

ゲームグラフィックス特論

第1回 レンダリングパイプライン

はじめに

この講義について

概要

- 何を学ぶのか
 - 3DCG の基礎理論
 - リアルタイムレンダリングで用いられる各種の手法
 - OpenGL および GLSL (OpenGL Shading Language) による実装
- 何の役に立つのか
 - 3D ゲームの開発の基礎知識を身につける
 - 一般的なコンピュータのプログラミングにも使える
 - 3DCG の理論や、そこで用いられている数学や物理の基礎知識、および各種のアルゴリズムやプログラミングテクニック、数値計算法など
 - コンピュータや GPU のプログラミングが少し上手くなるように



はず

単位認定

- 課題と期末試験で評価する
 - 課題は毎回の講義内容をもとにプログラミングを行うものを課す
 - 個々の課題の評価
 - 未提出: 0点
 - 期限遅れ: 2点
 - 期限内に提出: 3点
 - 期限内に課題を達成: 4点
 - 高評価 (拡張課題の達成等) : 5点
 - 課題の評価の合計×40% + 期末試験の点数×60%

プログラミングスキルを向上するために

- ・与えられた問題に対して
 - ・解決方法を考える
 - ・実際にコードを書く
 - ・これを繰り返す
- ・課題
 - ・自分でプログラムを書いたかどうかに注目して採点する
- ・期末試験
 - ・自分でプログラムが書けないと解答できない問題を用意する

つもり

つもり

講義内容 (あくまで予定)

- ・レンダリングパイプライン
- ・GPU (Graphics Processing Unit)
- ・座標変換
- ・見かけ
- ・テクスチャ
- ・高度な陰影付け
- ・面光源および環境光源
- ・大域照明
- ・...

本日の内容

- ・ゲームグラフィックス
 - ・ゲームに使うコンピュータグラフィックス (CG) 技術
- ・グラフィックスライブラリ
 - ・コンピュータの図形表示機能を利用するソフトウェアの層
- ・レンダリングパイプライン
 - ・図形表示を行うための手順と機構
- ・GPU (Graphics Processing Unit)
 - ・図形表示を行うハードウェア

ゲームグラフィックス

インタラクティブな CG

ゲームグラフィックスとは

- ・ 内容はコンピュータグラフィックス (CG)
 - ・ リアルタイム 3DCG が中心
- ・ 時間に制約がある
 - ・ 画像品質より処理時間を優先する
 - ・ 制限時間内で最大の品質を目指す
- ・ ハードウェアの支援を用いる
 - ・ ハードウェア (GPU) の機能を活用する
 - ・ 必要なハードウェアの機能を要求 (開発) する

GPU (Graphics Processing Unit)

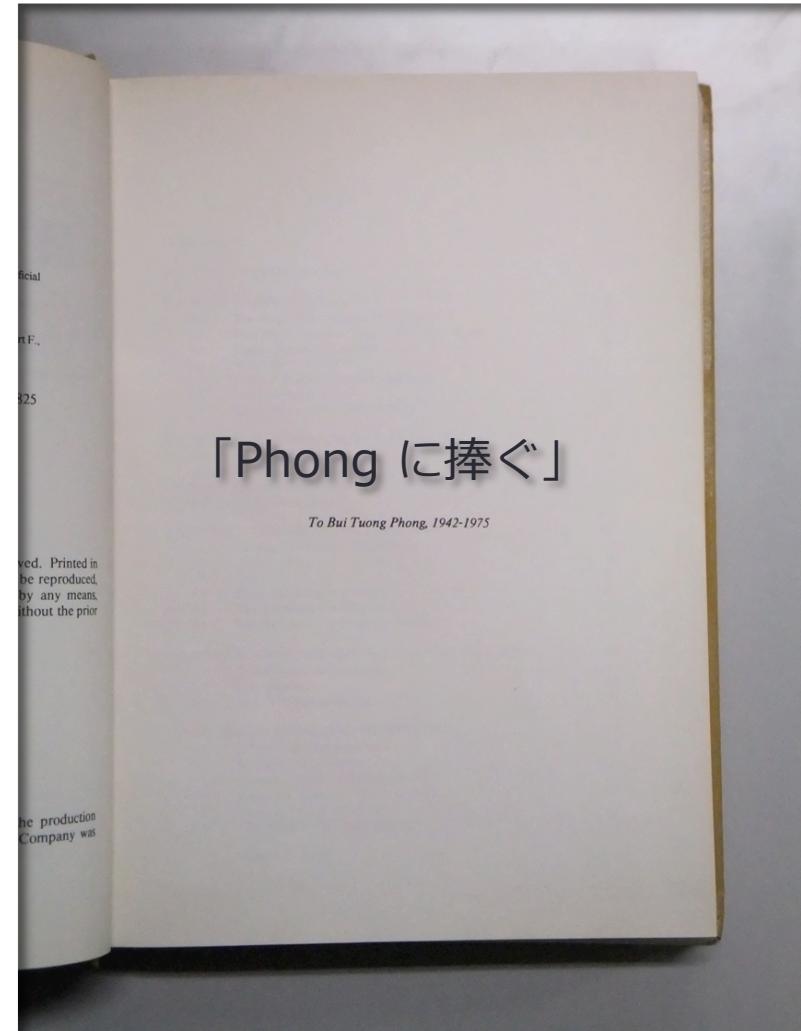
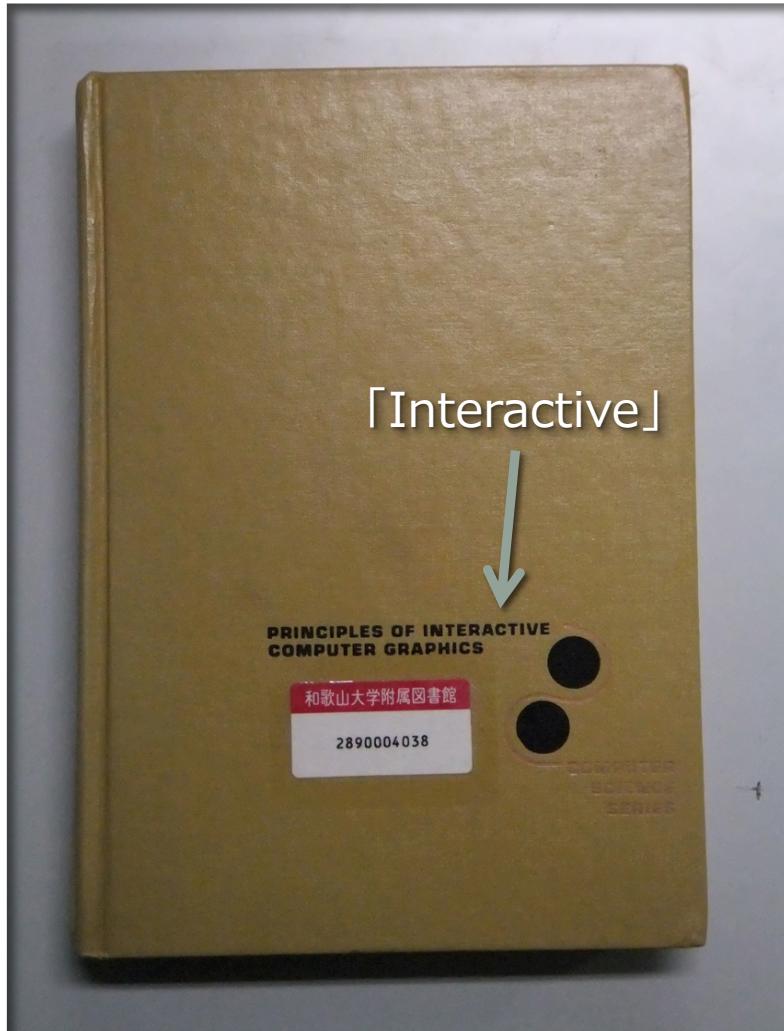
- 以前はグラフィックスアクセラレータなどと呼ばれた
 - 整数演算だけで可能な処理にしか対応していなかった
 - 座標変換や陰影付けなどの実数計算は CPU で行なっていた
- グラフィックスアクセラレータに**実数演算機能**を付けた
 - それまで CPU で行なっていた処理をグラフィックスアクセラレータ側で行なうことが可能になった
 - それを NVIDIA が **GPU** (Graphics Processing Unit) と名付けた
- GPU に求められる機能がどんどん多様化していった
 - ハードウェアの機能追加では対応できなくなつた
 - そこで GPU を**プログラマブル**にして対応しようとした
 - 画像処理や音声処理、人工知能などの**一般的な計算**に利用可能

なぜゲームグラフィックスなのか

- 3DCG の応用分野はそれほど広くない
 - リアルさのために時間を潤沢に消費できるような応用は限られる
 - ムービー制作, サイエンティフィックビジュアリゼーション, . . .
- ユーザインターフェースの技術としての需要はある
 - ユーザの操作に対して即座に応答を返す
 - ゲームコントローラによる高度な操作性の実現
- インタラクティブコンピュータグラフィックス
 - コンピュータとの対話 (interaction)

CG本来の目的

最初の CG 教科書（これは第 2 版）



現在の CG 技術

- ・グラフィックスハードウェアの**高速化**
 - ・インタラクティブに実現できることが増えた
- ・グラフィックスハードウェアの**低価格化**
 - ・リアルタイム 3DCG が利用可能な局面が増えた
- ・グラフィックスハードウェアの**高機能化**
 - ・使いこなすために技術開発の要素が増えた
- ・プログラマブルなグラフィックスハードウェアの登場
 - ・新技術をソフトウェア的に実装することが可能になった

CG の技術開発

- 要求されている機能を分析する
 - 再現したい現象をモデル化する, …
- 機能の実現方法を開発する
 - 理論や解法, アルゴリズムを考える, …
- 機能を実装する
 - 解法やアルゴリズムをコード化する
 - ライブラリや API の組み合わせを考える
 - …

ゲーム開発に必要な知識

- ・ 数学
- ・ 物理学
- ・ 人工知能
- ・ アルゴリズム
- ・ プログラム開発技法
- ・ 言語処理系の実装の知識
- ・ グラフィックス API / ミドルウェア
- ・ システム / ネットワークプログラミング
- ・ グラフィックスハードウェアの効果的な利用
- ・ そして、もちろん英語（最新の技術情報は英語で入ってくる）

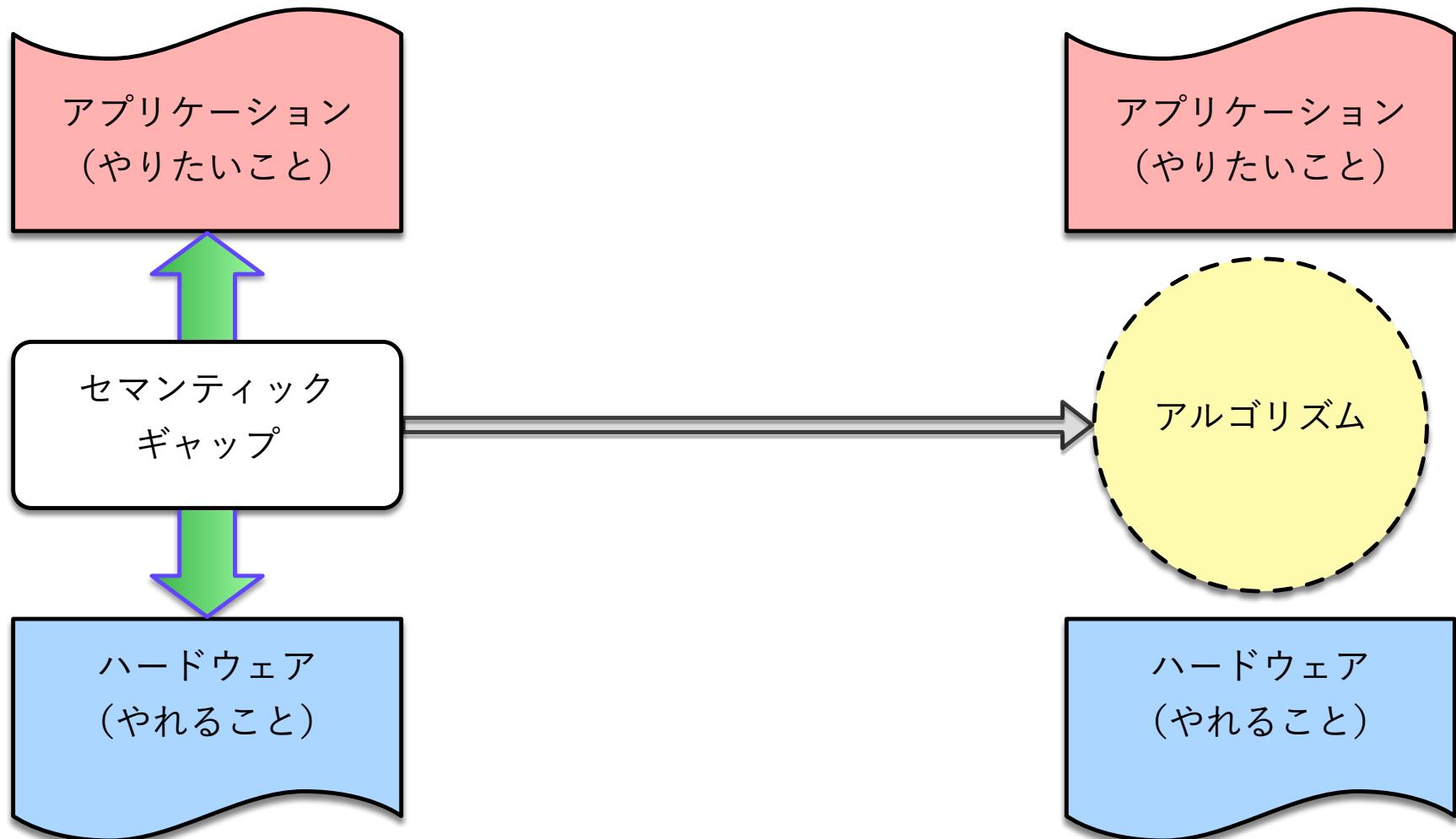


何でも知ってる
スーパーエンジニア

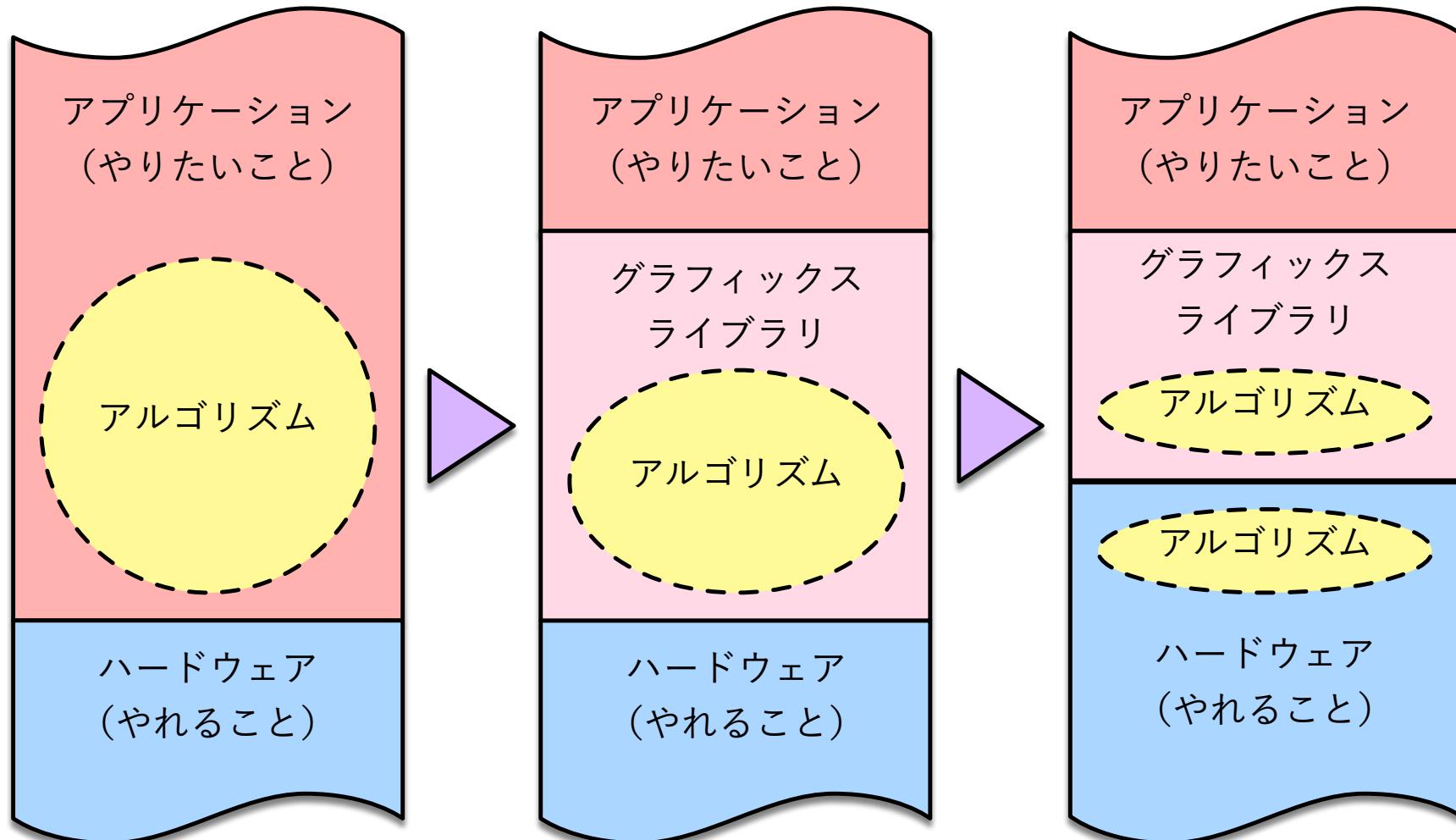
グラフィックスライブラリ

アプリケーションとハードウェアの仲立ち

セマンティックギャップとアルゴリズム



アルゴリズムの実装場所



グラフィックスライブラリ

- ・ アプリケーションソフトウェアにグラフィックスの機能を提供する
 - ・ 標準的なグラフィックスアルゴリズムを提供する
 - ・ ソフトウェア開発の手間やコストを減じる
 - ・ ソフトウェアのポータビリティを向上する
 - ・ ACM CORE, ISO GKS, GKS-3D, PHIGS, PHIGS+, …
 - ・ グラフィックスハードウェアの機能を呼び出す方法を提供する
 - ・ アプリケーションプログラムがグラフィックスハードウェアを制御するためのインターフェースを提供する
 - ・ API (Application Program Interface)
 - ・ ハードウェアの機能を抽象化する
 - ・ OpenGL, Direct3D, …

