく、同じ空間に存在する周囲の材質を映し出すことで、「空間の一体感」が現れます。

これらは材質の物理モデルを精密化するアプローチではなく、シーン内の「空間的関係」を増やす方向の表現力です。そのため、単純なプリミティブだけのシーンであっても十分に効果を発揮し、頂点数やアセット制作量を増やすことなく、画面の情報量を劇的に高めることが可能です。

WebGPU のコンピュートシェーダーをコアへ統合した意義は、単に特殊なデモを作ることではなく、標準の WebgApp と Shape で構築した既存シーンへ、こうした情報を段階的に追加できる拡張性にあります。

29.4　高水準 API によるエフェクトの統合

実際のアプリケーションでは、SSAO、シャドウマップ、SSR だけでなく、トゥーン、DoF、Bloom、エッジなどを組み合わせて使用します。これらを個別に手作業で接続する前に、まずは ComputeEffectPipeline を使用してください。

複数のエフェクトを統合する際に最も重要なのは、「直列に繋ぐこと」ではなく、「色空間の整合性を保つこと」です。どの段階で照明途中のリニアカラーを扱い、どの段階で表示用のトーンマッピングとガンマ変換を行うのかを整理しないと、見た目が不安定になります。

WebGPU コアは、この役割を統括する ComputeEffectPipeline、SSR 反射をカラーへ合成する ComputeEffectComposer、および表示変換を担当する ComputeEffectToneMapPass を提供しています。

```
import ComputeEffectPipeline from "./webg/ComputeEffectPipeline.js";

const effects = new ComputeEffectPipeline(app.getGPU(), {
  width: app.screen.getWidth(),
  height: app.screen.getHeight(),
  gbufferColorMode: "litMaterial",
  shadowMapSize: 1536,
  ssao: {
    enabled: true,
    radius: 22,
    strength: 1.55
  },
  shadow: {
    type: "directional",
    enabled: true,
    ambient: 0.18,
    directIntensity: 1.0
  },
  ssr: {
    enabled: true,
    intensity: 0.72
```

```
  },
  composer: {
    mode: "mix"
  },
  toneMap: {
    mode: "reinhard",
    exposure: 1.0,
    saturation: 1.0,
    gamma: 2.2,
    blackBackground: false
  },
  toon: {
    enabled: false,
    levels: 4
  },
  dof: {
    enabled: false
  },
  bloom: {
    enabled: false
  },
  edge: {
    enabled: false,
    blendMode: "black-multiply"
  }
});
await effects.ready;
```

この高水準 API は、内部で GeometryBufferPass や SsaoPass、ComputeSsrPass などの個別パスを所有し、最適な順序で呼び出します。各パスの入力・出力仕様は第 28 章で説明した通りであるため、高水準 API で問題が起きた際も、個別パスのデバッグビューを使って段階的に原因を切り分けることが可能です。

shadow.type では "directional"（平行光源）と "spot"（スポットライト）を選択できます。directional shadow はシーン全体を覆う表現に向き、spot shadow は懐中電灯のように特定の範囲を照らす表現に向いています。

```
const effects = new ComputeEffectPipeline(app.getGPU(), {
  width: app.screen.getWidth(),
```

```
  height: app.screen.getHeight(),
  shadow: {
    type: "spot",
    enabled: true,
    ambient: 0.18,
    directIntensity: 1.0,
    spot: {
      position: [0.4, 1.8, 2.8],
      direction: [0.0, -0.22, -1.0],
      fov: 70.0,
      innerAngle: 40.0,
      outerAngle: 50.0,
      near: 0.05,
      far: 42.0
    }
  }
});
```

Spot shadow の `position` や `direction` はフレームごとに変更可能です。一人称視点などで光源を視点に追従させる場合は、カメラ姿勢から算出した値を `renderScene()` と `encode()` の両方に渡してください。

`shadow.ambient` は影側の最低輝度、`shadow.directIntensity` は光が当たる側の直接光の強さです。影側が黒く潰れる場合は `ambient` を上げ、照明側が白っぽくなりすぎる場合は `directIntensity` を下げます。この 2 つは役割が異なるため、影の濃さを変えたいのか、光が当たる面の強さを変えたいのかを分けて調整してください。

`gbufferColorMode` は、G-buffer color へ何を入れるかを指定します。既定値は `"litMaterial"` です。このモードでは材質色に直接光の diffuse / specular を含めた色を G-buffer color へ書き込むため、ComputeEffectPipeline を標準シーンへ追加したときにも、通常描画に近い直接光のハイライトが残ります。高度な Deferred Lighting や、材質 Albedo を分離したデバッグを行う場合だけ、`"albedo"` を指定して材質色のみの G-buffer へ切り替えます。

`composer.mode` は SSR 反射の合成方法を選びます。既定値は `"mix"` です。`"mix"` は SSR の alpha に格納された反射重みに基づいて元の色と反射色を混ぜるため、反射を強くしても色が白っぽくなりにくく、材質色を残したいシーンに向いています。`"add"` は反射を加算する従来型の合成で、強い光沢や発光感を出しやすい一方、反射率が高いと画面全体が明るく見えやすくなります。

toneMap は最終表示のための変換をまとめた設定です。mode: "reinhard" は強い値をなだらかに圧縮して白飛びを抑える方式で、SSR や Bloom を含む通常のシーンに向きます。mode: "linear" は色を大きく圧縮しないため、元の色の鮮やかさを優先したい検証や、明るさを別の設定で十分に抑えている場合に使います。

exposure はトーンマッピング前の明るさ倍率です。全体が暗い場合は上げ、照明側が白っぽい場合は下げます。saturation は色の鮮やかさを調整します。影や反射の合成で色が眠く見える場合は少し上げ、派手すぎる場合は下げます。gamma は表示用のガンマ変換で、通常は 2.2 を基準にします。blackBackground は深度を持たない背景だけを黒にする設定で、反射や影の調整時に背景の明るさを切り離して確認したい場合に使います。

