

①GPUプログラミングの実際

床井浩平

こんなことを話します

- CPU と GPU (Graphics Processing Unit) のプログラミングの違い
- レンダリングパイプライン
- シェーダ
- ストリーミングプロセッシング
- グラフィックス API
 - OpenGL/Vulkan/METAL/DirectX12 というものがあるという話
- GPGPU (computing on Graphics Processing Units)
 - OpenCL/CUDA というものがあるという話

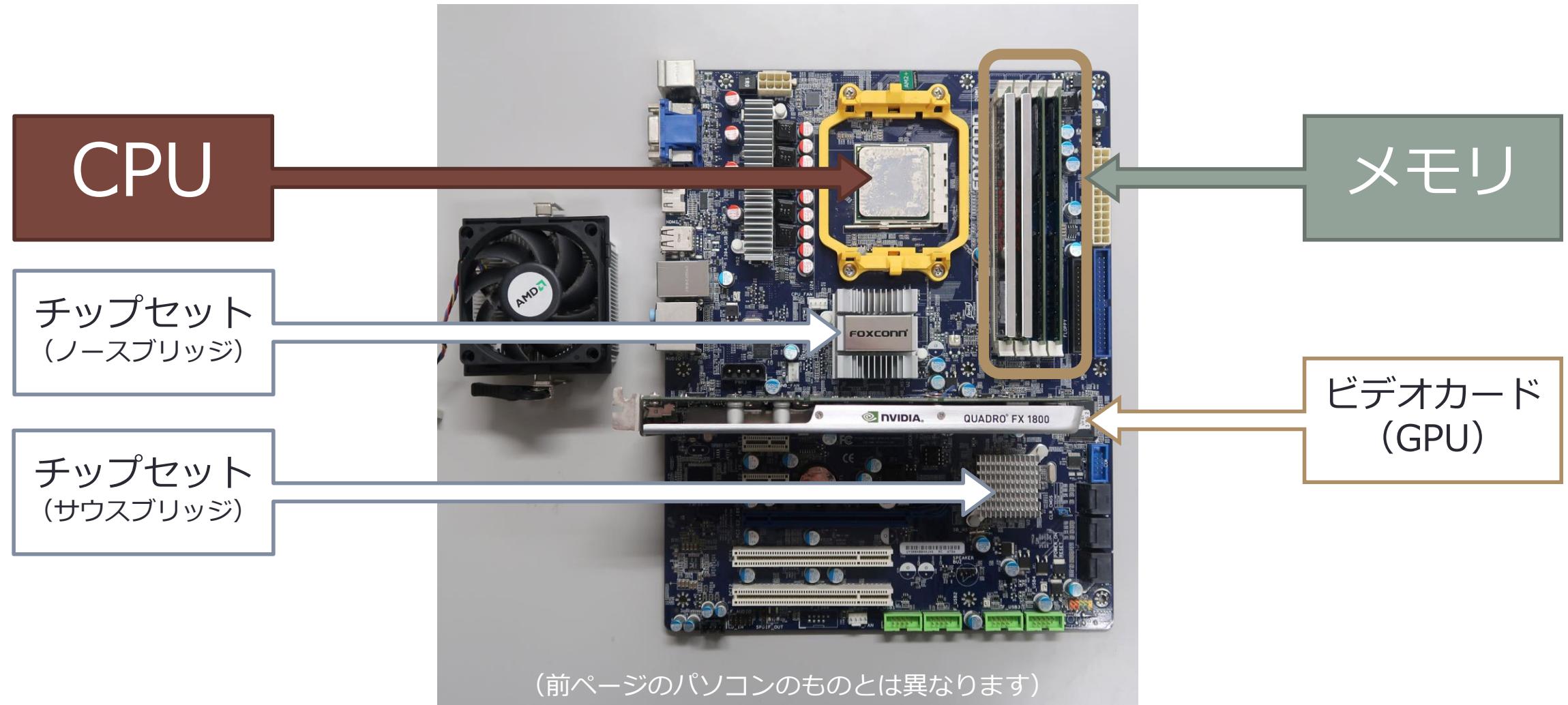
GPU プログラミング

汎用 CPU には GPU が付いている

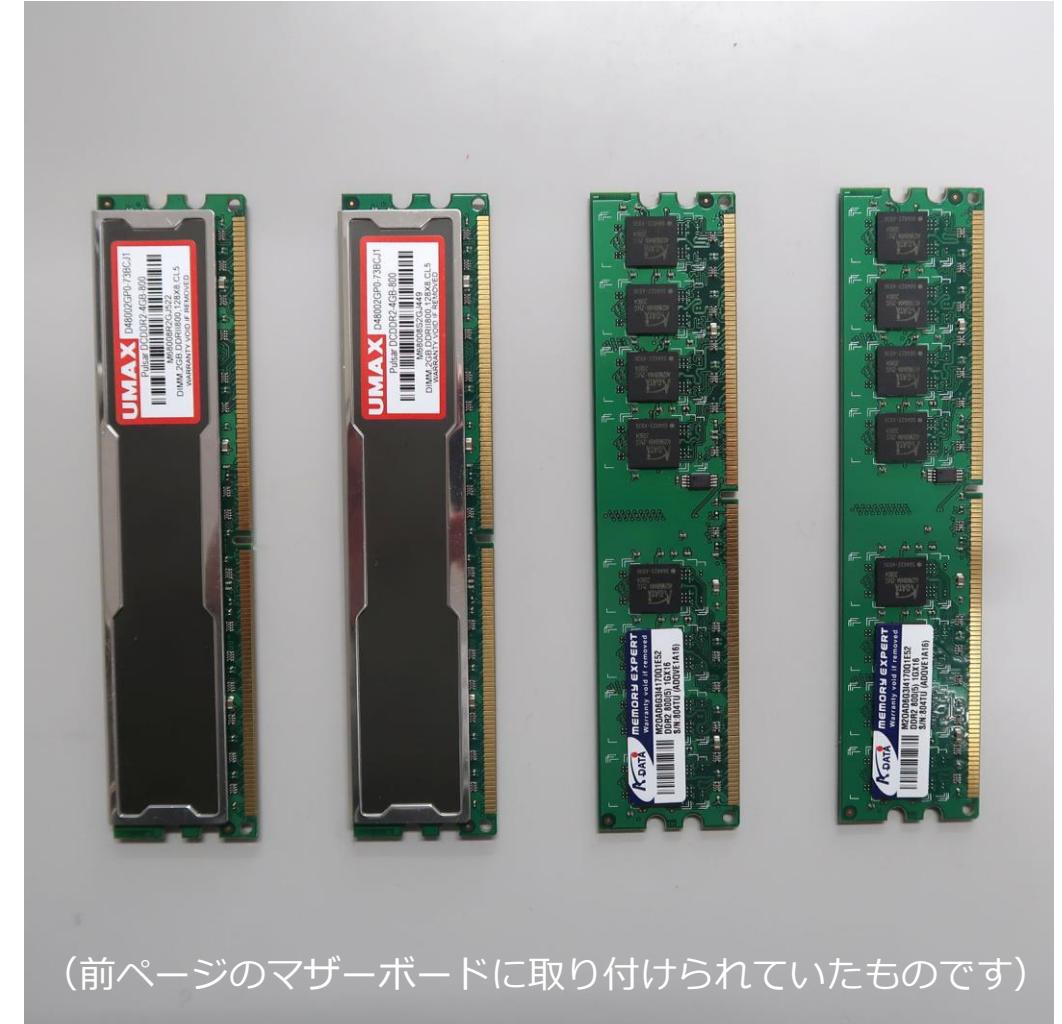
パソコンの外観と内部



マザーボード



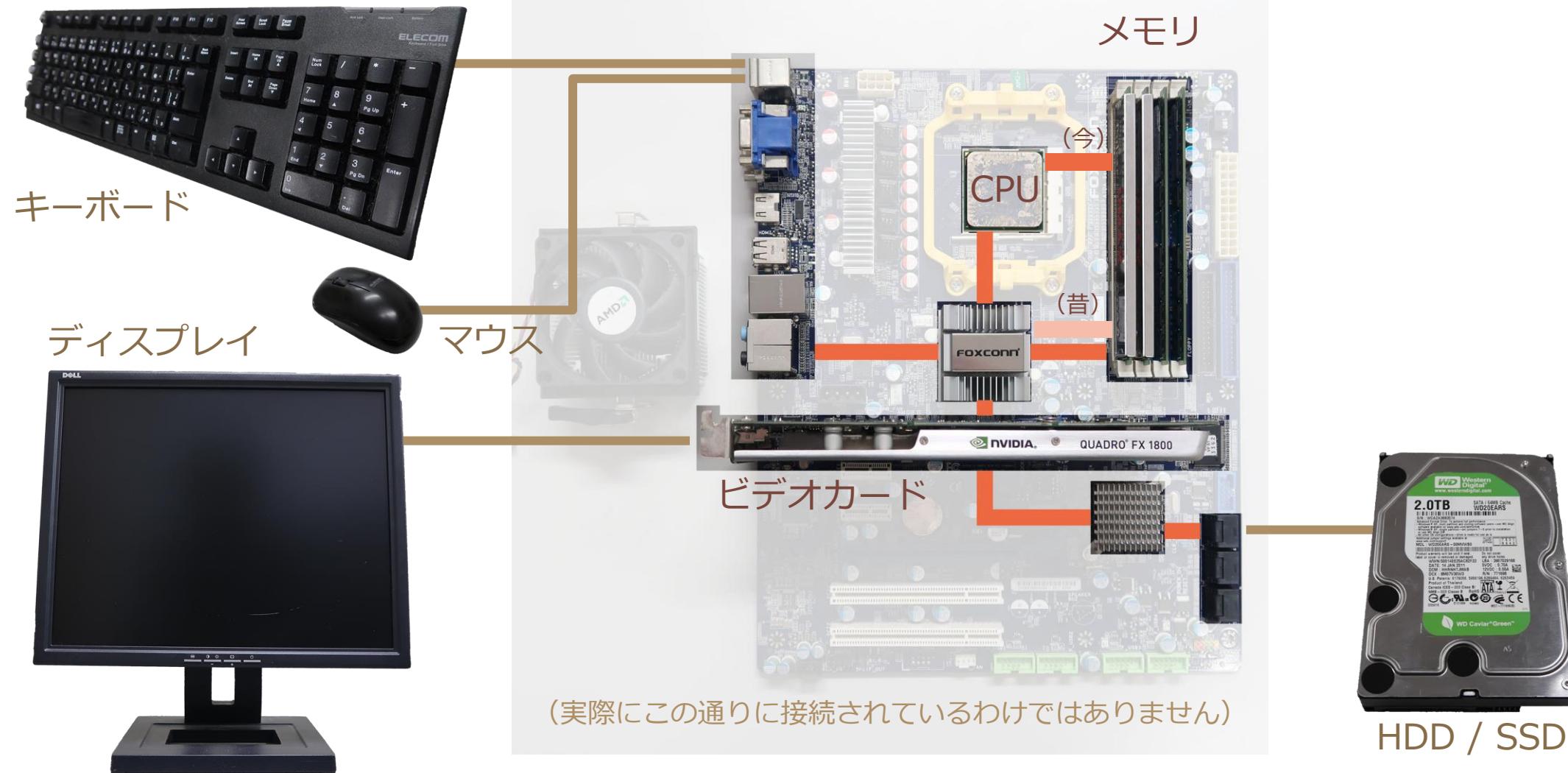
CPU とメモリ



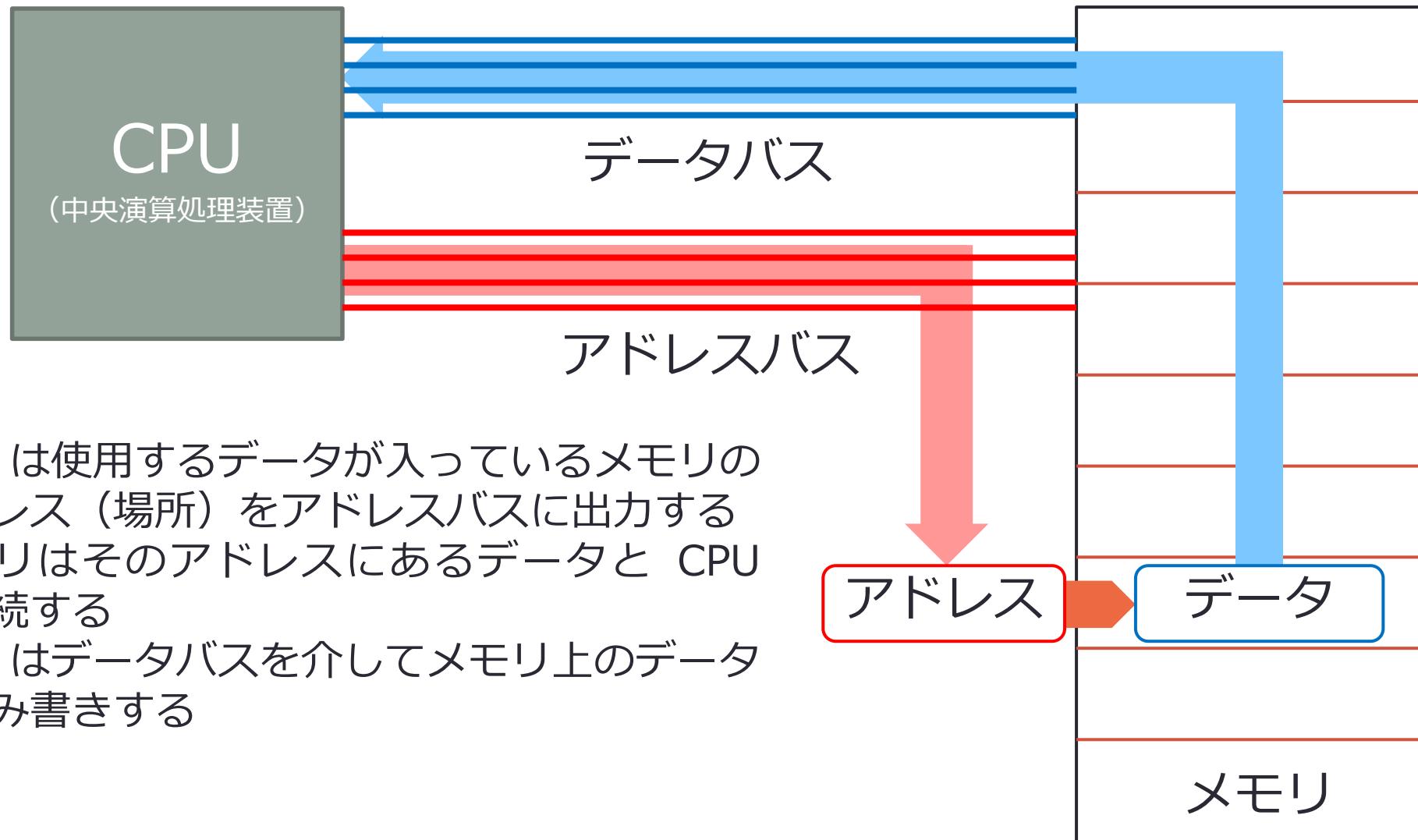
ビデオカード (GPU)



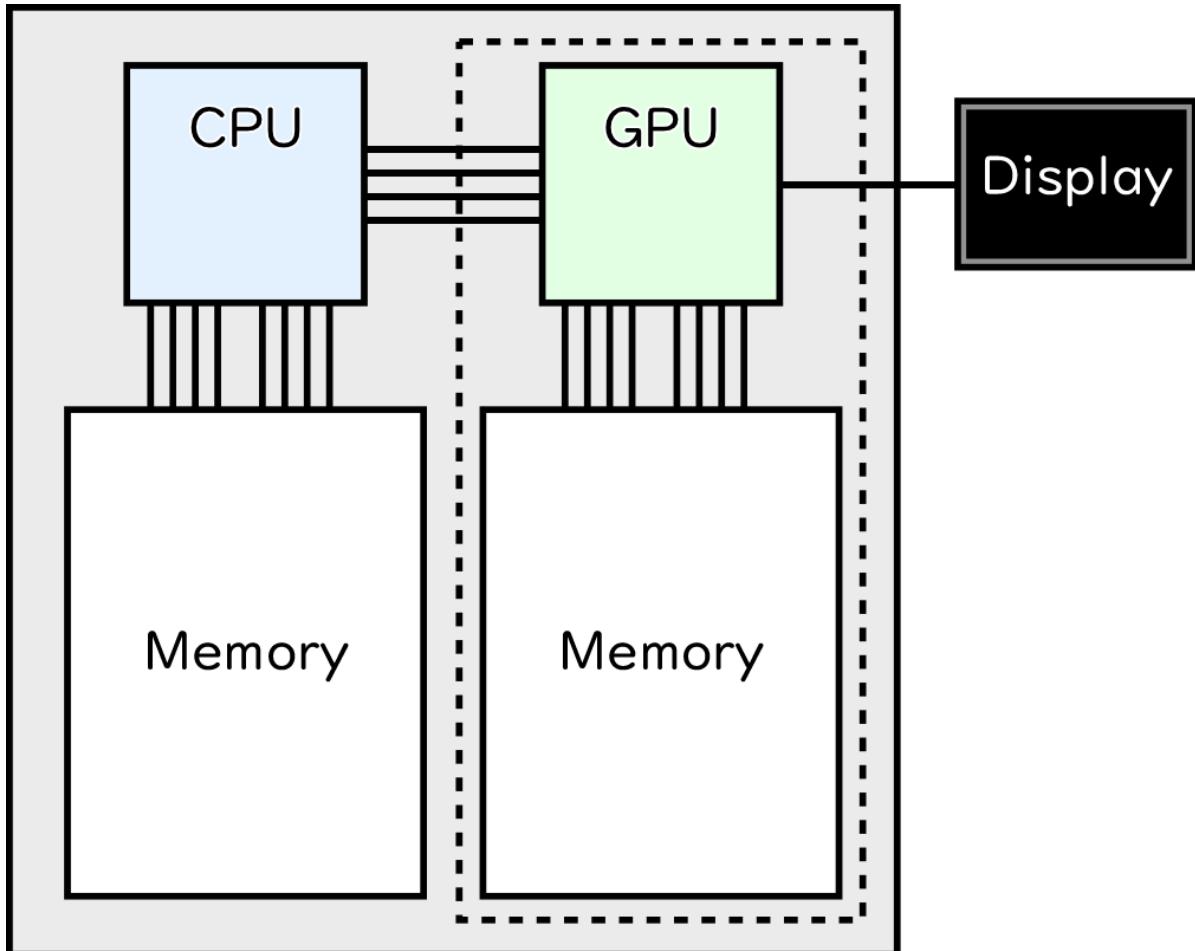
周辺機器との接続



CPU とメモリの関係

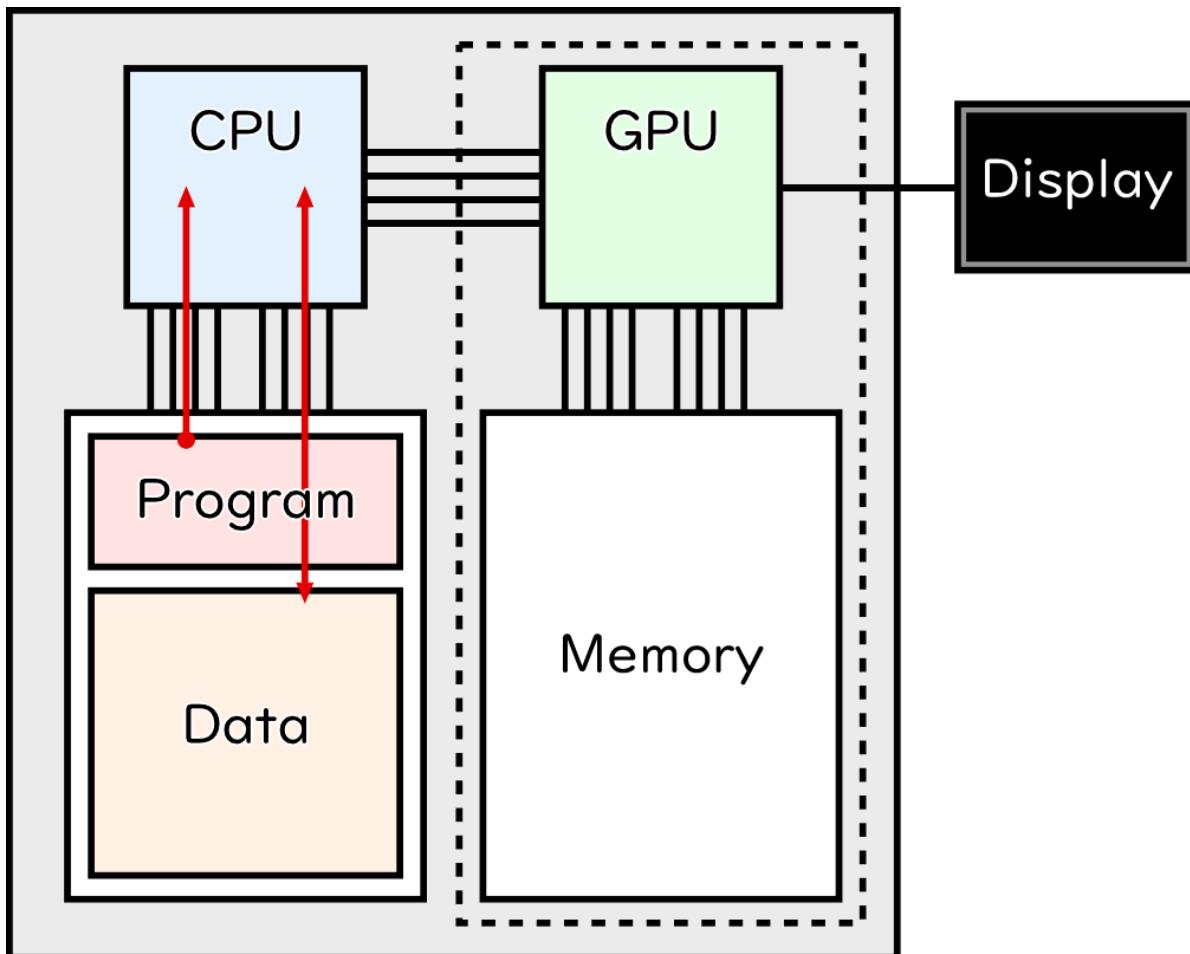


CPU と GPU



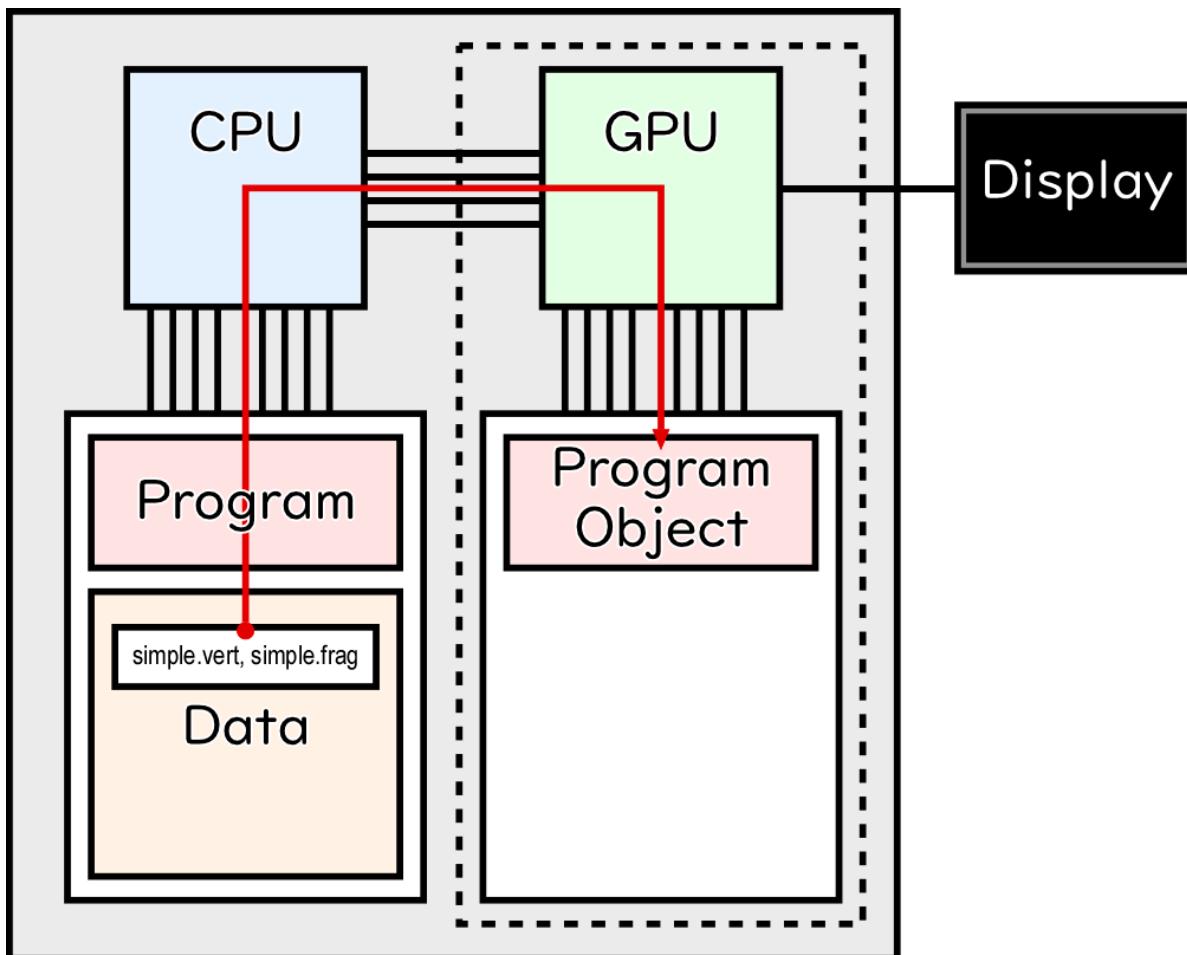
- GPU は CPU の I/O に接続されている
 - CPU と同じパッケージに入っている場合
 - Internal GPU (iGPU)
 - メモリは CPU のメインメモリと共有
 - PCI Express バスに装着されている場合
 - Discrete GPU (dGPU)
 - GPU 独自のメモリを持っている
 - CPU 側のメインメモリも使える
- 一般に dGPU の方が高性能
 - 発熱・消費電力の問題

CPU の動作



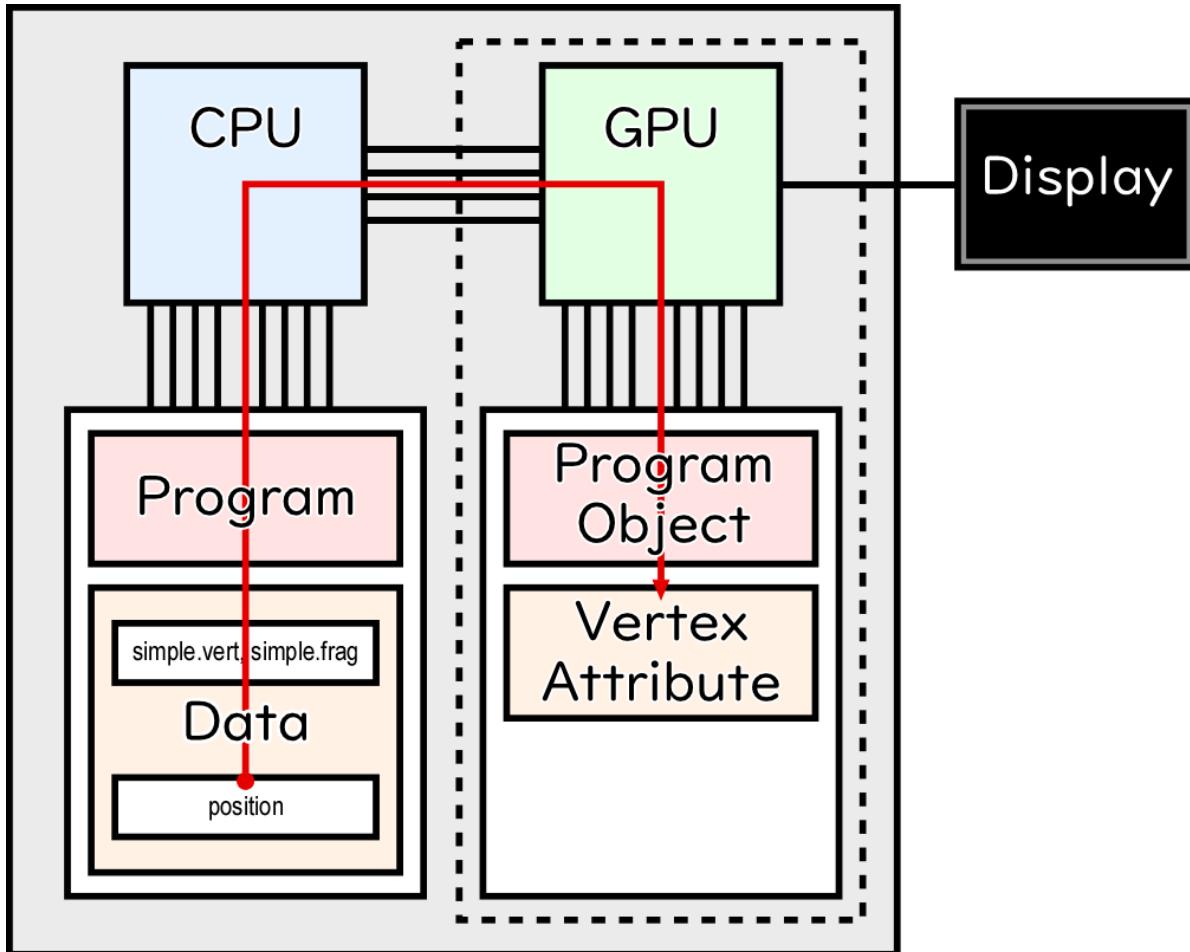
- CPU は…
 - メインメモリに格納されているプログラムを取り出す
 - プログラムの命令に従ってメインメモリに格納されているデータを取り出す
 - 取り出したデータを使って計算する
 - 計算結果をメインメモリに格納する