使われなく
なった

現在主流のグラフィックスライブラリ

- **OpenGL**

- Silicon Graphics (SGI) 社 が開発し後にオープンソースとなった
- 現在は Khronos グループが規格を策定している
- UNIX, Linux, FreeBSD, Windows, macOS など様々なプラットフォームで採用されている

- **OpenGL ES**

- OpenGL の組み込み機器向けのサブセット
- iOS, Android のほか WebGL でも採用されている

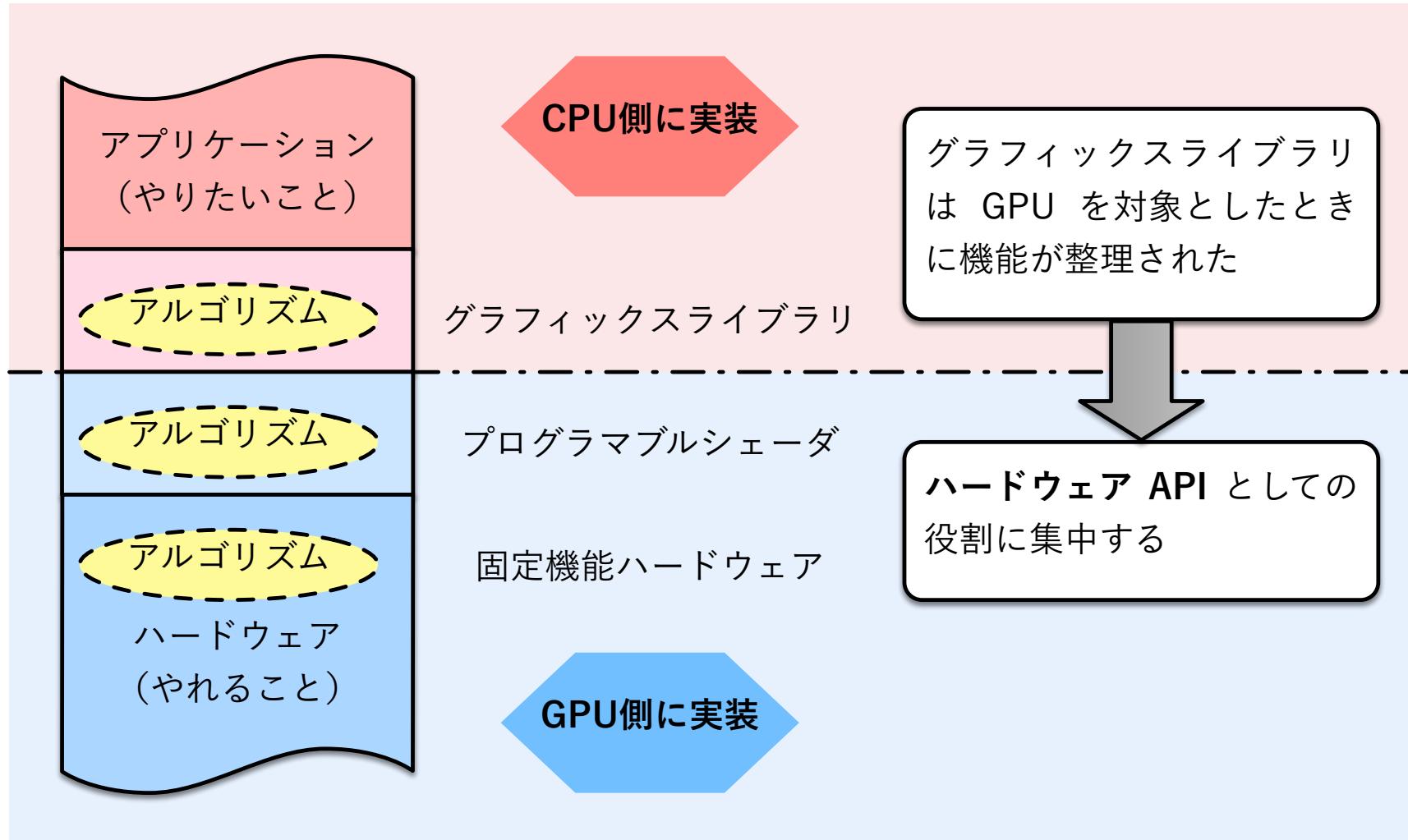
- **Direct3D**

- Microsoft 社のマルチメディア API である DirectX の 3D グラフィックス部分
- Windows のほか Xbox, Xbox 360, Xbox One で動作する

グラフィックスのアルゴリズム

- OpenGL / DirectX もアルゴリズムを実装している
 - ただし高水準の（複雑な）機能は別の層に移管する方向にある
 - ミドルウェア等
 - グラフィックスハードウェアを制御する機能しかもたない
 - データの入出力やサウンドなどは取り扱わない
 - グラフィックスハードウェアの機能を抽象化する
 - ハードウェアを直接制御するには非常に煩雑な手順が必要になる
- **Vulkan** (次世代 OpenGL) / **DirectX 12** の場合
 - ハードウェアを詳細に制御することが可能になっている
 - 抽象化のための CPU 処理のオーバーヘッドが無視できないため
 - Apple は macOS に同様な API として **Metal** を導入した

GPU におけるグラフィックスライブラリ



ハードウェア API とミドルウェア

- ・ハードウェア API の役割
 - ・グラフィックスハードウェアの機能の抽象化
 - ・高水準の機能 (アルゴリズム) の実装は控える
- ・高水準の機能の実装
 - ・アプリケーションプログラム内に実装
 - ・アプリケーションプログラムから GPU にダウンロード
 - ・シェーダプログラム
 - ・ミドルウェアの利用

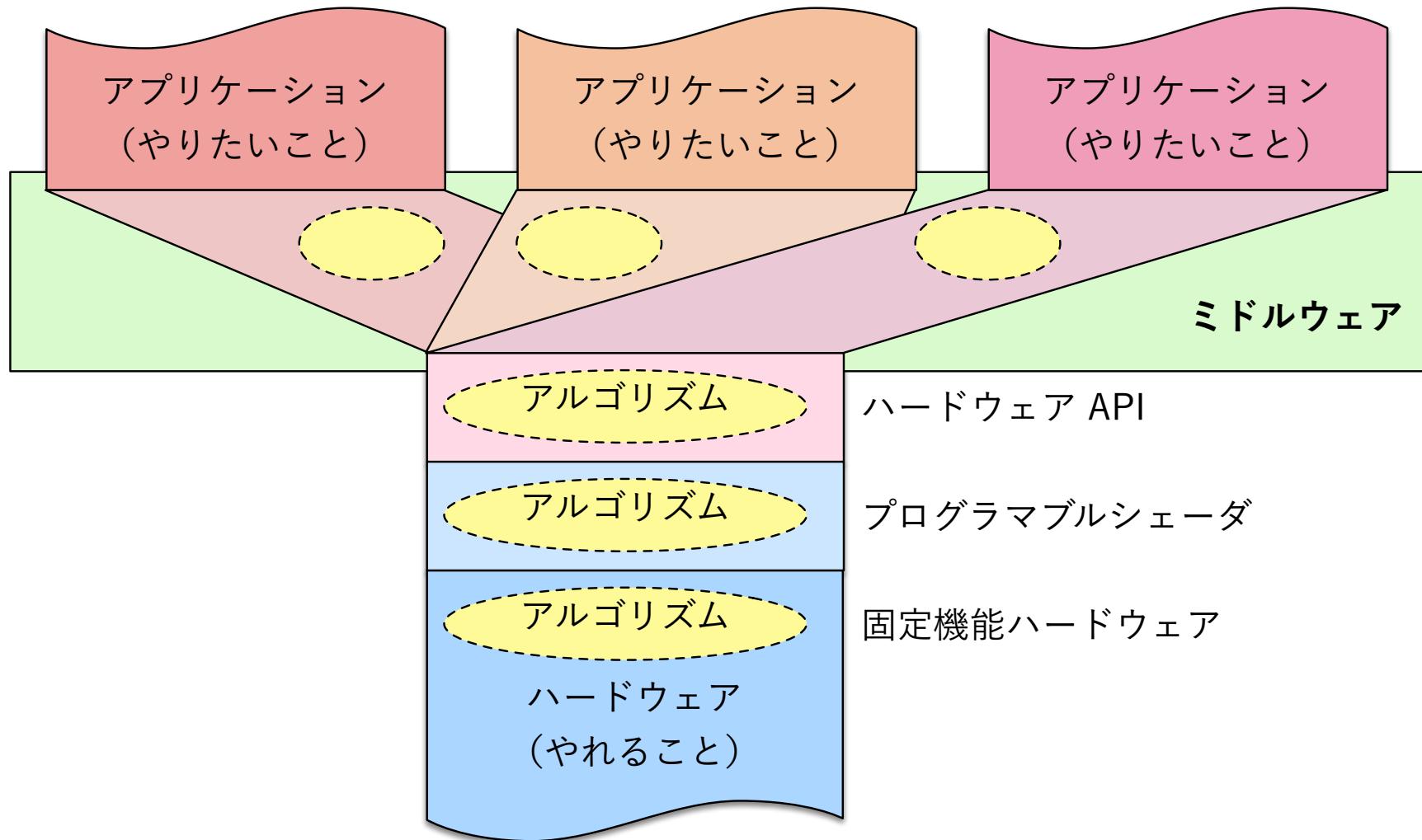


要求される機能が
多様化

ミドルウェア

- 目的に合わせて高水準の機能を提供するライブラリ
 - シーングラフ API (CG シーンの記述)
 - OpenSceneGraph, Scenix, OpenInventor (obsolete), …
 - 物理演算エンジン (剛体、柔軟体、流体などの挙動の再現)
 - Havok, Physx, Bullet, ODE, …
 - レンダリングエンジン (映像生成)
 - Mizuchi, OGRE, …
 - エフェクトエンジン (映像効果)
 - BISHAMON, YEBIS, …
 - ゲームエンジン (統合開発環境)
 - CryENGINE, Unreal Engine, Unity, MascotCapsule, OROCHI, chidori, Irrlicht, OGRE, Blender Game Engine, …