標準シーンをそのまま入力にする設計

WebGPU コアが目指すのは、「標準の WebgApp、Space、Shape でシーンを作成し、そこに高水準クラスを追加するだけで表現力が向上する」構成です。

利用者が影や AO を導入するたびに、G-buffer のアタッチメントやバインドグループをゼロから組み立てる必要はありません。高水準 API が以下の役割を担い、複雑な管理を自動化します。

- **シームレスな入力**: 標準 Space をそのまま受け取り、エフェクト用に Shape を再登録させる手間を省く。
- **経路の自動選択**: 有効なエフェクトに基づき、「完全な G-buffer 経路」か「通常のフォワード経路」かを自動決定する。
- **リソース追従**: カメラ依存ターゲットを画面リサイズへ自動的に追従させる。
- **効率的な管理**: シャドウマップなどの固定解像度リソースを個別に管理し、不要な再生成を避ける。
- **リソース共有**: 法線と深度を複数のパスで共有し、同じシーンを不要に再描画しない。
- **色空間の統制**: リニアカラー → マスク → トーンマッピング → ガンマ変換の順序を一箇所で管理する。
- **統一的な操作**: 各エフェクトの有効・無効、パラメータ、デバッグビューを統一したインターフェースで操作できる。

これにより、開発者はシーン構築に集中し、SSAO や SSR を「標準機能の一つ」として選択的に追加できるようになります。

実行例としては、**samples/compute_effect/compute_effect.html** が、標準の Space と Shape で作成したシーンへ各種エフェクトを切り替えながら適用する構成を示しています。

色処理フローの整合性と順序

複数のエフェクトを重ねる際、最も注意すべきは「色処理のフロー」です。例えば、SSR のときだけトーンマッピングを行い、無効にしたときにリニアカラーのまま出力すると、エフェクトの ON/OFF だけで画面全体の明るさが変わってしまいます。

また、トゥーン表現のように明暗を段階化する処理を表示変換（トーンマッピング）の後に行うと、圧縮された輝度域が同じ帯に集まってしまい、段階数が潰れて見えます。トゥーンは必ず「リニアで意味を持っている照明結果」に対して適用しなければなりません。

そのため、ComputeEffectPipeline では色処理を以下の順序に固定しています。

【エフェクト処理フロー】

1. **G-buffer / Shadow Map 生成**
2. **ComputeShadowPass / ComputeSpotShadowPass** (ライティング適用)
3. **SsaoPass** (遮蔽の乗算)
4. **ComputeEffectComposer** (SSR 反射の合成)
5. **ComputeToonPass** (リニア状態での色段階化)
6. **ComputeDofPass / ComputeBloomPass** (リニア空間でのボケ・にじみ)
7. **ComputeEffectToneMapPass** (表示用のトーンマッピングとガンマ変換)
8. **ComputeEdgePass** (最終的な輪郭線の合成)
9. **canvas 出力**

この順序により、照明ベースの効果はすべてリニア空間で完結させ、最後に一度だけ表示変換を行い、その上に「画面上の線」であるエッジを加えるという整合性が保たれます。

ComputeEffectComposer と ComputeEffectToneMapPass を分けているのは、反射の合成と表示変換が別の問題だからです。SSR は、反射色と反射重みを作る処理です。Composer は、その反射を元の照明結果へどのように混ぜるかを決めます。ToneMapPass は、合成済みのリニアカラーをディスプレイへ出せる明るさと色に整えます。

既定の mix 合成は、SSR 出力の alpha に格納された反射重みを使い、元の色と反射色を置き換え寄りに混ぜます。そのため、強い反射でも照明結果の色を保ちやすく、標準のシーンへ

追加しやすい設定です。反射を足し込む add 合成は、光沢が分かりやすく出る一方で、床や透明感のある面が多いシーンでは全体が白っぽくなることがあります。強い発光感や誇張した反射が必要な場合だけ、composer.mode: "add" を明示してください。

その後の toneMap では、まず exposure でリニアカラー全体の明るさを決め、mode で高輝度の圧縮方法を選びます。reinhard は白飛びを抑える安全な選択で、linear は元の色を確認したいときに有効です。最後に saturation で色の鮮やかさを整え、gamma で表示用の明るさカーブへ変換します。効果の ON/OFF で画面全体の印象が変わる場合は、まずこの順序でどの段階が原因かを確認すると切り分けやすくなります。

低解像度化による最適化

Compute Shader を使うポストエフェクトでは、すべての処理を canvas と同じ解像度で行う必要はありません。SSR、SSAO、DoF のような効果は、最終的な画面に「補助成分」として合成される部分を持っています。この補助成分は、通常のシーンカラーや G-buffer よりも多少低い解像度でも破綻しにくいことがあります。webg ではこの性質を利用し、効果ごとに低解像度の中間 target を持たせて GPU 負荷を下げています。

重要なのは、低解像度化の対象を「シーン全体」ではなく「その効果が生成する中間結果」に限定することです。シーンカラー、depth、normal を一律に縮小すると、輪郭、接触、深度比較、エッジ抽出などの基礎情報まで粗くなります。一方、G-buffer や最終出力をフル解像度のまま維持し、反射、AO、ぼかし画像だけを低解像度化すると、見た目の情報量を保ちながら、負荷の大きい部分だけを減らせます。

現在の主な低解像度化は、以下のように効果ごとに意味が異なります。

- **DoF**: ComputeDofPass は stageSmallScale、stageMediumScale、stageLargeScale を使い、多段階ぼかし用の画像をそれぞれ別解像度で作ります。既定値は 0.7、0.5、0.3 です。ボケが大きい段階ほど高周波の細部は見えにくくなるため、より低い解像度を使いやすくなります。
- **SSR**: ComputeSsrPass の resolutionScale は、反射結果を書き込む target だけを低解像度化します。既定値は 0.7 です。G-buffer はフル解像度のまま読み、低反射 pixel では reflectivityThreshold によって ray marching 自体を省略します。
- **SSAO**: SsaoPass の resolutionScale は、raw AO target だけを低解像度化します。既定値は 0.7 です。最終的な合成はフル解像度で行い、depth と normal の重み付けによって境界をまたいだ AO のにじみを抑えます。