GPU のプログラム



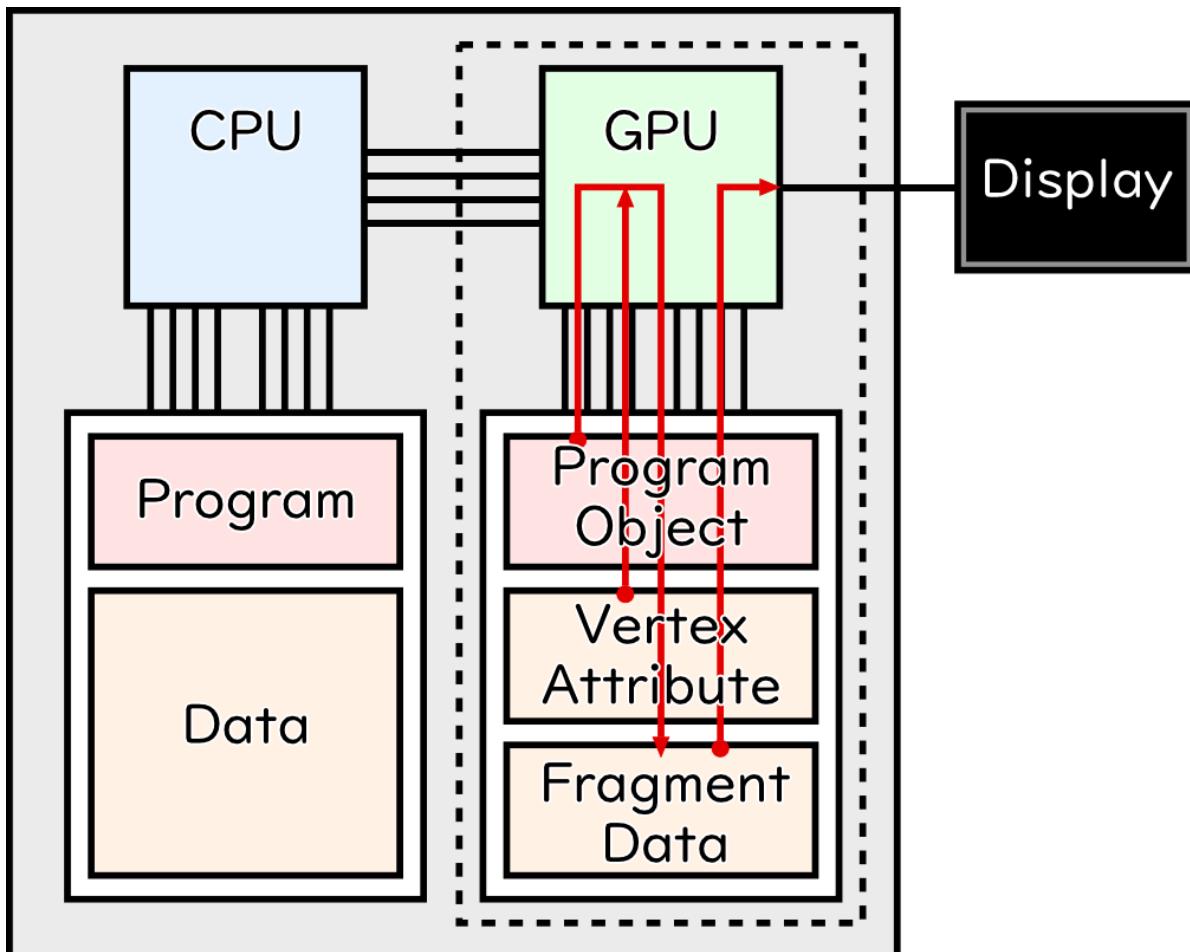
- GPU のプログラムは…
 - ソースプログラムを CPU のメインメモリ上に文字列として置く
 - CPU のプログラム（メインプログラム）実行時に GPU のソースプログラムをビルドしてプログラムオブジェクトを作成する
 - CPU はプログラムオブジェクトを GPU のメモリ上に転送する

GPU のデータ



- GPU のデータは…
 - CPU のメインメモリ上に準備する
 - 頂点属性
 - テクスチャ
 - CPU がメインメモリから GPU のメモリに転送する

GPU の動作



- GPU の動作は…

- CPU の描画コマンド（ドローコール）などによってプログラムオブジェクトが起動される
- プログラムオブジェクトは頂点属性やテクスチャをもとにレンダリングを行う
- レンダリング結果はフラグメントデータとして GPU のメモリ上に置かれる
- GPU はフラグメントデータから映像信号を生成してディスプレイに表示する

OpenGL / WebGL によるグラフィクスプログラミング

- GPU は CPU とは別にプログラミングする
 - プログラミング言語（シェーダ）も異なる
 - メモリも異なる（共有していても相互に直接参照できない）
- GPU のプログラムは GPU のメモリを直接操作できない
 - シェーダプロセッサからは GPU のメモリが見えない
 - ポインタのような仕組みはない
 - メモリにプロセッサが割り当てられる
- CPU から送られたデータは基本的に Read Only
 - 次段に送るだけで書き戻したりはできない（一方通行）
 - ストリームプロセッシング

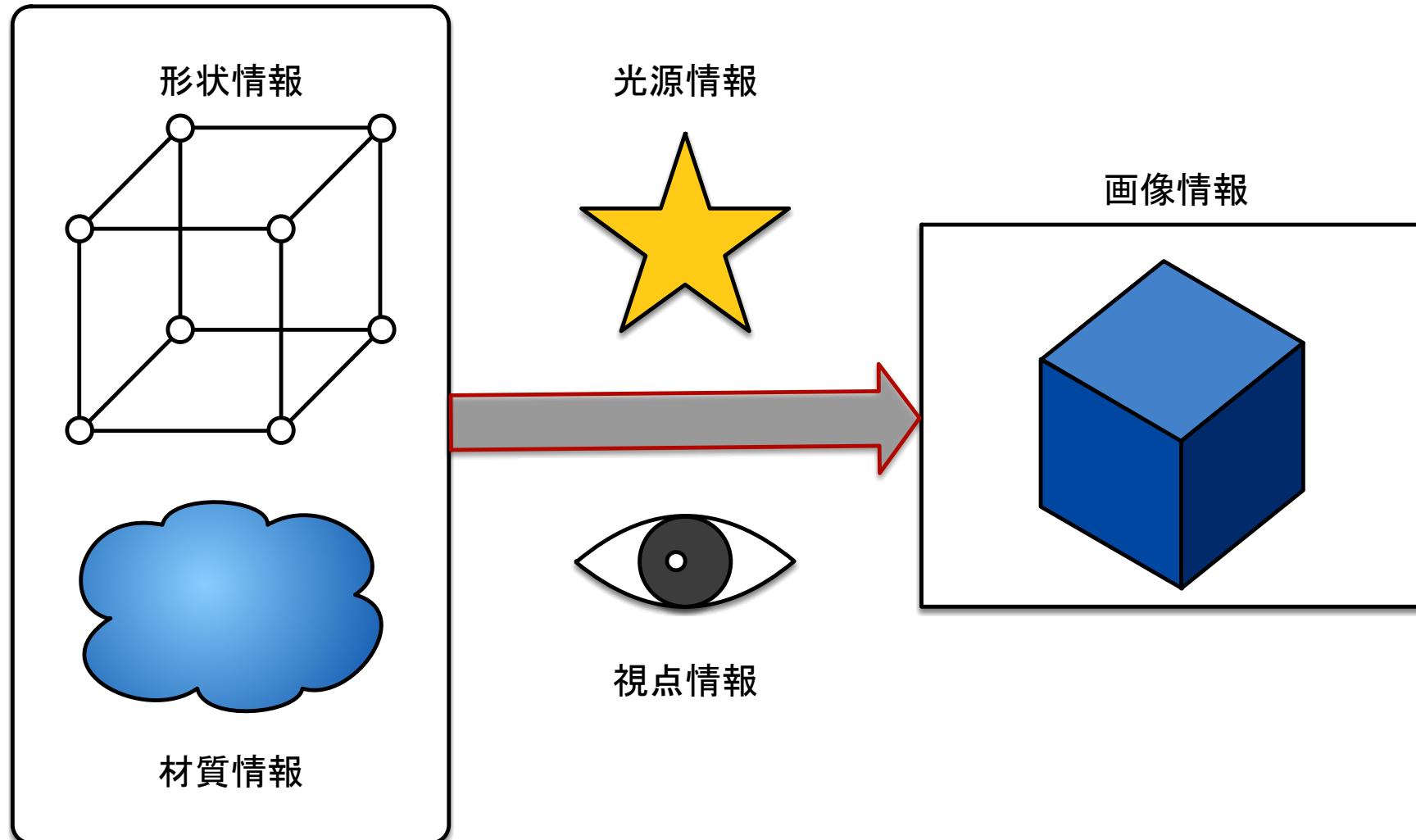
現在の CPU には GPU が付いている

- GPU は CPU と独立して並行動作する
 - ヘテロジニアスなマルチプロセッサ
- GPU の計算能力は非常に高い
 - CPU しか使わなければコンピュータの能力を活かせていない
- GPU のプログラミングモデルは CPU とは異なる
 - プログラミングの考え方を変えないといけない
- グラフィックス処理以外の汎用計算への応用 (GPGPU)
 - CUDA (NVIDIA)
 - OpenCL (Khronos)

レンダリング

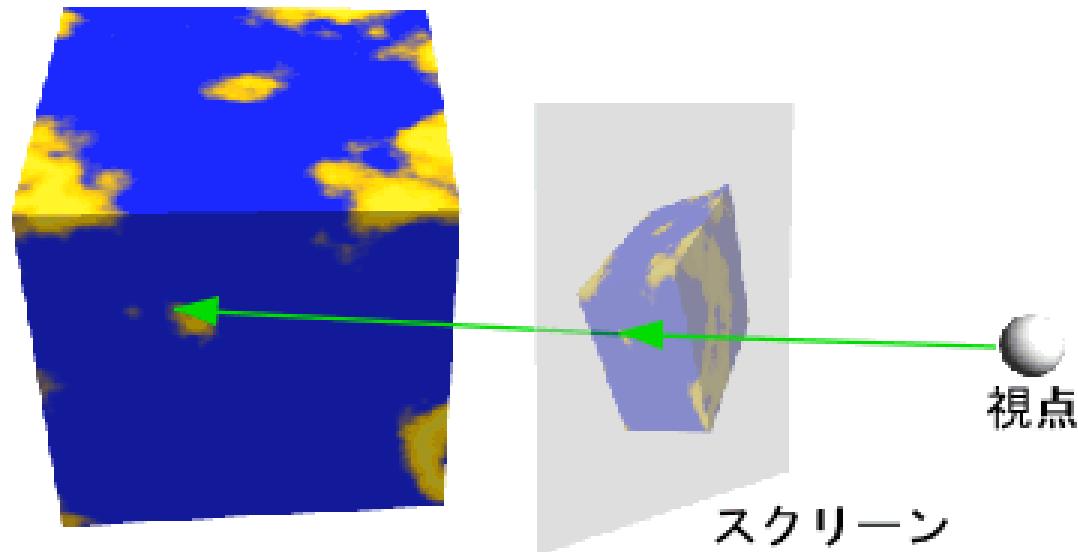
映像生成

レンダリングとは



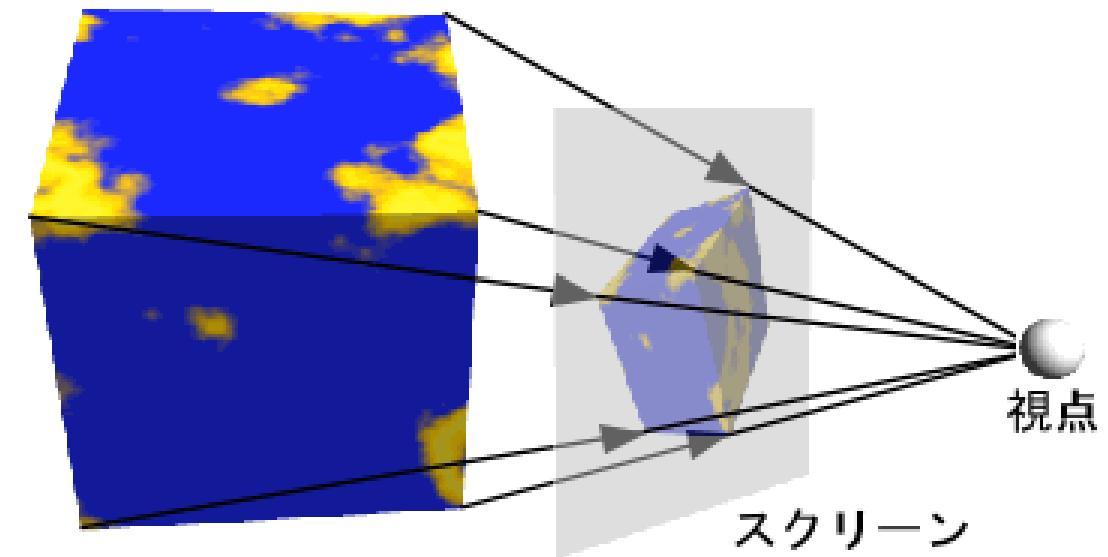
レンダリングの二つの方向

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

- ・複雑な光学現象の再現が容易
 - ・高品質な映像生成が行える
 - ・レイキャスティング法
 - ・レイトレーシング法
- ・画素ごとに独立した処理
 - ・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

- ・一般に複雑な光学現象の再現が困難
 - ・リアルさに欠ける場合がある
 - ・スキャンライン法
 - ・デプスバッファ法
- ・ハードウェアによる補間処理
 - ・コヒーレンシ（一貫性）が活用できる
 - ・リアルタイムレンダリングで用いられる

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、単位 **fps** (Frames Per Second)

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● 表示の遅れが操縦に影響する場合（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と正常に表示できない
 - フレームが欠落したり **ティアリング** (図形の裂け目) が発生したりする
- 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もある
 - **G-SYNC** (NVIDIA)、 **FreeSync** (AMD)
 - 対応ディスプレイ、 対応ビデオカードが必要

リアルタイムレンダリングの課題

- 通常、3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
 - 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）
 - ネットワーク、通信
- 画面表示以外の処理をこなしながら**規定時間内**に画面表示を完了する

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
 - ・3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

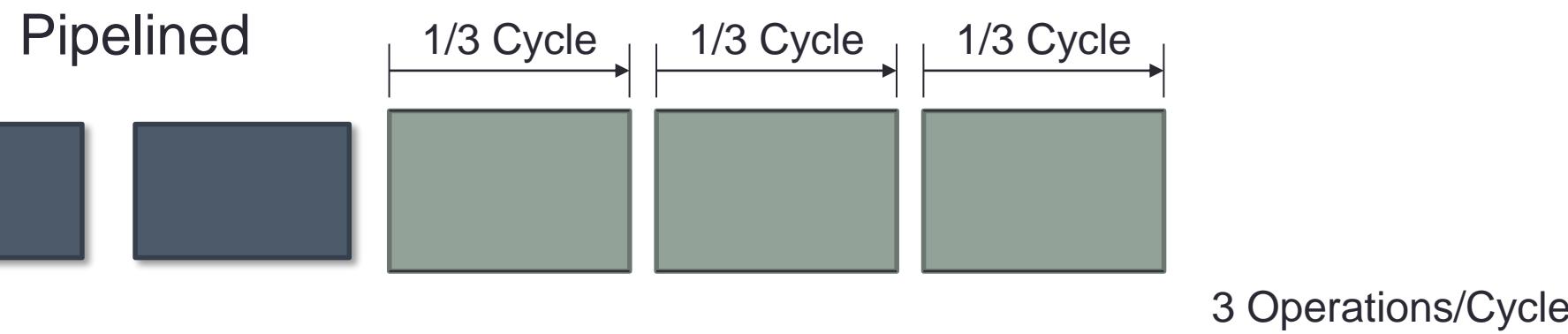
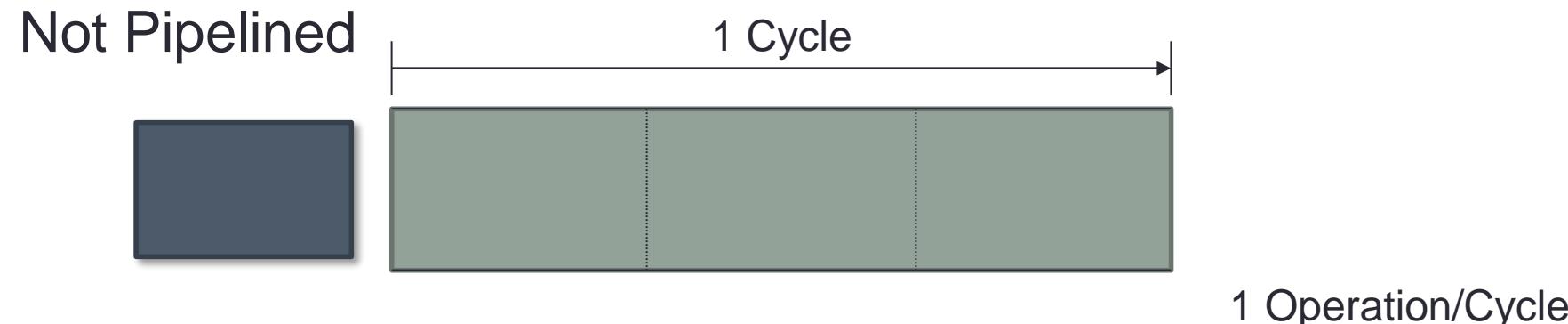
レンダリングパイプライン

グラフィックス処理の流れ

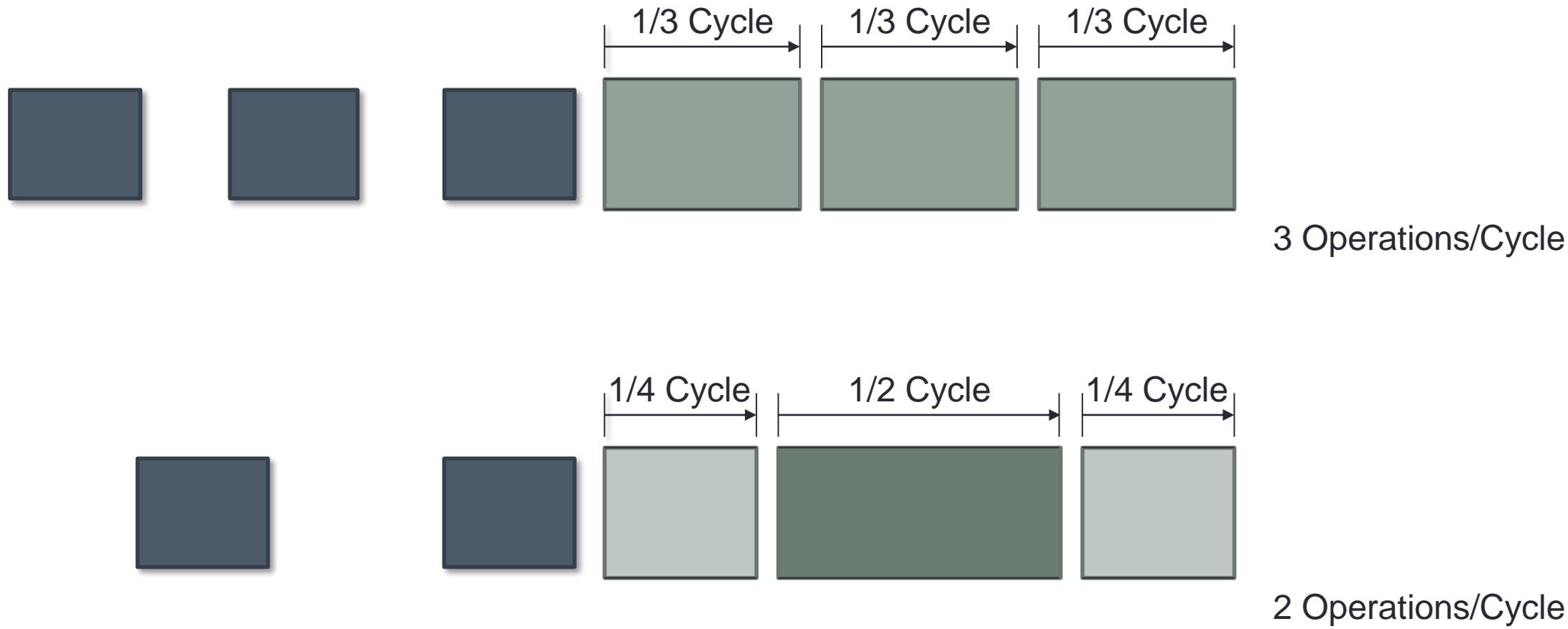
パイプライン処理

- ・ひとまとめりの処理を複数の段階（ステージ）に分割して実装する
 - ・ステージごとに処理を順送りする
- ・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すると
 - ・連続した処理の全体の処理速度を最大 n 倍スピードアップできる
 - ・ただし 1 個の処理の処理速度は変わらない
- ・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
 - ・このようなステージをボトルネックという
 - ・他のステージはボトルネックの処理が終わるまで待つ（ストールする）

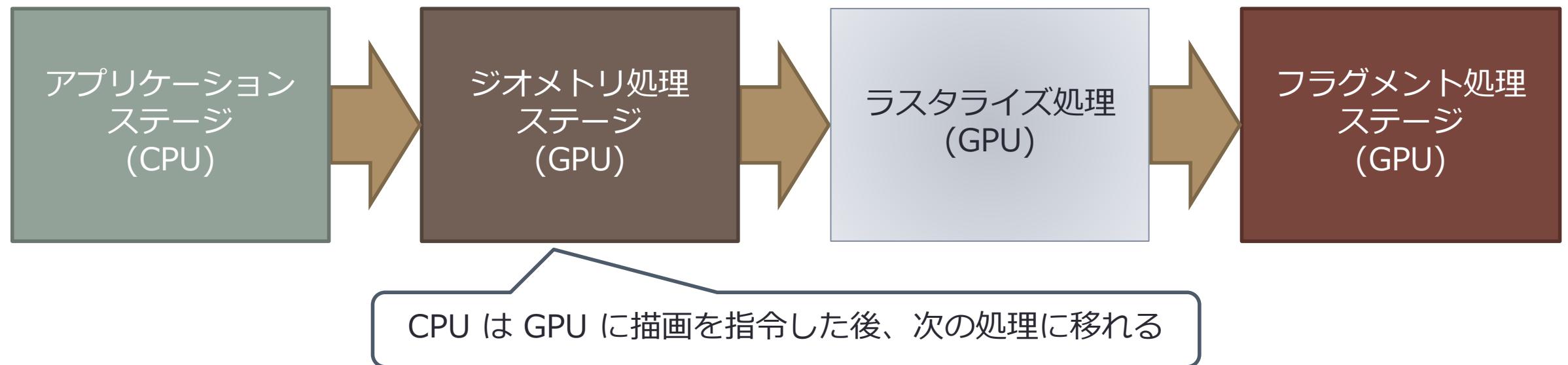
パイプライン処理による速度向上



ボトルネック



グラフィックス処理のパイプライン



(実際のハードウェアのステージ構成は実装依存)

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

- ・ソフトウェア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えることが可能
- ・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
 - ・形状変形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
 - ・テクスチャのアニメーション

他のステージでは実行できない
全ての種類の処理

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 (並列処理の単位) 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
 - ・今後のメニーコア (多数コア) 化への対応は課題

そのため Vulkan / METAL / DirectX 12 は機能を細かく分割した

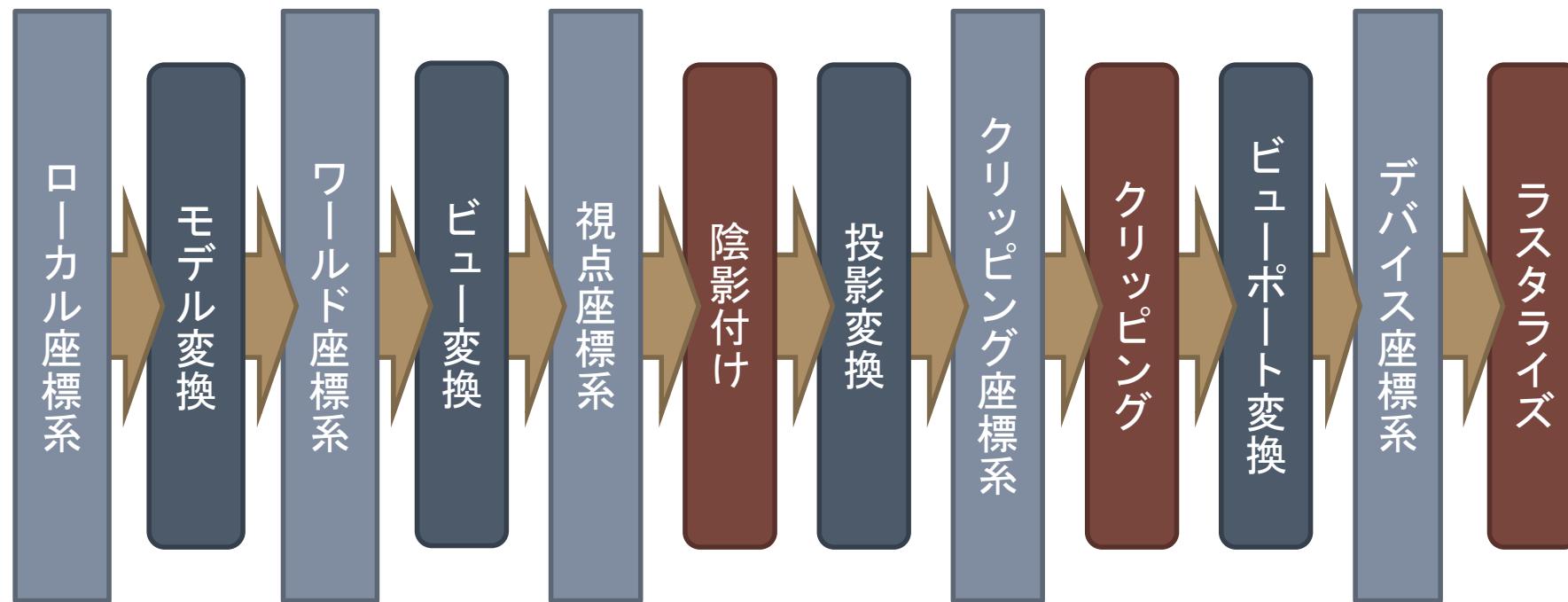
ジオメトリ処理ステージ

頂点単位の処理

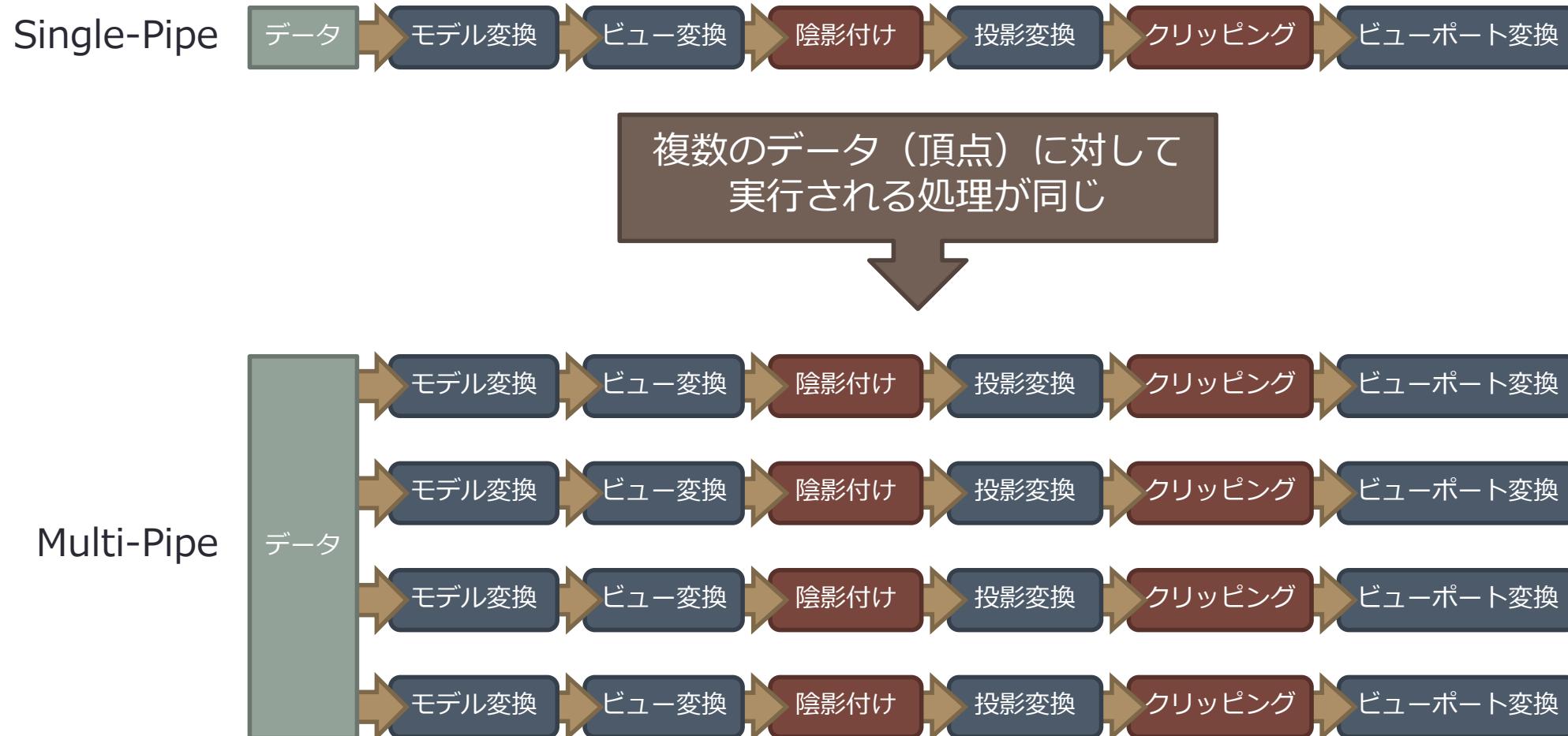
ジオメトリ処理

- **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



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

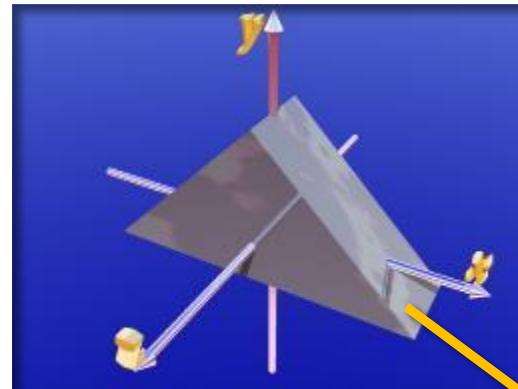
ビューボリューム

- ・視野となる空間

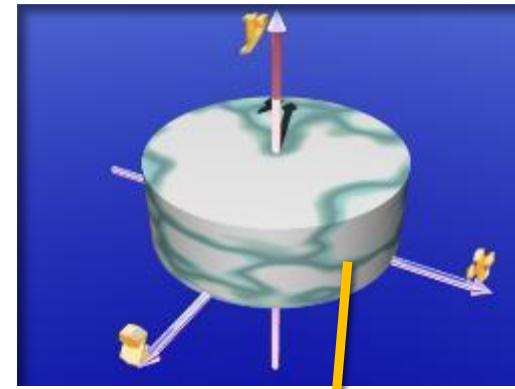
この二つはまとめて実行することが多い
(モデルビュー変換)