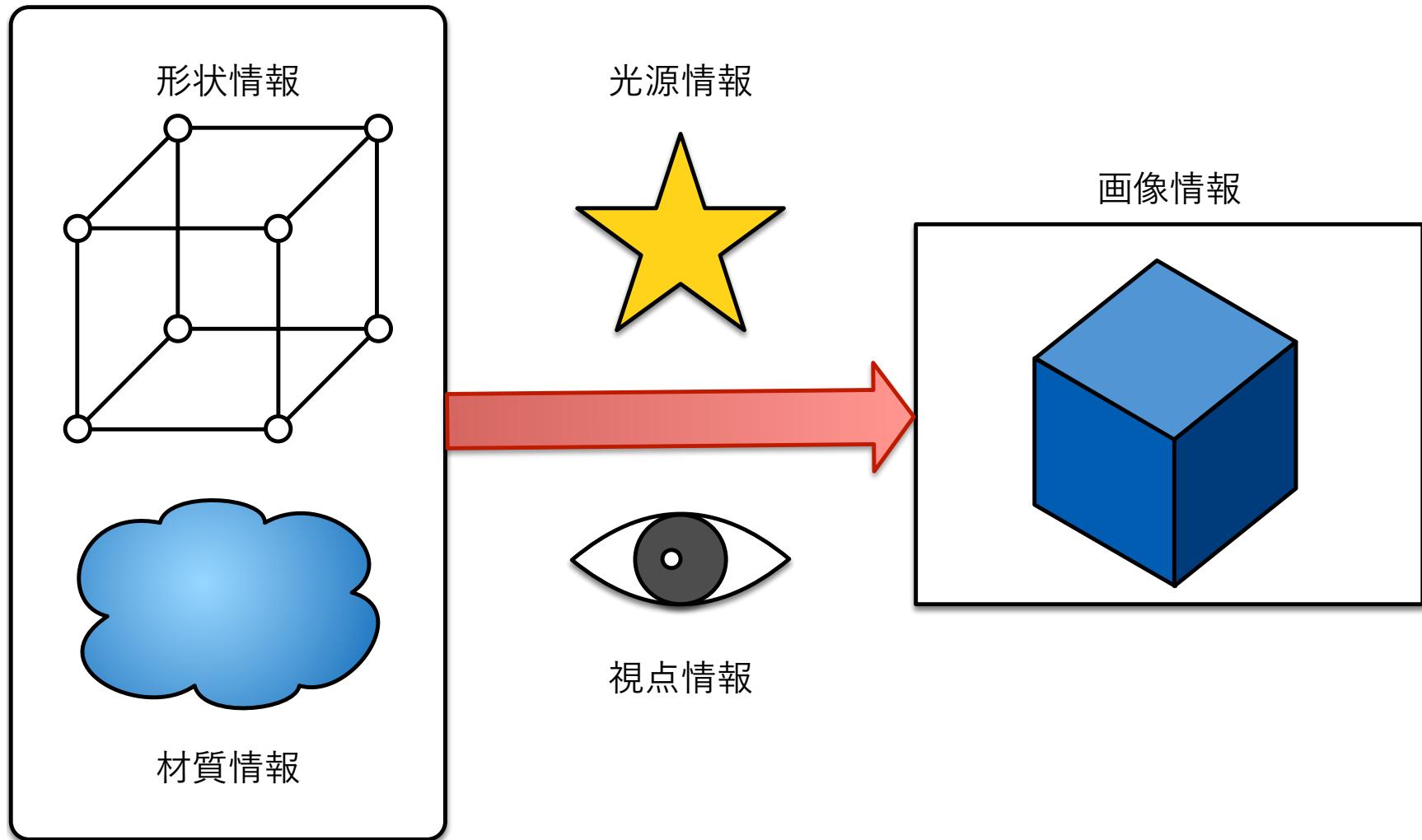
多様なアプリケーションとミドルウェア



レンダリング

映像生成

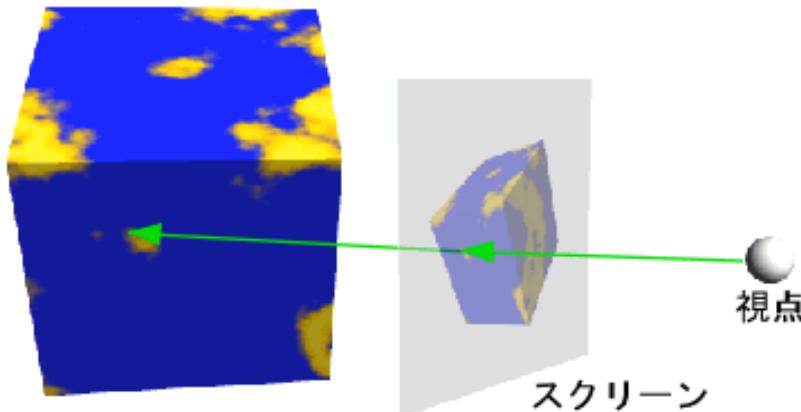
レンダリングとは



レンダリングの二つの方向

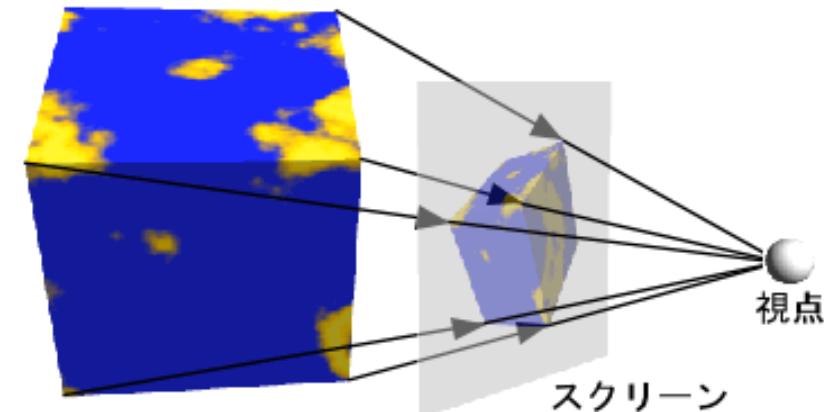
サンプリングによる方法

スクリーン上の1点に
何が見えるか調べる



ラスタライズによる方法

物体の表面形状を
スクリーン上に投影する



サンプリングとラスタライズ

- ・サンプリングによる方法
 - ・レイキャスティング法, レイトレーシング法
 - ・複雑な光学現象の再現が行いやすい
 - ・高品質な映像生成が行える
 - ・一般に処理に時間がかかる
 - ・リアルタイムレンダリングでは用いにくい
- ・ラスタライズによる方法
 - ・スキャンライン法, デプスバッファ法
 - ・一般に複雑な光学現象の再現には向かない
 - ・リアルさに欠ける場合がある
 - ・ハードウェアの支援による高速化が行いやすい
 - ・リアルタイムレンダリングで用いられる

現在では

- この二つは互いに近づいている
- この二つは組み合わせて使用される

リアルタイムレンダリングとは

- ・コンピュータによる高速な画像生成
 - ・画像による観測者の**反応**を次に**表示**される画像に反映する
 - ・この**反応→表示**のサイクルが十分高速なら観測者は**没入感**を得る
 - ・**フレームレート**が重要になる
 - ・1秒間に生成する画像の枚数, **fps** (Frames Per Second)
- ・通常, 三次元のシーンを対象にする
 - ・単に図形を描くだけではない
 - ・物理シミュレーション等も含まれる
 - ・陰影付け
 - ・アニメーションの生成
 - ・シミュレーション (運動, 衝突, 変形, 破壊, 流体, …)
 - ・インターフェース機器からのデータ入力 (ゲームパッド, センサ)

フレームレート

| | |
|--------|---|
| 1 fps | <ul style="list-style-type: none">● 1枚1枚の画像が順に現れるように見える● 対話的操縦はほぼ不可能 |
| 8 fps | <ul style="list-style-type: none">● ぎこちないが動画として知覚される● 対話的操縦が可能になる |
| 15 fps | <ul style="list-style-type: none">● 動きはほぼ滑らかに感じられる● 対話的操縦に違和感がない |
| 60 fps | <ul style="list-style-type: none">● 一般的なフラットパネルディスプレイの表示間隔 |
| それ以上 | <ul style="list-style-type: none">● 以下の用途で用いられる場合がある● 表示の遅れが操作に影響する場合（ゲーム等）● 時分割多重による立体視を行う場合● Head Mounted Display で頭の動きに追従する場合 |

フレームレートとリフレッシュレート

- ・フレームレート (単位 **fps**)
 - ・コンピュータが 1 秒間に生成可能なフレーム (画像) の数
 - ・シーンの複雑さによって変化する
- ・リフレッシュレート (単位 **Hz**)
 - ・ディスプレイが 1 秒間に表示するフレームの数
 - ・ディスプレイの特性値であり固定

表示はリフレッシュレートに同期する

- 60Hz ならフレームレートを 60 fps 以上にできない
 - リフレッシュレートを無視してフレームレートを上げることはできなくはないが正常な表示とならない
 - フレームが表示されなかったりティアリングが発生したりする
- レンダリングに時間がかかると fps はリフレッシュレートの整数分の 1 になる
 - 60fps → 30fps → 20fps → 15fps → 12fps → 10fps → …
 - 1 フレーム 16ms 以下なら 60fps で表示できる
 - しかし 17ms かかると 30fps になる
- フレームートがリフレッシュレートに制限されない技術
 - **G-SYNC** (NVIDIA), **FreeSync** (AMD)
 - 対応ディスプレイ, 対応ビデオカードが必要

グラフィックスハードウェアの重要性

- ・リアルタイムレンダリングの条件
 - ・三次元空間を対象とすること
 - ・立体図形の表示が行えること
 - ・陰影付けが行えること
 - ・インタラクティビティ（**対話性**）をもつこと
 - ・アニメーションが生成できること
 - ・コントローラ等からの入力を即座に処理できること
- ・グラフィックスハードウェアの支援が必要
 - ・ほとんどの 3D グラフィックスアプリケーションで要求される
 - ・グラフィックスハードウェアは現在の PC に必須
 - ・現在では多くの **CPU** に内蔵されている

レンダリングパイプライン

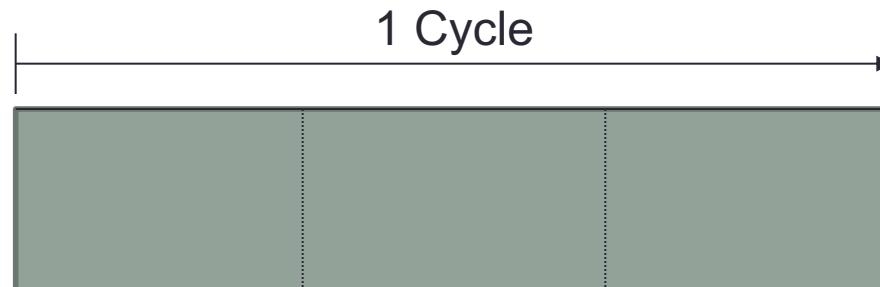
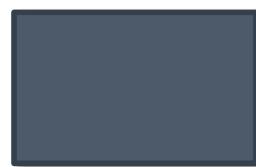
グラフィックス処理の流れ

パイプライン処理

- ・ 非パイプライン構造を n 個のパイプライン化ステージに分割
 - ・ 最大 n 倍スピードアップできる
- ・ 最も遅いステージがパイプライン全体の速度を決定する
 - ・ 他のステージはその間遊んでいる
 - ・ ボトルネック

パイプライン処理による速度向上

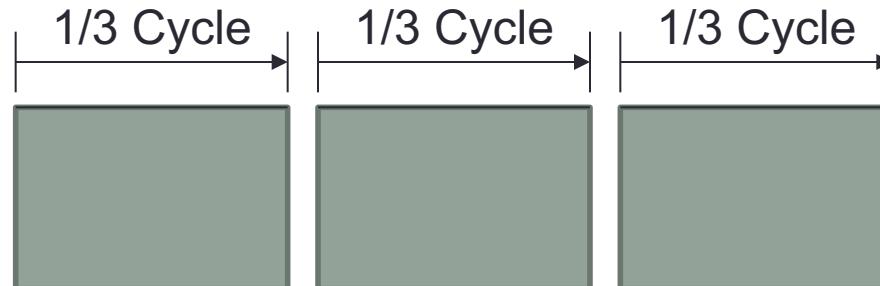
Not Pipelined



1 Cycle

1 Operation/Cycle

Pipelined



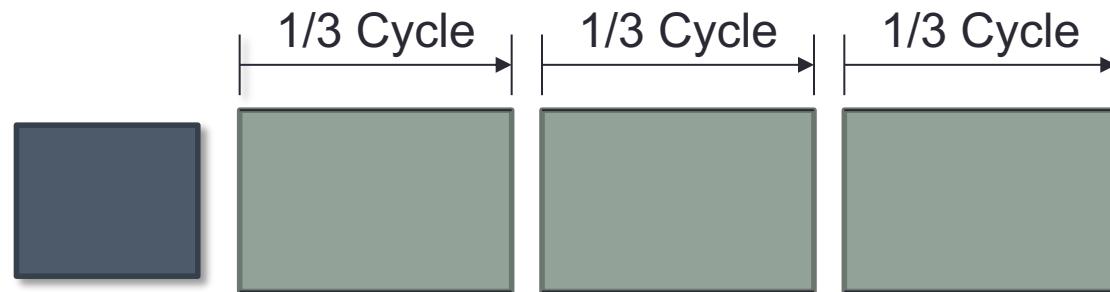
1/3 Cycle

1/3 Cycle

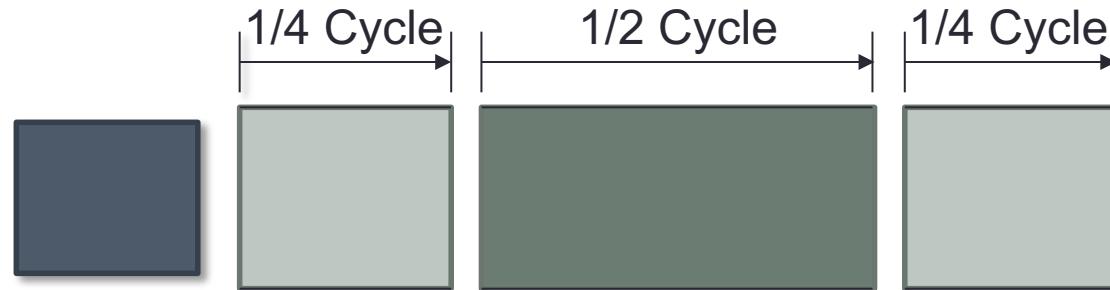
1/3 Cycle

3 Operations/Cycle

ボトルネック



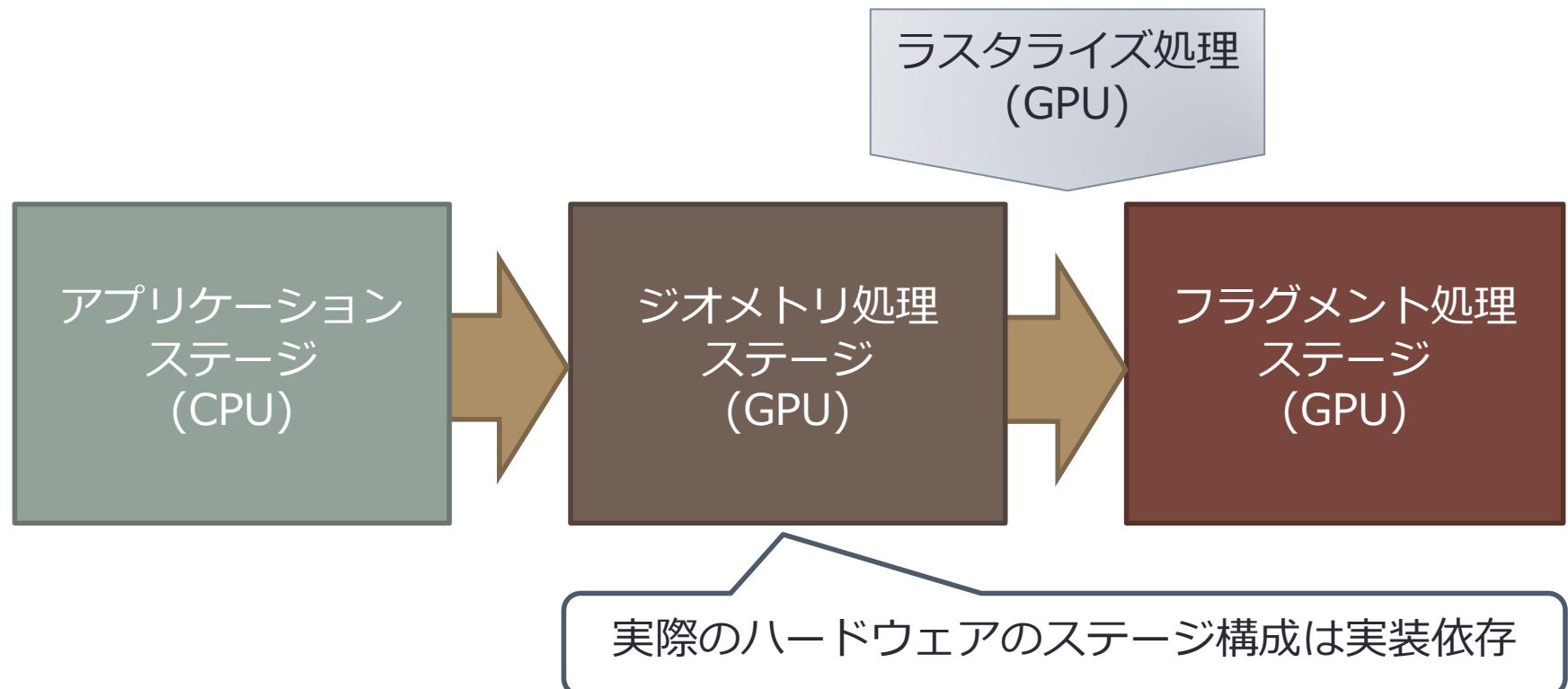
3 Operations/Cycle



2 Operations/Cycle

グラフィックス処理のパイプライン

- ハードウェアの概念上のステージ



アプリケーションのステージ

CPU 側のソフトウェアで行うこと

アプリケーションステージ

- ・開発者はすべてを制御できる
 - ・このステージは常にソフトウェアで実行される
 - ・実装を変更して動作を変更することが可能
- ・図形の情報を次のステージに送る
 - ・形状は基本図形（レンダリングプリミティブ）で表現されている
 - ・点, 線分, 三角形
 - ・アプリケーションステージの最も重要な役割
- ・ジオメトリデータ
 - ・アプリケーションステージから出力される図形データ
 - ・頂点の情報（位置, 色, 法線ベクトルほか）の集合 + 図形の種類

アプリケーションステージでの作業

- ・他のソースからの入力の面倒を見る
 - ・マウス, キーボード, ジョイスティック, イメージセンサ, …
- ・衝突検出
 - ・衝突が検出されたら画面表示や力覚デバイスに反映する, など
- ・時間に関する処理
 - ・座標変換によるアニメーション
 - ・一部ジオメトリステージで実現可能
 - ・スキニング・ジオメトリモーフィング
 - ・一部ジオメトリステージで実現可能
 - ・テクスチャアニメーション
- ・他のステージでは実行できない全ての種類の計算

アプリケーションステージでの最適化

- 次のステージに送るデータの量を減らす
 - カリング (Culling)
 - 画面表示に寄与しないデータをあらかじめ削除する
 - 階層的な視錐台カリング, オクルージョンカリング
- マルチコア CPU による並列処理
 - ただし次のステージ (GPU) の入り口 (バス) はひとつ
 - 複数のスレッドから同時にデータを流し込めない
 - スレッドごとに異なる処理を分担する
 - カリング, ジオメトリデータ送出, 衝突検出, シミュレーション, …
 - 処理内容が違うと時間がそろわないと同期をとるのが難しい
 - 将来のメニイコア化への対応は課題
 - Vulkan / DirectX 12 はそのために GPU の機能を細かく分割した