これらの倍率は、必ず同じ値に揃える必要はありません。同じ 0.7 という値が使われていても、それは「共有できる同一の縮小バッファ」を意味しません。DoF の中間画像は色をぼかした結果、SSR の中間画像は反射色と反射重み、SSAO の中間画像は遮蔽係数です。内容、形式、合成方法、必要な境界処理が異なるため、同じ縮小倍率にしても自動的にメモリや dispatch を共有できるわけではありません。

同じ倍率に揃える利点があるとすれば、設計上の分かりやすさと、リサイズ時の寸法管理が単純になることです。たとえば SSR と SSAO をどちらも 0.7 にすると、画質と負荷の比較はしやすくなります。しかし GPU 負荷の最適化としては、各効果の見え方に合わせて別々に調整するほうが自然です。DoF の large stage は 0.3 でも成立しやすい一方、SSAO を 0.3 まで下げると接触影や隅の暗がりが欠けやすくなります。SSR も、鏡面の細い反射や画面端の反射を重視する場面では 0.7 より高い値が必要になることがあります。

したがって、統合パイプラインでの基本方針は「まず既定値で成立させ、重い効果から個別に下げる」です。DoF はボケ段階ごと、SSR は反射 target、SSAO は raw AO target というように、何を低解像度化しているかを分けて考えます。低解像度化で画質が崩れる場合は、倍率を単純に戻すだけでなく、早期 return、サンプル数、blur radius、sample step、反射率閾値など、効果ごとの計算量パラメータも合わせて調整します。

G-buffer 利用の最適化（必要なときだけ使う）

高水準 API は、常に G-buffer 描画を行うわけではありません。以下のいずれかが有効な場合にのみ「G-buffer 経路」を選択します。

- シャドウマップ / SSAO / SSR / トゥーン / geometry edge

これらがすべて無効で、DoF、Bloom、color edge、または表示変換のみを使用する場合は、通常のフォワード描画結果を sceneTarget へ出力し、そこからコンピュートパスを開始する効率的な経路を辿ります。

【経路の分岐】

- **G-buffer が必要な場合**: Space → GeometryBufferPass → lighting / SSAO / SSR / Toon ... → tone map → Edge → canvas
- **G-buffer が不要な場合**: Space → sceneTarget（通常描画） → DoF / Bloom ... → tone map → color Edge → canvas

なお、ComputeToonPass は入力としてシーンカラーのみを必要としますが、G-buffer 経路に分類されています。これは、G-buffer のマテリアルカラーから一貫したリニアライティング結果を生成し、それをトゥーンへ渡すというパイプライン構造を維持するためです。

また、シャドウマップを無効にした場合でも、G-buffer 経路では「影評価 pass」自体は省略せず、単に「常に影がない（比較成功）」というパラメータで動作させます。これにより、シャドウの ON/OFF にかかわらず、ダイレクトライティングとアンビエントライティングの照明モデルそのものは一貫して維持されます。

動作確認済みサンプルの完全なフレーム接続

samples/compute_effect/main.js の実装に基づき、ComputeEffectPipeline を WebgApp へ接続し、エフェクトを実行時に切り替える実装例を示します。

1. パイプラインの生成

```
const app = new WebgApp({
  document,
  autoDrawScene: false,
  renderMode: "continuous",
  frameTiming: true,
  clearColor: [0.045, 0.065, 0.09, 1],
  viewAngle: 52,
  projectionFar: 120
});
await app.init();

const gpu = app.getGPU();

const pipeline = new ComputeEffectPipeline(gpu, {
  width: app.screen.getWidth(),
  height: app.screen.getHeight(),

  shadow: {
    ambient: 0.10
  },

  toon: {
    floor: 0.14
```

```
  }
});

const copyPass = new FullscreenPass(gpu, {
  targetFormat: gpu.format
});

await Promise.all([
  pipeline.ready,
  copyPass.init()
]);
```

※ autoDrawScene: false に設定し、pipeline.renderScene() による制御に切り替えます。

2. エフェクト状態の管理

```
const state = {
  ssaoEnabled: true,
  shadowEnabled: true,
  ssrEnabled: true,
  shadowAmbient: 0.10,
  directIntensity: 1.0,
  composerMode: "mix",
  toneMapMode: "reinhard",
  exposure: 1.0,
  saturation: 1.0,
  gamma: 2.2,
  blackBackground: false,

  toonEnabled: false,
  toonLevels: 4,

  dofEnabled: false,
  bloomEnabled: false,

  edgeEnabled: false,
  edgeThickness: 2,
  edgeBlendMode: "black-multiply"
```

```
};
```

3. リサイズへの追従

```
onUpdate: ({ screen }) => {
  pipeline.resize(
    screen.getWidth(),
    screen.getHeight()
  );
}
```

4. シーンの描画（renderScene）

```
onBeforeDraw: () => {
  pipeline.renderScene(
    app.space,
    app.projectionMatrix,
    app.eye,
    app.clearColor,
    {
      shadowEnabled: state.shadowEnabled,
      ssaoEnabled: state.ssaoEnabled,
      ssrEnabled: state.ssrEnabled,
      toonEnabled: state.toonEnabled,

      edgeEnabled: state.edgeEnabled,
      edgeGeometryEnabled: true,

      timestampWrites:
        app.getGpuRenderTimestampWrites(true, true)
    }
  );
}
```