モデル変換

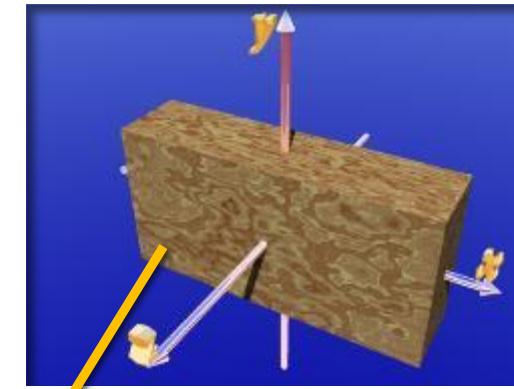
ローカル座標系



ローカル座標系

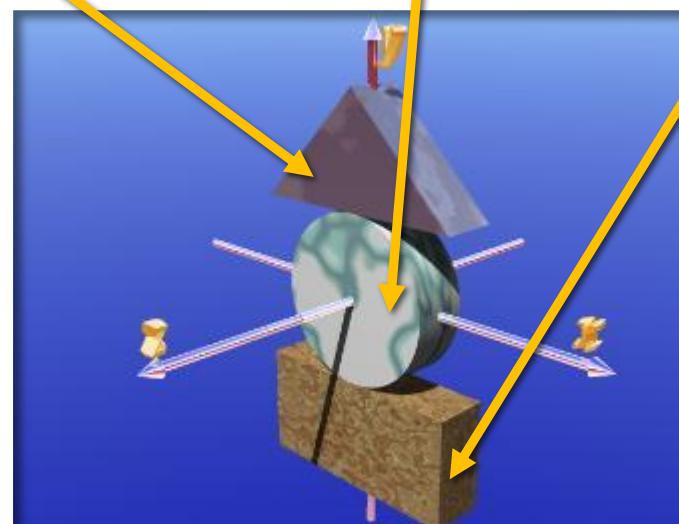


ローカル座標系



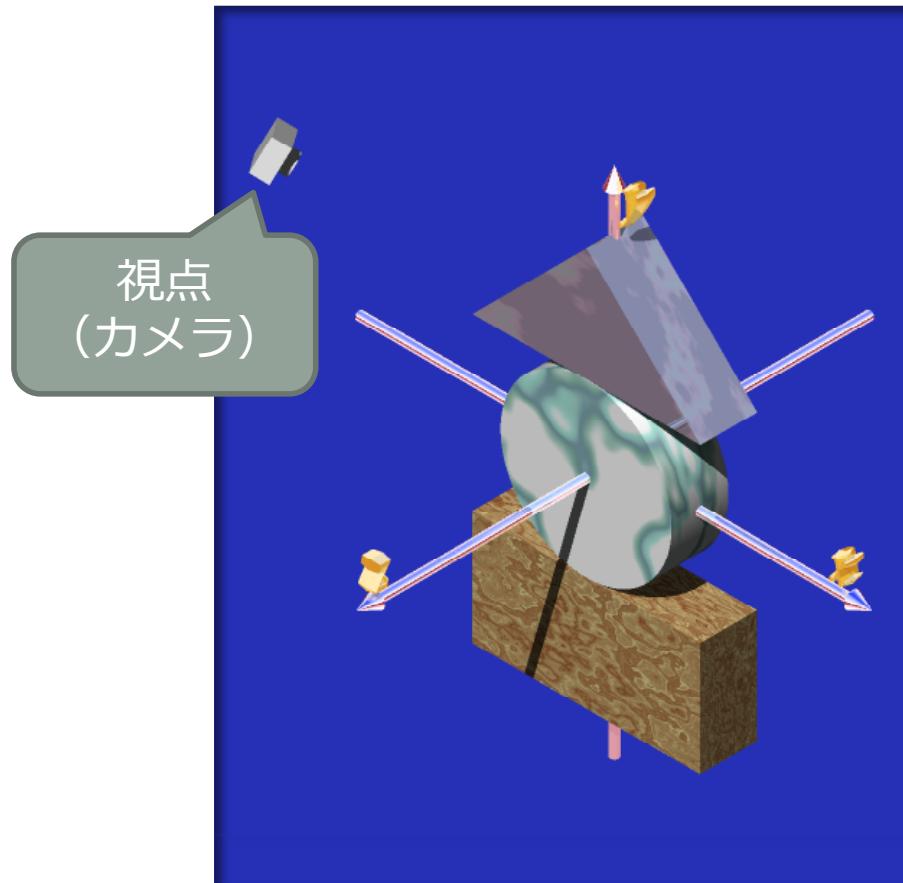
ワールド座標系上に配置

ワールド座標系



ビュー変換

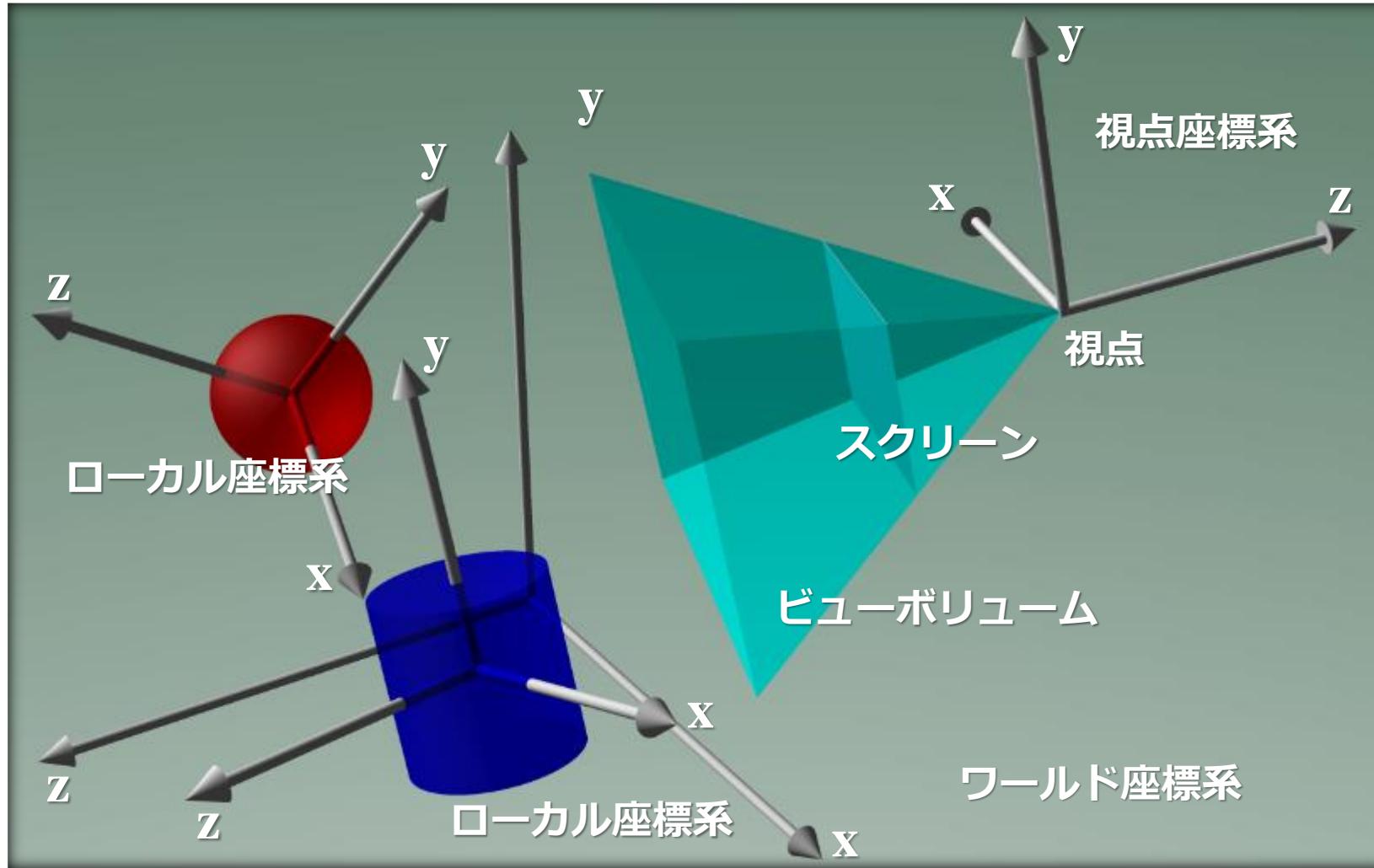
ワールド座標系



視点座標系



視点とスクリーンの関係

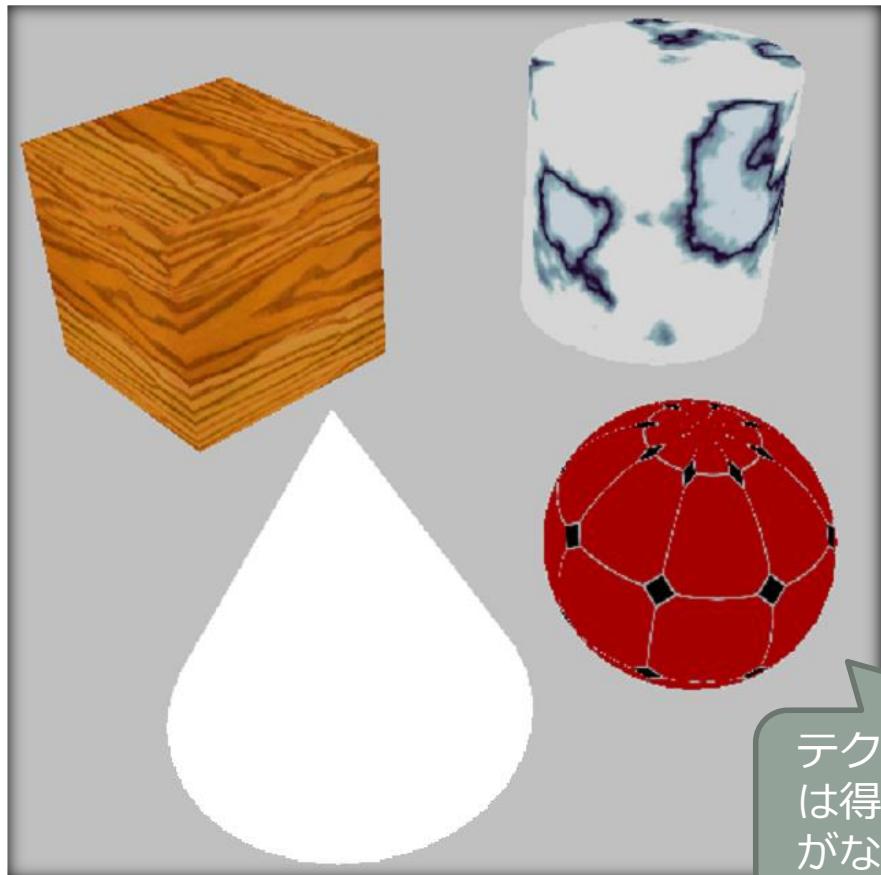


頂点の陰影付け

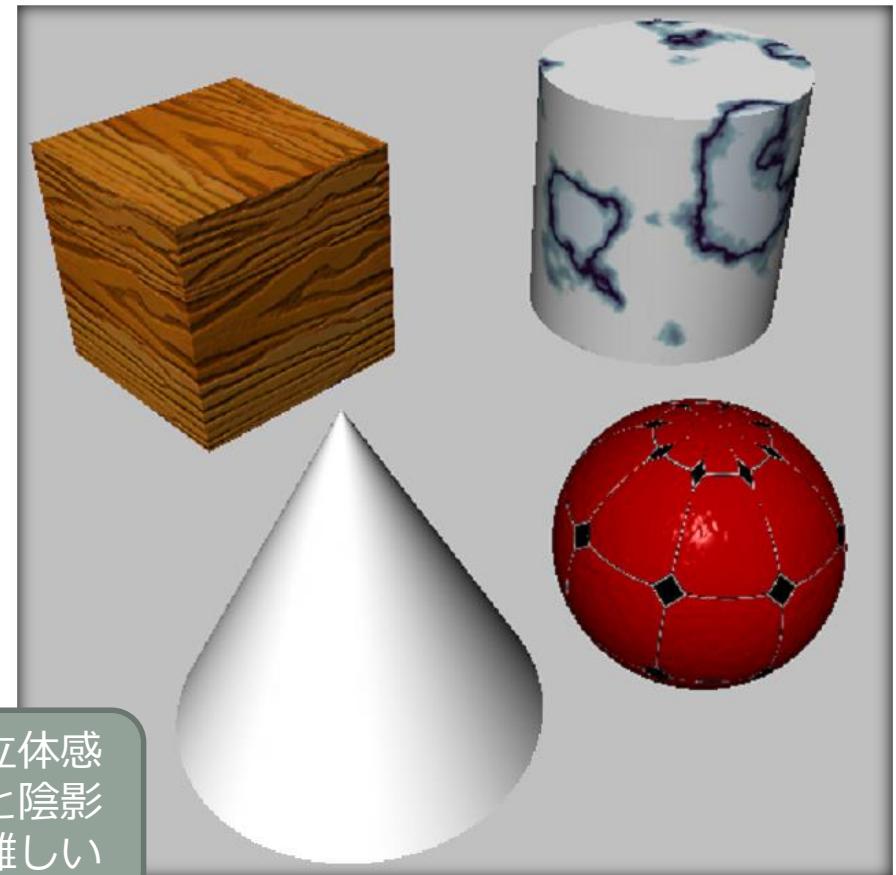
- ・**陰影**によってオブジェクトにリアルな外観を与える
 - ・シーンに1つ以上の**光源**を設定する
- ・**陰影付けの式**によってオブジェクトの**各頂点の色**を計算する
 - ・実世界の**光子**（フォトン）と物体表面との相互作用を近似する
 - ・実世界では光子は**光源から放出**され、**物体表面で反射・吸収**される
 - ・リアルタイムレンダリングでは、この計算に多くの時間を割けない
- ・本物の反射（映りこみ）や屈折、影（シャドウ）は含まない
 - ・これらは**擬似的手法**を用いて実現する場合が多い
 - ・プログラム可能 GPU によりかなり精密に計算で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

陰影付けの有無

陰影付けなし



陰影付けあり

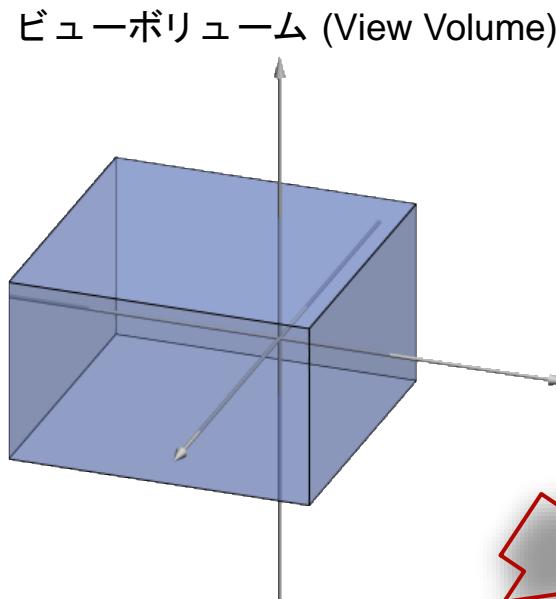


テクスチャによっても立体感
は得られるが、単色だと陰影
がなければ形の認識が難しい

投影変換

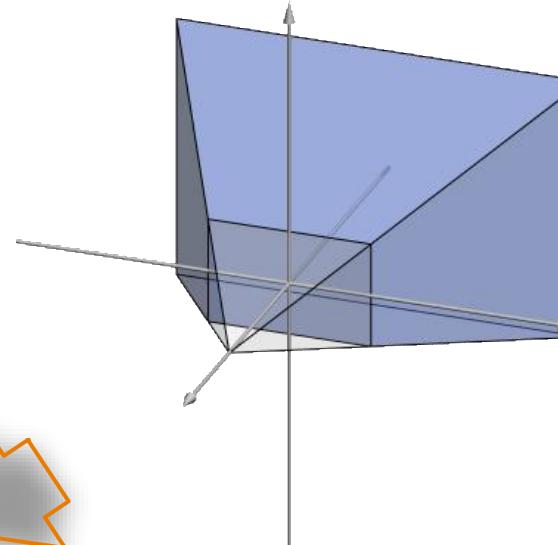
- ・ビューボリューム内の物体の座標値をクリッピング座標系 (Clipping Coordinate) 上の座標値に変換する
 - ・正規化デバイス座標系 (Normalized Device Coordinate, NDC)
 - ・ $(-1, -1, -1), (1, 1, 1)$ を対角の頂点とする x, y, z 軸に沿った立方体
 - ・標準ビューボリューム (Canonical View Volume) とも呼ばれる
- ・直交投影 (平行投影)
 - ・直方体のビューボリュームを用いる
- ・透視投影 (遠近投影)
 - ・錐台のビューボリューム (視錐台, View Frustum)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

ビューボリューム

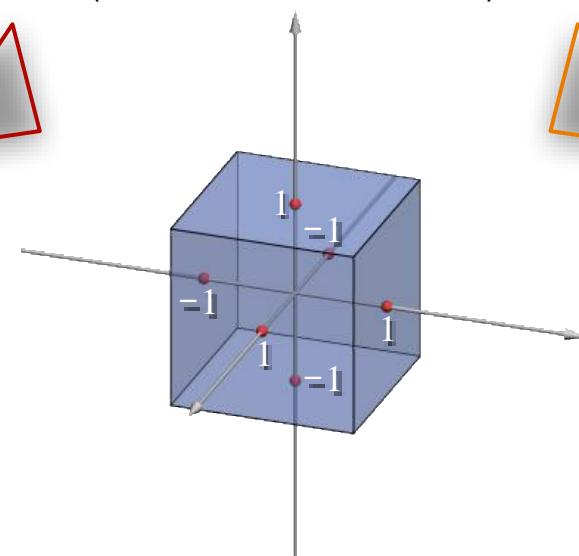


標準ビューボリューム
(Canonical View Volume)

視野錐台 (View Frustum)



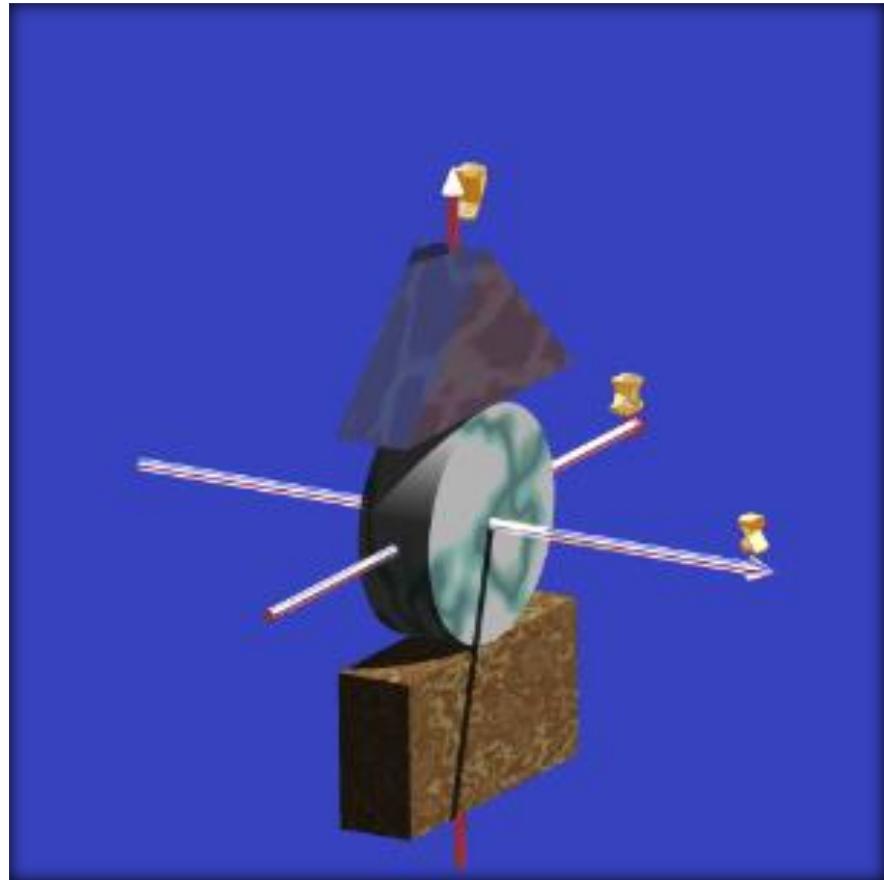
直交投影
(Orthographic Projection)



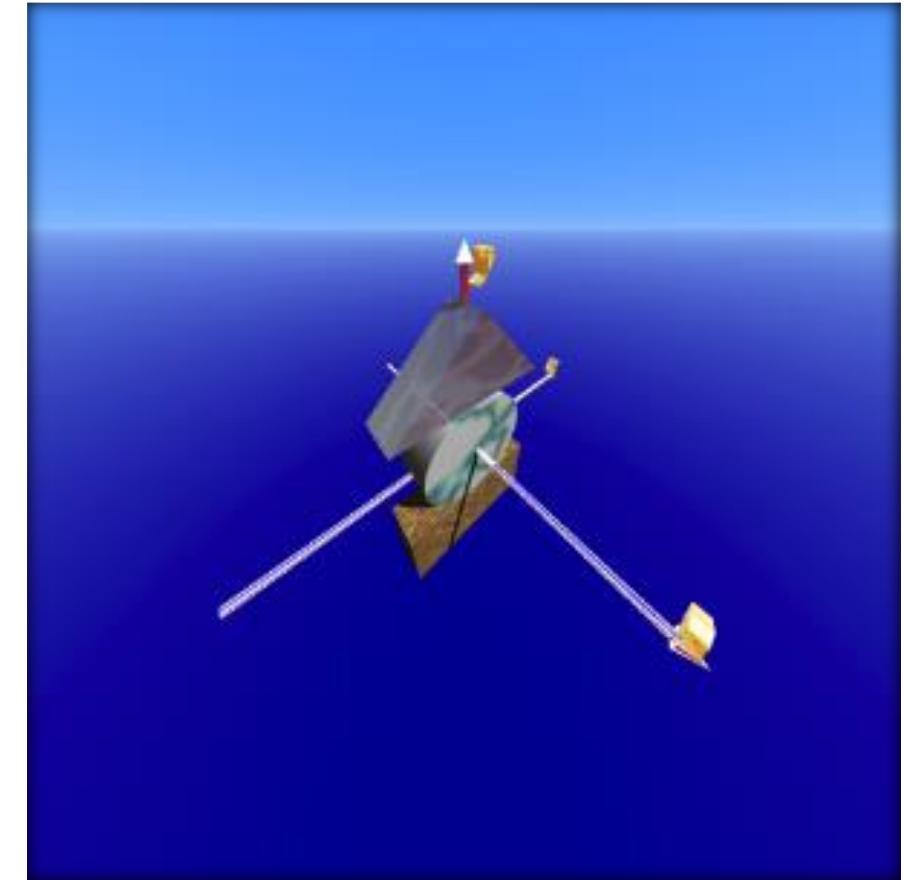
透視投影
(Perspective Projection)

