THE BIG PICTURE

Z-bufferからThree.jsへ

50年のレンダリング技術史を、**「見える面の決定」→「GPUへの委譲」→「シーン管理の抽象化」**という3つの問いで読み解く。

技術の積み重なり

1974	**Z-buffer**	画素ごとの前後関係を解決
1992	**OpenGL**	GPU描画を共通APIで扱う
2004	**GLSL / Shader**	描画計算をプログラム可能に
2010–11	**Three.js + WebGL**	3Dシーンをブラウザへ
2020s	**WebGPU + TSL**	描画とGPU並列計算を接続

問い 1

どの面が見える？

隠面処理とZ-buffer。現在でも不透明物の表示を支える。

問い 2

何をGPUで計算？

頂点変換・ラスタライズ・シェーダー・深度比較・GPU計算。

問い 3

誰が描画を管理？

Three.jsがScene・Camera・MaterialをGPU命令へ橋渡しする。

結論：Three.jsはZ-bufferを置き換えたのではない。**Z-bufferを含むGPU描画パイプラインの上に、シーン管理・材質・照明・影・レンダリング制御を積み上げたライブラリ**である。

対象：CGの基礎を知りたい学習者・WebGL/WebGPU開発者。年度は主な公開・成立時期の目安。資料参照 [S1–S17]。

01 HISTORY

描画技術の年表

レンダリング史は単に画質が上がった歴史ではない。**可視判定・ライティング・計算主体・API設計**が別々に発展し、現在のGPUパイプラインに合流した。

年代	転換点	何が解決されたか
1960s–70s	隠面処理の研究	Painter's algorithm、スキャンライン法、面の分割などを検討。「奥の面を描かない」が中心課題。
1974	**Catmull の Z-buffer**	画素ごとに最も手前の深度を保存。面同士の複雑な順序づけを大幅に単純化。[S1,S2]
1970s–80s	Phong照明、テクスチャ、レイトレーシング	可視判定とは別に「表面が何色・どの明るさに見えるか」を高度化。
1980s–90s	描画ハードウェアの普及	三角形処理・Z-buffer・テクスチャ等を専用回路で高速に実行。
1992	**OpenGL 1.0**	ハードウェアに依存しない共通描画API。固定機能中心の設計。[S3]
2000s	GPUプログラマブル化	頂点・ピクセルごとの演算をシェーダーで記述。GLSLがOpenGL 2.0の標準機能に（2004）。[S4]
2010–11	**Three.js と WebGL 1.0**	JavaScriptからGPU描画へ。Three.jsはScene/Camera/Materialを提供。[S5,S7,S17]
2017	WebGL 2.0	OpenGL ES 3.0系の機能をWebで利用可能に。[S6]
2020s	WebGPU / WebGPURenderer / TSL	描画とCompute、低レベルGPU資源管理、新しいシェーダー記述を導入。[S11,S12]

> **技術史の読み方：**Z-buffer、シェーダー、API、Three.jsは代替関係ではなく**異なる層の技術**。GPU内部で同時に使われている。

02 VISIBILITY

Z-buffer：1画素の前後関係を決める

Z-buffer（Depth Buffer）は、画素（厳密には描画サンプル）ごとに**現在最も手前にある面の深度値**を記録する。次の面が手前なら、色と深度を更新する。



模式図：通常の深度設定では 0 が近く、1 が遠い。赤い領域のみ、0.7 より手前の 0.3 が採用される。

通常の深度テストの考え方

```
if (incomingDepth < storedDepth) { color = newColor; depth = incomingDepth; }
```

実際の比較関数はAPI設定による。Three.jsの標準Materialでは

`LessEqualDepth`

が初期値。[S9]