5. エフェクトチェーンの実行（encode）

```
onAfterDraw3d: () => {
  const verticalFov =
    app.screen.getRecommendedFov(app.viewAngle) *
    Math.PI / 180;

  const projection = createGBufferProjectionParams(
    app.projectionNear,
    app.projectionFar,
    verticalFov,
    app.screen.getAspect()
  );

  app.eye.setWorldMatrix();
  const cameraWorld =
    app.eye.worldMatrix.clone();

  gpu.endPass();

  const finalColor = pipeline.encode(
    gpu.commandEncoder,
    {
      projection,
      cameraWorld,

      ssaoEnabled: state.ssaoEnabled,
      shadowEnabled: state.shadowEnabled,
      ssrEnabled: state.ssrEnabled,

      toonEnabled: state.toonEnabled,
      toon: {
        levels: state.toonLevels
      },

      shadow: {
        ambient: state.shadowAmbient,
        directIntensity: state.directIntensity
      },

      composer: {
        mode: state.composerMode
      },
```

```
      toneMap: {
        mode: state.toneMapMode,
        exposure: state.exposure,
        saturation: state.saturation,
        gamma: state.gamma,
        blackBackground: state.blackBackground
      },

      dofEnabled: state.dofEnabled,
      bloomEnabled: state.bloomEnabled,

      edgeEnabled: state.edgeEnabled,
      edgeGeometryEnabled: true,
      edge: {
        colorEnabled: false,
        blendMode: state.edgeBlendMode,
        thickness: state.edgeThickness
      },

      projectionNear: app.projectionNear,
      projectionFar: app.projectionFar,

      timestampWrites:
        app.getGpuTimestampWrites(true, true)
    }
  );

  app.screen.beginPass({
    clearColor: app.clearColor,
    colorLoadOp: "clear",
    depthView: null
      });

  copyPass.draw(finalColor);
  app.screen.clearDepthBuffer();
}
```

6. リソースの破棄

```
window.addEventListener("pagehide", () => {
  app.stop();
  copyPass.destroy?.();
  pipeline.destroy();
}, { once: true });
```

renderScene() と encode() の設定同期

高水準パイプラインは、リソースを「作成する段階（renderScene）」と「利用する段階（encode）」に分かれています。このため、両方の設定フラグを一致させる必要があります。

```
const frameOptions = {
  shadowEnabled: state.shadowEnabled,
  ssaoEnabled: state.ssaoEnabled,
  ssrEnabled: state.ssrEnabled,
  toonEnabled: state.toonEnabled,
  edgeEnabled: state.edgeEnabled,
  edgeGeometryEnabled: true
};

pipeline.renderScene(app.space, ..., frameOptions);
gpu.endPass();
const finalColor = pipeline.encode(gpu.commandEncoder, { ...frameOptions, ... });
```

もし renderScene でフォワード経路（G-buffer なし）を選択したのに、encode で geometry edge（G-buffer 必要）を有効にした場合、必要な法線や深度が存在しないため、G-buffer mode mismatch エラーを投げます。これは不整合を黙って隠すのではなく、設計上の誤りを早期に検出するための仕様です。

内部リソースの管理戦略

ComputeEffectPipeline は、コンストラクタ時に内部のパスとターゲットをすべて生成し、ready プロミスへまとめます。

エフェクトを無効にすることは、「**リソースを破棄すること**」ではなく、「**そのフレームの**

実行経路から除外すること」を意味します。これにより、実行中にエフェクトを ON/OFF しても、GPU リソースの再生成待ちが発生せず、瞬時に表示を切り替えることが可能です。

また、GPU タイムスタンプは、個別のパスではなくエフェクトチェーンの「最初」と「最後」にのみ割り当てます。これにより、チェーン全体としてのコンピュートコストを正確に計測できます。

29.5　ローレベル API による個別制御

高水準 API を提供しつつも、`GeometryBufferPass` などの個別パスを直接操作できる入口を残しています。これは、仕組みを理解し、最適化するための「学習と研究の窓口」としての役割を持たせるためです。

G-buffer という共通の入口

SSAO、SSR、シャドウマップ評価は、すべて「画面上のピクセルに対応する法線と深度」を必要とします。そのため、これらは `GeometryBufferPass` を共通の入口として利用します。

個別パスを組む場合は、`WebgApp` に `autoDrawScene: false` を指定し、`onBeforeDraw` で G-buffer 描画を行います。

```
onBeforeDraw: () => {
  gbuffer.renderSpace(
    app.space,
    app.projectionMatrix,
    app.eye,
    app.clearColor
  );
}
```

`colorMode` の選択基準

- **@<tt>{colorMode: "lit"}**: SSAO のみを追加する場合に適しています。ライティング済みのカラーに AO を乗算します。

- @<tt>{colorMode: "material"}: シャドウマップや SSR を併用する場合に必須です。照明前のアルベドや、反射率（アルファ値）を保持する必要があります。

個別パスの導入例

SSAO の追加

G-buffer のレンダーパスを閉じ、SsaoPass を記録し、最後に FullscreenPass で出力します。

```
gpu.endPass();
const output = ssaoPass.encode(gpu.commandEncoder, gbuffer.getBindingResources(), {
  projection,
  enabled: true,
  view: "composite"
});
// ... FullscreenPass.draw(output)
```

シャドウマップの追加

ShadowMapPass（光源視点深度）と ComputeShadowPass（カメラ視点評価）を組み合わせます。

```
onBeforeDraw: () => {
  shadowMap.renderSpace(app.space, light.viewProjection);
  gbuffer.renderSpace(app.space, app.projectionMatrix, app.eye, app.clearColor);
}
// ... shadowPass.encode()
```

bias や normalBias は、シーンのスケールや解像度に合わせて明示的に設定し、シャドウアクネを抑制します。

SSR の追加

ComputeSsrPass を使い、G-buffer から反射情報を抽出します。マテリアルのアルファ値

を「反射率」として利用するため、setMaterial 時のアルファ値で反射の強さを制御します。

複数パスの同時利用と合成フロー

SSAO、シャドウマップ、SSR を同時に使う場合、単純な直列接続は不可能です。

【不整合の例】 - SSR は元のマテリアルアルファを読みますが、シャドウパスの出力アルファは 1 であるため、反射率が失われます。- SSR の結果に後からシャドウを適用すると、反射像まで影の影響を受けてしまいます。