直交投影と透視投影

直交投影

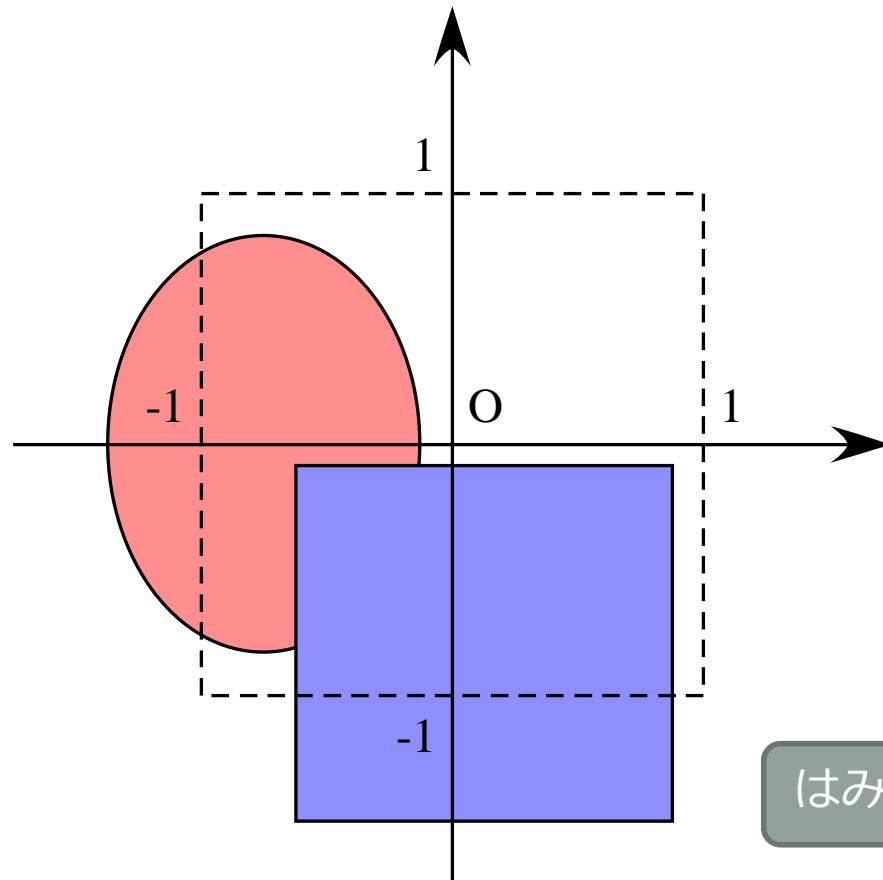


透視投影

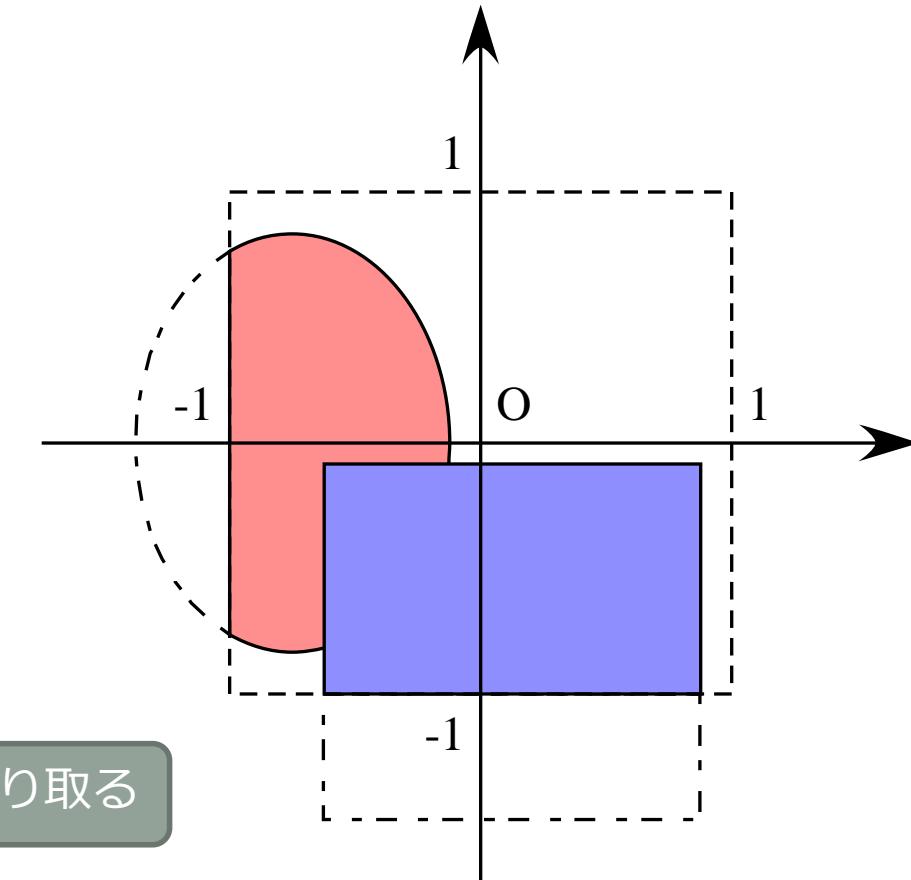


2次元のクリッピング

クリッピング前



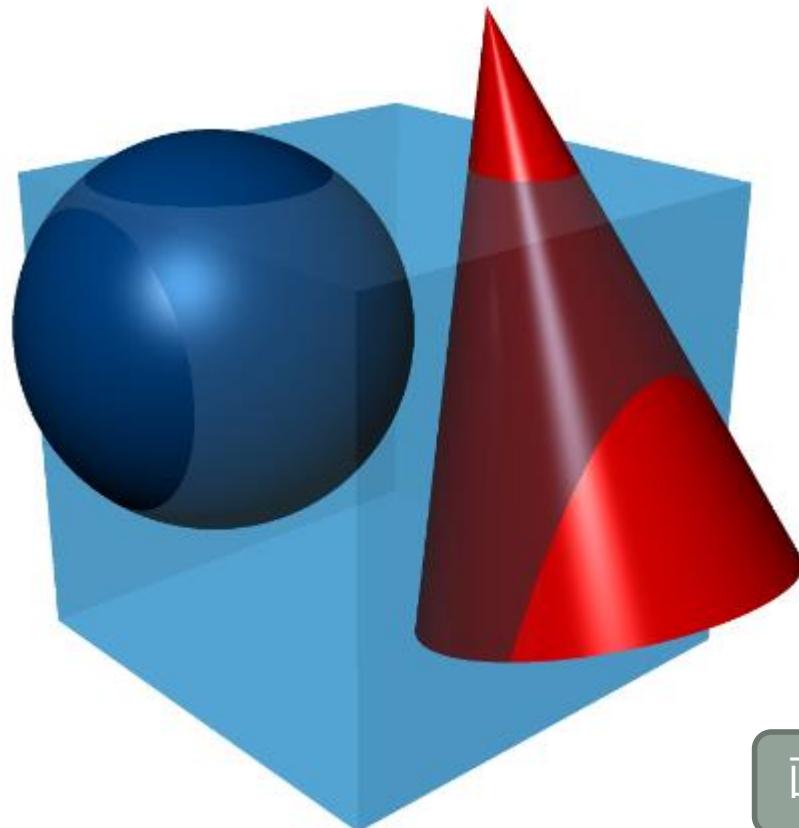
クリッピング



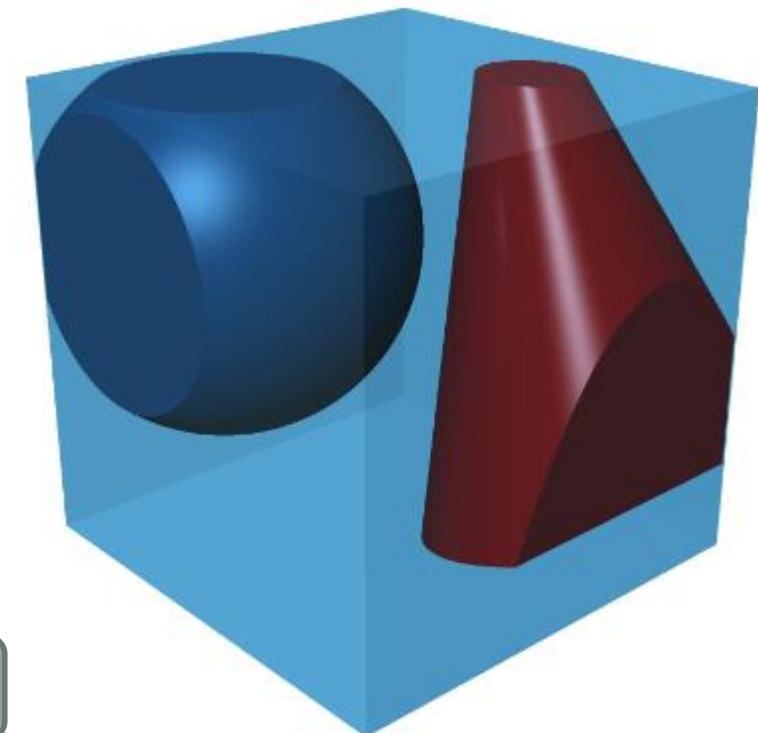
はみ出たところを削り取る

ビューボリュームによるクリッピング

クリッピング前



クリッピング後



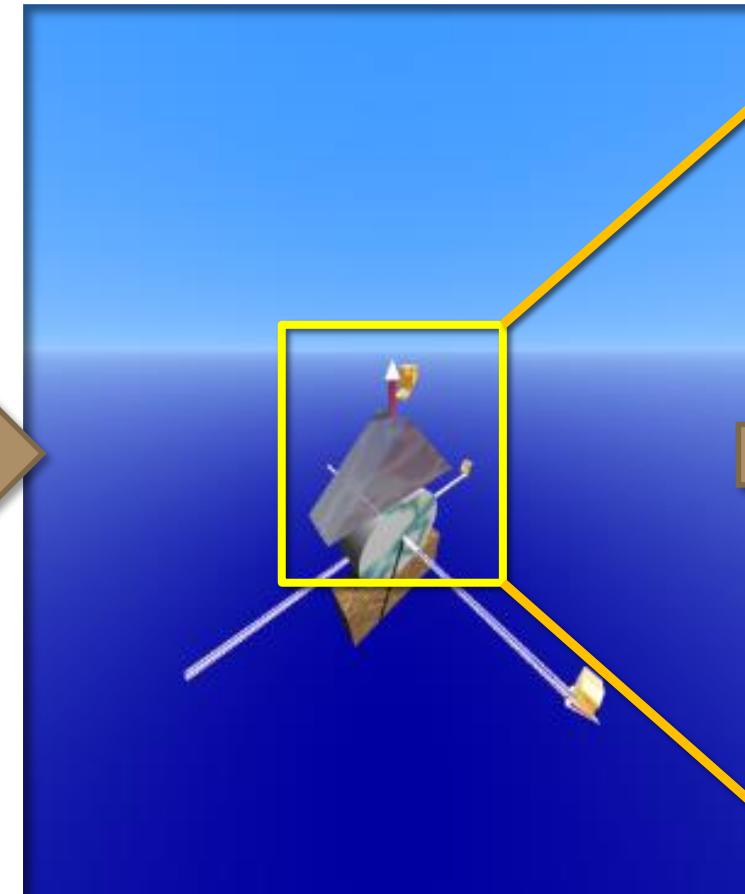
画像はイメージです

投影とクリッピング

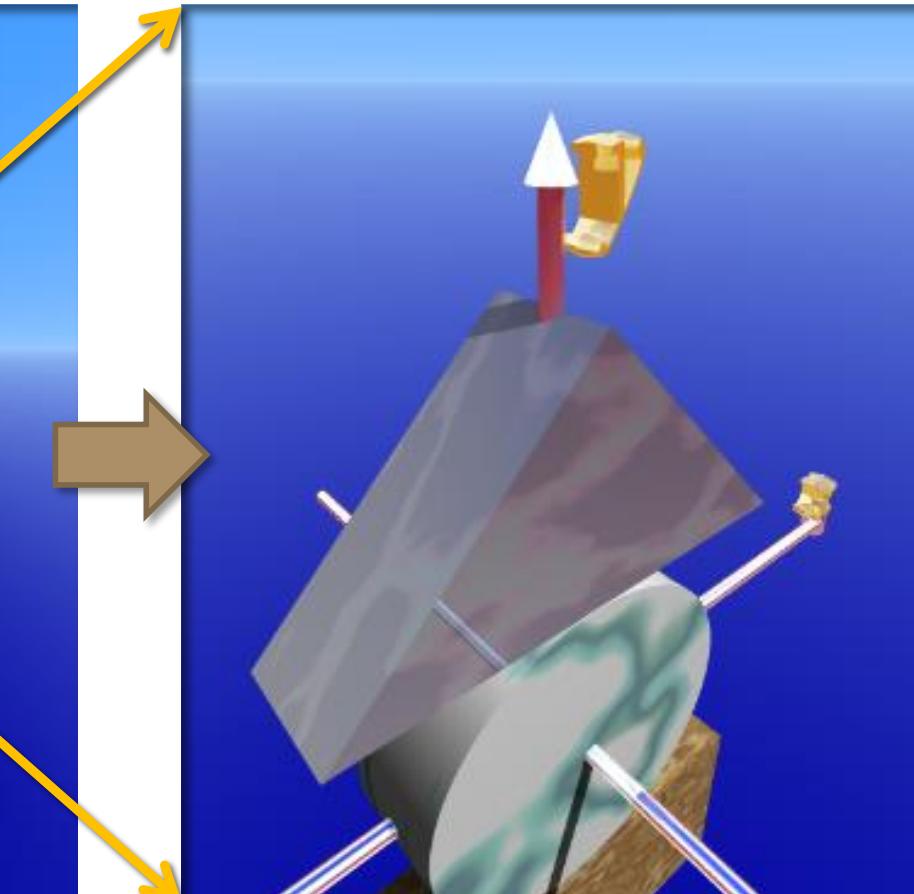
視点座標系



スクリーンの投影像

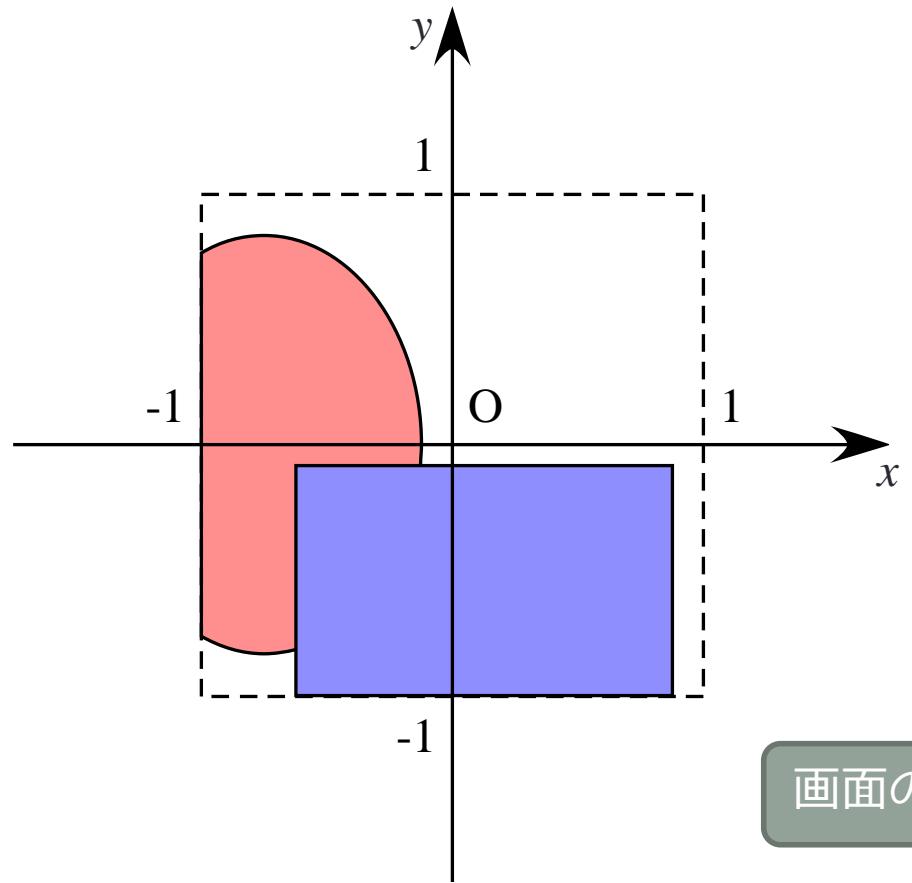


クリッピング座標系

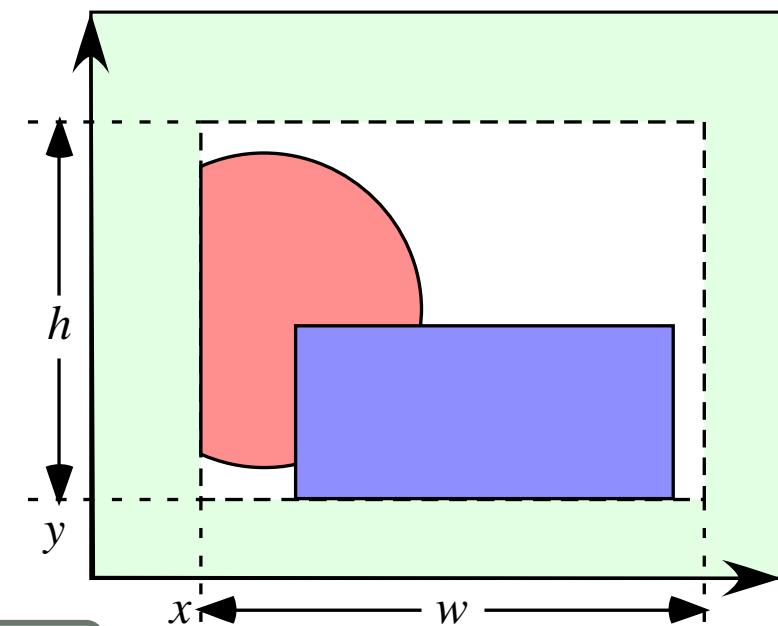


ビューポート変換

クリッピング座標系



デバイス座標系



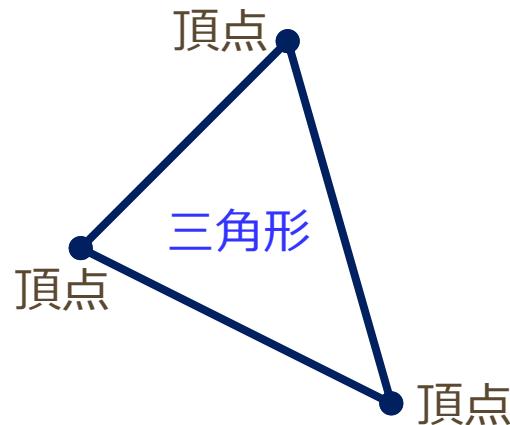
画面の表示領域内に収める

ラスタライズ処理

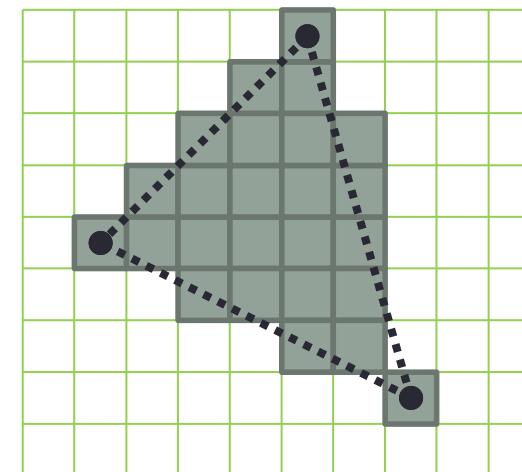
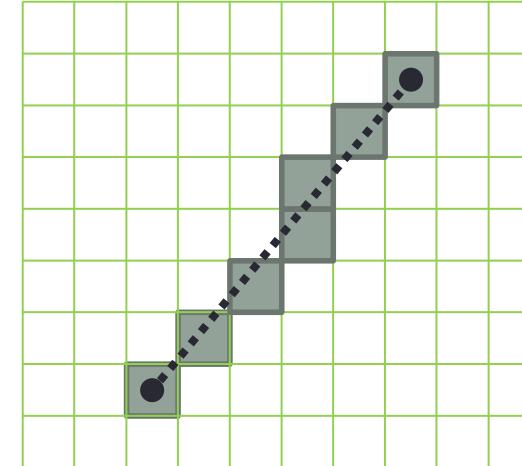
画像化

ラスタライズ処理

ジオメトリデータ



フラグメントデータ

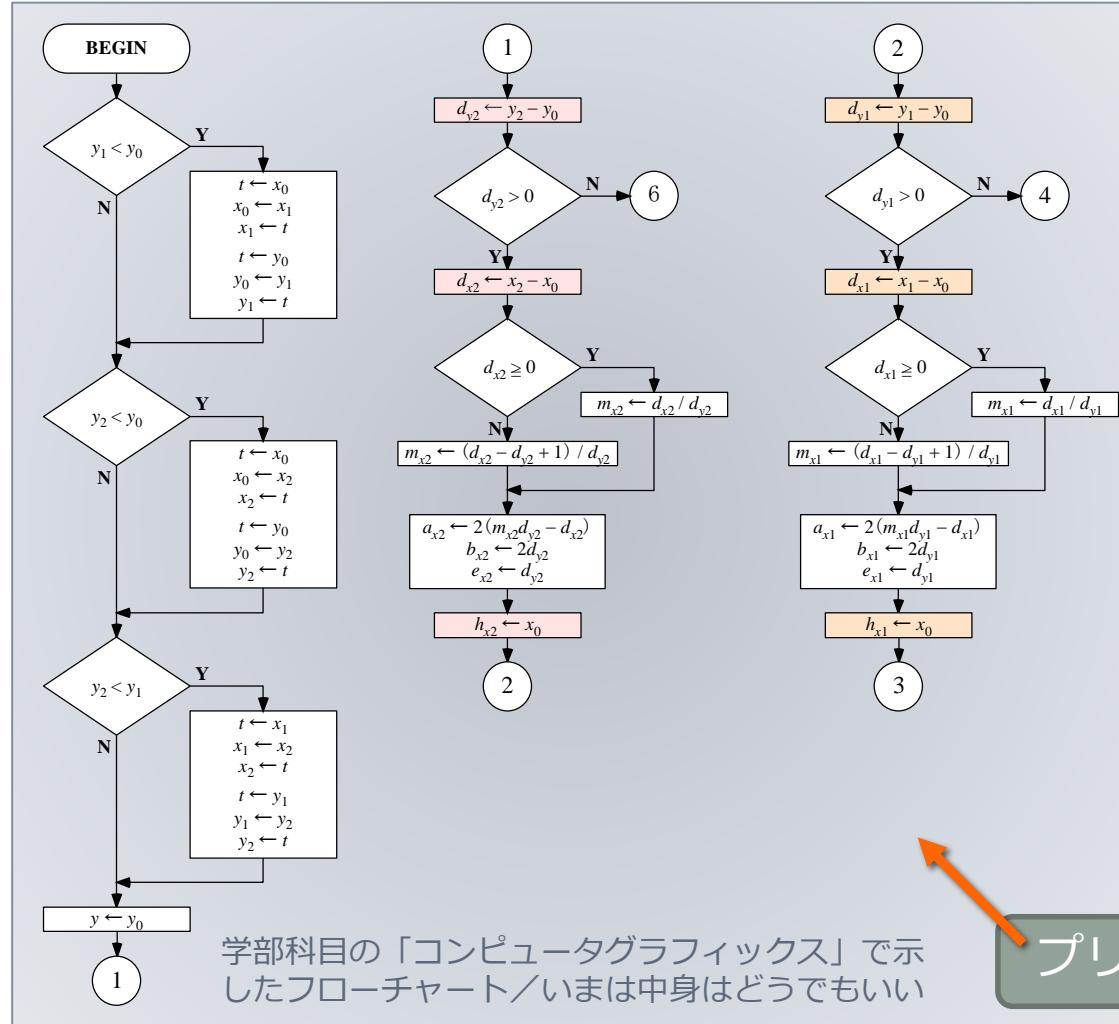


ラスタライザ

- ・図形データから画像データを生成するハードウェア
 - ・プリミティブセットアップ
 - ・スキャンコンバージョン（塗りつぶし）のための係数等の計算
 - ・スキャンコンバージョン（走査変換）
 - ・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→ラスターデータ
- ・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）

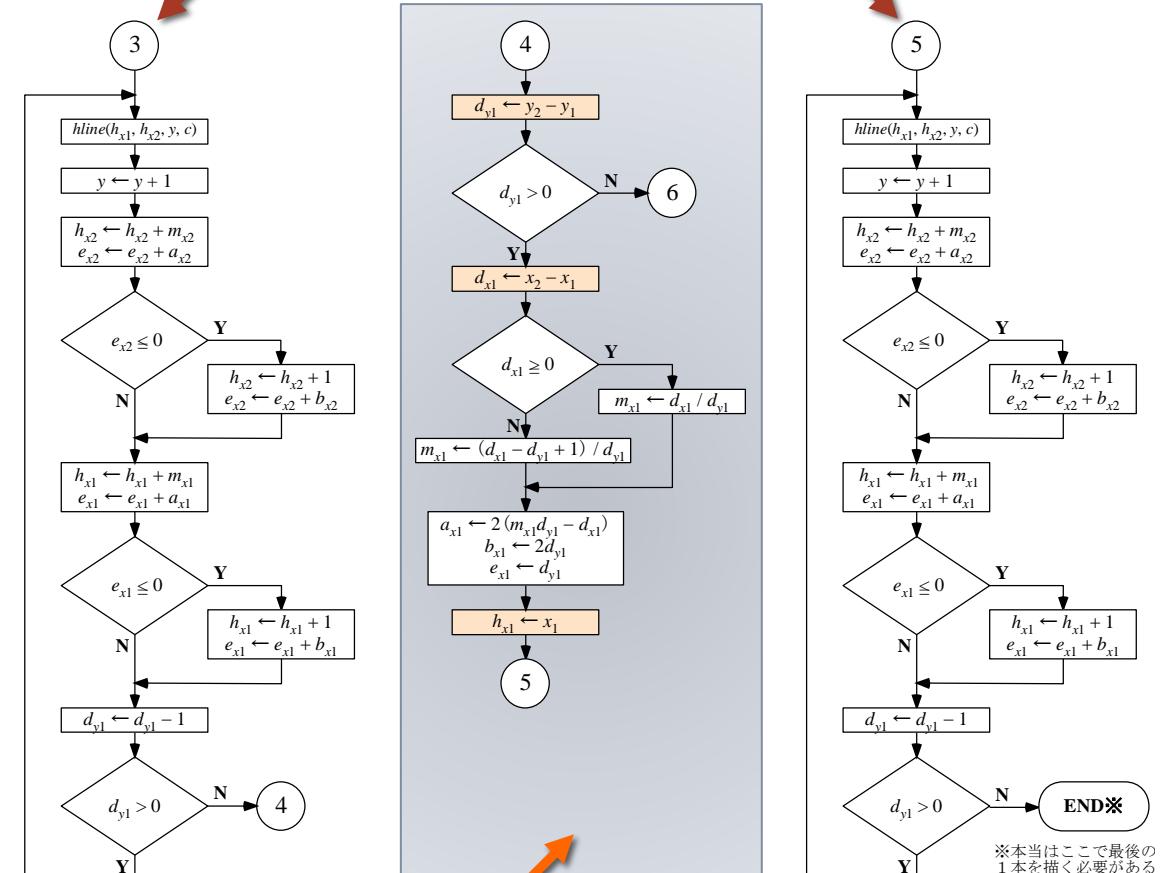
他に、法線ベクトル、テクスチャ座標、…

プリミティブセットアップ



プリミティブセットアップ

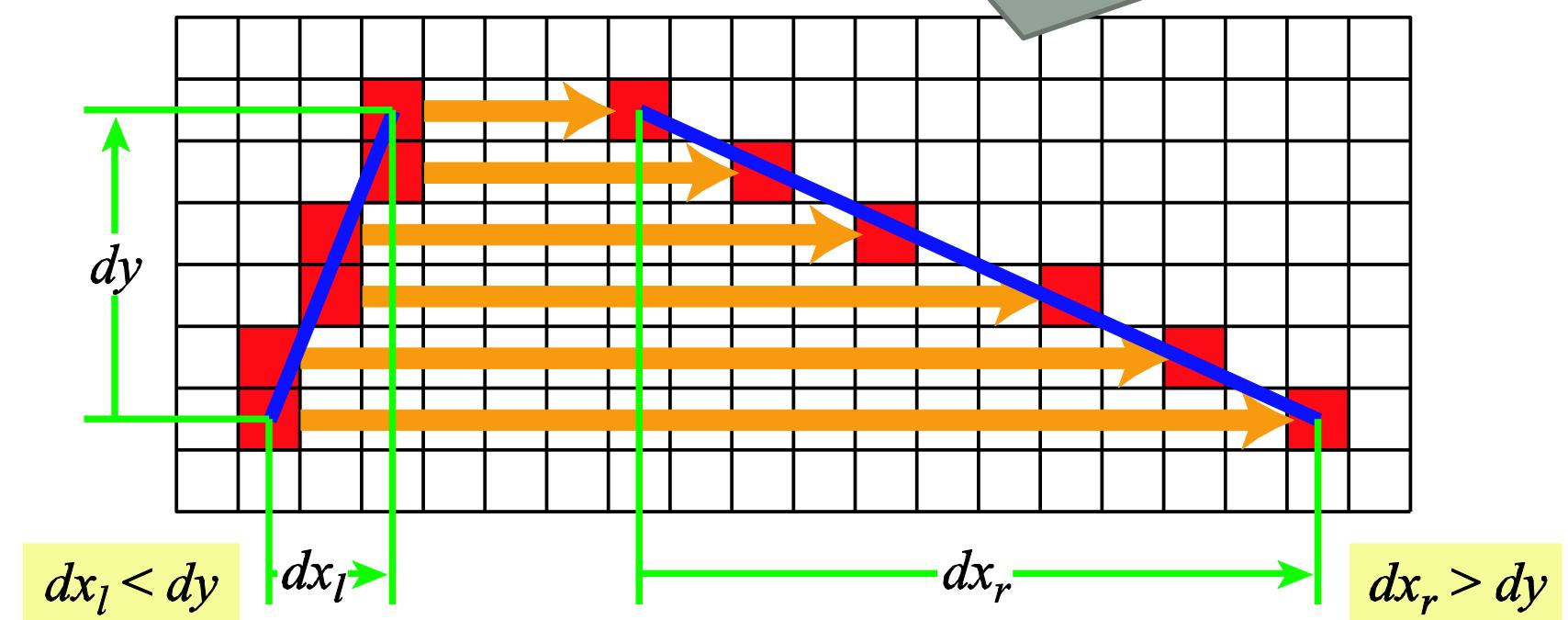
スキャンコンバージョン



スキャンコンバージョン (走査変換)

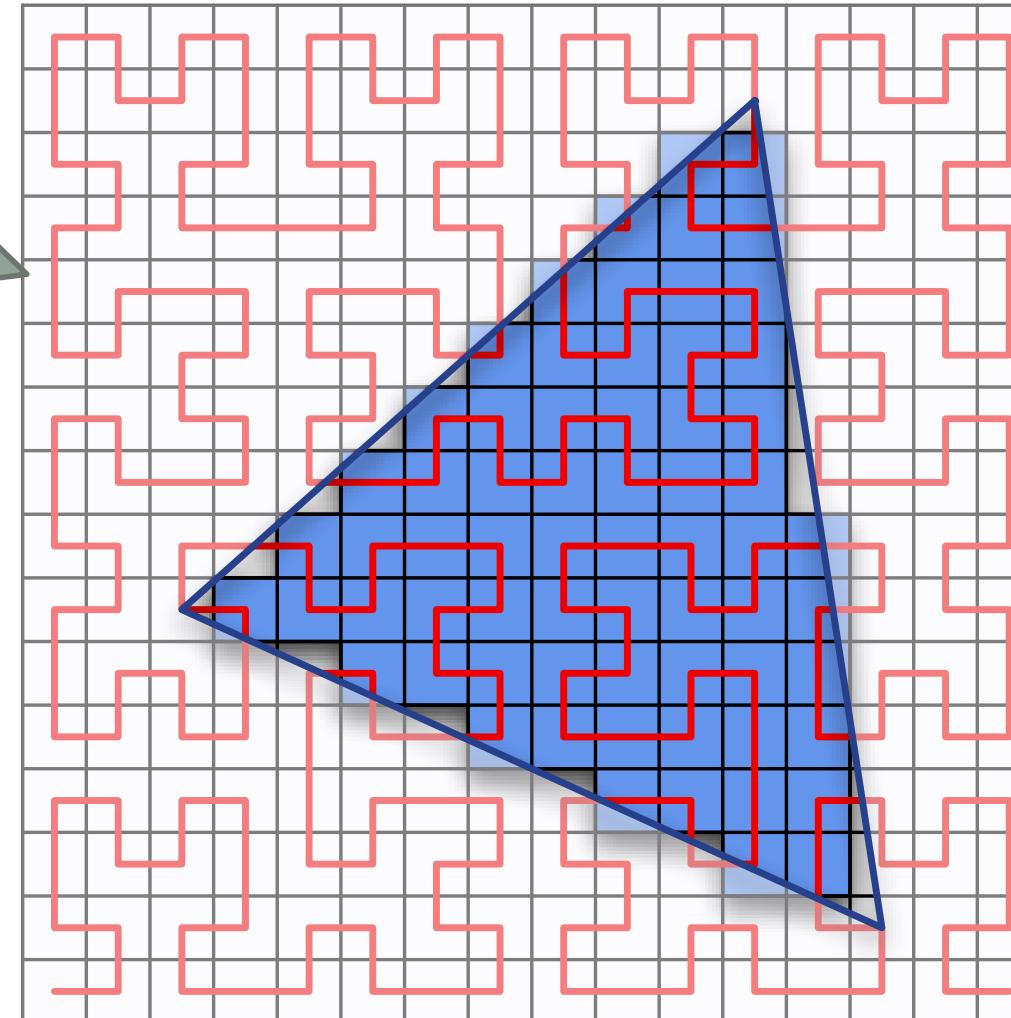
- ・頂点色の補間
- ・奥行きの補間
- ・テクスチャ座標の補間
- ・画素の選択