従来のPainter's algorithm

奥から手前の順に面を塗る。交差する面・循環する遮蔽関係では単純なソートでは解決できず、分割などが必要。

Z-buffer方式

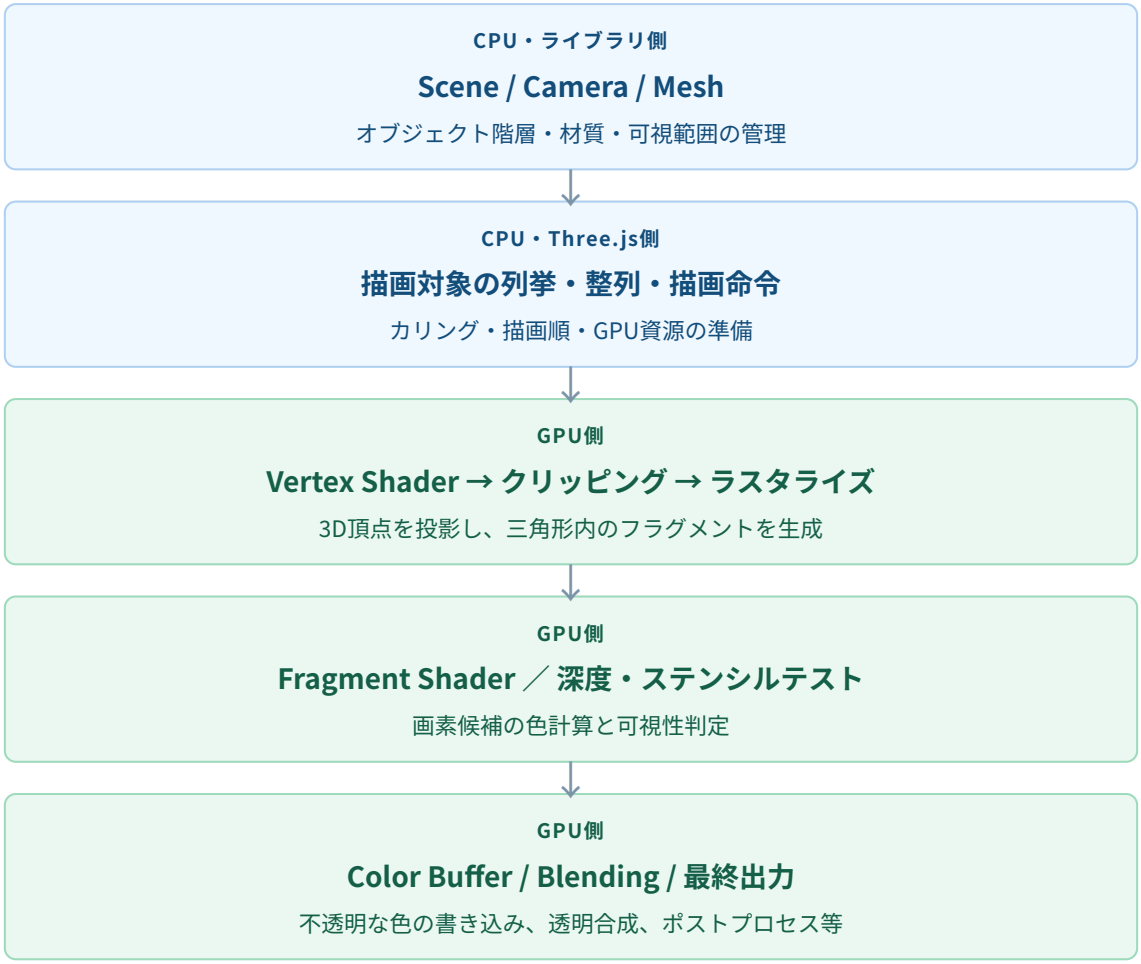
各画素で前後を比較するため、不透明な交差ポリゴンでも基本的に描画順序に依存しない。メモリ容量と深度精度が代償。

重要：Z-bufferは「影を描く装置」でも「透明度を合成する装置」でもない。基本的には**可視性の判定**である。影と透明は別の処理が必要。原点：[S1,S2]。