そのため、「**G-buffer からの分岐と合流**」という構造をとります。

【正しい処理フロー】 1. ShadowMapPass → ライト空間深度を作成。2. GeometryBufferPass → マテリアル / 法線 / カメラ深度を作成。3. ComputeSsrPass → マテリアルアルファを読み、反射色を RGB へ、反射重みを alpha へ持つ**独立した反射ターゲット**へ書き出す。4. ComputeShadowPass → アルベドへダイレクトライティングと影を適用。5. SsaoPass → 影付きカラーに AO を乗算。6. ComputeEffectComposer → 「AO 済みカラー」と「SSR 反射」を add または mix で合成。7. ComputeEffectToneMapPass → reinhard または linear で最終的な表示変換。8. FullscreenPass → canvas 出力。

この設計により、SSR 反射を計算する際に元のマテリアル情報を保持しつつ、ライティングと反射を適切な順序で合成できます。

29.6　高水準とローレベルの共存設計

二層構造のメリット

WebGPU コアは、高水準 API とローレベル API を同一のリソース仕様の上に構築しています。

- **高水準 API**: 複雑な依存関係（リソースの生成、処理順、色空間の管理）を隠蔽し、迅速な開発を可能にします。
- **ローレベル API**: 各段階の入出力を明示し、デバッグ表示や独自エフェクトの研究、詳細なパラメータ調整を可能にします。

この二層構造により、「簡単に使えるが、中身はブラックボックスではない」という状態を

実現しています。

実装上の原則

高水準層は、ローレベルパスを所有して組み合わせるだけの構成であり、独自のシェーダーやレンダラーを別に持つことはありません。これにより、修正内容の重複を避け、仕様の一貫性を保証しています。

また、高水準 API であっても、不整合を黙って隠す「過剰なフォールバック」は提供しません。- 必要な G-buffer 入力がない → **エラー** - 不適合なエフェクト順序 → **エラー**

不整合を明示的にエラーとすることで、開発者がリソースの依存関係を正しく理解し、意図した通りの描画パイプラインを構築することを促します。

29.7 設計検証のまとめ

本章の実装を通じて、以下の設計方針を確定させました。

1. **標準シーンとの完全統合**: `Space` と `Shape` をそのまま利用し、エフェクト追加のためにデータ構造を変える必要はない。
2. **リソース共有の最適化**: カメラ G-buffer を複数のエフェクトで共有し、描画回数を最小限に抑える。
3. **色空間の厳格な管理**: リニアカラー → 表示変換 → エッジ合成 の順序を固定し、視覚的な破綻を防ぐ。
4. **柔軟な経路選択**: G-buffer が必要なエフェクトがすべて無効な場合は、効率的なフォワード経路へ自動的に切り替える。
5. **責任境界の明確化**: シーン構築は利用者が、中間リソースの管理と実行順序の制御は高水準 API が担う。

29.8 まとめ

本章では、標準の `WebgApp`、`Space`、`Shape` で構築したシーンに、SSAO、シャドウマップ、SSR などの高度な表現を統合する方法を学びました。

通常の開発では `ComputeEffectPipeline` を使用することで、複雑な中間リソースの管理や色処理フローを意識することなく、高品質な視覚効果を追加できます。一方で、個別のパスを直接操作することで、エフェクトがどのような幾何学情報（法線・深度）から構築されているかを深く理解し、シーンに合わせた微調整を行うことができます。

PBR のような物理的な正確さだけでなく、こうした「空間的な関係」を補完するエフェクトを段階的に追加することで、アセットの複雑さを増やすことなく、画面の情報量と説得力を最大限に高めることが可能です。

付録 A

コーディング AI の皆さまへ

A.1　webg 利用者を支援するためのガイドライン

この付録は、webg を用いて 3D アプリケーションを開発する利用者を、コーディング AI が支援する場面を想定して作成されました。ここで想定する AI は、一般的な 3D エンジンや外部ライブラリの知識をそのまま適用するのではなく、webg という自己完結したライブラリの設計思想に基づき、何をどの順番で参照すべきかを正しく判断する支援者としての役割を担います。

本書において極めて重要な点は、webg が外部ライブラリに依存せず、描画からシーン管理、アニメーションに至るまで、必要な実装と解説をすべて内部に完結して持っていることです。AI が技術支援を行う際は、「一般的な WebGPU の実装はこうであるはずだ」あるいは「Three.js ではこうだから同様だろう」といった推測よりも、本書の記述および webg 自体のソースコードを優先して参照することが、正確な回答への最短ルートとなります。

一般論を補助的に活用することはあっても、最終的な判断基準は webg の本文、サンプル、unittest、およびコア実装に置くことが推奨されます。本付録の目的は、利用者が「何をしたいか」「どこで困難に直面しているか」に応じて、AI がどの章を起点にし、どのサンプルを参照し、どの実装まで掘り下げるべきかを素早く判断するための「読書地図」を提供することにあります。

A.2　自己完結した設計思想の理解

webg を支援する AI がまず理解すべきは、本ライブラリが単なる「外部ライブラリを繋ぎ

合わせた薄いラッパー」ではなく、描画、シーン、モデル、アニメーション、UI、入力、診断、ポストプロセスに至るまでを自前で実装した、独立した設計思想を持つライブラリであるということです。

一般的な Web 系の AI 支援では、「カメラ制御なら Three.js 的な手法を」「UI オーバーレイなら DOM 側で処理を」といった連想が起こりやすくなります。しかし webg では、独自の設計流儀が定義されています。

- cameraRig → cameraRod → eye という 3 段構成のカメラシステム
- ModelAsset → build() → instantiate() というリソース展開の分離
- clip → pattern → action → state というアニメーションの階層構造
- Scene JSON によるシーン定義の粒度
- SmoothShader を標準入口とするマテリアル管理