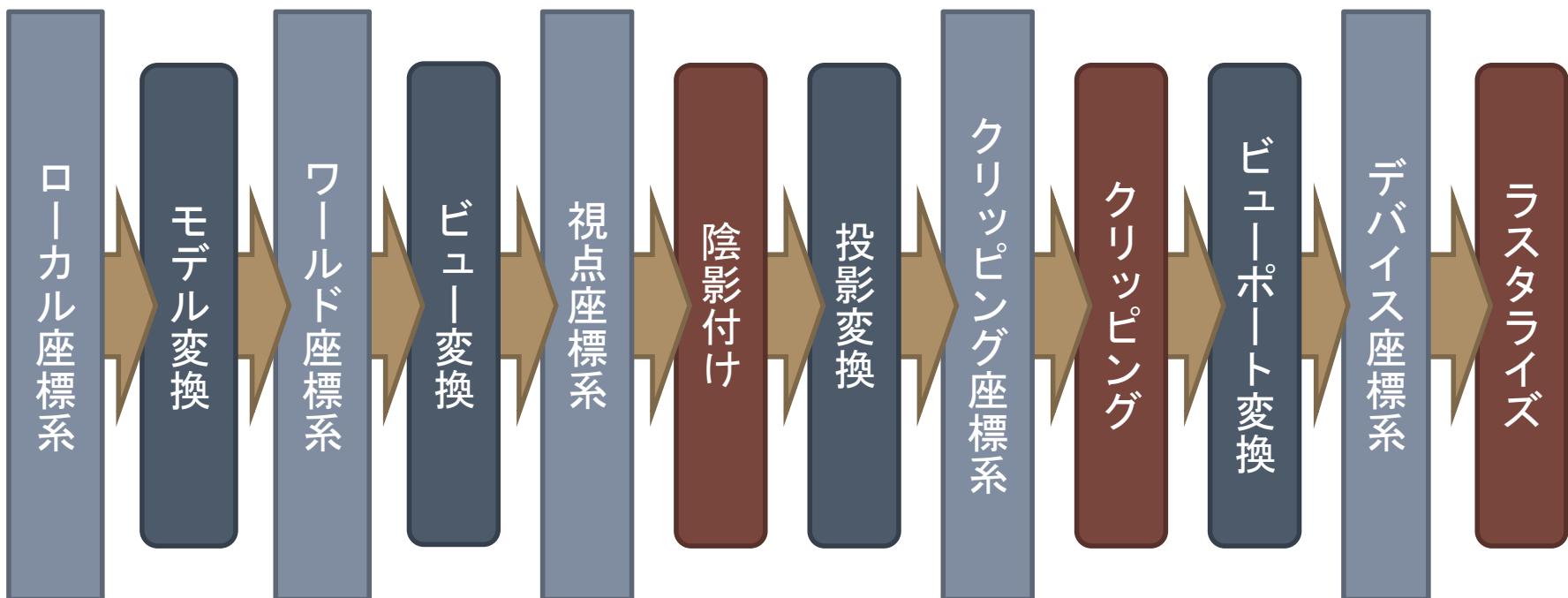
ジオメトリ処理ステージ

頂点単位の処理

ジオメトリ処理

- **頂点**単位に行う処理
 - 頂点位置の座標変換, 頂点の陰影付け, …
- **図形**単位に行う処理
 - クリッピング
- 複数の**固定機能**のステージで構成されている
 - より小さな複数のパイプラインステージに分割する場合もある
- 数値計算の負荷が高い
 - 陰影計算には非常に多くの**実数計算**が含まれる
 - テクスチャ座標の算出なども行われる

ジオメトリ処理のパイプライン

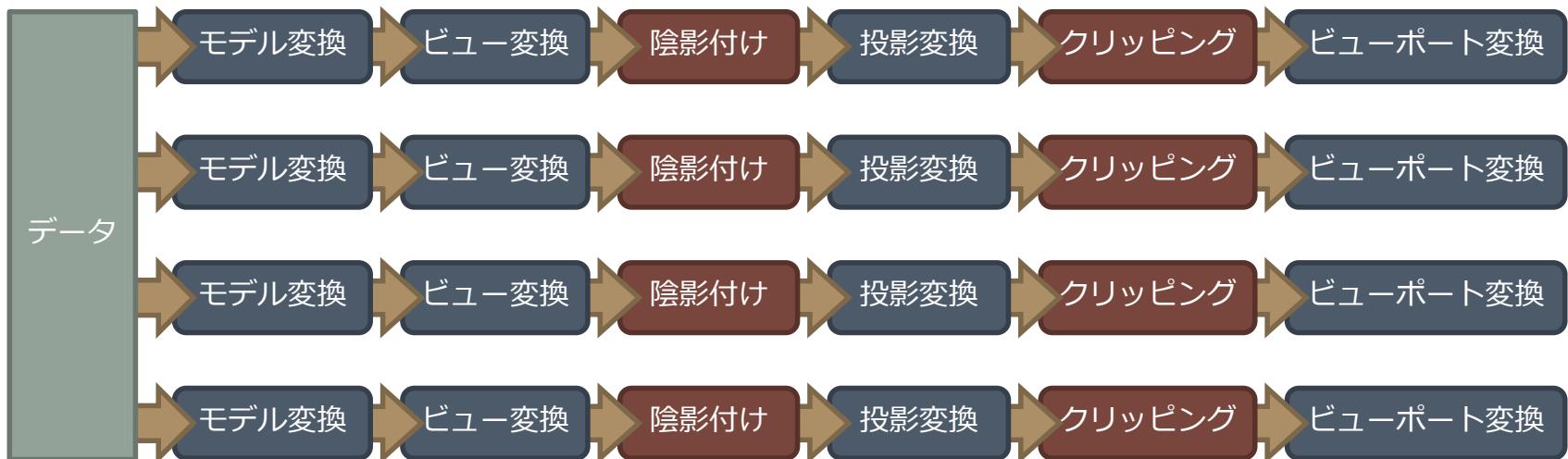


ジオメトリ処理のパイプラインの並列化

Single-Pipe



Multi-Pipe



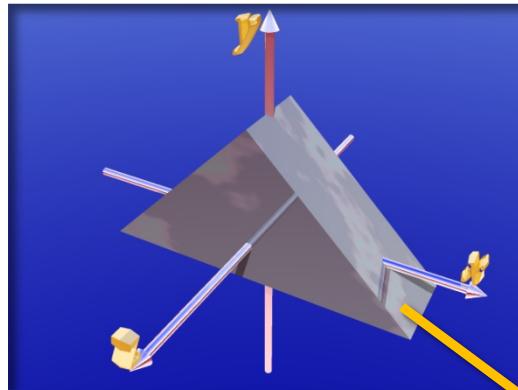
モデル変換とビュー変換

- ・オブジェクト（図形）はローカル座標系上で定義されている
 - ・個々のオブジェクト（モデル）がもつ独自の座標系
 - ・モデル座標系とも言う
- ・**モデル変換**
 - ・シーンを構成するために物体をワールド座標系上に配置する
 - ・モデル変換後はすべてのオブジェクトがこの空間内に存在する
- ・**ビュー変換**
 - ・視点（カメラ）はワールド座標系上に配置する
 - ・視点から見た画像を生成する
 - ・視野変換とも言う
- ・**ビューボリューム**
 - ・視野となる空間

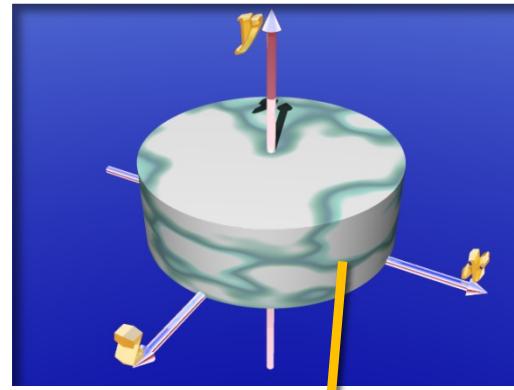
この二つはまとめて実行することが多い
(モデルビュー変換)