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

画素をヒルベルト曲線でたどって三角形内の画素を選択



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

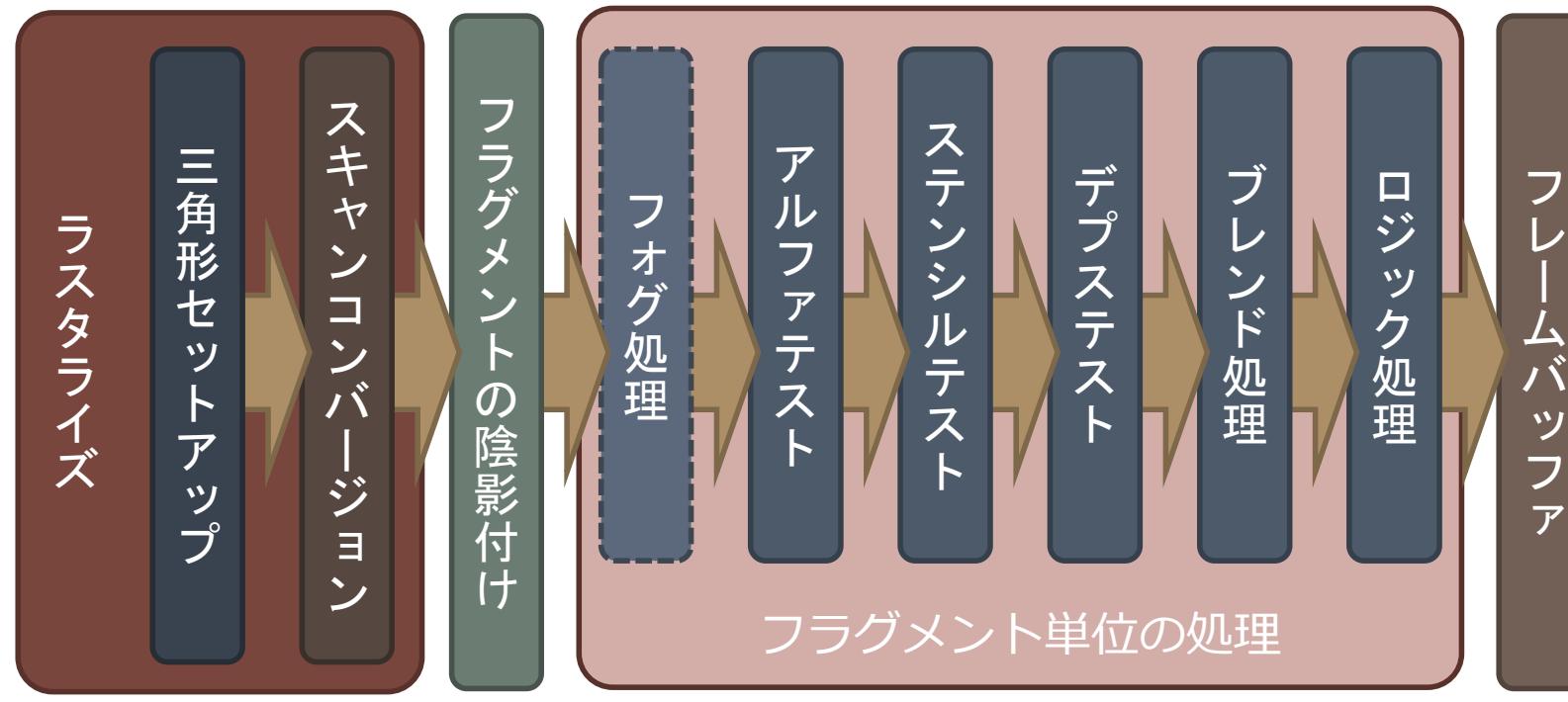
フラグメント処理ステージ

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** (不透明度), **ステンシルテスト** (型抜き), **デプステスト** (深度)
- 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** (論理演算), **ブレンドオペレーション** (合成)
- **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



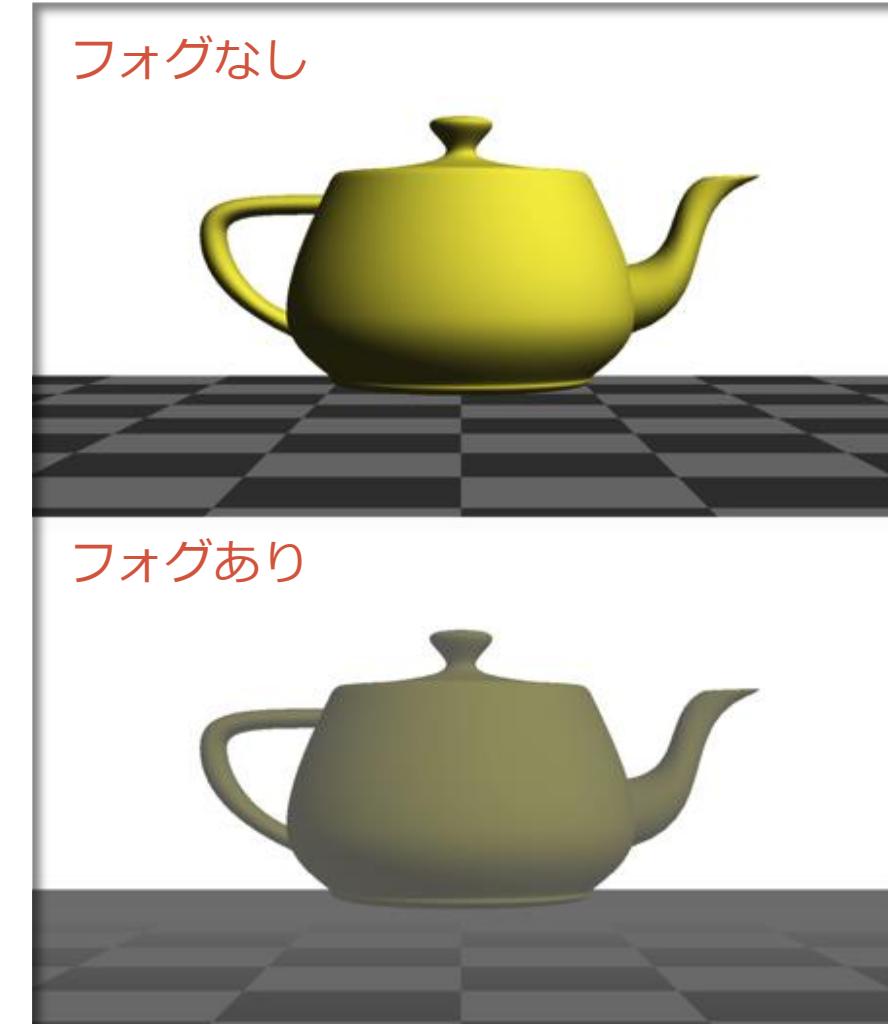
フラグメントの陰影付け

- 頂点パラメータの補間値の取得
 - デプス(深度)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
 - 頂点における法線ベクトルの補間値を使って照明計算を行う場合もある

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色の合成
- ロジック処理
 - スクリーン上で論理演算

半透明処理等



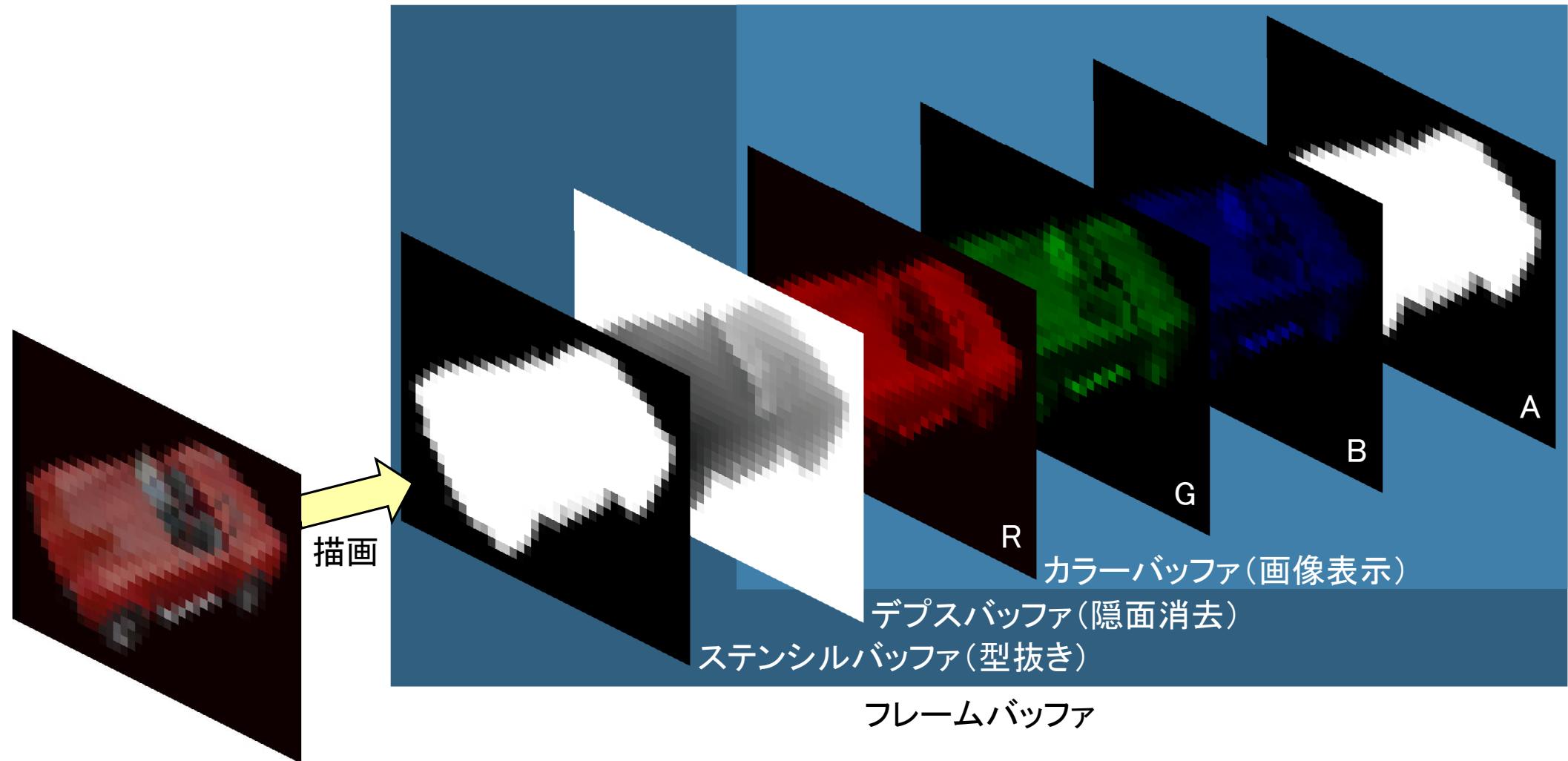
フレームバッファ

画像を保持する

フレームバッファ

- ・図形の描画先
 - ・フラグメント処理の結果が描き込まれる
- ・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
 - ・Stencilバッファ
 - ・表示図形の「型抜き」に使う
 - ・Accumulation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）

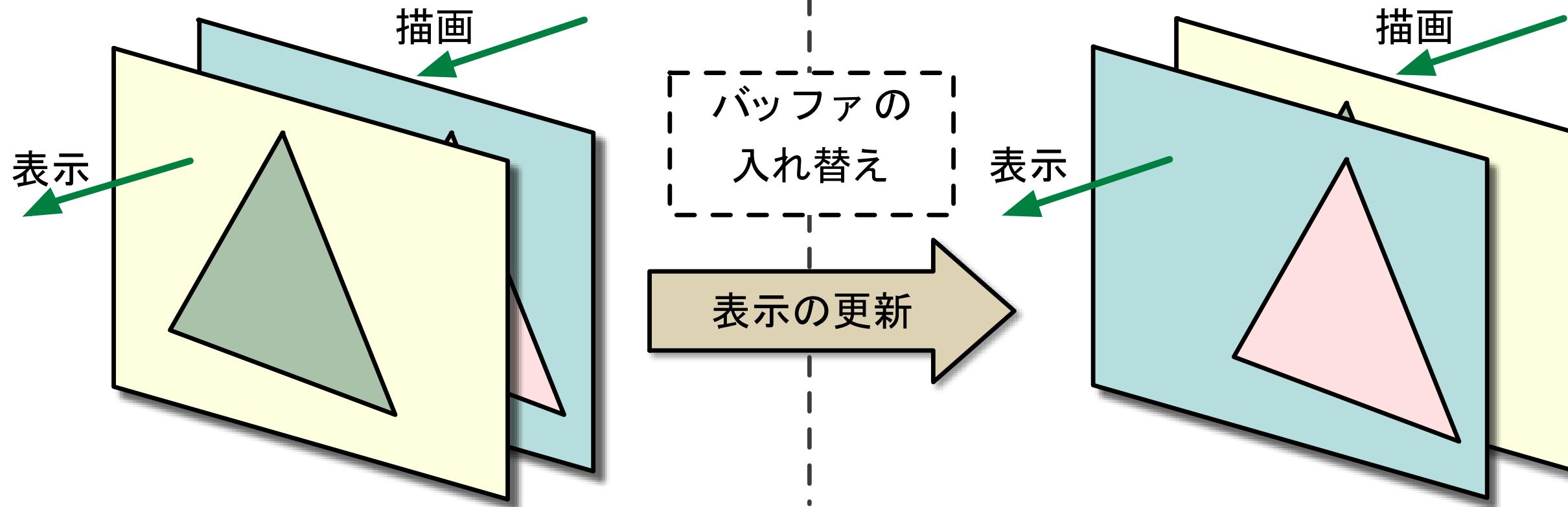
フレームバッファのイメージ



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
 - 二つのカラーバッファを使って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フロントバッファと入れ替える
 - ディスプレイの表示更新のタイミング (垂直帰線消去時)

ダブルバッファリング



デプスバッファ (Zバッファ)

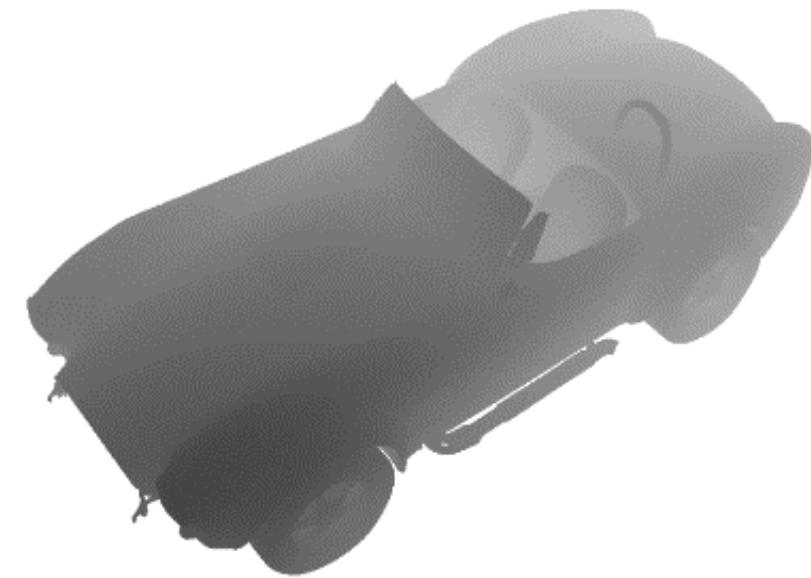
- 隠面消去処理に用いる
 - カメラから見える（他の図形より手前にある）ものを表示する
- カラーバッファと同じサイズをもつ
 - カメラに最も近い図形のデプス（深度）を各画素に格納する
 - 描画しようとする図形の深度と既に格納されている深度を比較する
 - 深度が小さいほうの図形の色を表示し、その深度を格納する
- 図形を任意の順序で描画できる
 - 物体の交差を表現できる
 - 半透明の図形は視点から遠いものから順に描く必要がある

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)

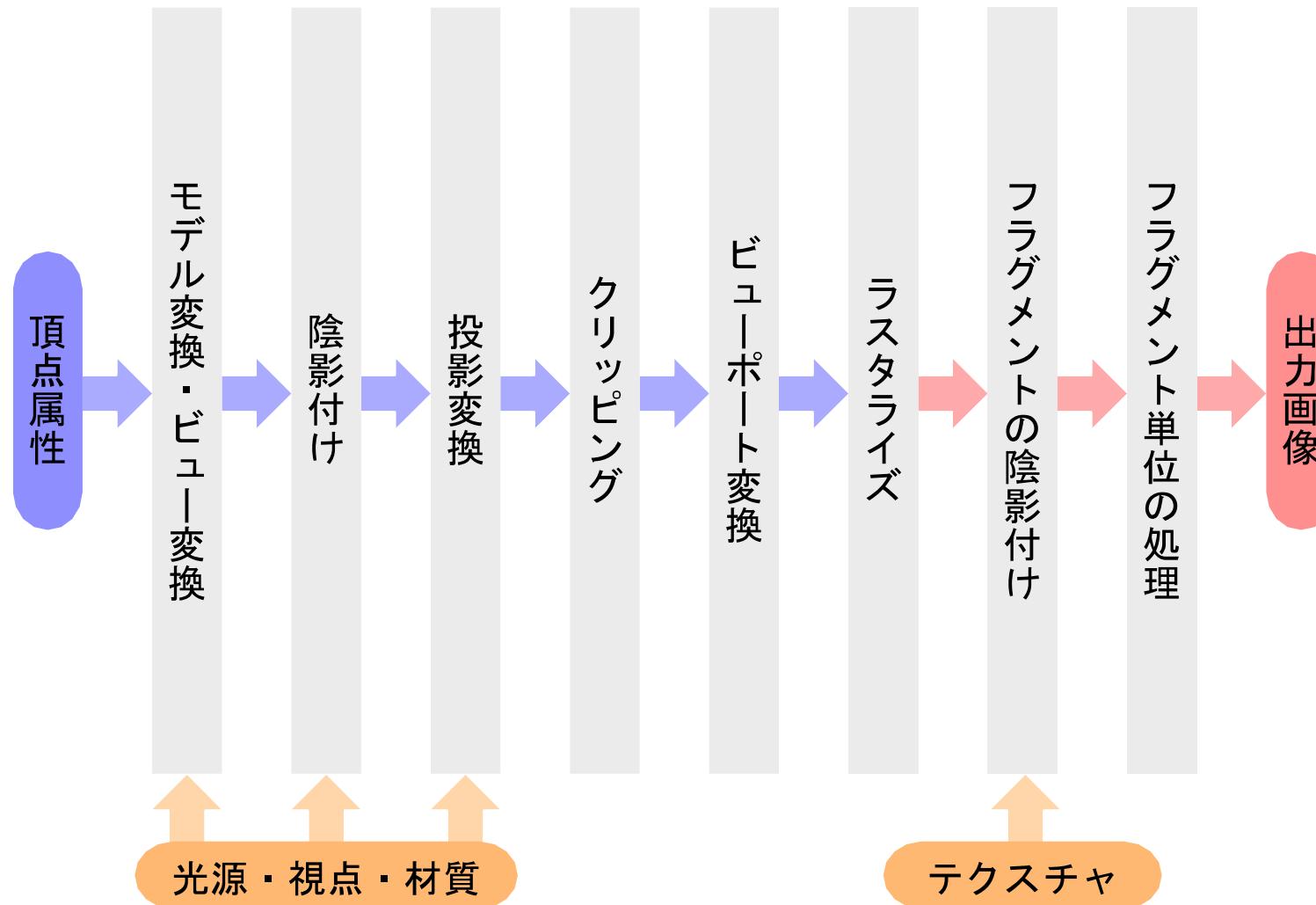
オフスクリーンバッファ

- ・直接画面には表示されないフレームバッファ
 - ・ダブルバッファリング時のバックバッファ
 - ・表示に使われるフレームバッファの表示領域外
 - ・フレームバッファオブジェクト (FBO)
- ・**FBO** (Frame Buffer Object)
 - ・ソフトウェア開発者が GPU 上に確保したメモリをバッファに用いる
 - ・それらのバッファの任意の組み合わせでフレームバッファを構成する
 - ・それぞれのバッファに何を格納するかはソフトウェア開発者が決める
 - ・レンダリング結果を利用した様々な効果に用いられる
 - ・レンダリング結果をテクスチャとして参照する (Render to Texture)

レンダリングパイプラインの ハードウェア化

ハードウェアアクセラレーションから GPU まで

最初レンダリングパイプラインはソフトウェア実装



つまり CPU で処理されていた



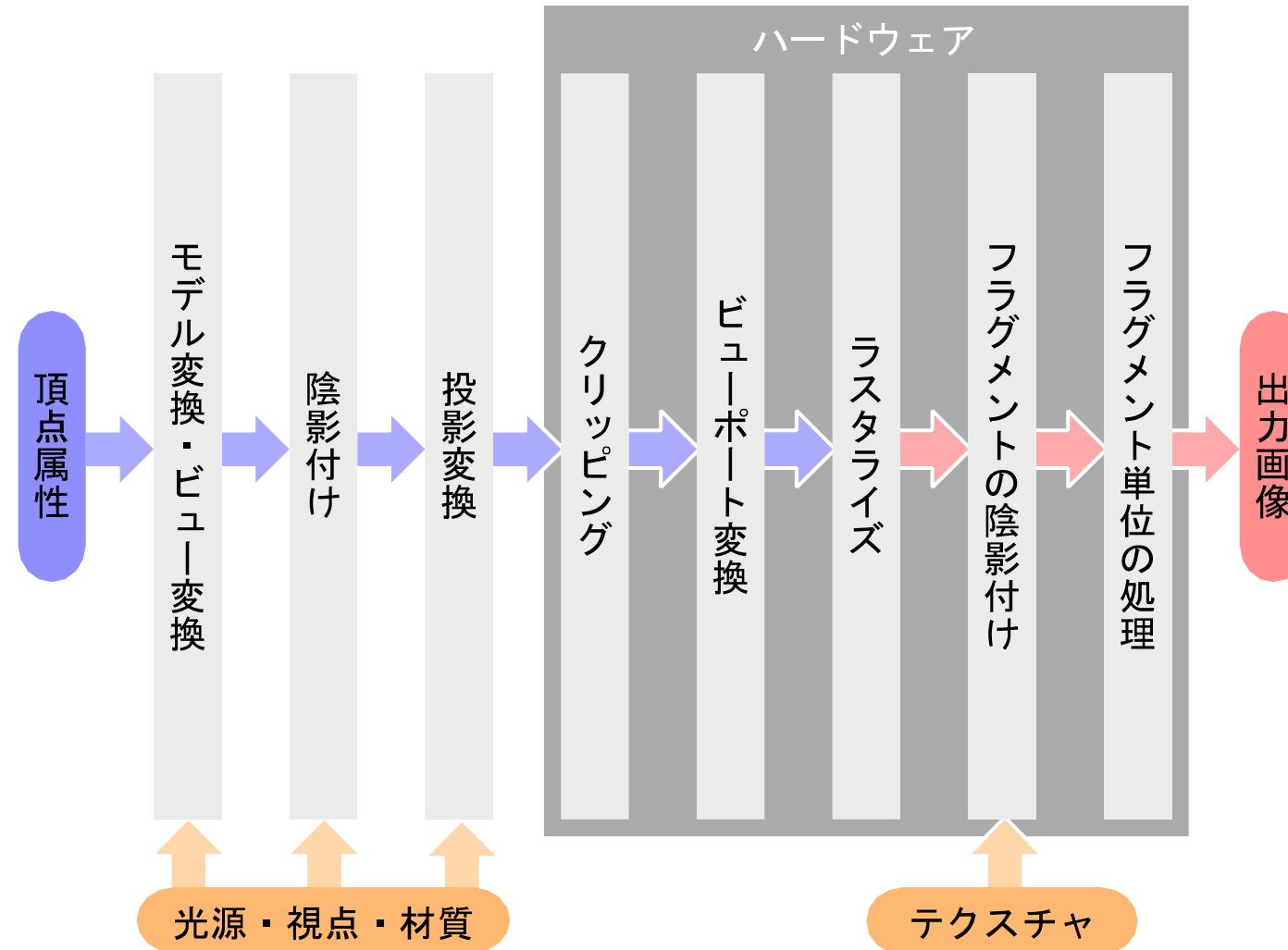
レンダリングパイプラインのソフトウェア実装の問題

- ・単純な処理を大量に繰り返す
 - ・計算時間がかかる
- ・計算結果の格納先（フレームバッファ）が CPU の外部にある
 - ・データの入出力のために CPU が待たされる
- ・CPU はグラフィックス表示以外にもやることがある
 - ・物理シミュレーション、アニメーション等

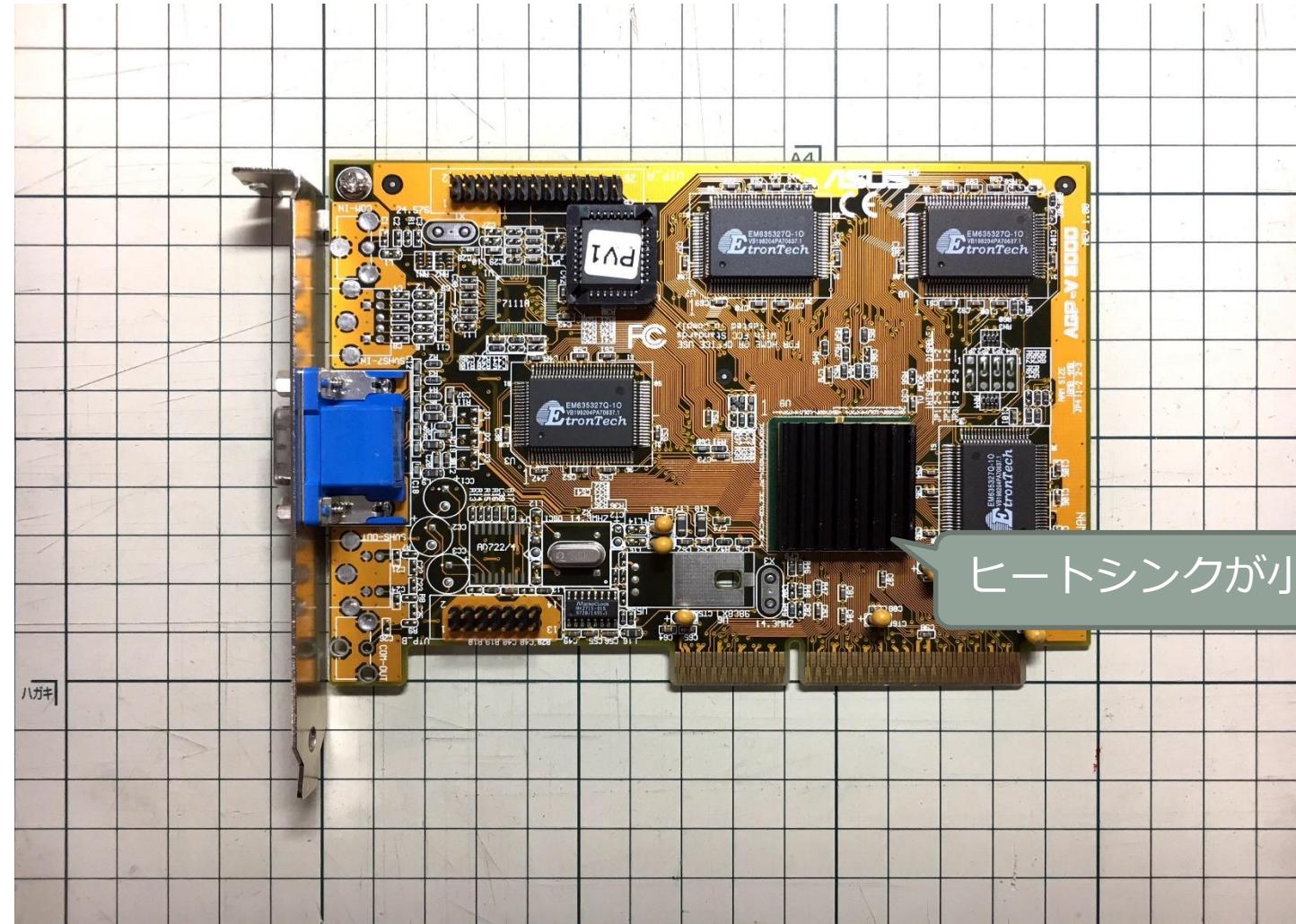
結論

レンダリングは別のハードウェアに任せる

ラスタライズ処理・フラグメント処理のハードウェア化



初期のグラフィックスアクセラレータ (RIVA 128)



フラグメント処理ハードウェア

1. クリッピング

- ・クリッピングディバイダ（二分法による交点計算）

2. 三角形セットアップ

- ・少数の整数計算と条件判断

3. スキャンコンバージョン

- ・単純な整数の加減算とループ

4. 隠面消去

- ・デプス値の補間とデプスバッファとの比較による可視判定

5. フラグメントの陰影付け

- ・頂点のテクスチャ座標値を補間
- ・テクスチャメモリからサンプリング
- ・頂点の陰影を補間してテクスチャの値に乗じる

整数演算だけで実装可能
(浮動小数点演算は高コスト)

固定小数点演算

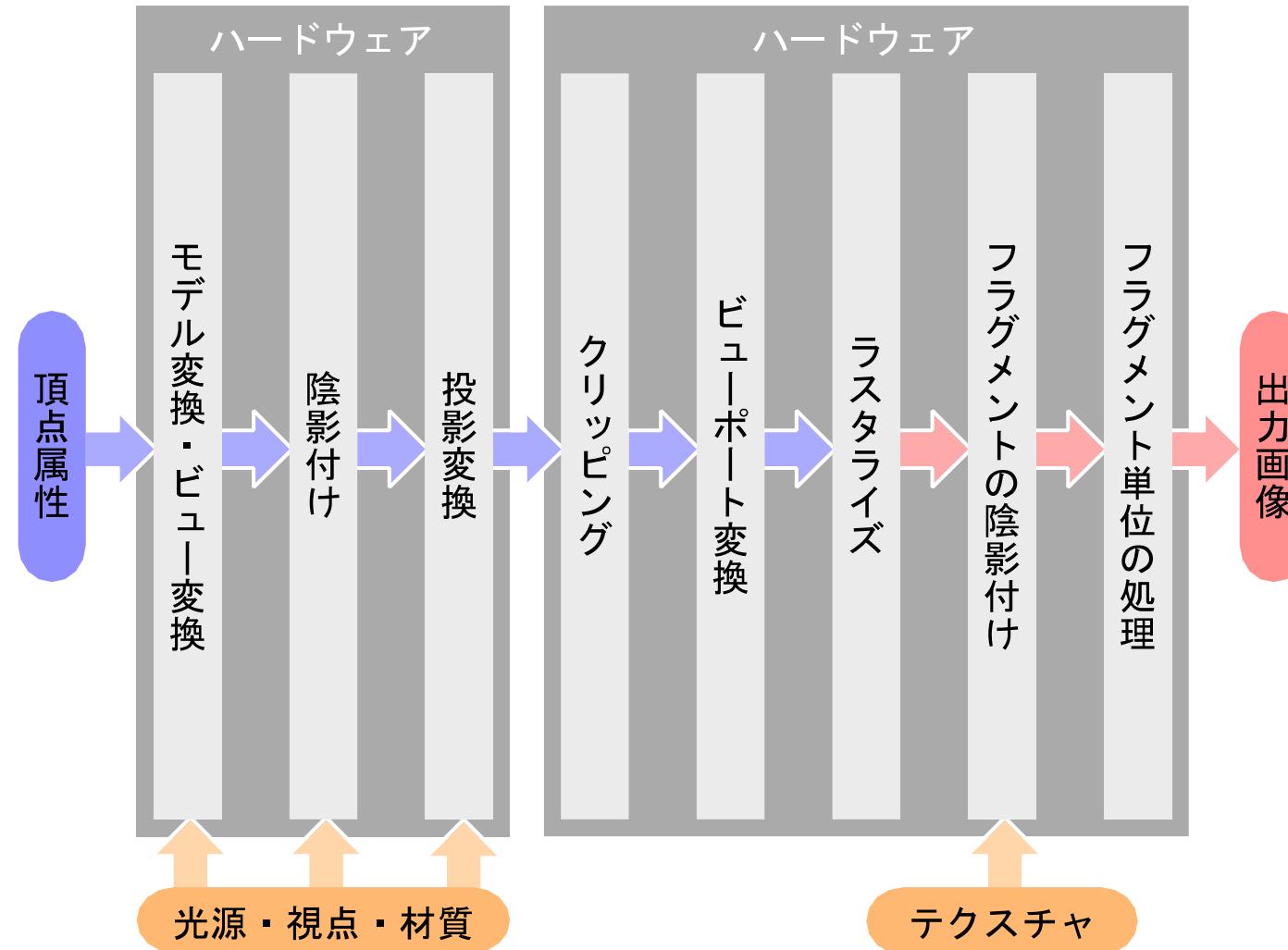
CPU によるジオメトリ処理がボトルネック化

- ・ラスタライズ処理・フラグメント処理が高速化された
 - ・CPU はより大量のデータをグラフィックスハードウェアに送れるようになった
 - ・CPU の処理が遅いとグラフィックスハードウェアの性能が活かされない
- ・ジオメトリ処理のコストは高い
 - ・大量の実数計算（浮動小数点演算）が必要
 - ・高性能で高精度な浮動小数点演算ハードウェアは高価
 - ・CPU はジオメトリ処理以外にもやることがある