AI は、まず webg 独自の定義を正しく読み取り、その後に必要に応じて一般的な 3D コンセプトと対応付けるという手順で思考することが重要です。

A.3　遵守すべきテクニカル・ルール

AI がコードを生成する際、以下の項目を怠ると「文法的に正しくても何も表示されない」あるいは「実行時にクラッシュする」という事態を招きます。提案するコードに以下の要素が含まれているか、必ず確認してください。

1. 描画における必須ステップ

- 初期化の待機: await screen.ready（または await app.init()）の完了を待たずに、描画処理やリソース生成を行ってはいけません。GPU デバイスの準備が完了していることが前提となります。
- バッファの確定: Shape に頂点データを追加した後は、必ず shape.endShape() を呼び出してください。これを忘れると GPU バッファが確定せず、形状が描画されません。
- 描画ループの順序: clear → draw → present の実行順序を厳守してください。

2. WebgApp のライフサイクル

- プロパティへのアクセス: app.space、app.eye、app.getGPU() などのプロパティは、必ず await app.init() が完了した後にアクセスしてください。
- 更新ループの登録: 毎フレーム実行される処理は、app.start({ onUpdate: ... }) のハンドラ内に記述してください。

3. 座標系と回転用語の定義

- 右手座標系: +X=右, +Y=上, +Z=前 として定義されています。
- 回転用語: webg では yaw / pitch / roll という用語を使用します。一般論に基づく計算式を適用せず、CoordinateSystem 等の webg 内部定義に従って回転を制御してください。

A.4　目的別リソース・ナビゲーション

本書は webg の全体像を把握するための構造的なガイドラインとして機能します。AI は、ユーザーが現在どのレイヤーの話をしているかを判定し、以下の対応表に基づいて参照先を選択してください。

ユーザーの目的	起点とする章	参照すべきサンプル / キーワード
最初の 3D オブジェクトを表示したい	第 4, 5 章	low_level, high_level
WebgApp でアプリの土台を構築したい	第 5, 6 章	high_level
Orbit / Follow / First-person カメラの実装	第 5, 6 章	high_level, eye_rig
シェーダーやマテリアルを調整したい	第 7 章以降	shapes, smooth_shader
glTF / GLB / Collada モデルの読み込み	第 10, 12, 13 章	gltf_loader, collada_loader
Scene JSON でシーン全体を定義したい	第 5, 10, 11 章	scene
アニメーションの状態遷移を制御したい	第 12, 13 章	animation_state, janken
HUD やパネルなどの UI を実装したい	第 5, 14 章以降	各種 UI サンプル
低レイヤーでメッシュやシェーダーの構築	第 22 章以降	low_level, 各 unittest

A.5　API が見つからない場合の探索プロトコル

webg は API の数が多いため、AI が最初の検索で目的の関数に到達できないことがあります。その場合は、思いつきで外部ライブラリの API を代用せず、以下の順に探索してください。

1. 付録 B からクラス名・機能名を探す: book/付録B_API一覧.md は webg の API 索引です。まず WebgApp、Shape、Texture、ModelAsset、SceneLoader のようなクラス名、または raycast、normal map、dialogue、particle のような機能語で検索してください。
2. 章本文で設計意図を確認する: API 名だけでは使い方を判断できない場合は、上の「目的別リソース・ナビゲーション」の章へ戻り、概念、ライフサイクル、推奨パターンを確認してください。
3. サンプルで呼び出し方を確認する: 実際の使い方は samples/ 以下を優先して確認します。サンプルは main.js だけでなく、同じディレクトリ内の補助 *.js に実装が分かれている場合があります。
4. unittest で最小動作を確認する: ある API の最小条件や境界挙動を知りたい場合は unittest/ 以下を確認してください。
5. 最後に webg/*.js を仕様として読む: 付録 B やサンプルで分からない場合、webg/*.js の実装が最終的な仕様です。公開 API の有無、引数名、戻り値、例外条件をここで確認してください。

検索コマンドを使える AI は、次のように「文書 -> サンプル -> unittest -> 実装」の順で範囲を広げると見落としを減らせます。

```
rg -n "ClassName|methodName|feature keyword" book/付録 B_API 一覧.md book/*.md
rg -n "methodName|feature keyword" samples unittest webg
rg -n "^export |export default|methodName" webg/*.js
```

API 名が分からない場合は、まず book/付録B_API一覧.md の見出しだけを確認すると、どのクラスに属する機能かを素早く絞り込めます。

```
rg -n "^(##|###|####) " book/付録 B_API 一覧.md
```

よくある確認先は、次のように目的別に分けて考えると判断しやすくなります。

- **アプリ全体の構成、描画更新、入力、HUD、診断を調べる場合**: まず `WebgApp` と第 5 章を確認してください。カメラ操作は第 6 章、UI 表示の使い分けは第 14〜16 章、入力と衝突判定は第 16〜17 章、診断やデバッグ表示は第 18〜19 章が入口になります。実装上の最終確認は `webg/WebgApp.js`、`webg/InputController.js`、`webg/Diagnostics.js` を参照します。
- **カメラや視点操作を調べる場合**: 第 6 章を入口にし、標準のオービットカメラ、フォローカメラ、1 人称視点の違いを確認してください。`createOrbitEyeRig()` や `createFollowEyeRig()` のように `WebgApp` 側で毎フレーム更新されるものと、`new EyeRig(...)` で直接作ってアプリ側が更新するものでは責任範囲が異なります。実装は `webg/EyeRig.js` と `webg/WebgApp.js` を確認します。
- **形状生成、頂点、material、wireframe を調べる場合**: `Primitive` と `Shape` が主な入口です。基本的な形状生成は第 22〜24 章、material の値が描画や G-buffer へどう渡るかは第 24 章と第 27〜29 章を確認してください。wireframe は通常の面描画とは扱いが異なるため、Shadow Map、SSR、G-buffer の対象に含めるかどうかを個別に判断します。
- **texture、画像読み込み、normal map、procedural texture を調べる場合**: `Texture` と第 7 章を最初に確認します。normal map や procedural texture の実例は第 24 章と `samples/proctex` が参考になります。texture の生成元、GPU への転送、sampler 設定、shader 側の参照方法は分かれているため、サンプルのコードだけでなく `webg/Texture.js` も確認してください。
- **ModelAsset、json.gz、シーンデータの読み書きを調べる場合**: `ModelLoader`、`ModelAsset`、`SceneAsset` と第 10〜11 章を確認します。読み込みだけなら `ModelLoader`、webg 独自の JSON 形式や gzip 圧縮済み asset を扱う場合は `ModelAsset`、シーン全体の保存や復元は `SceneAsset` が関係します。アニメーションや skeleton を含む場合は第 12〜13 章も合わせて確認してください。
- **animation clip、action、transition を調べる場合**: `Animation`、`Action` と第 12〜13 章が入口です。clip の読み込み、再生時間、ループ、transition、bone や node への適用は別々の層に分かれています。実データを使う場合は `samples/gltf_loader`、`samples/collada_loader`、`samples/compute_json` の animation 対応箇所を確認すると、loader と再生処理の接続が分かります。