03 PIPELINE

GPUは三角形をどう画面にするか

現在の一般的なラスタライズ描画は、**3Dの形状を画素候補（フラグメント）へ変換し、深度と色を確定する**処理の連続である。



Vertex Shader

モデル座標 → ワールド座標 → カメラ座標 → クリップ座標。法線やUVなども次段へ渡す。

Fragment Shader

ライティングやテクスチャなどから色を計算。GPUは条件が合えば深度テストを先行実行（Early-Z）できる。

注意：上図は概念上の処理順。GPU実装ではEarly-Z等の最適化により、深度テストとフラグメント処理の実際の順序は変わる。

04 APIs

OpenGL → WebGL → WebGPUで変わったこと

「描画できる技術」から「描画命令を組み立てる技術」へ。低レベルAPIの進化は、GPUプログラミングの自由度と責任範囲を広げた。

項目	固定機能中心のOpenGL	WebGL 2	WebGPU
登場の文脈	ネイティブ3D共通API	ブラウザでOpenGL ES 3.0相当	現代GPU向けWeb API
描画計算	固定機能が中心（初期）	GLSL ESシェーダー	WGSLシェーダー
Z-buffer	標準的な深度テスト	標準的な深度テスト	標準的な深度テスト
GPU並列計算	新しいOpenGLにはComputeあり	汎用Computeなし	Compute Pipelineあり
制御単位	状態変更・Draw呼び出し	状態変更・Draw呼び出し	Pipeline・Buffer・Bind Group・Pass等
設計上の特徴	長い互換性の歴史	広いブラウザ互換性	明示的なGPU資源・処理の組立て

> **重要な区別：**OpenGLは歴史的に固定機能からプログラマブルへ進化した。**「OpenGL＝固定機能」ではない**。OpenGL 2.0のGLSL、OpenGL 4.3以降のComputeを含む。[S3,S4]

ブラウザでの技術階層

アプリのシーン・モデル・カメラ・操作

Three.js（高水準ライブラリ）

WebGLRenderer → WebGL 2 ｜ WebGPURenderer → WebGPU（または WebGL 2）

ブラウザ実装 → GPUドライバ / ネイティブAPI → GPU

WebGLはOpenGLそのものではなく、OpenGL ES系の描画モデルをWeb用に規格化したもの。[S5,S6]　WebGPUはW3Cで標準化が進行中。[S12]

05 THREE.JS

Three.jsが引き受けた仕事

WebGLだけでは「三角形を描く」低レベル命令が中心。Three.jsはその手前に**3Dアプリケーションを記述する語彙**を提供した。[S7]

ユーザーが書く

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(
  60, width / height, 0.1, 100
);
const cube = new THREE.Mesh(
  new THREE.BoxGeometry(1, 1, 1),
  new THREE.MeshStandardMaterial({color:
0x55aaff})
);
scene.add(cube);
renderer.render(scene, camera);
```

ライブラリが引き受ける

- Scene Graph（親子座標変換）
- カメラ行列・投影変換
- BufferGeometryとGPU頂点データの管理
- Materialからシェーダーを組み立てる
- フラスタムカリング・描画順制御
- GPU状態の設定・ドローコール発行
- ライト、シャドウ、環境マップ
- リソースのキャッシュや再利用

レンダリング時の概念的な流れ

① Scene Graph	② 可視判定	③ Render List	④ GPUに送信	⑤ 画面へ
メッシュやカメラを列挙	フラスタムカリング等	不透明／透明を分類・整列	Buffer / Shader / Draw	Depth Test / Blending

ポイント：renderer.render(scene, camera) は「画像を直接計算する単一命令」ではない。**多様なGPU処理を準備・発行する入口**である。[S10,S15,S16]

06 THREE.JS DEPTH

Three.jsに残るZ-bufferの課題

不透明物：Z-bufferが得意

手前の面だけを残す。通常はポリゴンを厳密に奥から描かなくてもよい。

透明物：順序がまだ必要

色を合成するためには背後の色が必要。オブジェクト単位の整列だけでは交差する面等が破綻しうる。[S8]

Three.jsの深度関連設定