結論

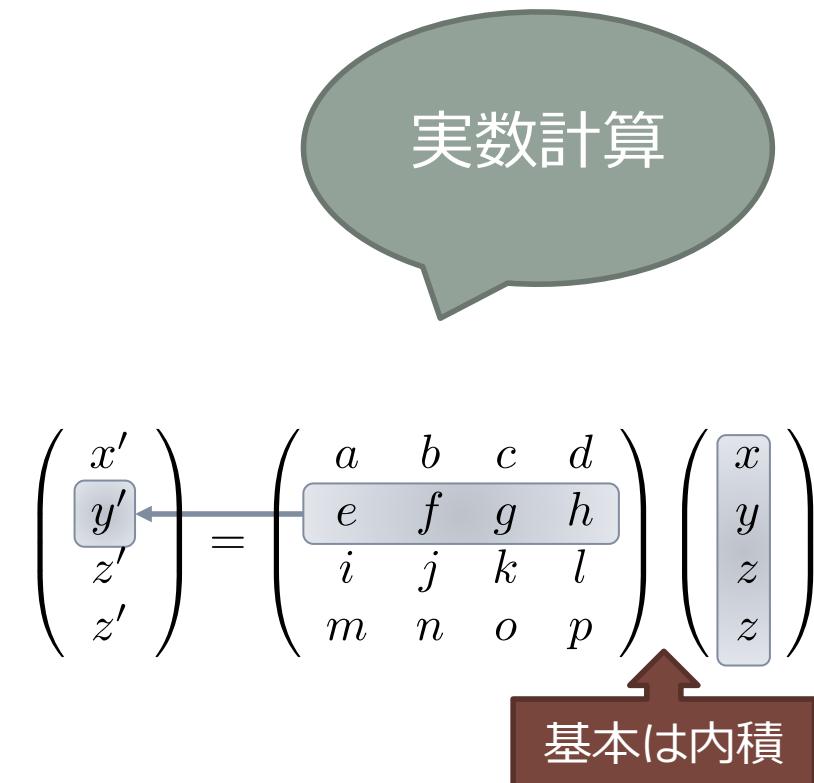
ジオメトリ処理も別のハードウェア (GPU) に任せる

ジオメトリ処理のハードウェア化

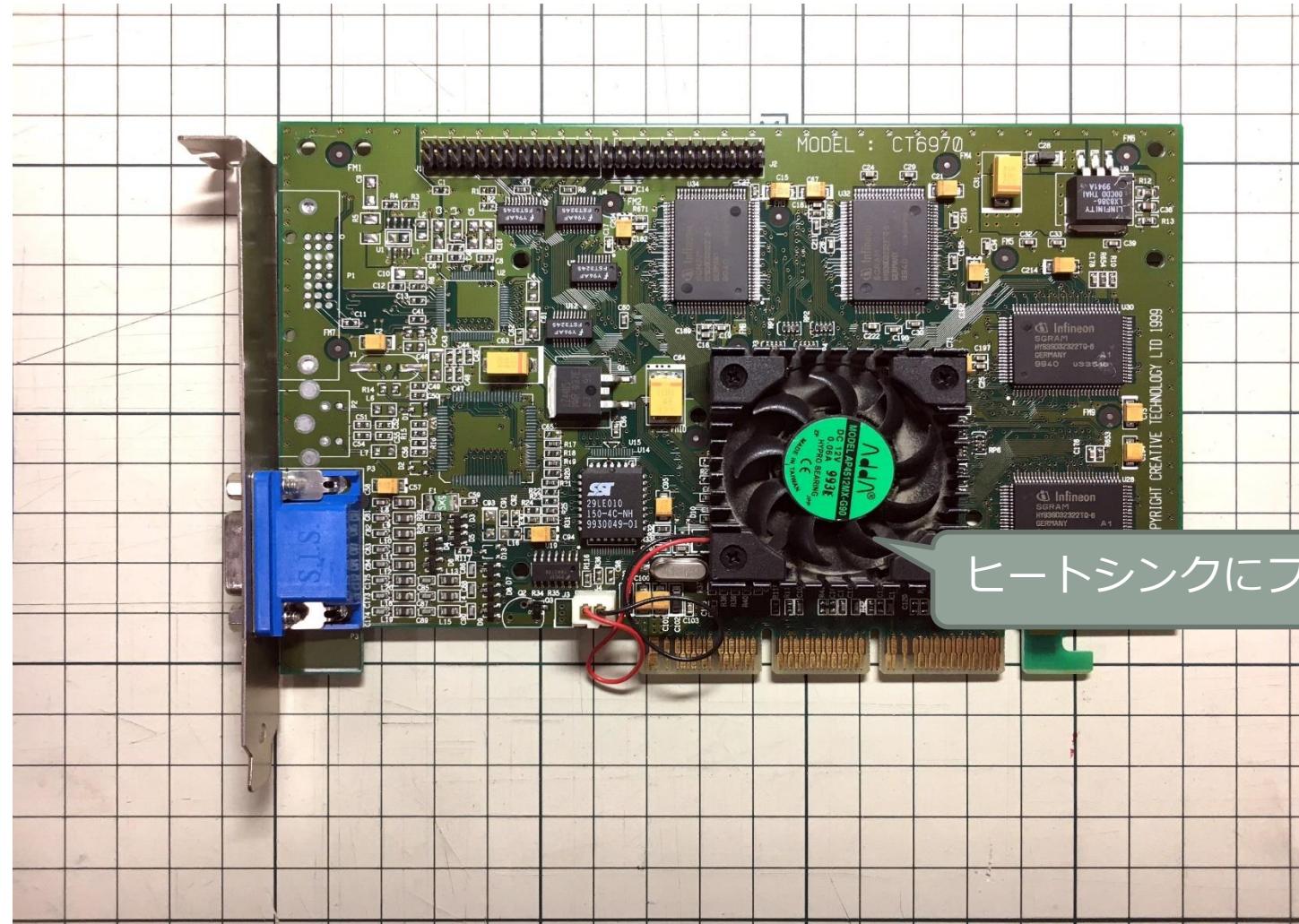


ジオメトリ処理ハードウェア

- ・浮動小数点演算ハードウェアの導入
 - ・主として**積和計算**を実行する
- ・座標変換 (Transform)
 - ・4要素のベクトル同士の**内積**
 - ・4要素のベクトルと4×4行列の積 (内積4回)
 - ・4×4行列どうしの積 (ベクトルと行列の積4回)
- ・照明計算 (Lighting, 陰影付け)
 - ・四則演算, 逆数
 - ・平方根の逆数, 指数計算
 - ・内積計算, 外積計算
 - ・条件判断, クランプ (値の範囲の制限)
- ・この機能は**ハードウェア T&L** (Transform and Lighting) と呼ばれる



ハードウェア T&L を持った GeForce 256



プログラマブルシェーダの導入

ユーザによる機能追加

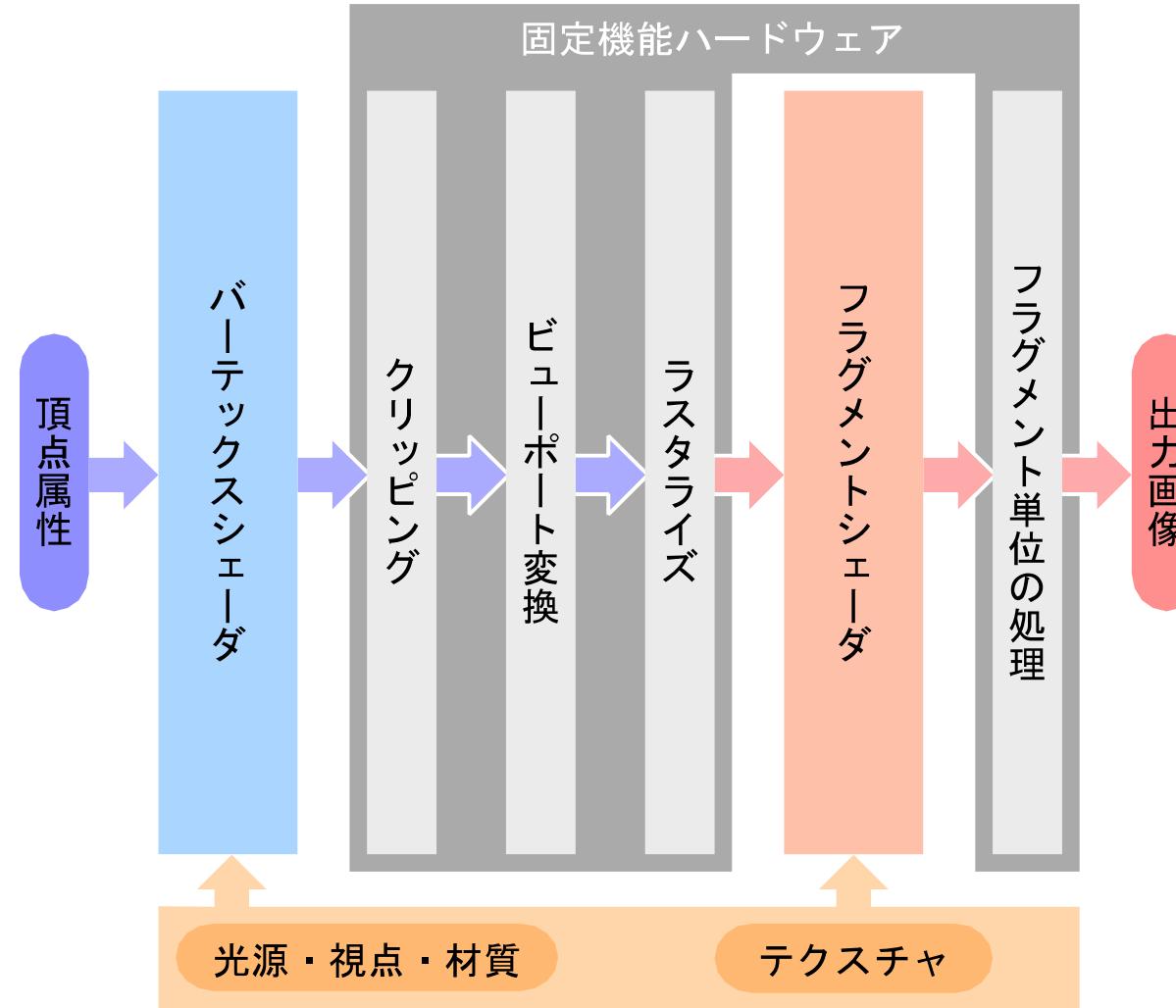
ハードウェアによる機能追加の限界

- ・レンダリングに対する要求（品質・速度）には際限がない
 - ・時代とともにより高度な体験が求められる
- ・高度な機能や複雑なアルゴリズムのハードウェア実装はコスト高
 - ・要求に応じて機能を追加すれば回路が大規模化する
 - ・回路が大規模化すれば開発コストや製品価格も高くなる
 - ・追加した機能が必ずしもすべて利用されるとは限らない

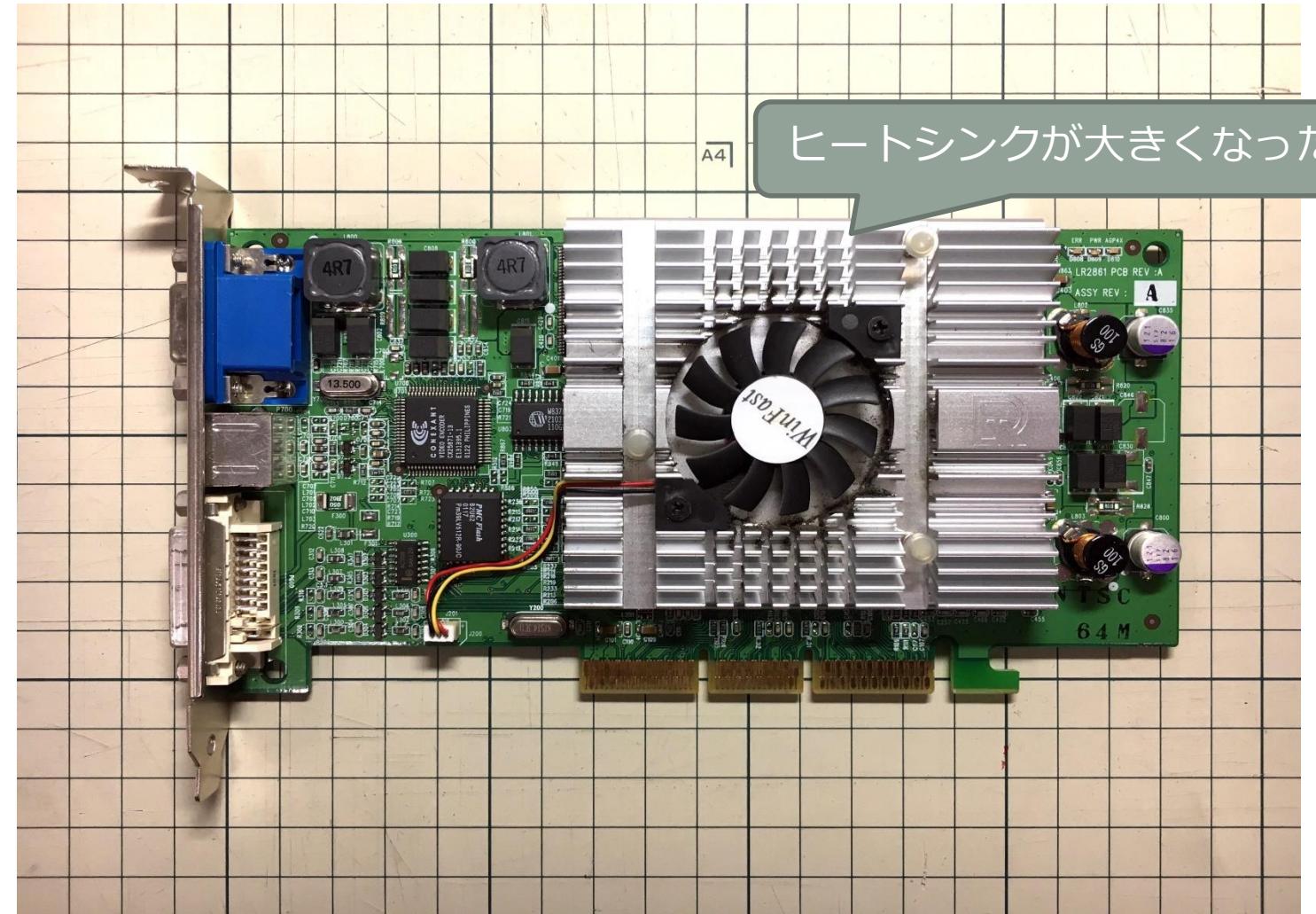
結論

機能追加はソフトウェア開発者に任せる

プログラマブルシェーダによる置き換え



プログラマブルシェーダが導入された GeForce 3



プログラマブルシェーダ

- ・バーテックスシェーダ
 - ・ジオメトリ処理を担当する
 - ・座標変換、頂点の陰影付け
 - ・入力された**頂点ごと**に実行される
- ・フラグメントシェーダ
 - ・フラグメント処理を担当する
 - ・画素の色の決定
 - ・テクスチャのサンプリング、合成 (**マルチテクスチャ**)
 - ・頂点色の補間値との合成
 - ・デプス値の補正 (ポリゴンオフセット)
 - ・出力する**画素ごと**に実行される

最初 (DirectX 8) は
ループすらなかった

制御構造を備えて
汎用プログラミング
可能に (DirectX 9)

OpenGL 2.0

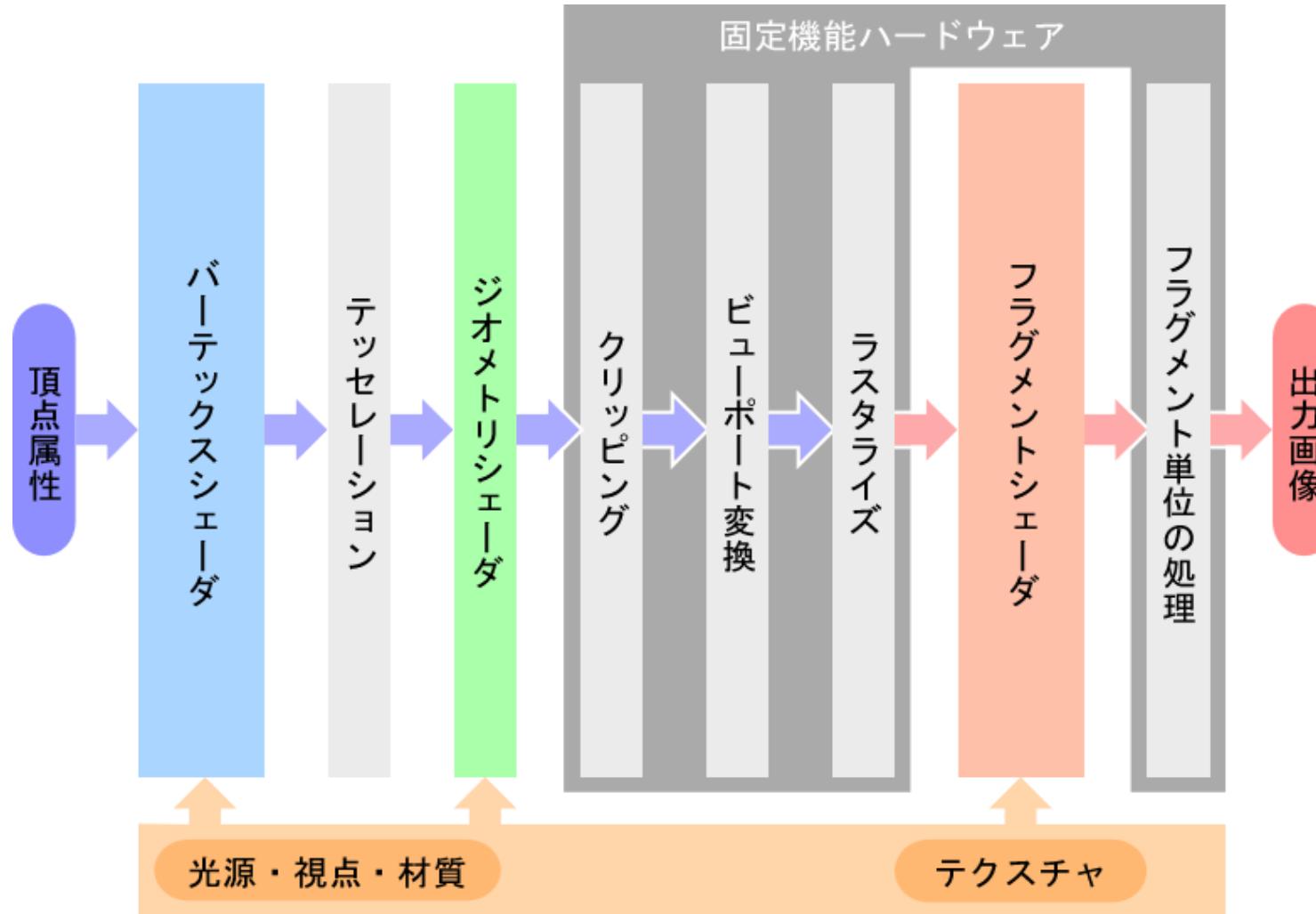
バスを介したデータ転送がボトルネック化

- CPU と GPU を接続しているバスのデータ転送速度には制限がある
 - CPU や GPU の内部のデータ転送に比べて遅い
 - データ転送に時間がかかると CPU や GPU に何もできない**待ち時間**が発生する
 - リアルタイム性を向上するにはジオメトリデータの転送量を減らす必要がある
- CPU から GPU へのデータ転送は依然多い
 - 大量のジオメトリデータが送られる
 - 滑らかな曲面は多数のポリゴン（三角形）で近似される

結論

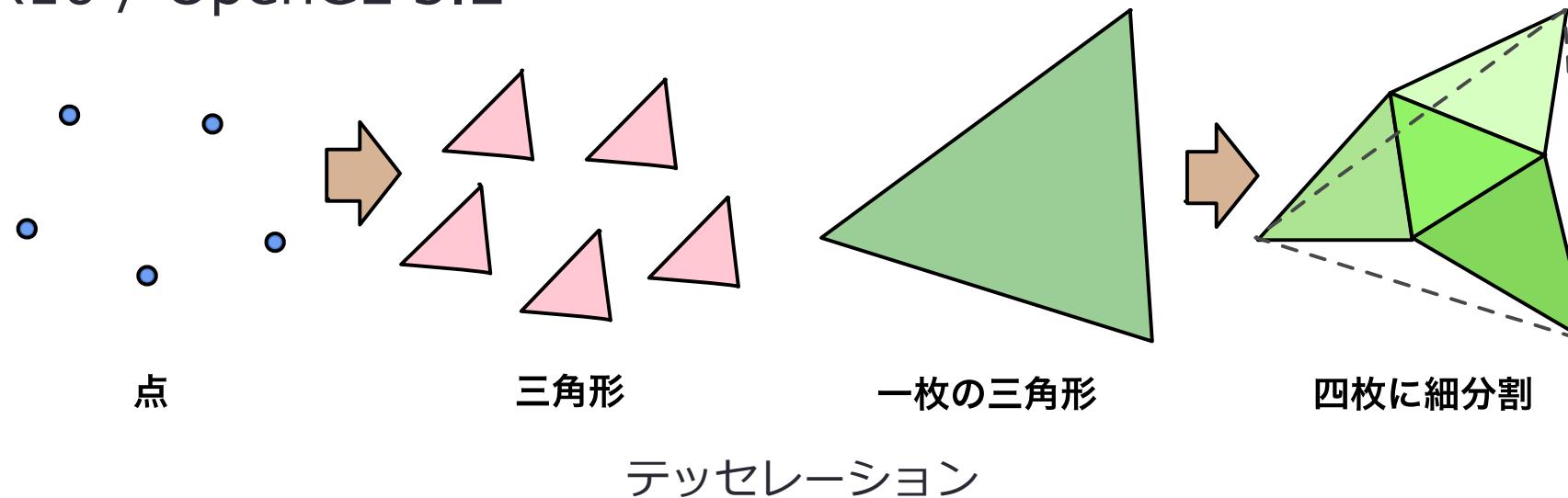
ジオメトリデータを GPU 側で生成する

ジオメトリシェーダの追加

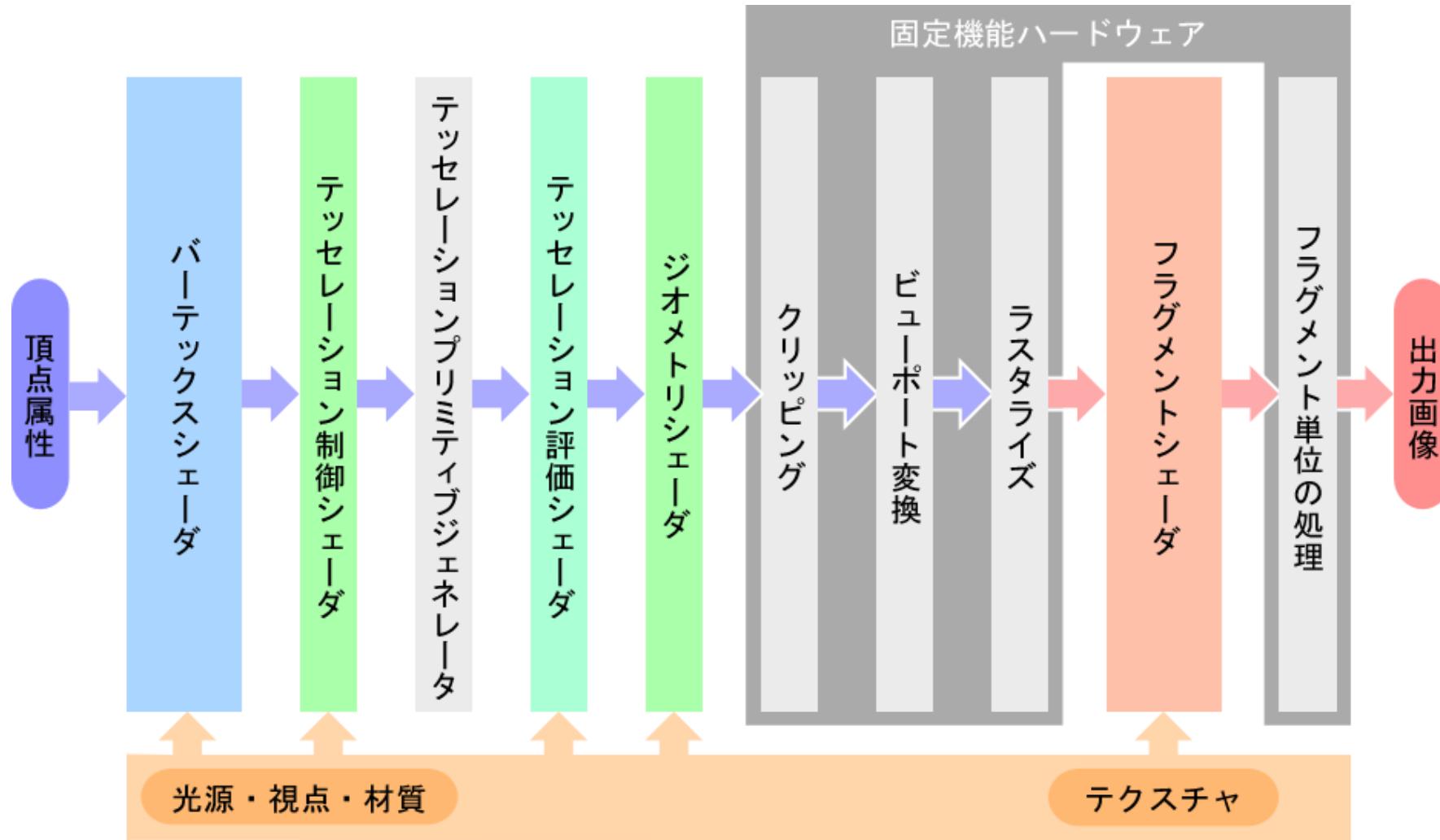


ジオメトリシェーダ

- ・ジオメトリデータの生成や細分化を行う
 - ・テッセレーション (Tessellation)
 - ・テッセレータ (テッセレーションプリミティブジェネレータ) を制御する
 - ・オプション (使用しなくても良い)
 - ・Direct X10 / OpenGL 3.2