- **DOM overlay、help panel、debug dock、CommandPalette を調べる場合**: UI の役割分担は第 14 章、詳細な設計は第 15 章、タッチ入力との関係は第 16 章を確認します。説明やエラー表示は `OverlayPanel` / `OverlayPanelPresets`、一時的な設定変更は `CommandPalette`、診断表示は `DebugDock` が主な担当です。`CommandPalette` は title 行だけをドラッグハンドルにでき、低頻度の設定変更を常時表示ボタンから分離する用途に向いています。
- **raycast、picking、collision を調べる場合**: `Space` と第 17 章を入口にします。ポインタ位置から 3D 空間の対象を選ぶ場合は raycast、移動体と壁や床の接触を扱う場合は collision の処理フローを分けて考えてください。境界条件や最小例は `unittest/raycast` を確認します。
- **反発、摩擦、物理 body、回転付き box 接触を調べる場合**: `PhysicsNode`、`PhysicsSpace` と第 26 章を確認します。実例としては `samples/physics_bounce` が入口になり、契約的な挙動確認には `unittest/physics_space_contracts` を使います。物理挙動は見た目だけでは原因を判断しにくいため、速度、接触法線、反発係数、摩擦係数をログや unittest で確認してください。
- **Compute Shader を使うポストプロセスや空間表現を調べる場合**: 第 27〜29 章、`ComputeEffectPipeline`、`ComputeEffectComposer`、`GeometryBufferPass`、`ComputeSsrPass`、`ShadowMapPass` を確認します。個別の効果だけを見る場合は `samples/compute_effect` が入口です。G-buffer の色が material color なのか lighting 済み色なのか、SSR の合成が `mix` なのか `add` なのかで見え方が大きく変わるため、`gbufferColorMode`、`composer.mode`、tone map、gamma、exposure、saturation の設定を合わせて確認してください。

ファイル名とクラス名は多くの場合 `webg/ClassName.js` に対応しますが、例外もあります。`formatJSON()` は `webg/JsonFormat.js`、UI theme は `webg/WebgUiTheme.js`、help / error の option builder は `webg/OverlayPanelPresets.js`、`SkinningConfig` は定数と関数の module です。迷ったら `rg -n "export default class ClassName|export class ClassName|export function functionName" webg` で確認してください。

A.6 UI コンポーネントの選択指針

利用者が「画面に情報を表示したい」と要望した場合、AI は目的に応じて以下のコンポーネントを適切に使い分ける提案を行ってください。

- 操作説明やヘルプ: `app.showOverlayPanel(buildHelpPanelOptions(...))` または

app.showOverlayPanel({ collapsible: true, ... })
- 動的な数値や状態の表示: app.message.setLines("status", [...], options) または HUD (setHudRows)（簡潔な情報を整然と表示）
- 会話形式の演出やチュートリアル: OverlayPanel の buttons / choices と、アプリ側 controller を組み合わせる
- 詳細な情報やエラー理由の提示: app.showOverlayPanel({ format: "pre", scrollY: true, ... }) または buildErrorPanelOptions()

A.7 リソース参照の優先順位

AI は以下の優先順位で参照先を選択し、根拠に基づいた提案を行ってください。

1. book/付録B_API一覧.md（API 名の索引）: API 名、クラス名、代表的メソッドを確認する入口です。API の数が多い場合は、まず付録 B で所属クラスを絞り込んでから本文や実装へ進んでください。
2. samples/（実装意図の把握）: まず対応する *.txt（解説）を読み、そのサンプルの「目的」を理解した上で main.js（実装）を参照してください。コードの断片的な模倣ではなく、設計意図を汲み取った提案を行うためです。
3. unittest/（局所的な検証）: 特定機能の挙動確認や、最小単位での動作検証が必要な場合に参照してください。
4. webg/ 本体実装（最終的な仕様定義）: 本文やサンプルで意味が曖昧な場合、最終的な仕様を確認するための正解として参照してください。

A.8 API レイヤーの分離と整合性

AI が最も避けるべきは、高レイヤー（ハイレベル）API と低レイヤー（ローレベル）API を不用意に混在させることです。

- 原則: まず高レイヤー（ハイレベル）API (WebgApp 等) で解決できないかを検討してください。
- 文脈の尊重: 利用者が WebgApp を使用している場合は、その構造を維持したまま追加・修正する提案を優先します。逆に、低レイヤー（ローレベル）で実験的に実装している利用者の場合は、その文脈を尊重し、最小限の差分で提案してください。

`ModelAsset` と `SceneAsset` の区別

- `ModelAsset`: 単一モデルの共通表現（メッシュ、スケルトン、アニメーション）。`runtime.instantiate()` を通じて個別のインスタンスを生成します。
- `SceneAsset`: シーン全体の初期状態（カメラ、HUD、配置済みプリミティブ、配置済みモデル）。
- 判断基準: 「モデルを複数配置したい」のか「シーン全体の初期状態を保存したい」のかを切り分け、参照先（第 10 章か第 11 章か）を正しく選択してください。