モデル変換

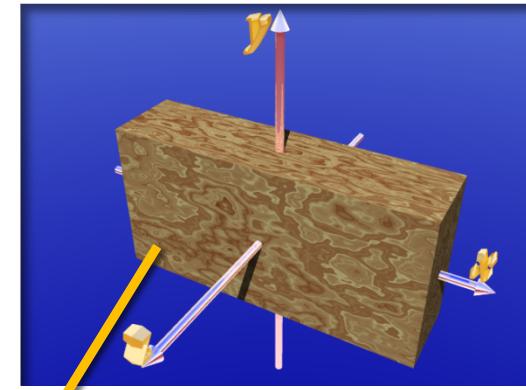
□ カル座標系



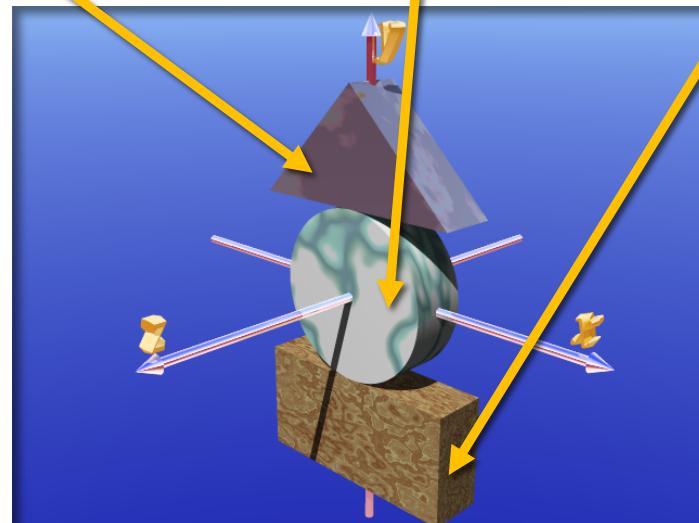
□ カル座標系



□ カル座標系

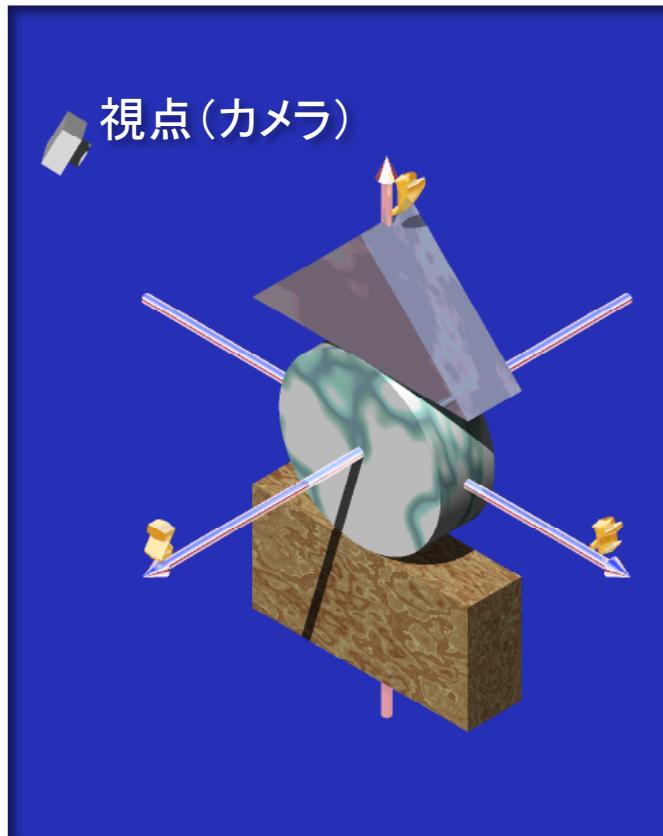


ワールド座標系

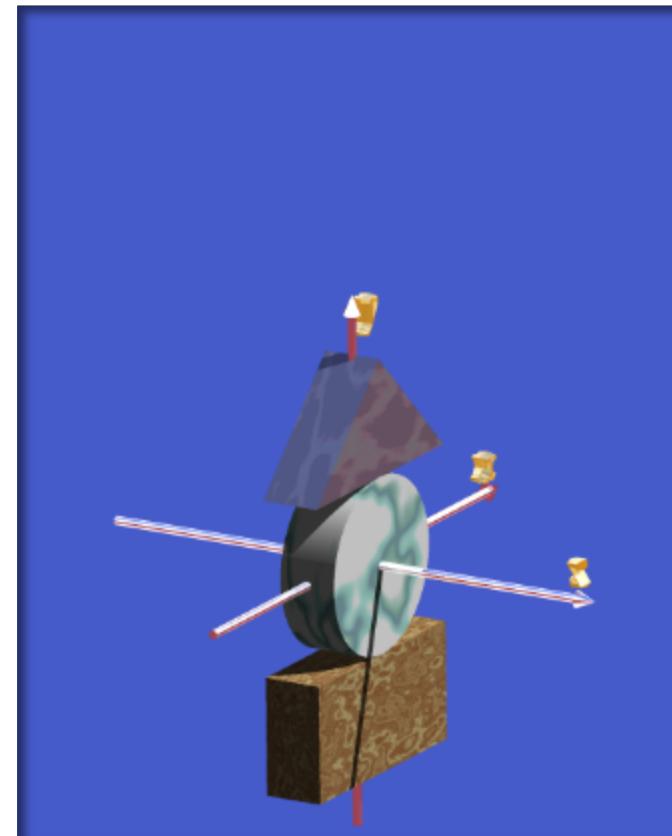


ビュー変換

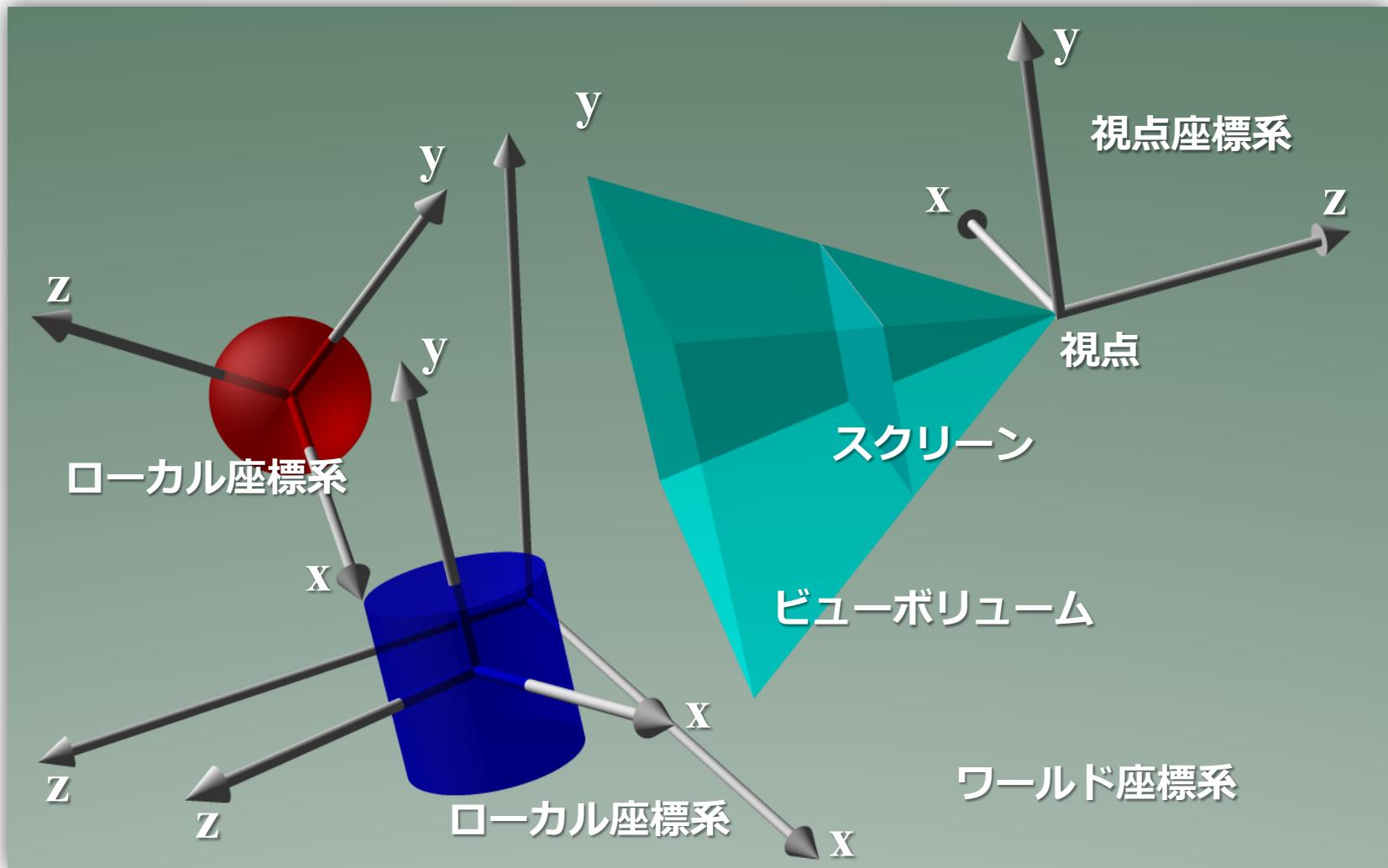
ワールド座標系



視点座標系



視点とスクリーンの関係

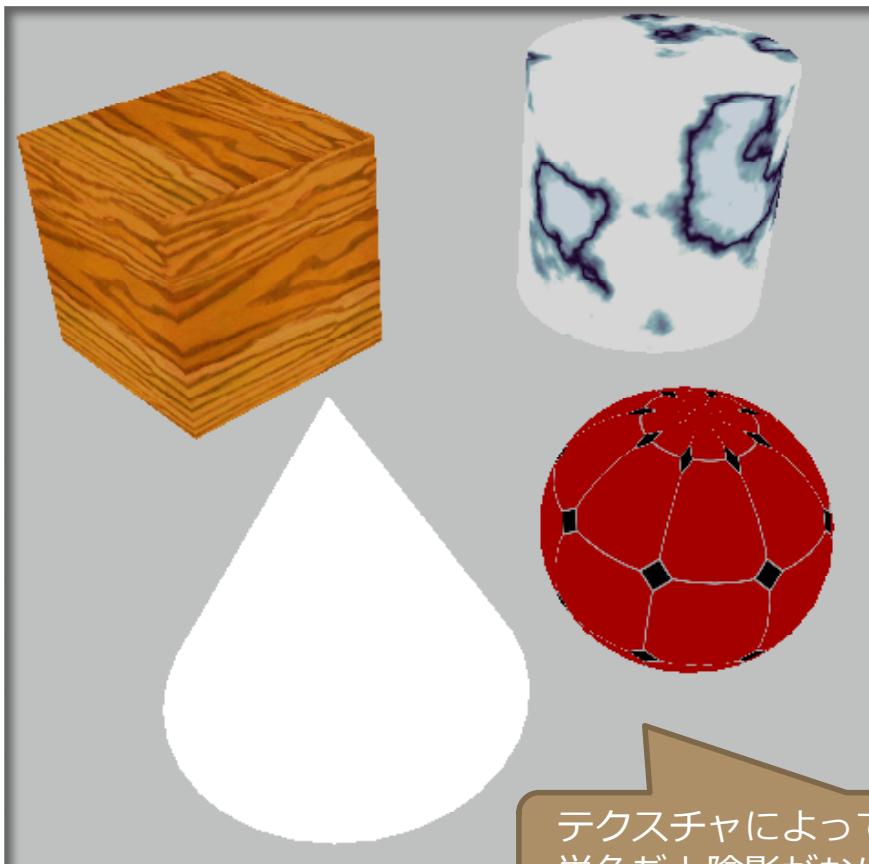


頂点の陰影付け

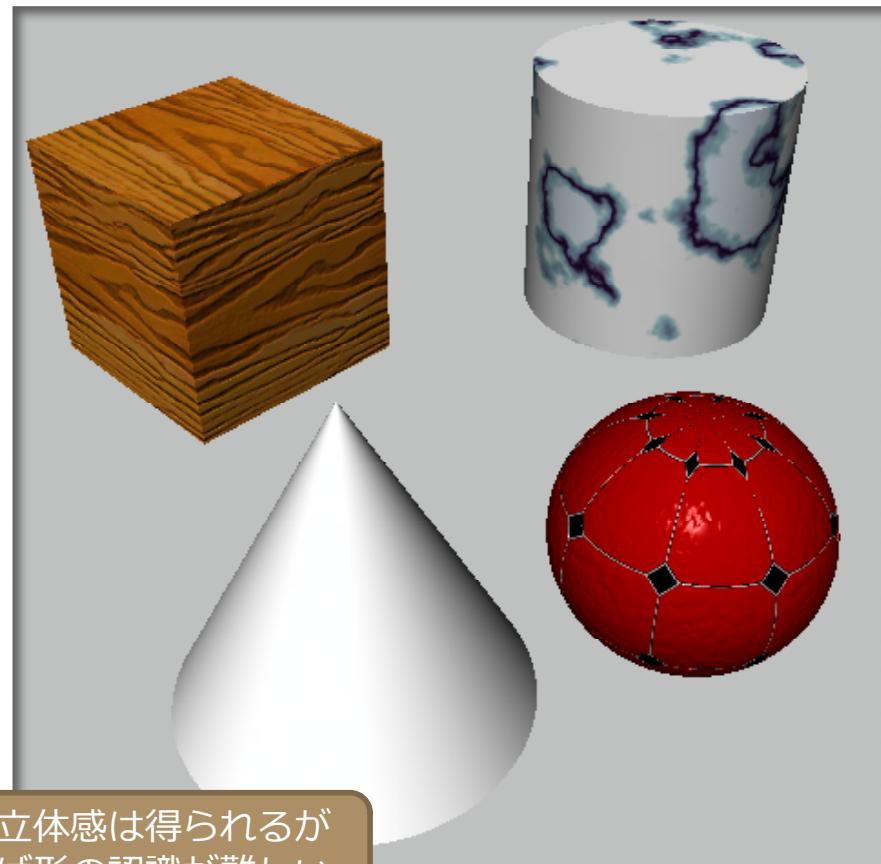
- ・ 陰影によってオブジェクトにリアルな外観を与える
 - ・ シーンに 1 つ以上の光源を設定する
- ・ 陰影付けの式によってオブジェクトの各頂点の色を計算する
 - ・ 実世界の光子（フォトン）と物体表面との相互作用を近似する
 - ・ 実世界では光子は光源から放出され、物体表面で反射・吸収される
 - ・ リアルタイムレンダリングでは、この計算に多くの時間を割けない
- ・ 本物の反射（映りこみ）や屈折、影（シャドウ）は含まない
 - ・ 擬似的手法を用いる
 - ・ GPU によりかなり精密に計算できるようになってきた

陰影付けの有無

陰影付けなし



陰影付けあり



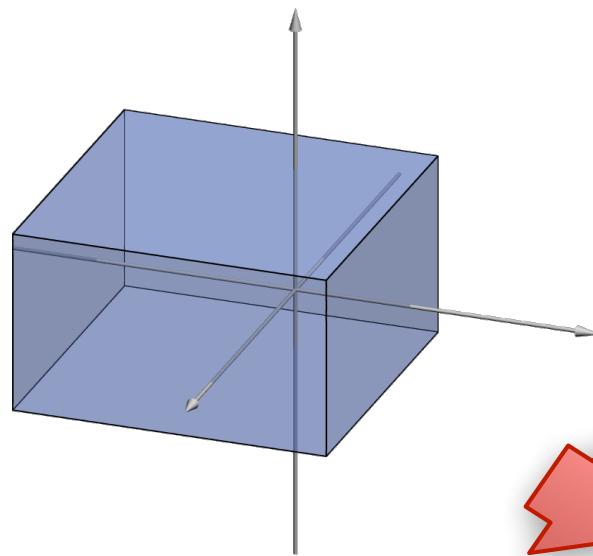
テクスチャによっても立体感は得られるが
単色だと陰影がなければ形の認識が難しい

投影変換

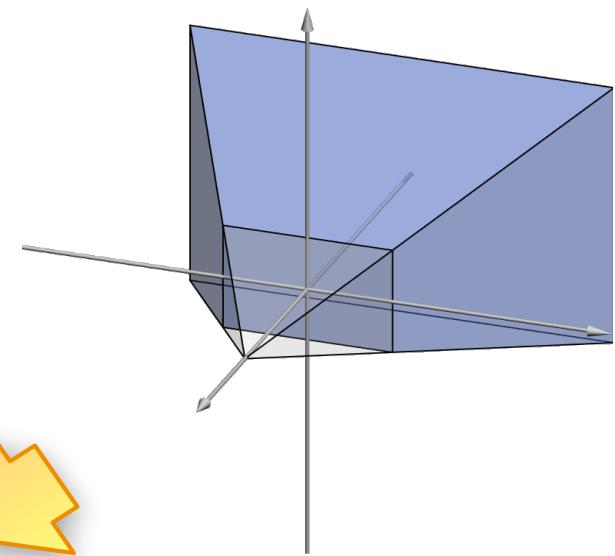
- ・ビューボリューム内の物体の座標値をクリッピング座標系 (Clipping Coordinate) 上の座標値に変換する
 - ・正規化デバイス座標系 (Normalized Device Coordinate, NDC)
 - ・ $(-1, -1, -1), (1, 1, 1)$ を対角の頂点とする x, y, z 軸に沿った立方体
 - ・標準ビューボリューム (Canonical View Volume) とも呼ばれる
- ・直交投影 (平行投影)
 - ・直方体のビューボリュームを用いる
- ・透視投影 (遠近投影)
 - ・錐台のビューボリューム (視錐台, View Frustum) を用いる

ビューボリューム

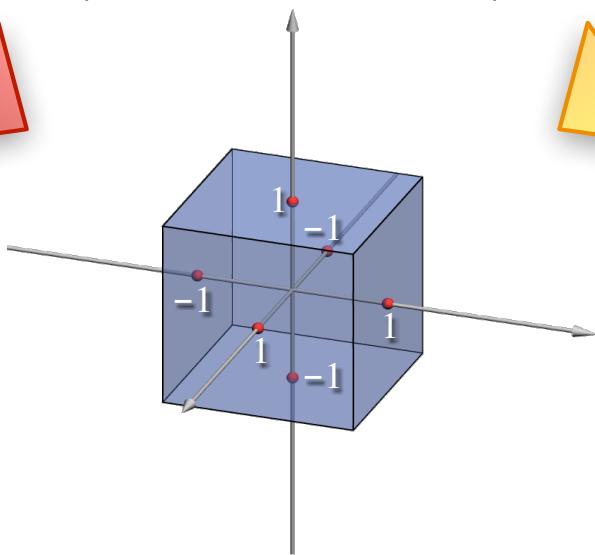
ビューボリューム (View Volume)



視野錐台 (View Frustum)



標準ビューボリューム
(Canonical View Volume)



直交投影
(Orthographic Projection)

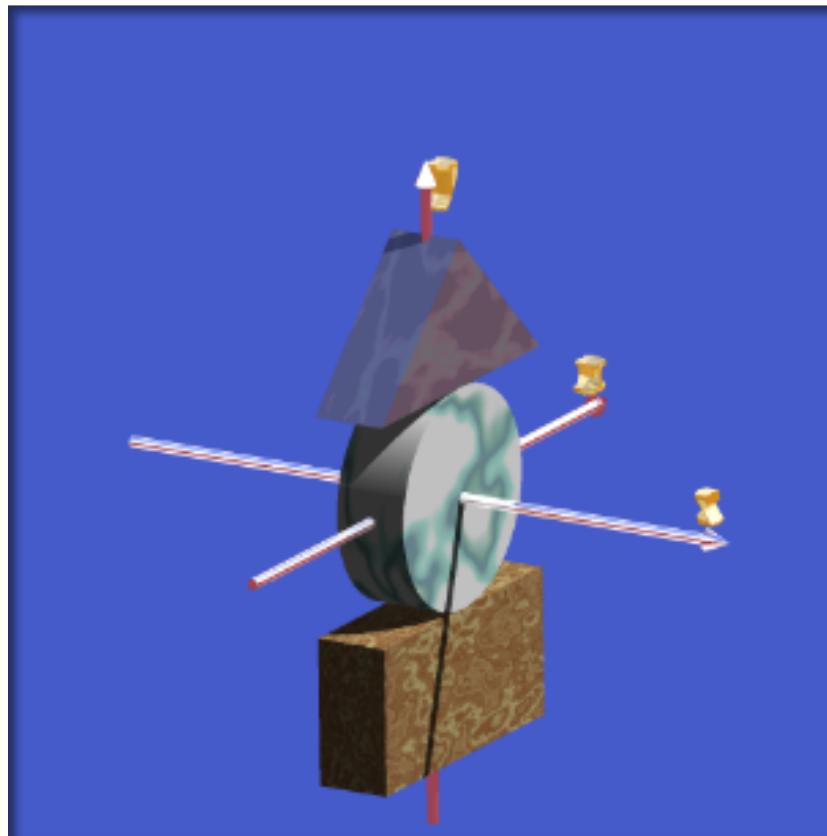


透視投影
(Perspective Projection)