```
const material = new THREE.MeshStandardMaterial({ color: 0x6699cc });
material.depthTest  = true;  // 保存済み深度との比較
material.depthWrite = true;  // 通過した深度を書き込む

const glass = new THREE.MeshStandardMaterial({
  color: 0x77ccff, transparent: true, opacity: 0.4,
  depthWrite: false          // 一般的な透明表現での選択肢
});
```

Z-fighting

ほぼ同じ深度の2面で、深度精度の限界によりチラつく。カメラのnearを極端に小さくしない・共面を避ける・必要ならpolygonOffset等を使う。

遠距離シーン

透視投影の深度分布は通常非線形。reversedDepthBuffer（対応環境）やlogarithmicDepthBufferが選択肢になる。[S10,S13]

落とし穴：renderOrder は描画順の制御。**深度テスト自体を無効にする機能ではない**。また、透明度0.5の2枚を重ねる処理は単なる深度比較だけでは正しく表現できない。[S8,S9]

07 LIGHT & MATERIAL

深度の次に発展した「見た目」の技術

Z-bufferは**どれが見えるか**を扱う。照明・影・質感・ポスト処理は**どう見えるか**を扱う。Three.jsは後者もライブラリとして統合してきた。

照明モデル 1

Lambert / Phong など、表面法線と光源方向に基づく明暗計算。

テクスチャ・法線マップ 2

形状を増やさずに色や凹凸感を表現。

Shadow Mapping 3

光源から見た深度を別に描画し、遮蔽される位置を判定。Z-bufferの応用。

PBR / Material 4

Metalness・Roughness・環境マップ等を使い物質ごとの反射を近似。[S14]

ポストプロセス 5

描画結果をテクスチャとして再処理。Bloom、被写界深度、色調整など。

色空間・トーンマッピング 6

線形光での計算と表示向け変換を区別し、HDR等の表現を扱う。

混同しやすい点：Shadow Mapは「光源視点の深度画像」。カメラ視点で可視面を決めるZ-bufferと関連するが、**役割の異なる追加レンダーパス**である。

08 MODERN RENDERER

WebGPU時代のThree.js

Three.jsはWebGL専用ライブラリではない。現在のWebGPURendererはWebGPUを優先し、WebGL 2バックエンドへフォールバックできる。[S11]

Scene / Camera / Mesh / Material

TSL（Three.js Shading Language）＋ Node Material

WebGPURenderer（共通レンダラー層）

WebGPU Backend
WGSL / Compute / Render Pass

WebGL 2 Backend
GLSL / Raster Rendering

WebGPUで増えた可能性

- Compute Shaderによる並列計算
- GPUバッファを中心とした処理
- 描画とシミュレーションの連携
- パスの制御・MRT・ポスト処理

引き継がれる基礎

- 三角形、ラスタライズ
- Vertex / Fragment Shader
- Z-buffer・深度テスト
- 透明合成の順序問題
- 材質とカメラの抽象化

注意すべき移行ポイント

ShaderMaterial、RawShaderMaterial、onBeforeCompile()を使った既存WebGLコードはWebGPURendererにそのまま移せない場合がある。公式はNode Material / TSLへの移行を案内する。現行レンダラーはなお実験的と記載されている。[S11]

> **WebGPUの本質：**WebGPUによってZ-bufferを使わなくなるのではない。**GPUで何をどう計算するかの選択肢**が広がる。

09 TAKEAWAYS

結局、Three.jsとは何なのか

THREE.JS IN ONE SENTENCE

Three.jsは、**50年かけて育った描画アルゴリズムとGPUパイプライン**を、ブラウザの**Scene・Camera・Mesh・Material**という操作モデルに変換して使いやすくした3Dライブラリである。

5つの理解チェック

1. **Z-bufferは現在も使う？** → はい。不透明オブジェクトの可視判定の基本。
2. **Three.js自身がすべての画素をCPUで計算する？** → いいえ。多くの描画計算はGPUへ依頼。
3. **Three.jsなら透明物の順序問題は完全に解決？** → いいえ。オブジェクト整列では解けない例がある。
4. **WebGPUはWebGLよりいつでも速い？** → いいえ。負荷や実装による。Compute連携は大きな強み。
5. **レイマーチングもThree.jsが必須？** → いいえ。直接WebGPUでも書ける。シーン管理が必要なら併用。