テッセレーションの詳細な制御



テッセレーションの詳細な制御

- **テッセレーション制御シェーダ**

- テッセレーションプリミティブジェネレータによるポリゴン生成（細分化）を制御するプログラマブルシェーダ

- **テッセレーションプリミティブジェネレータ**

- ポリゴンの生成／細分化を行う固定機能ハードウェア

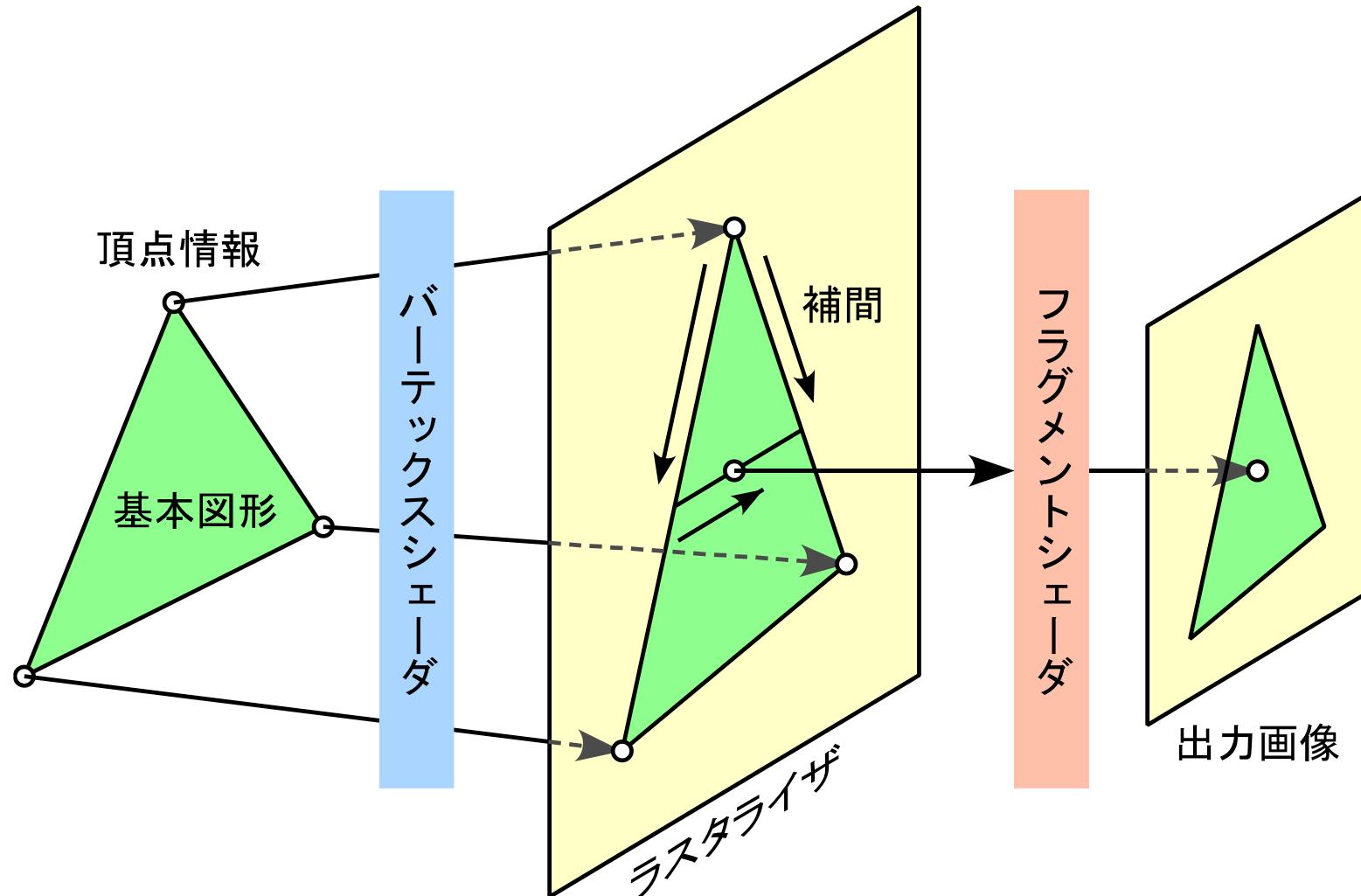
- **テッセレーション評価シェーダ**

- テッセレーションプリミティブジェネレータから出力されたジオメトリデータを処理するバーテックスシェーダに相当

- DirectX 11 / OpenGL 4.0 以降

- 後段でジオメトリシェーダも使用できる（同時利用可）

プログラマブルシェーダとラスタライザの関係



プログラマブルシェーダに対するラスタライザの役割

- ・前段から頂点情報を受け取る
 - ・前段はバーテックスシェーダか、ジオメトリシェーダあるいはテッセレーション評価シェーダ
- ・図形の塗りつぶし（**画素の選択**）を行う
 - ・フラグメントシェーダに出力先となる画素を割り当てる
- ・頂点の属性値（座標、色など）の**補間**を行う
 - ・フラグメントシェーダの入力となるデータを用意する
- ・フラグメントシェーダを起動する

シェーダプログラム

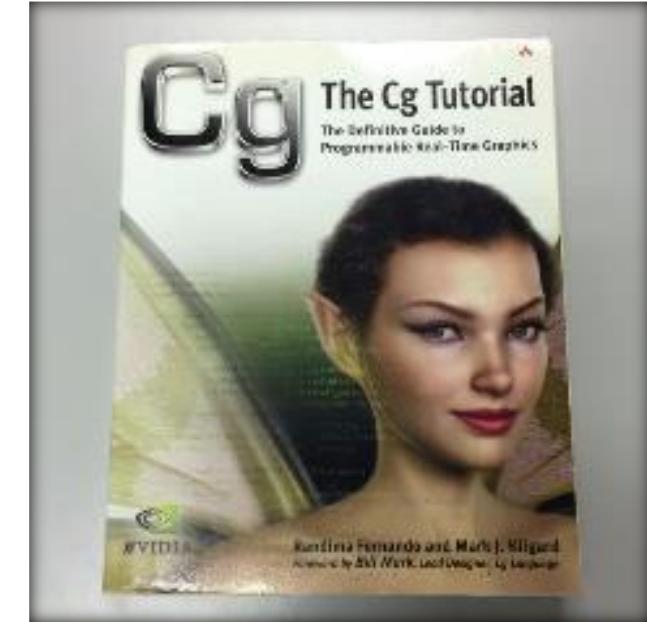
ソースプログラムのコンパイルとリンク

シェーダプログラムは GPU で実行される

- ・グラフィックスハードウェアのドライバソフトウェアが処理する
 - ・シェーダのソースプログラムのコンパイル
 - ・コンパイルされたオブジェクトプログラムのリンク
 - ・リンクされたシェーダのプログラムの **GPU** への転送
- ・データやシェーダプログラムを GPU 上のメモリへ転送する
 - ・図形データ、画像データ、定数データ
 - ・プログラムオブジェクト
- ・描画に使用する GPU 上のシェーダプログラムを指定する
- ・データと図形を指定して描画開始を指示する

シェーディング言語

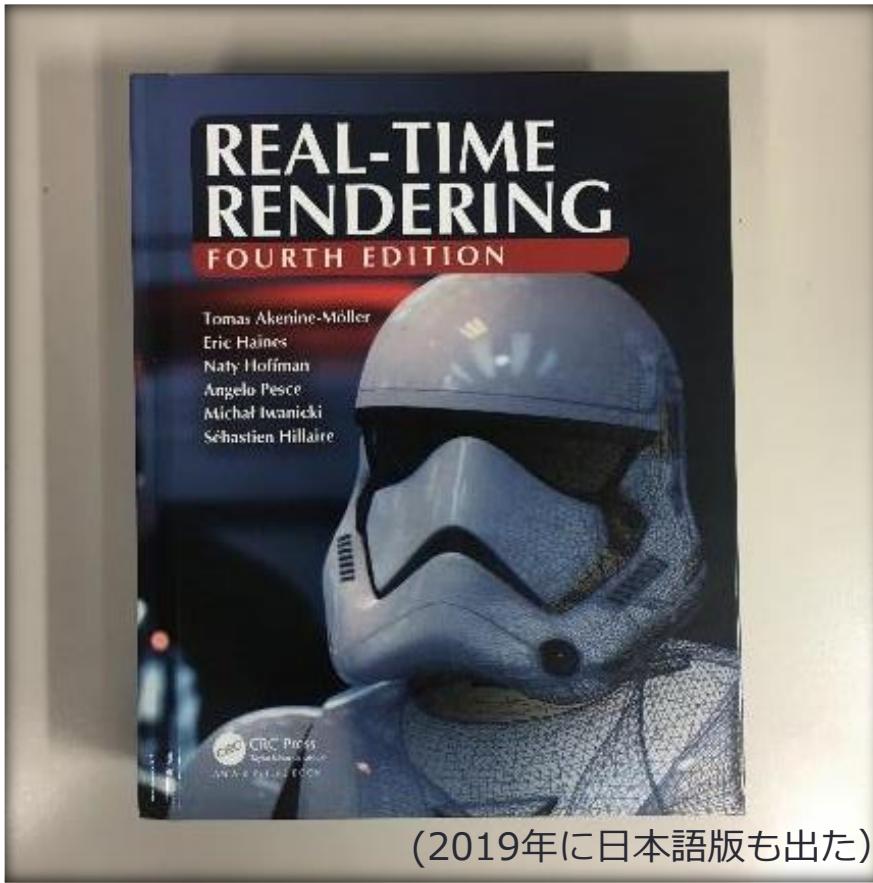
- ・シェーダのプログラミングに用いる
 - ・**Cg** (C for Graphics)
 - ・DirectX と OpenGL に対応
 - ・NVIDIA により開発
 - ・**HLSL** (High Level Shading Language)
 - ・DirectX に対応
 - ・Microsoft が NVIDIA の協力を得て開発
 - ・**GLSL** (OpenGL Shading Language, GLslang)
 - ・OpenGL に対応
 - ・OpenGL ARB (現在は Khronos Group) により開発
 - ・いざれも**C言語**に似せてある



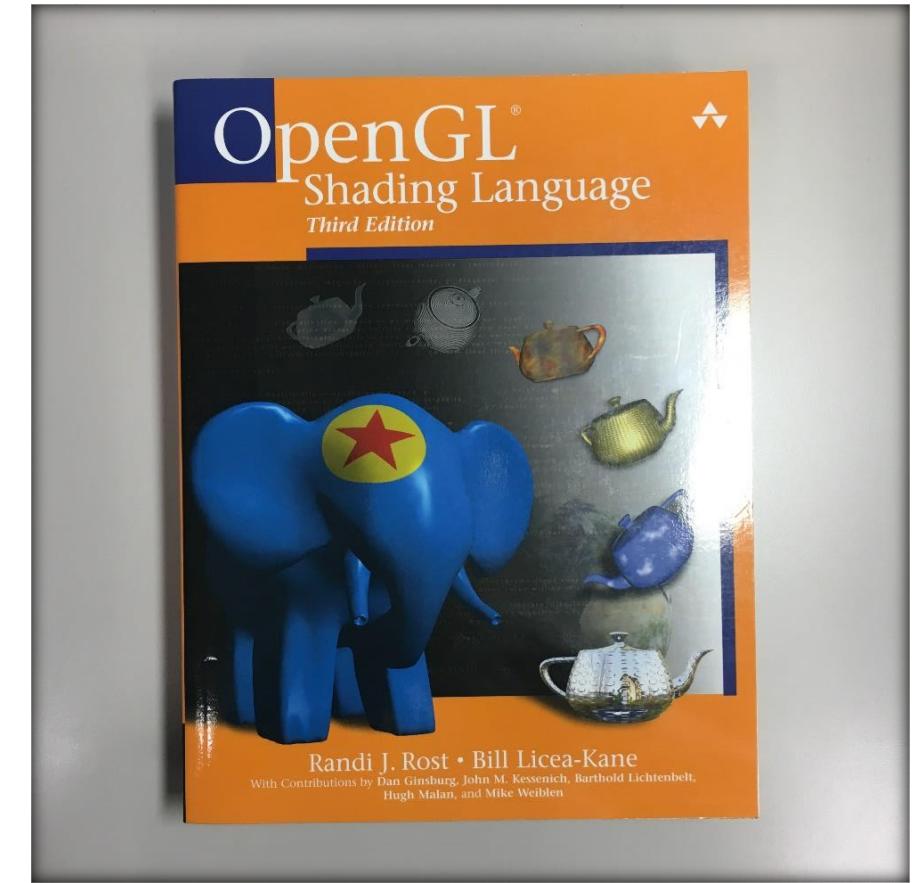
この講義では **GLSL** を採用します

参考書

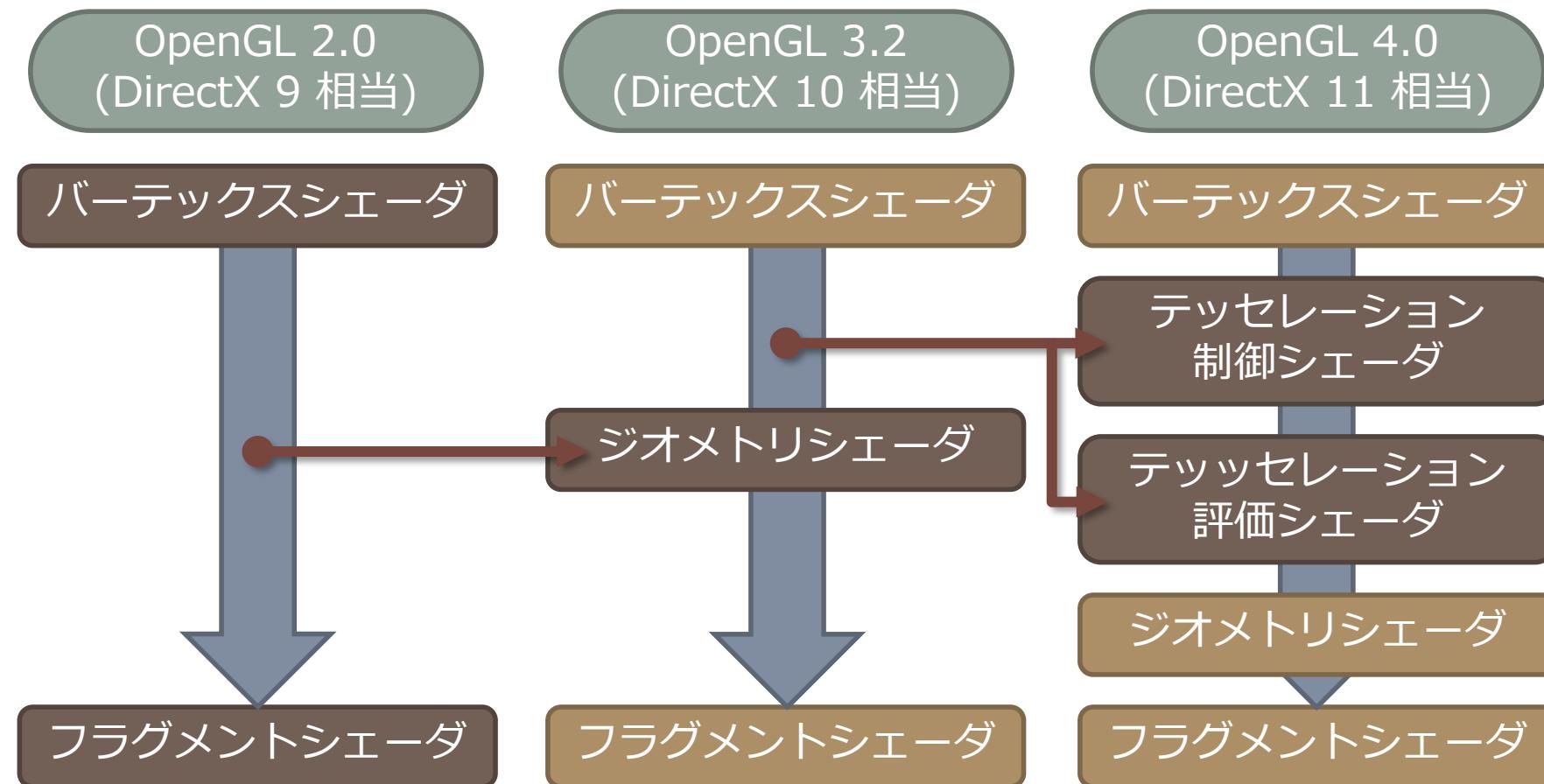
Real-Time Rendering (4th Ed.)



OpenGL Shading Language

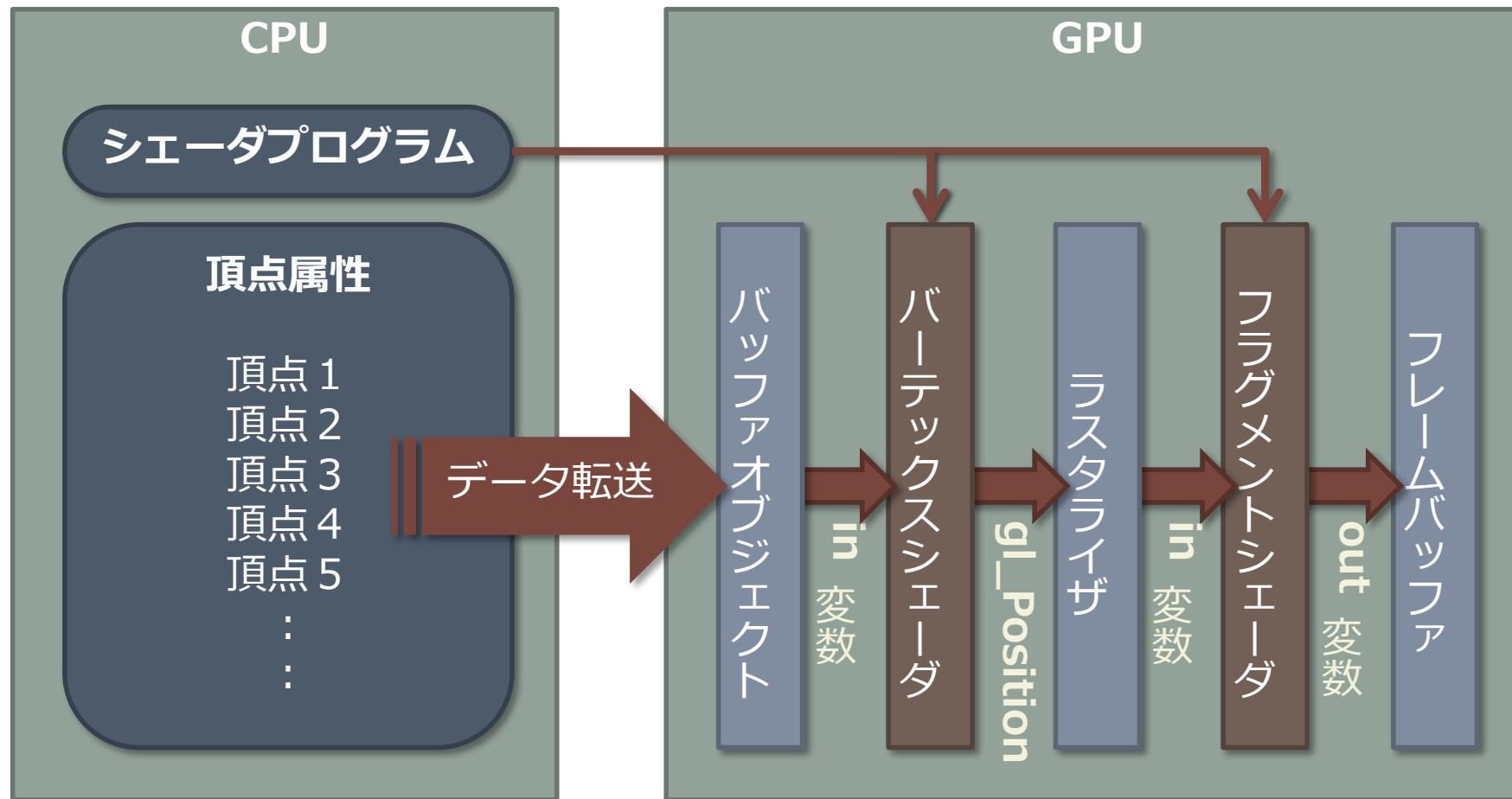


シェーダプログラムの種類



Vulkan (OpenGL Next Generation) / DirectX 12 / METAL は低レベルから作り直した

先にシェーダプログラムと頂点属性を GPU に転送



バーテックスシェーダのソースプログラム

```
#version 410

// シェーダの入力変数の宣言
in vec4 pv;

// バーテックスシェーダのエントリポイント
void main(void)
{
    gl_Position = pv;
}
```

バーテックスシェーダ
に入力される頂点属性

頂点ごとに実行される

#version 410

- GLSL version 4.1 を使用する宣言

in

- シェーダの入力変数の宣言

vec4

- 単精度実数 (float) 型の 4 つの要素を持つベクトルのデータ型

pv

- CPU から頂点属性を受け取るユーザ定義変数

main

- シェーダプログラムのエントリポイント

gl_Position

- バーテックスシェーダの出力先の組み込み変数

GLSL のデータ型

スカラー	ベクトル ($n=2\sim 4$)	要素のデータ型
bool	<code>bvecn</code>	論理型、true / false
int	<code>ivecn</code>	符号付き 32bit 整数
uint	<code>uvecn</code>	符号なし 32bit 整数
float	<code>vecn</code>	单精度実数
double	<code>dvecn</code>	倍精度実数

ベクトルの要素へのアクセス

メンバー	要素	vec4 pv; のとき	取り出される値 (Swizzling)
.x .s .r	第 1 要素	pv.xy	pv の第 1、第 2 要素からなる vec2 型の値
.y .t .g	第 2 要素	pv.rgb	pv の第 1、第 2、第 3 要素からなる vec3 型の値
.z .p .b	第 3 要素	pv.q	pv の第 4 要素の float 型の値
.w .q .a	第 4 要素	pv.yx	pv の第 1、第 2 要素の順序を入れ替えた vec2 型の値
		pv.brg	pv の第 1、第 2、第 3 要素を第 3、第 1、第 2 の順にした vec3 型の値

バーテックスシェーダーの入出力

- **in 変数 (attribute 変数)**

- 頂点属性（情報、座標値等）を得る
- CPU 側のプログラムから値を設定する
- バーテックスシェーダからは読み出しのみ

- **vec4 gl_Position**



これに値を設定することが
バーテックスシェーダの仕事

- 描画する図形の頂点位置を代入する
- バーテックスシェーダで読み書き可能
- クリッピングの対象になる

- **out 変数 (varying 変数)**

- 次のステージに送る頂点位置以外のデータ

フラグメントシェーダのソースプログラム

```

#version 410

// シェーダの出力変数
out vec4 fc;

// フラグメントシェーダのエントリポイント
void main(void)
{
    fc = vec4(1.0, 0.0, 0.0, 1.0);
}

```

カラー バッファに
出力する画素の色

画素ごとに実行される

カラー バッファに
結合されている

out

- シェーダの出力変数の宣言

fc

- フラグメントシェーダの出力先（カラー バッファの画素）に使うユーザ定義変数

vec4(…)

- （…）内のデータの vec4 型への明示的な型
変換（キャスト）

フラグメントシェーダの入出力

- **in 変数 (varying 変数)**
 - 前のステージから受け取るデータが格納されている
 - バーテックスシェーダからラスタライザを介して送られてくるのは頂点属性の**補間値**
- **vec4 gl_FragCoord**
 - 画素の画面上の位置 (gl_Position の補間値) を格納している組み込み変数, `read only`
- **vec4 gl_FragDepth**
 - 画素のデプス値を格納する組み込み変数, 初期値は `gl_FragCoord.z`
 - デプス値を代入して隠面消去処理を制御できる
- **out 変数**
 - 次のステージに送るデータを代入する
 - フラグメントシェーダではフレームバッファのカラーバッファに結合されている



これに値を設定することが
フラグメントシェーダの仕事

あるいは破棄
(`discard`)

シェーダプログラムのコンパイル手順

1. バーテックスシェーダのシェーダオブジェクトを作成する
 - GLuint `vertShader` = **glCreateShader(GL_VERTEX_SHADER);**
2. バーテックスシェーダのソースプログラムを読み込む
 - **glShaderSource(vertShader, lines, source, &length);**
3. バーテックスシェーダのソースプログラムをコンパイルする
 - **glCompileShader(vertShader);**
4. フラグメントシェーダのシェーダオブジェクトを作成する
 - GLuint `fragShader` = **glCreateShader(GL_FRAGMENT_SHADER);**
5. フラグメントシェーダソースプログラムを読み込む
 - **glShaderSource(fragShader, lines, source, &length);**
6. フラグメントシェーダのソースプログラムをコンパイルする
 - **glCompileShader(fragShader);**

シェーダプログラムのリンク手順

1. プログラムオブジェクトを作成する

- GLuint **program** = **glCreateProgram()**;

2. プログラムオブジェクトにシェーダオブジェクトを取り付ける

- **glAttachShader**(**program**, **vertShader**);
- **glAttachShader**(**program**, **fragShader**);

3. シェーダオブジェクトはもういらないので削除する

- **glDeleteShader**(**vertShader**);
- **glDeleteShader**(**fragShader**);

4. シェーダプログラムをリンクする

- **glLinkProgram**(**program**);

宿題ではこれらをまとめた
createProgram()
という関数を用意しています

バーテックスシェーダの in 変数の準備

- ・バーテックスシェーダの in 変数は **index** (番号) で識別する
- ・`glLinkProgram()` の前に変数名に **index** を割り当てる
 - ・ **glBindAttribLocation(program, 0, "pv");**
 - ・ `glLinkProgram(program);`
 - ・ 頂点属性を入力する in 変数 **pv** の **index** を **0** に設定してリンクする
 - ・ `glLinkProgram()` の後に変数の **index** を調べる方法もある
 - ・ `glLinkProgram(program);`
 . . .
 - ・ GLint **pvLoc** = **glGetAttribLocation(program, "pv");**
 - ・ `glBindAttribLocation()` を使わなければ **index** は自動的に割り振られる
 - ・ 頂点属性の入力に用いる in 変数 **pv** の **index** を **pvLoc** に求める

フラグメントシェーダの out 变数の準備

- フラグメントシェーダの out 变数にはフラグメントデータを出力するカラーバッファの番号を指定する
- glLinkProgram() の前に変数名に番号を割り当てる
 - **glBindFragDataLocation(program, 0, "fc");**
 - glLinkProgram(program);
 - フラグメントデータを出力する変数 fc に 0 番のバッファを指定する
- 番号とカラーバッファは glDrawBuffers() で対応付ける
 - GLenum buffers[] = { GL_BACK_LEFT, GL_BACK_RIGHT };
 - **glDrawBuffers(2, buffers);**
 - 0 番に GL_BACK_LEFT, 1 番に GL_BACK_RIGHT が対応付けられる
 - デフォルトは 0 番が GL_BACK_LEFT

index をシェーダのソースで指定することもできる

- OpenGL 3.3 / GLSL 3.30 以降の機能
 - それ以前では GL_ARB_explicit_attrib_location 拡張機能

バーテックスシェーダ

```
#version 150 core
#extension GL_ARB_explicit_attrib_location: enable
layout (location = 0) in vec4 pv;
layout (location = 1) in vec4 nv;
...
```

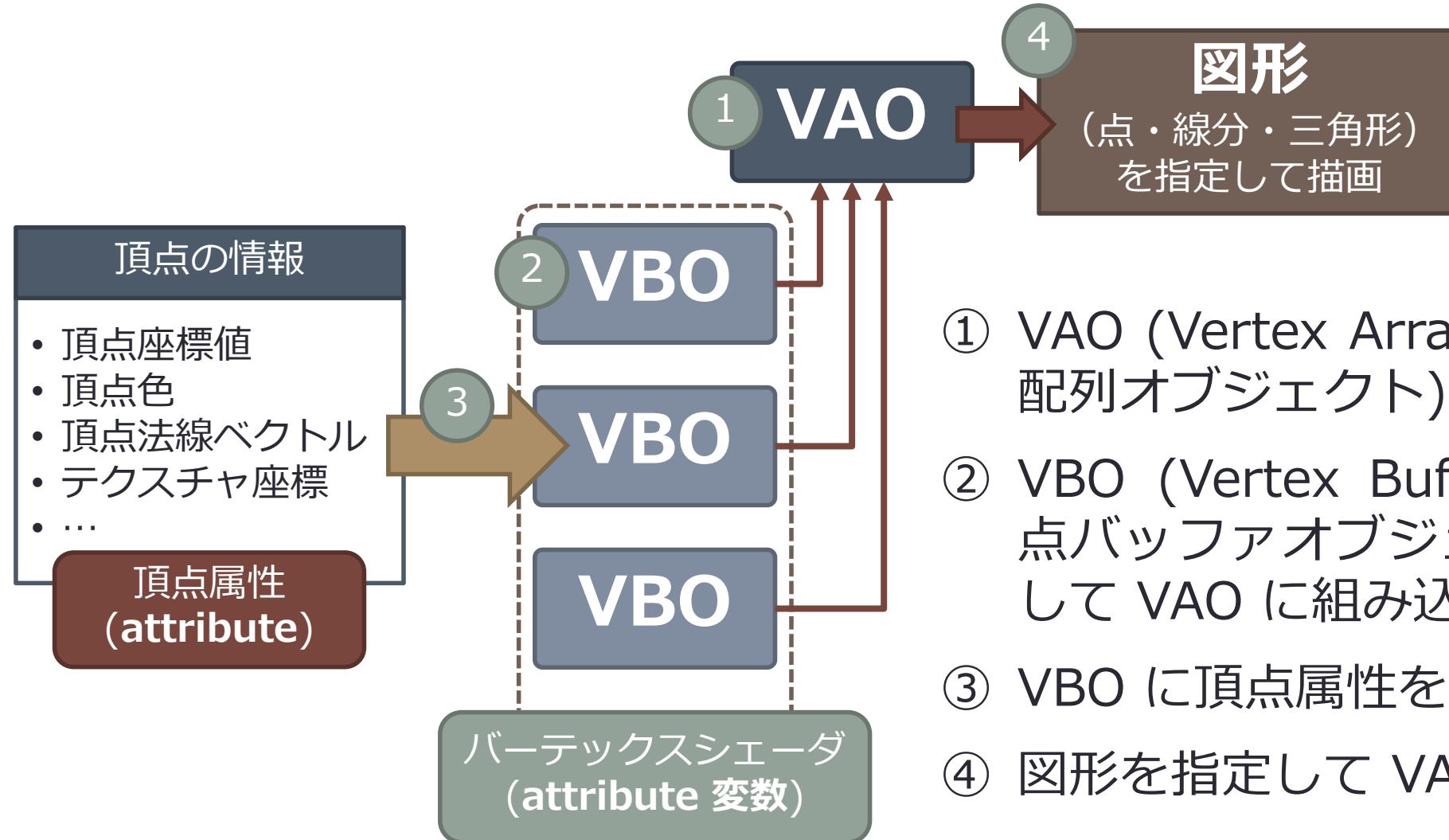
フラグメントシェーダ

```
#version 150 core
#extension GL_ARB_explicit_attrib_location: enable
layout (location = 0) out vec4 fc;
layout (location = 1) out vec4 fv;
...
```

図形の描画手順

頂点属性と基本図形

OpenGL による図形描画の手順



VAO を作って VBO を組み込み VAO を指定して描画

① VAO (Vertex Array Object, 頂点配列オブジェクト) の準備

1. VAO を作成する
2. VAO を結合する

② VBO (Vertex Buffer Object, 頂点バッファオブジェクト) の準備

1. VBO を作成する
2. VBO を結合する (これにより現在結合している VAO に組み込まれる)
3. VBO のメモリを確保し頂点属性を転送する
4. バーテックスシェーダの in 変数の index に VBO を結合する
5. バーテックスシェーダの in 変数の index を有効にする