直交投影と透視投影

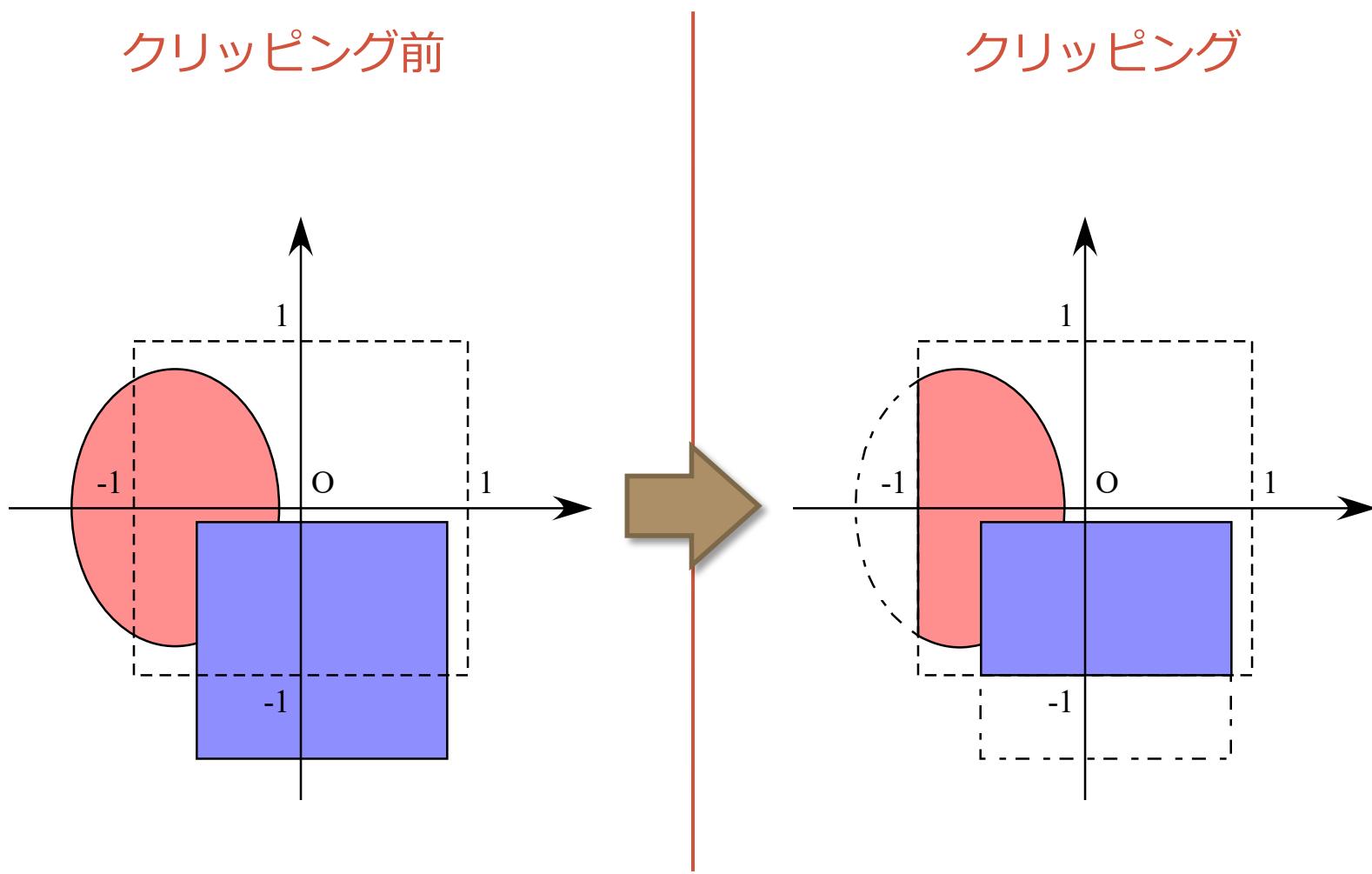
直交投影



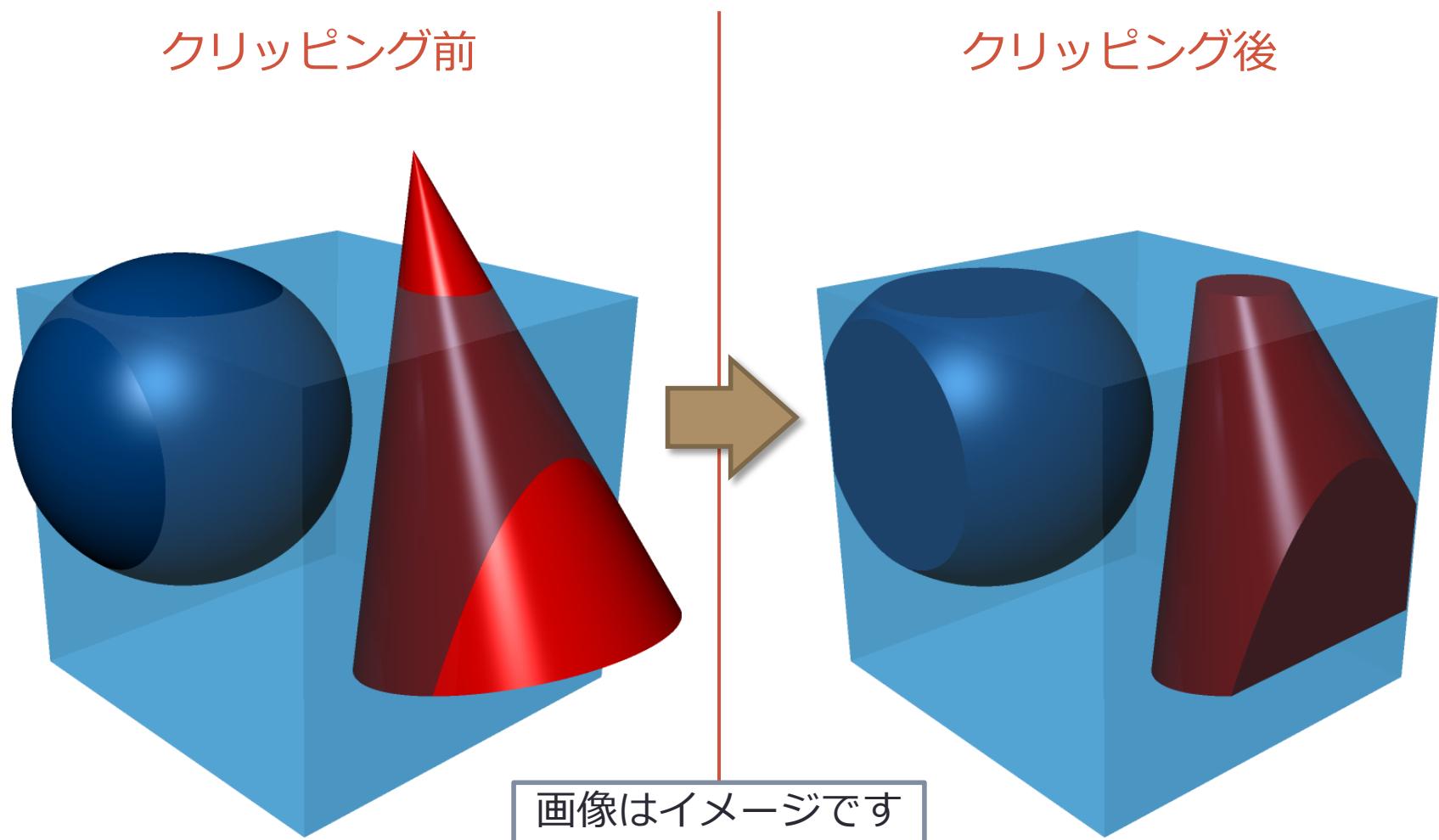
透視投影



二次元のクリッピング



ビューボリュームによるクリッピング

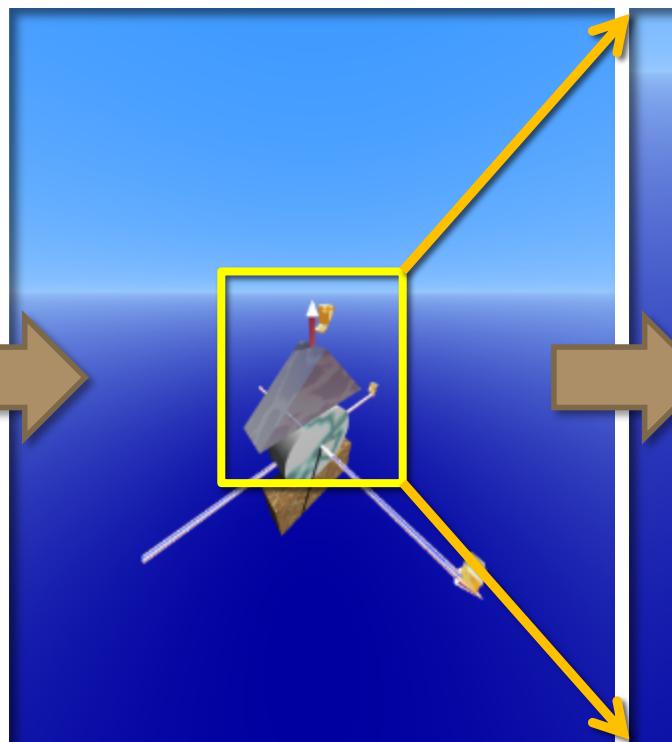


投影とクリッピング

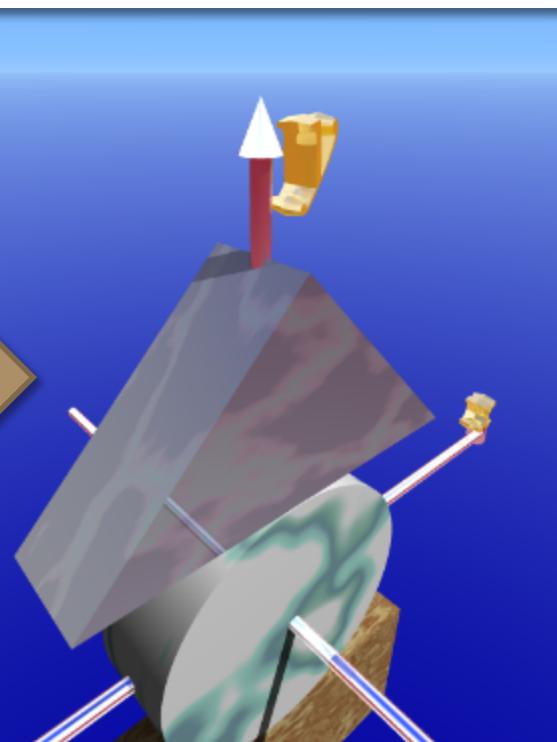
視点座標系



スクリーンの投影画像

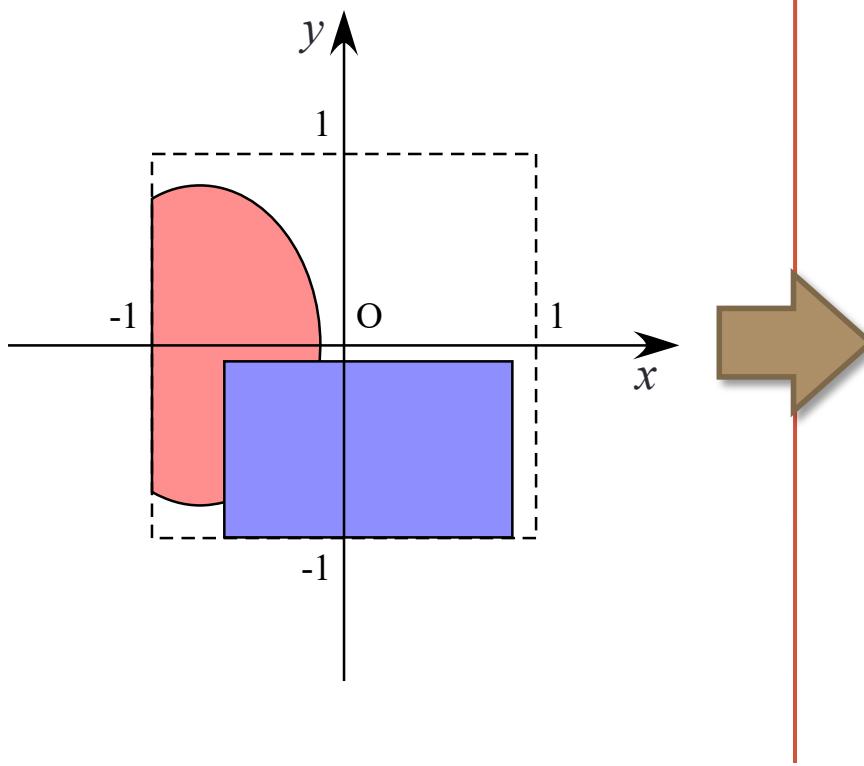


クリッピング座標系

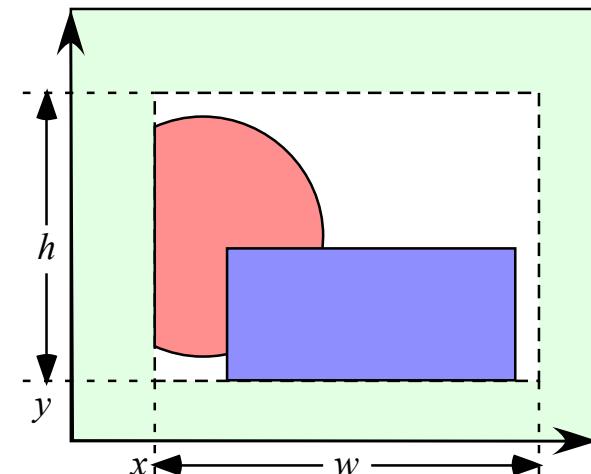


ビューポート変換

クリッピング座標系



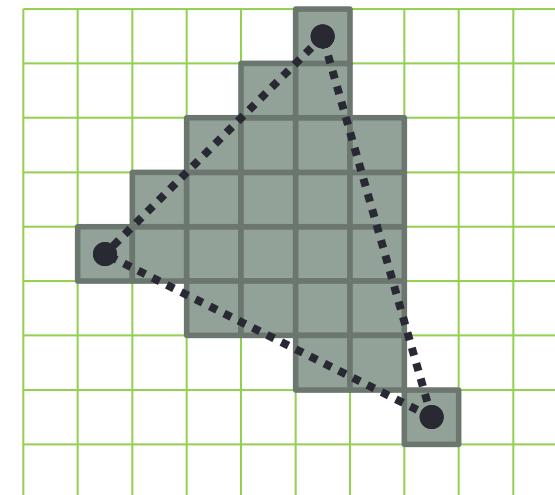
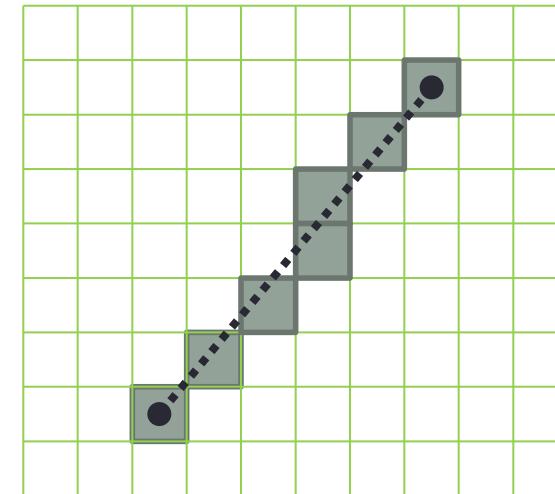
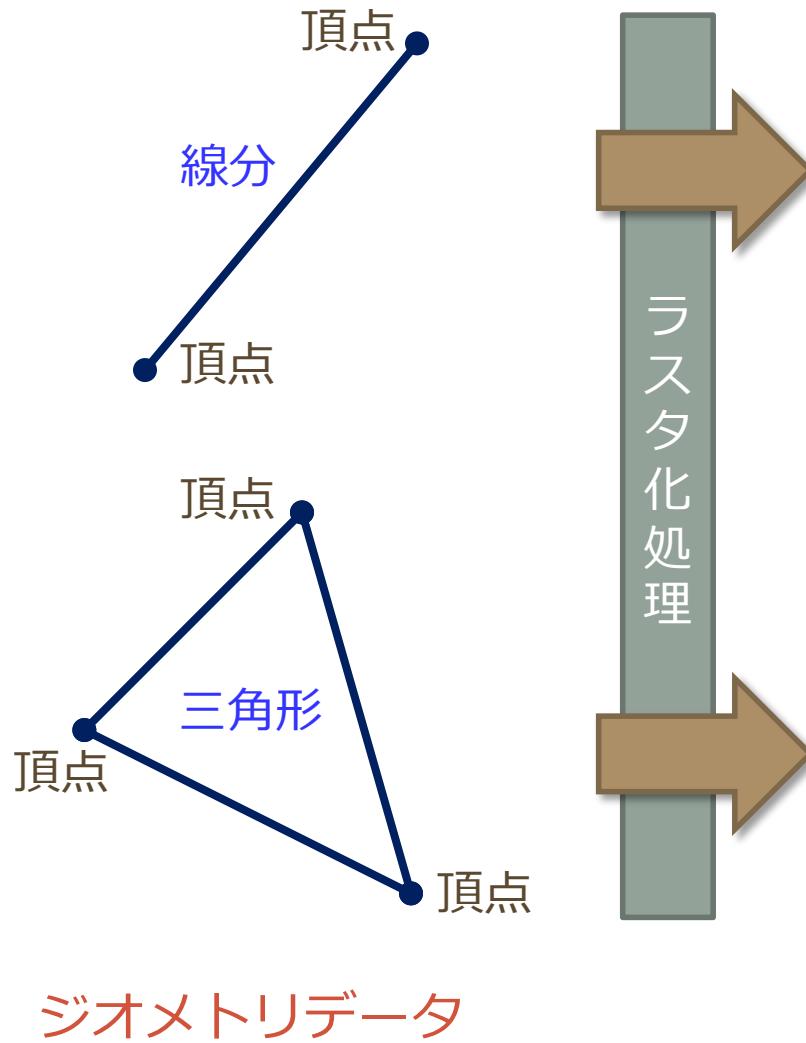
デバイス座標系