アニメーションとシェーダーの切り分け

- アニメーション: 問題が「clip（データ）」にあるのか、「Action（区間/ポーズ）」にあるのか、「AnimationState（状態遷移）」にあるのかを最初に切り分けてください。
- シェーダー: 通常は `SmoothShader` を標準の入口として提案してください。WGSL や bind group の低レイヤー（ローレベル）な修正は、高レイヤー（ハイレベル）なマテリアルパラメータで解決できない場合にのみ提案してください。

A.9 AI が維持すべき基本姿勢

1. 最も抽象度の高い API を優先的に検討する: `WebgApp` や `loadModel()` など、最も抽象化された API で解決できないかをまず検討してください。
2. 本書の定義を信頼する: 用語、構造、サンプルの位置づけは一貫しています。外部エンジンの流儀を無理に当てはめず、本書の定義をそのまま適用してください。
3. 問題の層を切り分ける: 現象を一つの原因にまとめず、「どの層（描画、カメラ、モデル、アニメーション、UI）の問題か」を切り分けてから解決策を提示してください。
4. コアライブラリの直接的な修正を回避する: アプリケーションの実装において `webg` コア機能 (`webg/*.js`) の修正が必要と感じた場合でも、クラスを継承するなどしてアプリケーション側で対応することを提案してください。もしコア機能のバグであると判断できる場合は、利用者にその旨を伝えてください。

`webg` は、設計・実装・文書化が一貫して行われているライブラリです。AI は本書を「一次的な参照地図」として活用し、利用者が迷わず実装できるようガイドしてください。

あとがき

ここまでお読みいただき、ありがとうございました。

本書は、JavaScript と WebGPU を用いて 3D アプリケーションを構築するためのライブラリ `webg` を題材に、単なる使い方だけでなく、その設計思想や内部構造までを一冊でたどれるようにすることを目指して執筆しました。最初の 3D グラフィックスの基礎から始まり、`WebgApp` による標準的なアプリ構成、モデルやシーンの読み込み、アニメーション、UI、レイキャストと衝突判定、物理エンジン、そしてローレベル（低レイヤー）API やスキニングの仕組みに至るまで、非常に広範な領域を扱っています。そのため、読者の皆様によっては、一部の内容が「専門的な領域に深く踏み込みすぎている」と感じられたかもしれません。

しかし、あえて表面的な操作方法だけで終わらせない構成にしたのは、3D アプリケーション開発における「見通しの重要性」を重視したためです。正常に動作している間は仕組みを理解しているように見えても、表示が崩れたり、入力が意図通りに動作しなかったりした途端、急に見通しが悪くなるものです。そのような局面で真に役立つのは、「どの API を呼べばよいか」という断片的な知識ではなく、「いま自分はどの層の問題に向き合っているのか」を判断できる体系的な理解、すなわち「見取り図」です。本書が一貫して目指したのは、その見取り図を読者の皆様に提示することでした。

また、本書を単なる API リファレンスに留めるのではなく、「サンプルコード」「本文」「コア実装」の三者が互いに有機的に結びついた資料でありたいと考えました。サンプルは学習のための教材であり、本文は読書順のガイドであり、実装は必要に応じて参照できる一次資料です。これら三者を往復しながら理解を深めていくことこそが、`webg` のようなライブラリを習得する上での近道になると信じています。

こうした構成は、現代における「コーディング AI」との共存という視点からも重要になっています。近年、人間だけでなく AI と共にプログラムを構築する機会が飛躍的に増え、技術文書に求められる役割も変化してきました。断片的な Tips を並べるのではなく、用語が一貫しており、層の境界が明確で、どの章がどの実装やサンプルに対応しているかが明快であること。それらを一次情報として提供することが、人間にとっても AI にとっても不可欠になって

います。本書がこのような形式となっているのは、人間にとっての理解しやすさと、AI が誤解なく解釈できる形式であることを同時に追求した結果です。

もちろん、本書に記した内容がすべて最終形であるわけではありません。webg 自体にも今後追加できる機能や、より洗練された形に整理できる部分があります。また、AI による支援が進歩したことで、ライブラリへの機能追加は AI に指示するだけで完結してしまうかもしれません。しかし、AI に任せきりにすることで実装速度は上がりますが、設計思想の見通しが効かなくなれば、いずれメンテナンスの限界が訪れます。AI が扱えるコンテキスト長には制限があり、システムが巨大化すれば、コードをすべて読み切って完全に理解することには限界があるからです。

そのため、本書は単なる解説書にとどまらず、AI に正しく実装させるための「構造化されたコンテキスト（実践的なプロンプト）」としての役割を持たせています。設計思想という「軸」を人間が把握し、それを AI に正しく伝えることができれば、AI は最強の武器になります。

この本を手に取ってくださった方の中には、最初から最後まで通読された方もいれば、必要な章だけを抽出して読まれた方もいるはずです。どのような読み方であっても構いません。本書が、webg を使って何かを創造しようとする際の入り口となり、行き詰まったときの参照先となり、そして内部構造を確かめたくなったときの足場となっていれば、著者としてこれ以上の喜びはありません。

最後に、本書の内容は、数多くのサンプルや実装を積み上げ、試行錯誤しながら整えてきたものです。もし本書を通じて webg に興味を持ち、実際に何かを作ってみようと思っていただけたなら、それこそが最大の成果です。立方体を一つ回転させるところからでも、glb を読み込んでキャラクターを動かすところからでも構いません。小さな一歩でも、実際に手を動かしてみることで、本書の内容はきっと異なる景色として見えてくるはずです。

本書が、その最初の一歩と、その先に待つ試行錯誤の両方において、少しでもお役に立てることを願っています。

ありがとうございました。

webg ではじめる WebGPU 3D アプリ開発

2026 年 7 月 3 日　第 2 版 試刷

著　者　水谷 純
連絡先　mizutani.jun@nifty.ne.jp