用途別の位置づけ

主目的	適した入口	理由
3Dモデルの表示、教材、Web 3D	Three.js	シーン管理が豊富、少ないコードで描画できる
独自のラスタライズ表現	Three.js + TSL / シェーダー	既存の描画機構と独自表現を組み合わせる
FFT、流体、粒子、独自レイマーチング	WebGPU直接利用も有力	Compute・バッファの制御自由度が高い

学習順序：Z-buffer → 座標変換 → ラスタライズ → Shader → WebGL → Three.jsの内部構造 → WebGPU。
個別技術を覚えるより、**「画素を誰が、どのデータで、どの順に決めるか」**を追うと理解しやすい。

APPENDIX

参考文献・一次資料

論文・標準仕様・Three.js公式マニュアルを優先。リンクは閲覧用で、引用番号は本文に対応。

No.	資料	URL
S1	Edwin Catmull, A Subdivision Algorithm for Computer Display of Curved Surfaces (1974), Utah	https://www-old.cs.utah.edu/docs/techreports/Catmull.74.pdf
S2	Sutherland et al., A Characterization of Ten Hidden-Surface Algorithms (1974)	https://doi.org/10.1145/356625.356626
S3	Kurt Akeley, The OpenGL Graphics System: A Specification (1992)	https://www.microsoft.com/en-us/research/publication/the-opengl-graphics-system-a-specification/
S4	Khronos, OpenGL 2.0 Specification (2004)	https://registry.khronos.org/OpenGL/specs/gl/glspec20.pdf
S5	Khronos, WebGL 1.0 Release (2011)	https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification
S6	Khronos, WebGL 2.0 Specification (2017)	https://registry.khronos.org/webgl/specs/2.0/
S7	Three.js Manual, Fundamentals	https://threejs.org/manual/pages/fundamentals.html
S8	Three.js Manual, Transparency	https://threejs.org/manual/pages/transparency.html
S9	Three.js Docs, Material / depthTest / depthWrite	https://threejs.org/docs/pages/Material.html
S10	Three.js Docs, WebGLRenderer	https://threejs.org/docs/pages/WebGLRenderer.html
S11	Three.js Manual, WebGPURenderer & TSL	https://threejs.org/manual/pages/webgpurenderer
S12	W3C, WebGPU Candidate Recommendation Draft (2026)	https://www.w3.org/TR/webgpu/
S13	NVIDIA, Visualizing Depth Precision (2021)	https://developer.nvidia.com/blog/visualizing-depth-precision/
S14	Three.js Docs, MeshStandardMaterial (PBR)	https://threejs.org/docs/pages/MeshStandardMaterial.html
S15	Three.js Manual, Creating a Scene	https://threejs.org/manual/pages/creating-a-scene.html
S16	Three.js Docs, StandardDrawingPipeline	https://threejs.org/docs/pages/StandardDrawingPipeline.html
S17	Three.js History, first release 2010 (course overview)	https://2019-spring-web-dev.readthedocs.io/en/latest/final/roen/index.html

作成：2026年11月4日。CG史の節目は成立時点と標準公開年が異なることがあるため、表では主な年を採用。図は本資料のために独自作成。