ラスタライズ処理

画像化

ラスタライズ処理



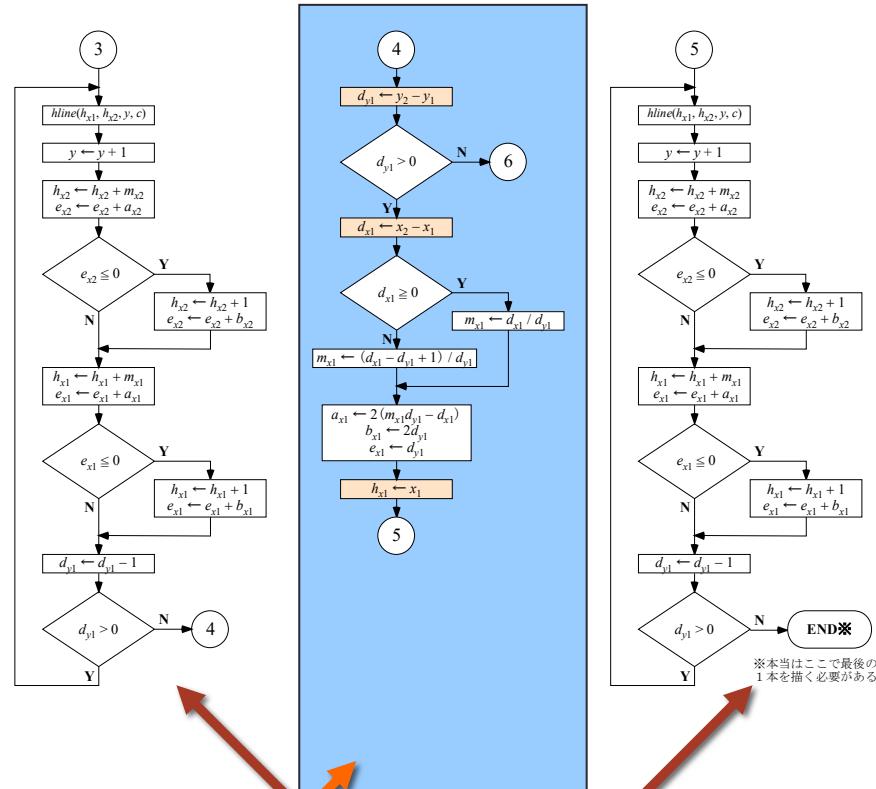
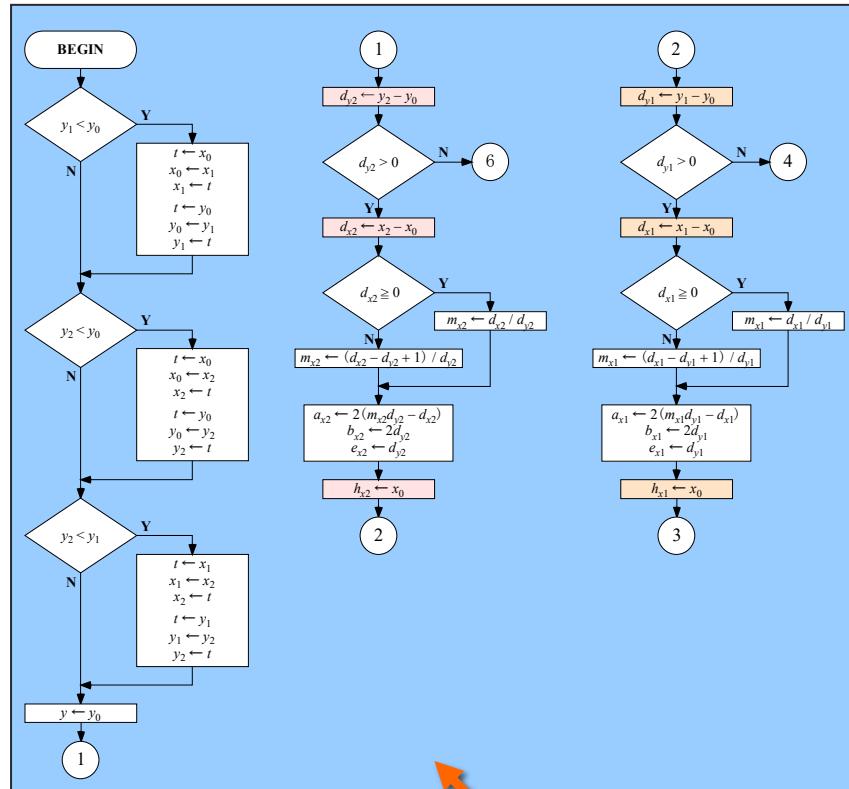
Fragment Data

ラスタライザ

- ・ 図形データから画像データを生成するハードウェア
 - ・ 三角形セットアップ
 - ・ 走査変換（塗りつぶし）のための係数等の計算
 - ・ 走査変換（スキャンコンバージョン）
 - ・ 図形（点・線分・三角形）によって覆われる画素を選択する
 - ・ 図形→ジオメトリデータ
 - ・ 画像→ラスタデータ
- ・ 頂点属性の補間機能をもつ
 - ・ 頂点情報を補間して選択された画素に与える
 - ・ 頂点色
 - ・ 画素の陰影計算に用いる
 - ・ 奥行き
 - ・ 隠面消去処理に用いる
 - ・ 他に、法線ベクトル、テクスチャ座標、…

プリミティブセットアップ

- プリミティブ（図形要素）の走査変換を行うための前準備



プリミティブセットアップ

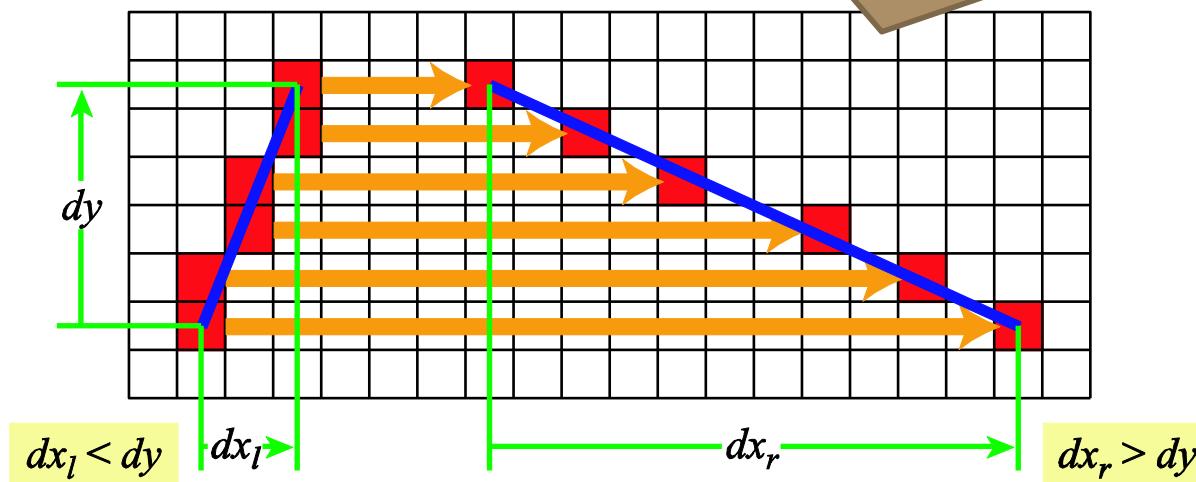
走査変換

※本當はここで最後の1本を描く必要がある

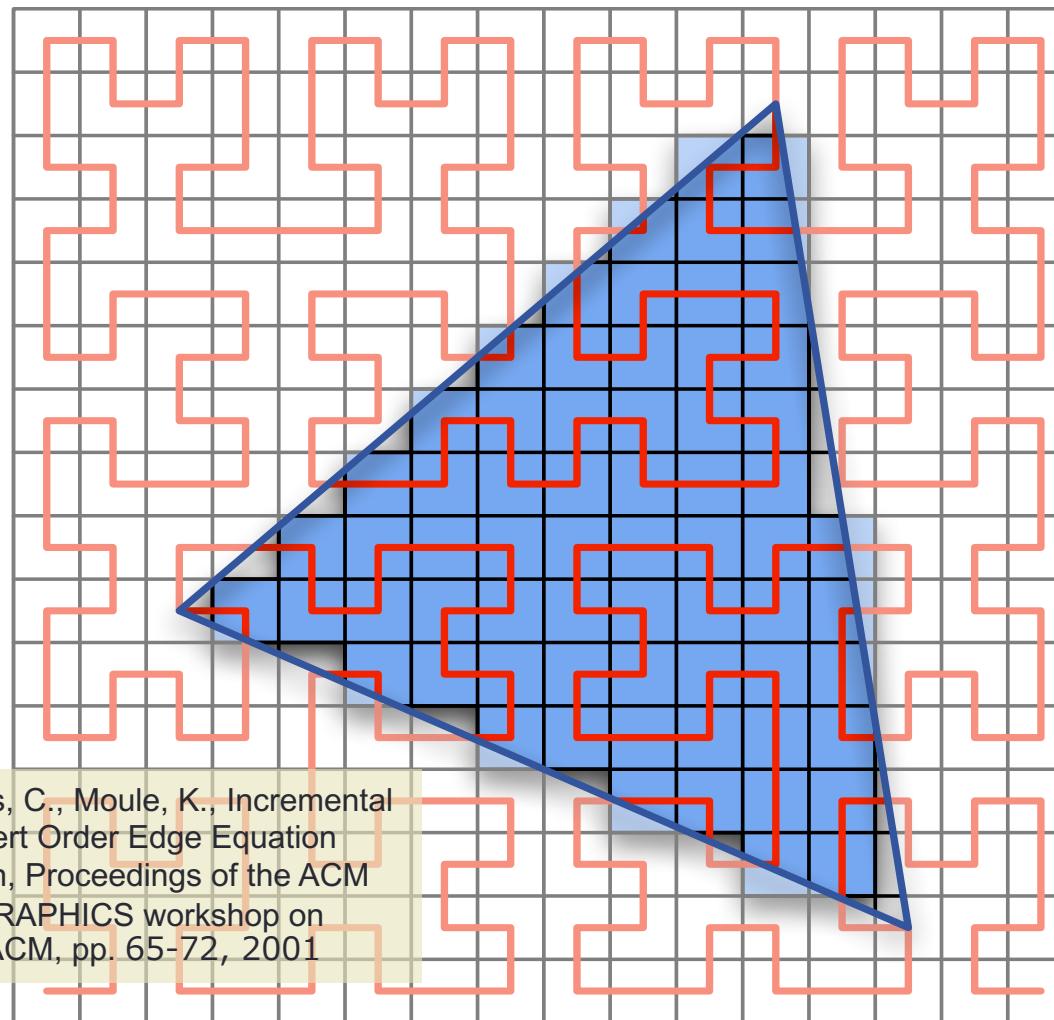
走査変換 (スキャンコンバージョン)

- ・頂点色の補間
- ・奥行きの補間
- ・テクスチャ座標の補間
- ・画素の選択

向かい合う辺上の点を結ぶ
水平線上にある画素を選択する



ヒルベルト曲線を使ったラスタライズ



McCool, M. D., Wales, C., Moule, K., Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM, pp. 65-72, 2001

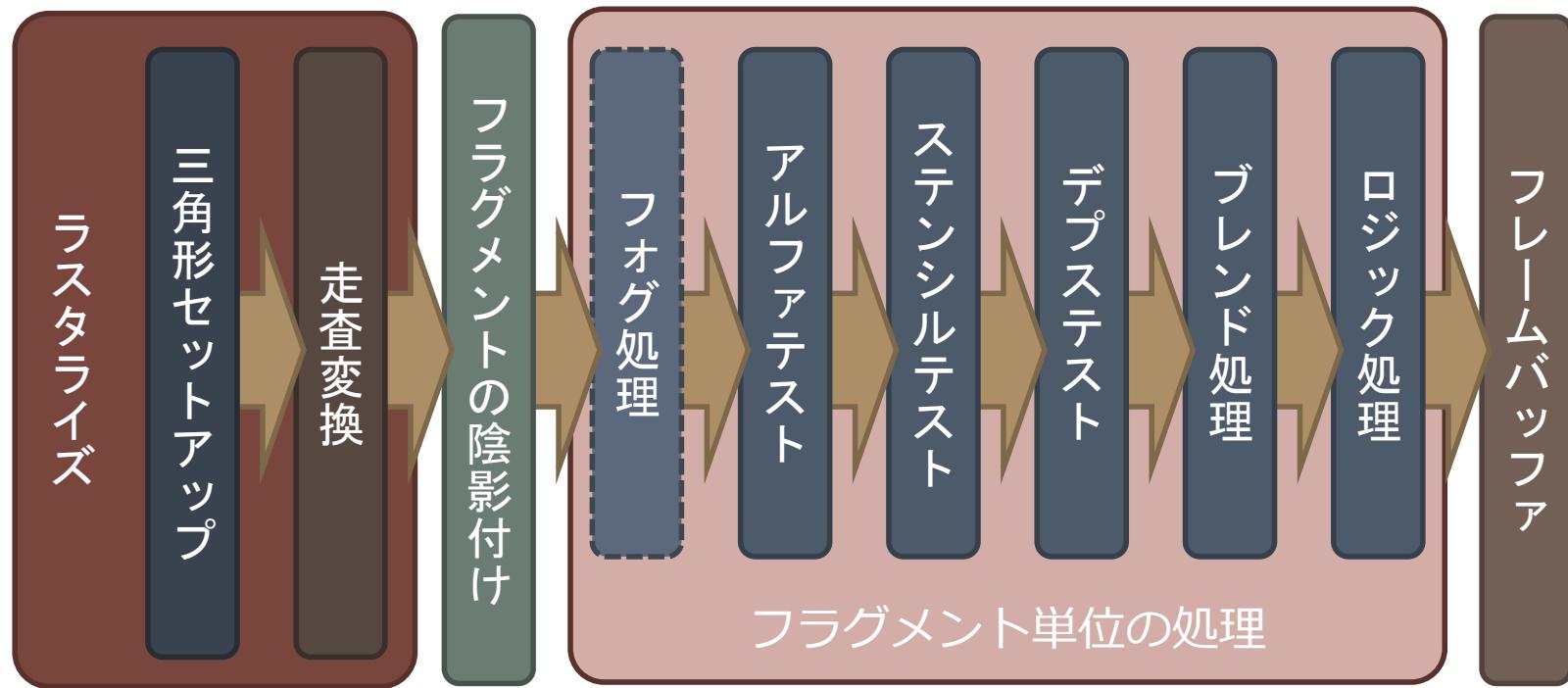
フラグメント処理ステージ

画素単位の処理

フラグメント処理

- 画素色の決定
 - フラグメント単位の陰影計算
 - テクスチャ座標の算出, テクスチャマッピング
- 可視判定
 - そのフラグメントにおいて図形が見えるか見えないか
 - アルファテスト, ステンシルテスト, デプステスト
- フレームバッファ上の処理
 - フラグメント単位の画像の合成処理
 - ロジックオペレーション, ブレンドオペレーション
- フレームバッファへの描き込み

フラグメント処理のパイプライン



フォグ処理はフレグメントの陰影付けに含まれる

フラグメントの陰影付け

- 頂点パラメータの補間値の取得
 - 深度値 (デプス値)
 - 色・陰影
 - テクスチャ座標値
 - (座標値)
 - (法線ベクトル)
- テクスチャのサンプリング
- 画素の色の決定
 - 頂点色の補間値とテクスチャ色の合成
 - (法線ベクトルの補間値を用いた陰影計算)

フラグメント単位の処理

- フォグ
 - 霧の効果, 大気遠近法
- アルファテスト
 - アルファ値にもとづく型抜き
- ステンシルテスト
 - ステンシルバッファによる型抜き
- デプステスト
 - デプスバッファによる可視判定
- ブレンド処理
 - カラーバッファ上での色の合成
- ロジック処理
 - スクリーン上での論理演算

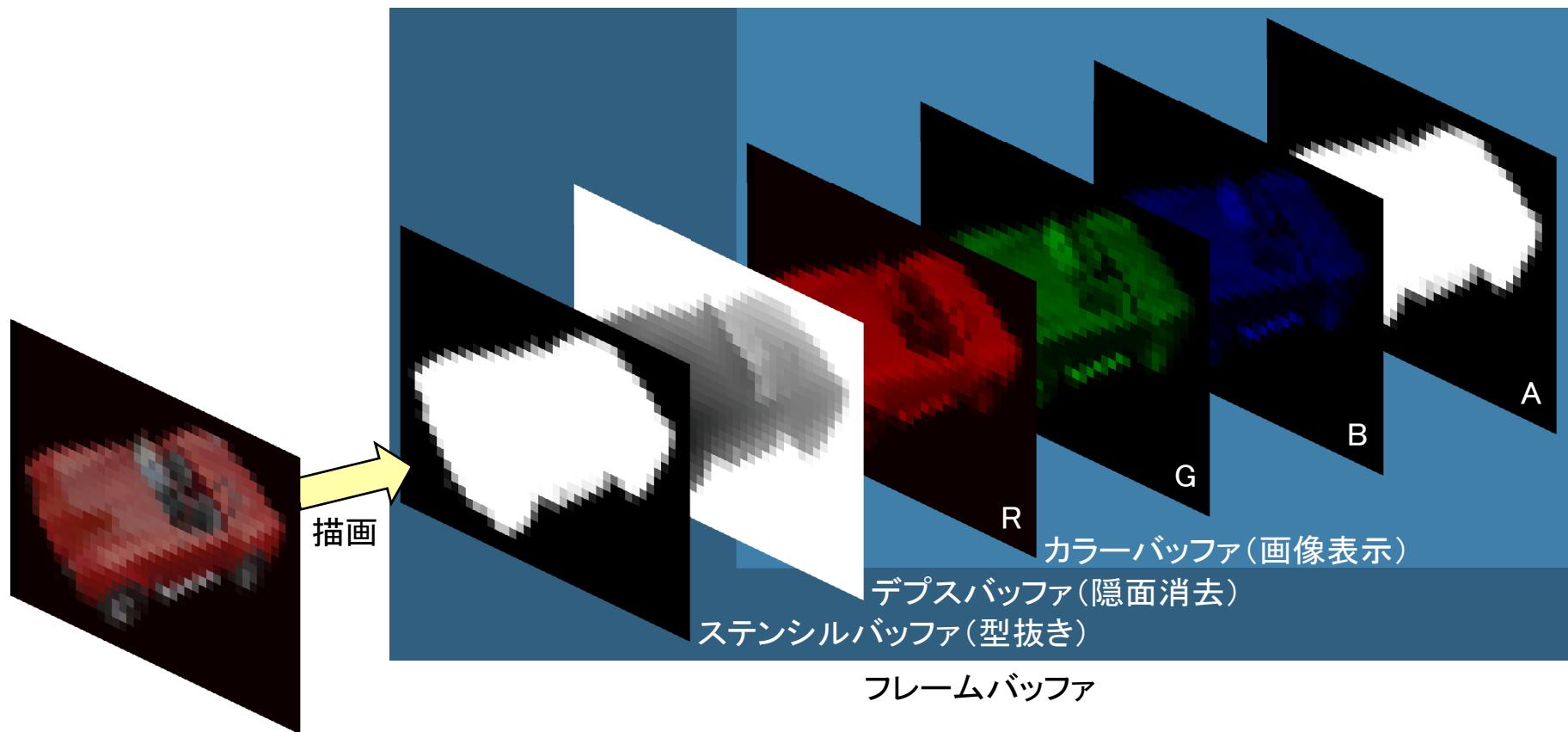
半透明等



フレームバッファ

画像を保持する

フレームバッファのイメージ



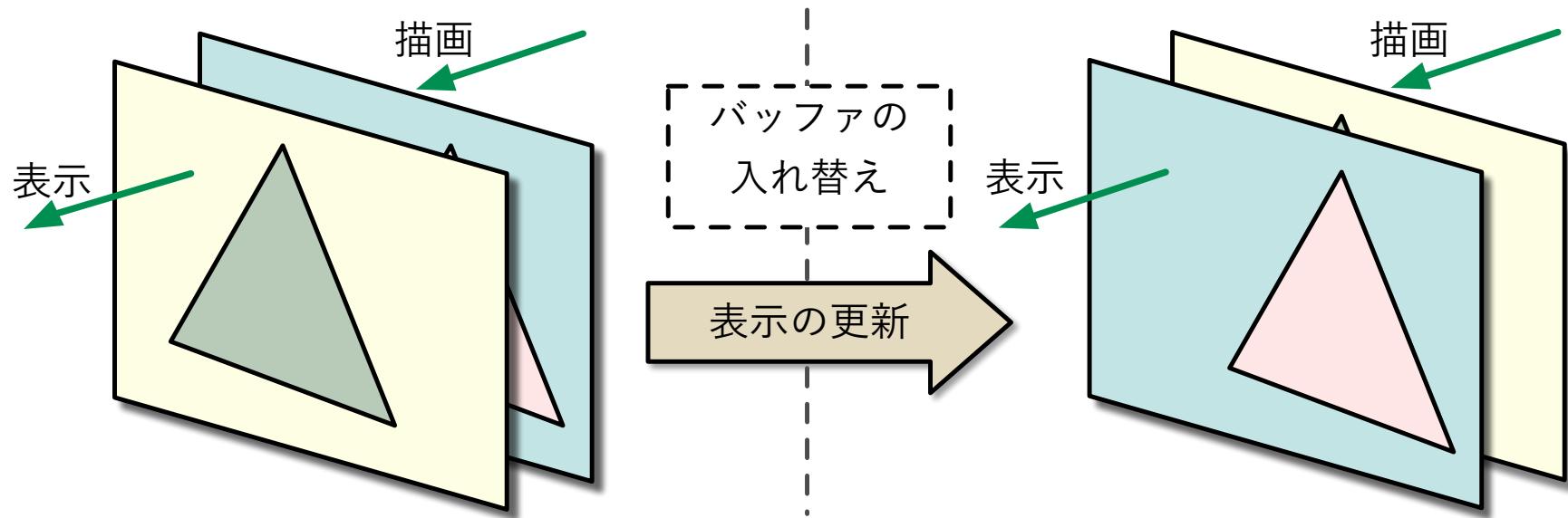
フレームバッファ

- ・様々なバッファの集合体
 - ・カラーバッファ
 - ・フロントバッファ, バックバッファ (ダブルバッファリングの場合)
 - ・デプスバッファ
 - ・隠面消去処理を行う
 - ・ステンシルバッファ
 - ・表示図形の「型抜き」を行う
 - ・アクチュムレーションバッファ
 - ・カラーバッファの内容を累積することができる (古い機能, FBO で代替)
- ・FBO (Frame Buffer Object)
 - ・ユーザが任意のバッファを組み合わせて構成する
 - ・各バッファに何を格納するかはユーザが決める

カラーバッファ

- レンダリング結果の画像を格納するバッファ
 - この内容が画面に表示される
- **ダブルバッファリング**
 - 描画過程が人に見えないようにする
 - フロントバッファ
 - 画面に表示されているバッファ
 - バックバッファ
 - 実際に描画を行うバッファ
- バッファの入れ替え (Swap Buffers)
 - バックバッファへの描画が完了したら
 - ディスプレイの表示更新のタイミング（垂直帰線消去時）で
 - フロントバッファとバックバッファを入れ替える

ダブルバッファリング



デプスバッファ (Zバッファ)

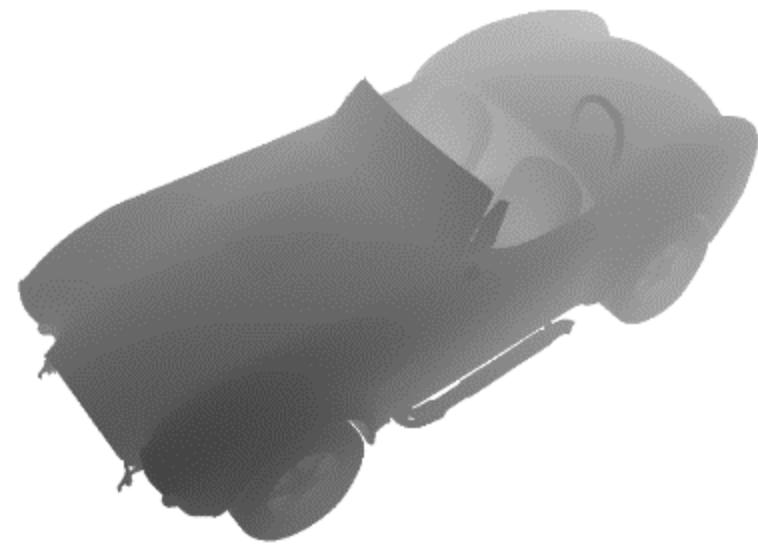
- **隠面消去処理**に用いる
 - カメラから見える（他の図形より手前にある）ものを表示する
- カラーバッファと同じサイズをもつ
 - カメラに最も近い図形の**深度（デプス）**を各画素に格納する
 - 描画しようとする図形の深度と既に格納されている深度を比較する
 - 深度が小さいほうの図形の色を表示し、その深度を格納する
- 図形を任意の順序で描画できる
 - 物体の交差を表現できる
 - 半透明の図形は視点から遠いものから順に描く必要がある

カラーバッファとデプスバッファ

カラーバッファ



デプスバッファ

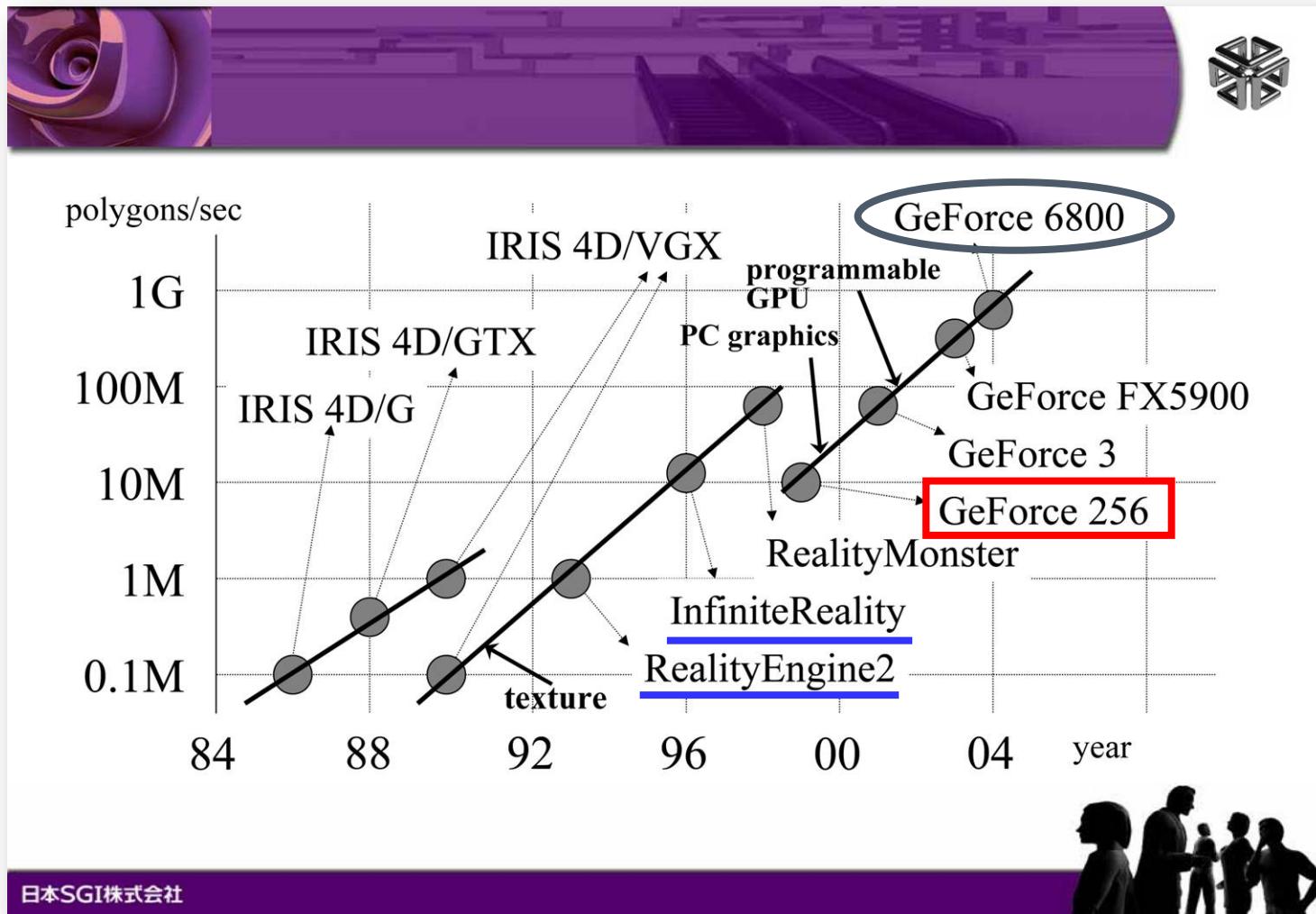


(明るいほど値が大きい = 遠い)

グラフィックスハードウェア の発展

GeForce 256

グラフィックスハードウェアの性能向上

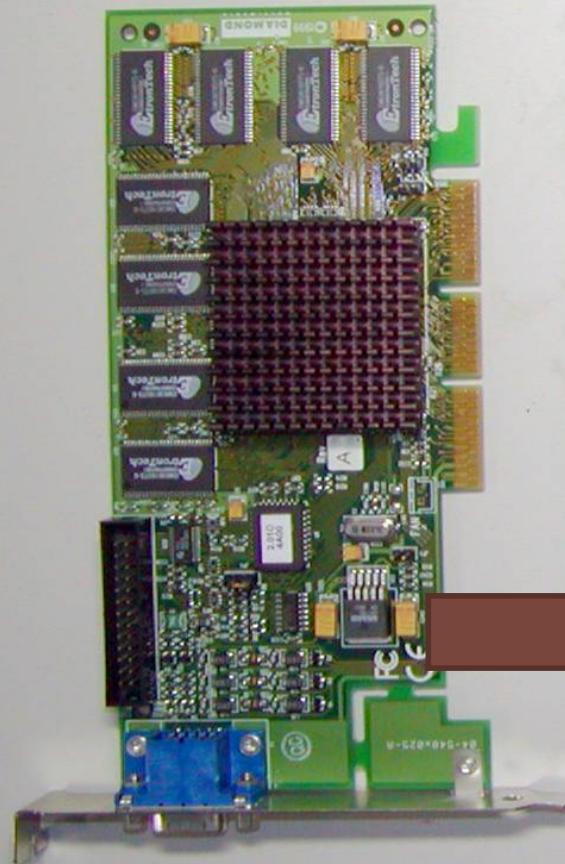


SGI Onyx



ビデオカード

NVIDIA RIVA TNT2
(ラスタライザがハードウェア)



NVIDIA GeForce 256
(ハードウェア T & L)



3DLabs WildCat VP870
(プログラマブルシェーダ)



冷却ファンが大型化

GeForce 256 が達成したこと

- 座標変換や陰影計算を実行するハードウェアを備えた
 - **Hardware T & L** (Transform and Lighting)
 - これらは**実数計算**が中心
 - 以前はアプリケーションステージで行われていた
 - それまでのグラフィックスアクセラレータはラスタライズのみ
- **GPU** (Graphics Processing Unit) と名づけられた
 - グラフィックス専用の計算を CPU の代わりに行う
 - ちなみに ATI (今の AMD) は VPU (Video Processing Unit) と名付けたものの普及しなかった

GeForce 256 以降

- ・パイプラインのハードウェアの設定を変更可能にした
 - ・固定機能パイプラインの設定を切り替えて多様な処理に対応する
 - ・ハードウェアが複雑になる
 - ・設定が複雑になる
- ・パイプラインのハードウェアをプログラム可能にした
 - ・プログラマブルシェーダ
 - ・さらに処理の自由度を進めた
 - ・開発者が独自のアルゴリズムを実装可能
 - ・高度にプログラマブルな画像処理装置
- ・図形表示以外の汎用の数値計算にも対応した
 - ・GPGPU (General Purpose GPU)
 - ・スーパーコンピュータや人工知能にも用いられている

新しい GPU

NVIDIA TITAN RTX



(298,000円, 280W)

AMD Radeon VII



(90,000～100,000円, 300W)

高い
でかい
電気を食う

新しい GPU の性能

Nvidia

GeForce 6800

▲
49
32
▼



VS

About

Release date ≤ Q2 2014.

XT 6800 GS GT

Nvidia vs AMD - Top 5 Games ☺

<http://gpu.userbenchmark.com/>

Nvidia

Titan RTX

▲
75
1,345
▼



¥ 275,047

Release date ≈ Q4 2018.

Titan Black X Pascal Xp V RTX

Nvidia vs AMD - Top 5 Games ☺

5,475倍以上

+547,402%

新しい GPU の性能

Nvidia

GeForce 6800

▲
49
32
▼



VS

About

Release date ≤ Q2 2014.

XT 6800 GS GT

Nvidia vs AMD - Top 5 Games

<http://gpu.userbenchmark.com/>

AMD

Radeon-VII

▲
76
848
▼



¥ 94,420

Release date: Q1 2019.

Vega-56 Vega-64 Vega-64LC Radeon-VII

3,228倍以上

Nvidia vs AMD - Top 5 Games

+322,798%

宿題

- OpenGL の開発環境を整備してください
 - 宿題のひな形は GitHub にあります
 - <https://github.com/tokoik/ggsample01>
 - GitHub にアカウントを作つて fork してもらって構いません
 - 講義の Web ページを参照してください
 - <https://www.wakayama-u.ac.jp/~tokoi/lecture/gg/>
 - 宿題プログラムの作成に必要な環境
 - Linux / Windows / macOS に対応しています
 - OpenGL のバージョン 4.1 以降が実行できる環境が必要です
- **自分の氏名**を記入した "学生番号.txt" (例: 60234567.txt) というテキストファイルを作つて **アップロード**してください
 - アップロード先
 - <https://www.wakayama-u.ac.jp/~tokoi/lecture/gg/upload/>

宿題のビルドと実行

- Windows 10 (Visual Studio 2017 以降)  [ggsample01.sln](#)
 - ソリューションファイル ggsample01.sln を開く
 - [デバッグ]→[デバッグの開始] (または [F5] キー, ▶)
- macOS 10.15 (Xcode 11 以降)  [ggsample01.xcodeproj](#)
 - プロジェクトファイル ggsample01.xcodeproj を開く
 - [Product]→[Run] (または ⌘R, ▶)
- Linux Mint 18.3 (gcc 5.4.0 以降)
 - make && ./ggsample01
 - 自前 Linux の場合は glfw をインストールしてください
 - sudo apt-get install libglfw3-dev (Linux Mint, Ubuntu 等)
 - Distribution に合わせて適宜 Makefile を編集してください