③ VBO に頂点属性 (attribute) を転送する

④ VAO を指定して図形を描画する

VAO (Vertex Array Object) の準備

1. N 個の VAO を作成する
 - GLuint `vao[N];`
 - `glGenVertexArrays(N, vao);`
2. i 番目の VAO を結合する
 - `glBindVertexArray(vao[i]);`

VBO (Vertex Buffer Object) の準備

1. N 個の VBO を作成する
 - `GLuint vbo[N];`
 - `glGenBuffers(N, vbo);`
2. i 番目の VBO にメモリを確保して頂点属性データを転送する
 - `glBindBuffer(GL_ARRAY_BUFFER, vbo[i]);`
 - `glBufferData(GL_ARRAY_BUFFER, size, data, usage);`
3. バーテックスシェーダの `in` 変数の `index` に VBO を割り当てる
 - `glVertexAttribPointer(index, size, type, normalized, stride, pointer);`
4. バーテックスシェーダの `in` 変数の `index` を有効にする
 - `glEnableVertexAttribArray(index);`

glBufferData() の引数 *size* と *data*

- *size*
 - GPU 上に確保するバッファオブジェクトのサイズ
 - 配列 *p* を全部送るなら `sizeof p`
- *data*
 - 確保したバッファオブジェクトに送るデータ
 - 配列 *p* のポインタ
 - `NULL` ならデータを転送しない（バッファオブジェクトの確保のみ行う）

glBufferData() の引数 *usage*

- バッファオブジェクトの使われ方を指定する
 - 性能を最適化するため
 - `GL_XXXX_YYYY` の形式の定数 (`GL_STATIC_DRAW` など)

XXXX (アクセスの頻度)		YYYY (アクセスの性質)	
STATIC	データは1回の変更に対して何回も使用される	DRAW	バッファ上のデータはアプリケーションから変更され描画に使用される
STREAM	データは1回から数回使用されるごとに1回変更される	READ	バッファ上のデータはGLからの読み出しにより変更されアプリケーションからの問い合わせ時にそのデータを返すために使用される
DYNAMIC	データは何回も変更され頻繁に使用される	COPY	バッファ上のデータはGLからの読み出しにより変更され描画や画像に関連したコマンドのソースとして使用される

配列変数に格納された頂点属性を VBO に転送する例

```
// 頂点属性は3次元 (x, y, z) の位置データ 1,000 個
GLfloat position[1000][3];
```

(中略)

// ①頂点配列オブジェクトを一つ作る

```
GLuint vao;
 glGenVertexArrays(1, &vao);
 glBindVertexArray(vao);
```

// ②頂点バッファオブジェクトを一つ作る

```
GLuint vbo;
 glGenBuffers(1, &vbo);
 glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

// ③頂点バッファオブジェクトに頂点属性を転送する

```
glBufferData(GL_ARRAY_BUFFER, sizeof position, position, GL_STATIC_DRAW);
```

VBO と attribute 変数の対応付け

1. バーテックスシェーダの `in` 変数の `index` を VBO に対応付ける

- **glVertexAttribPointer**(`index`, `size`, `type`, `normalized`, `stride`, `pointer`);
 - `size`: 一つの頂点データの要素数
 - `type`: 頂点データのデータ型
 - `normalized`: `GL_TRUE` なら固定小数点データを正規化する
 - `stride`: データの間隔
 - `pointer`: 頂点属性が格納されている場所の VBO の先頭からの位置
 - 引数 `pointer` はバイト数を `GLubyte *` 型に変換して設定する (`BUFFER_OFFSET` マクロ、後述)

2. バーテックスシェーダの `in` 変数の `index` を有効にする

- **glEnableVertexAttribArray**(`index`);

VBO をバーテックスシェーダの in 変数に結び付ける例

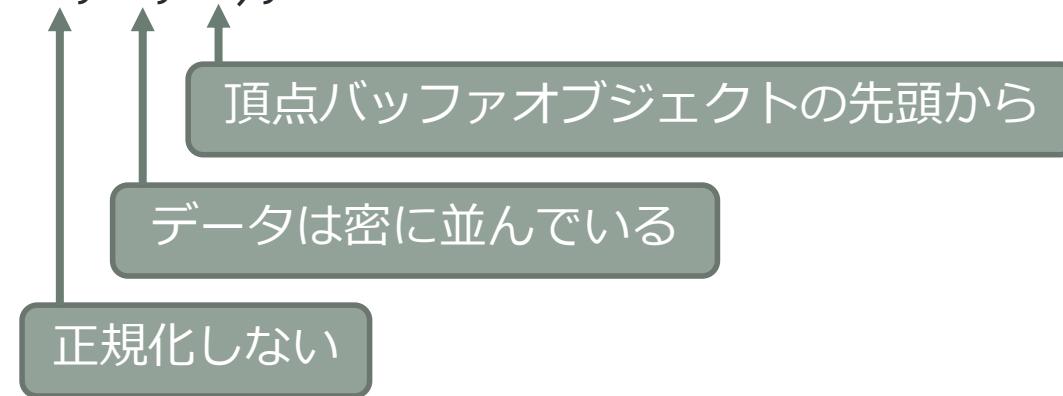
```
// 頂点属性は3次元 (x, y, z) の位置データ 1,000 個  
GLfloat position[1000][3];
```

(中略)

```
// ③頂点バッファオブジェクトに頂点属性を転送する  
glBufferData(GL_ARRAY_BUFFER, sizeof position, position, GL_STATIC_DRAW);
```

```
// 場所が 0 の in 変数で現在の頂点バッファオブジェクトから 3 次元の GLfloat 型のデータを取り出す  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

```
// 場所が 0 の in 変数を有効にする  
glEnableVertexAttribArray(0);
```



図形の描画

1. 使用するシェーダプログラムを選ぶ

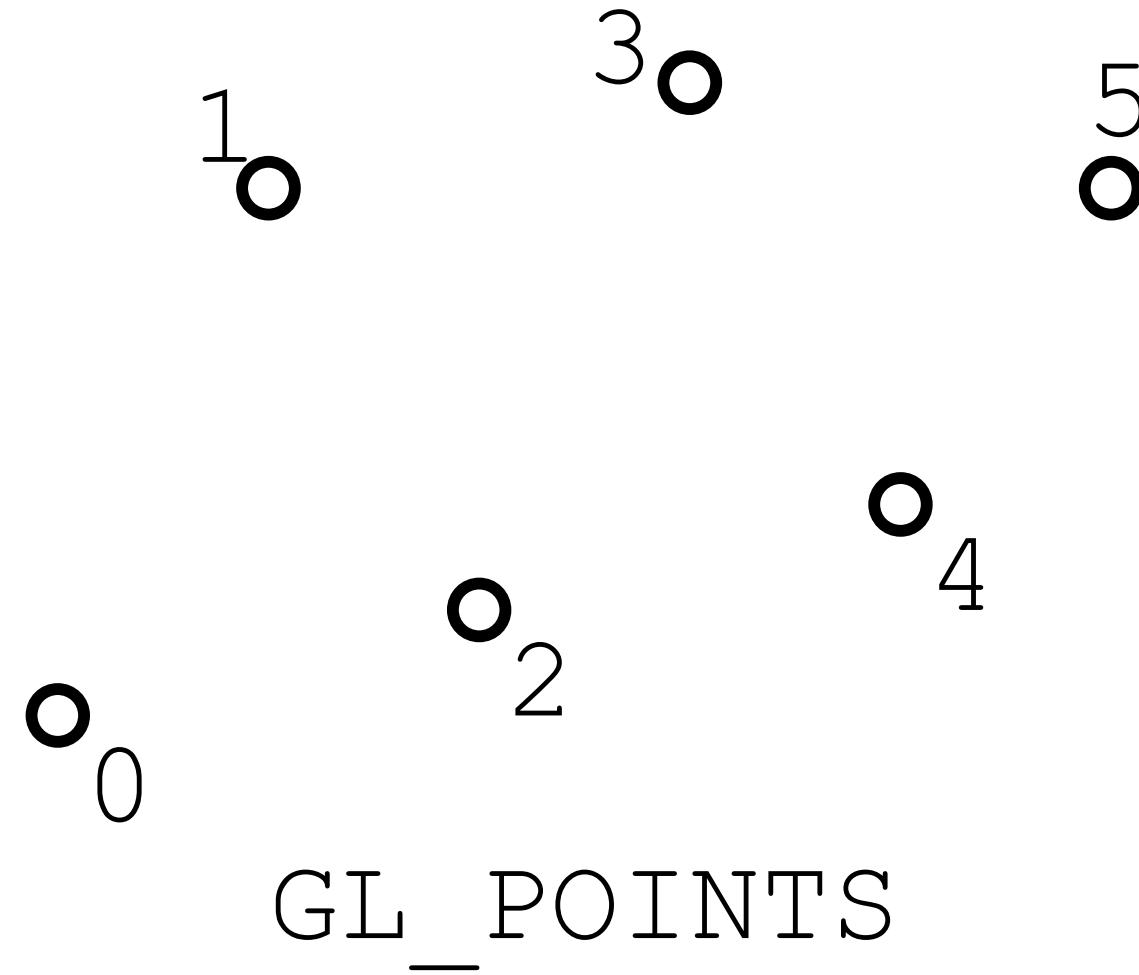
- **glUseProgram(program);**

2. i 番目の VAO を図形を描画する

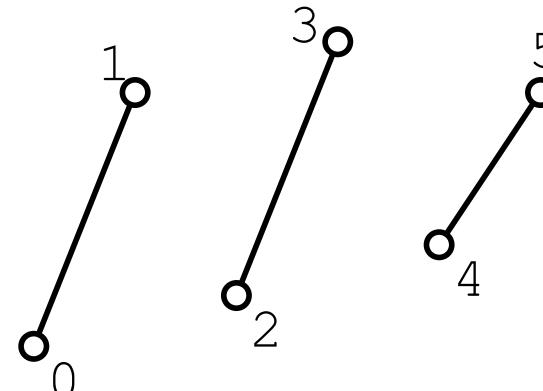
- **glBindVertexArray(vao[i]);**
- **glDrawArrays(mode, first, count);**

- $mode$: 描画する**基本図形**の種類
- $first$: 描画する頂点データの先頭の番号
- $count$: 描画する頂点データの数

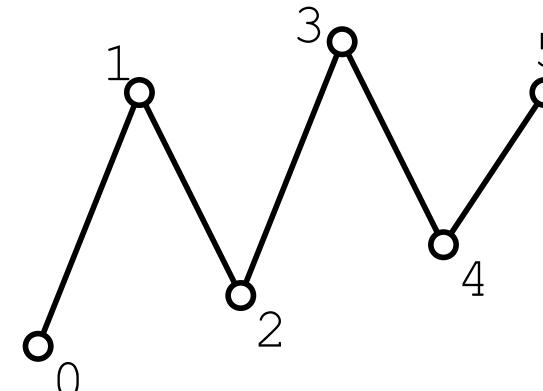
基本図形 – 点



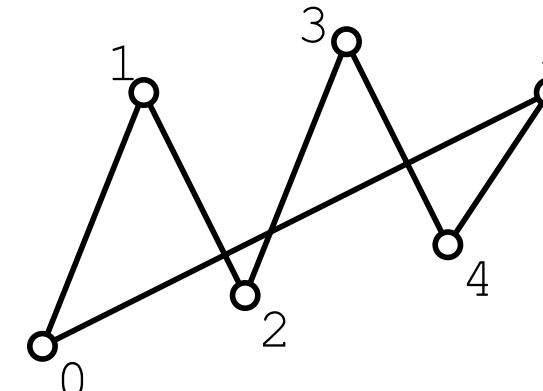
基本図形 – 線分と三角形



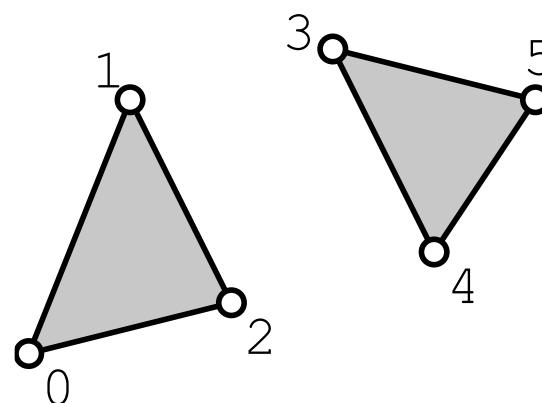
GL_LINES



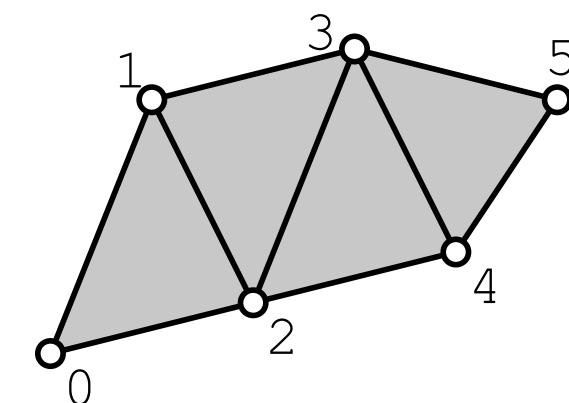
GL_LINE_STRIP



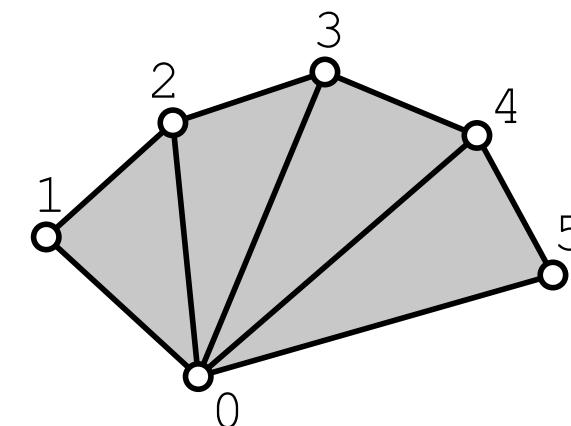
GL_LINE_LOOP



GL_TRIANGLES

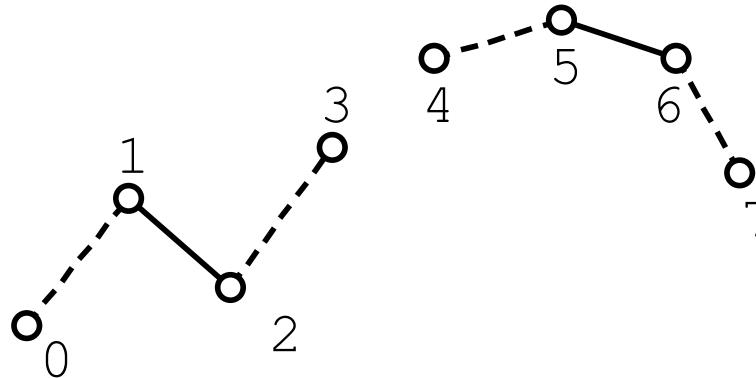


GL_TRIANGLE_STRIP

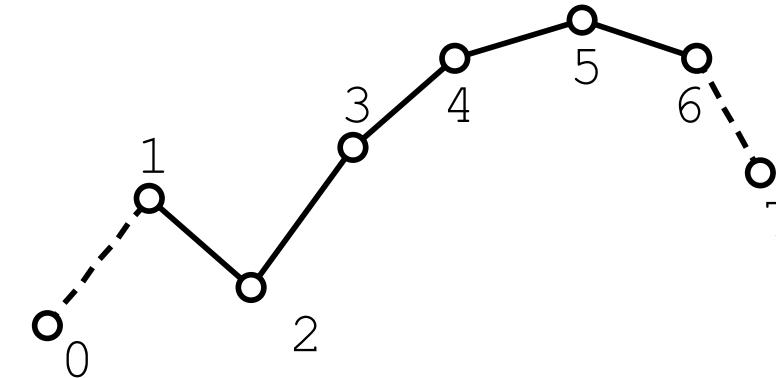


GL_TRIANGLE_FAN

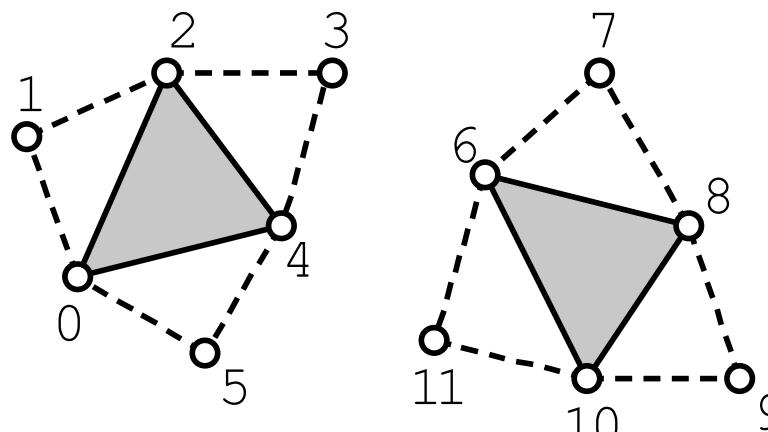
ジオメトリシェーダで追加されたもの



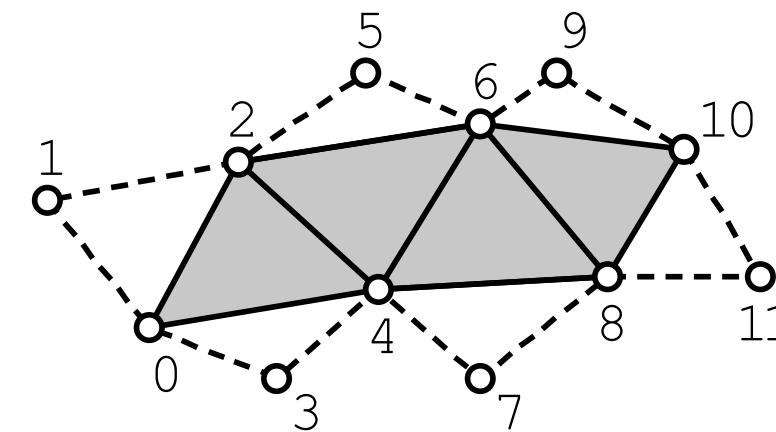
GL_LINES_ADJACENCY



GL_LINE_STRIP_ADJACENCY



GL_TRIANGLES_ADJACENCY

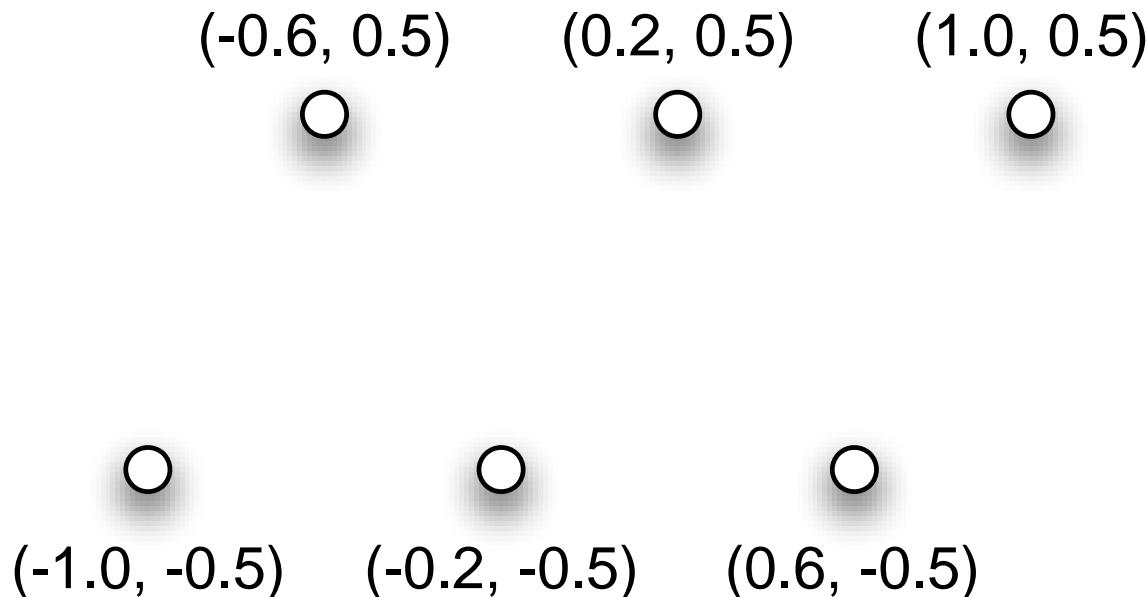


GL_TRIANGLE_STRIP_ADJACENCY

頂点配列による図形描画

頂点属性だけを使う

頂点属性データの例



```
// 頂点位置
static GLfloat position[][2] =
{
    { -1.0f, -0.5f },
    { -0.6f,  0.5f },
    { -0.2f, -0.5f },
    {  0.2f,  0.5f },
    {  0.6f, -0.5f },
    {  1.0f,  0.5f },
};
```

頂点配列オブジェクトの準備

```
// 頂点配列オブジェクトを作成する
```

```
GLuint vao;
```

```
glGenVertexArrays(1, &vao);
```

```
glBindVertexArray(vao);
```

```
// 頂点バッファオブジェクトを作成する
```

```
GLuint vbo;
```

```
glGenBuffers(1, &vbo);
```

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

```
glBufferData(GL_ARRAY_BUFFER, sizeof position, position, GL_STATIC_DRAW);
```

```
// 結合されている頂点バッファオブジェクトを in 変数から参照する
```

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
```

```
glEnableVertexAttribArray(0);
```

頂点属性が転送される

in 変数の index

1 頂点あたりのデータ数 (次元)

glDrawArrays() による描画

```
// シェーダプログラムを選択する  
glUseProgram(program);  
  
// 描画する頂点配列オブジェクトを選択する  
glBindVertexArray(vao);  
  
// 図形を描画する  
glDrawArrays(GL_POINTS, 0, 6);
```

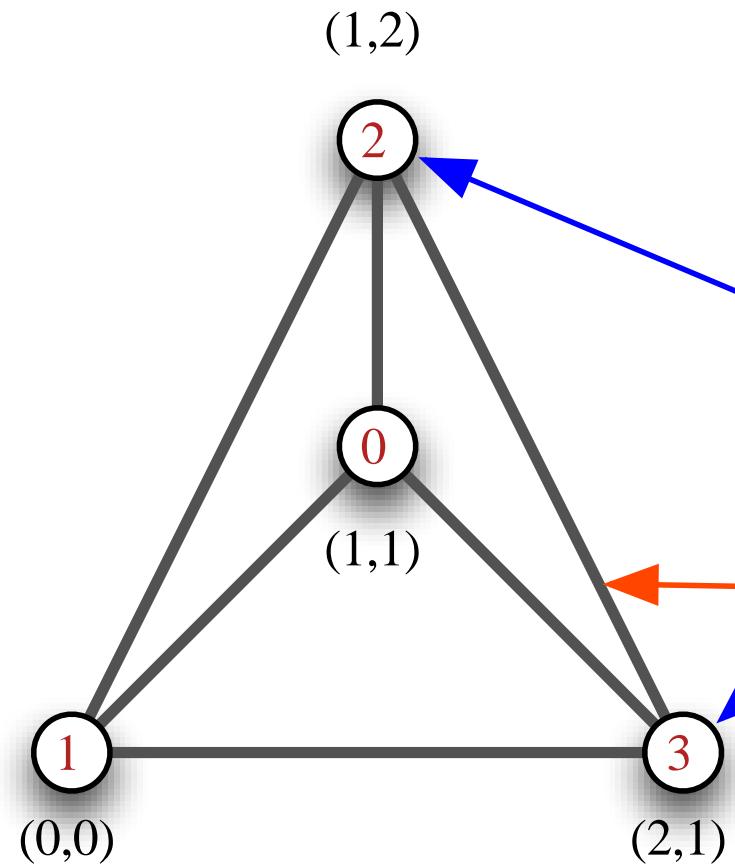
描画する最初の頂点番号

頂点の数

頂点インデックスを用いた図形描画

描画する頂点属性をインデックスで指定する

頂点インデックスを使った図形の表現



頂点属性
(位置)

番号	x	y
0	1	1
1	0	0
2	1	2
3	2	1

インデックス
(線分)

始点	終点
0	1
0	2
0	3
1	2
2	3
3	1

頂点インデックスのバッファオブジェクト

- ・頂点インデックスも GPU に送る必要がある
 - ・もうひとつバッファオブジェクトを使う
- ・配列 `edge` に格納された頂点のインデックスを j 番目のバッファオブジェクトに転送する
 - ・`glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[j]);`
 - ・`glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof edge, edge, usage);`
- ・VAO に頂点属性を格納した VBO と一緒に登録する

頂点インデックスを使った図形の描画

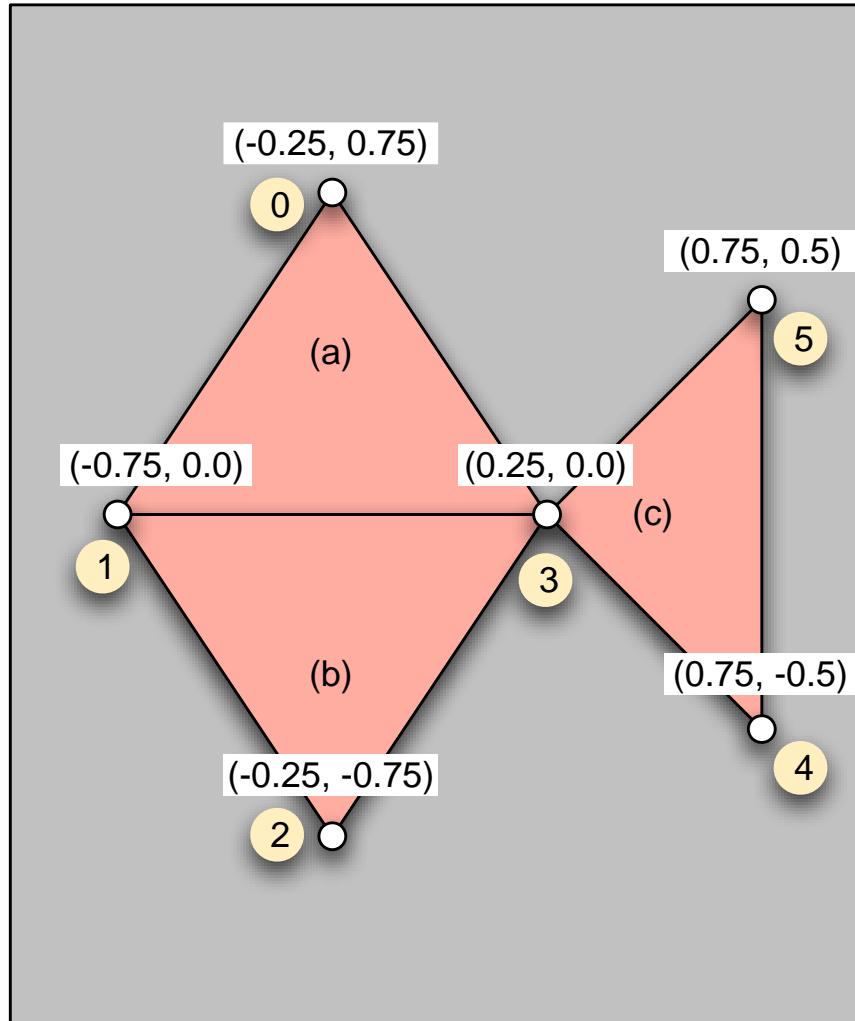
1. 使用するシェーダプログラムを選ぶ

- `glUseProgram(program);`

2. i 番目の VAO を図形を描画する

- `glBindVertexArray(vao[i]);`
- **`glDrawElements(mode, count, type, indices);`**
 - *mode*: 描画する**基本図形**の種類
 - *count*: 描画する頂点データの数
 - *type*: *indices*のデータ型 (VBO に格納したインデックスのデータ型)
 - *indices*: VBO 内で頂点インデックスが格納されている場所
 - 引数 *indices*はバイト数を `GLubyte *` 型に変換して設定する (`BUFFER_OFFSET` マクロ)

頂点インデックスを使った図形データ



```
static GLfloat position[][2] =
{
    { -0.25f,  0.75f }, // (0)
    { -0.75f,  0.0f  }, // (1)
    { -0.25f, -0.75f }, // (2)
    {  0.25f,  0.0f  }, // (3)
    {  0.75f,  0.5f  }, // (4)
    {  0.75f, -0.5f  } // (5)
};

static GLuint index[] =
{
    0, 1, 3, // (a)
    1, 2, 3, // (b)
    3, 4, 5 // (c)
};
```

頂点インデックス

頂点配列オブジェクトの準備

```
// 頂点配列オブジェクトを作成する
GLuint vao;
 glGenVertexArrays(1, &vao);
 glBindVertexArray(vao);
```

頂点属性だけを使う場合と同じ

```
// 頂点バッファオブジェクトを作成する
GLuint vbo;
 glGenBuffers(1, &vbo);
 glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof position, position, GL_STATIC_DRAW);
```

```
// 結合されている頂点バッファオブジェクトを in 変数から参照する
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(0);
```

インデックスのバッファオブジェクト追加

```
// インデックスのバッファオブジェクト
GLuint ibo;
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof index, index, GL_STATIC_DRAW);
```

glDrawElements() による描画

```
// シェーダプログラムの選択  
glUseProgram(program);  
  
// 図形を描画する  
glBindVertexArray(vao);  
glDrawElements(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0);
```

インデックス
配列の要素数

インデックス
配列のデータ型

バッファオブジェクトの先頭

動的な描画

バッファオブジェクトの内容の更新

バッファオブジェクトのデータの更新方法

- 既存のバッファオブジェクトにあとからデータを転送する
 - glBufferSubData**(*target*, *offset*, *size*, *data*);
 - target*: GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER
 - offset*: 転送先のバッファオブジェクトの先頭位置
 - size*: 転送するデータのサイズ
 - data*: 転送するデータ
- バッファオブジェクトからデータを取得することもできる
 - glGetBufferSubData**(*target*, *offset*, *size*, *data*);
 - target*, *offset*, *size*, *data*: 同上 (データの転送方向が反対になるだけ)
- バッファオブジェクトを CPU 側のメモリ空間にマップして読み書きできる
 - void *glMapBuffer**(*target*, *access*);
 - target*: 同上
 - access*: GL_READ_ONLY, GL_WRITE_ONLY, GL_READ_WRITE

glMapBuffer()

```
// GPU 上の頂点バッファオブジェクトをアプリケーションのメモリとして参照できるようにする
GLfloat (*p)[4] = (GLfloat (*)[4])glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);

// 8 番目のデータ p[7] にデータを設定する (access が GL_READ_ONLY 以外)
p[7][0] = 3.0f;
p[7][1] = 4.0f;
p[7][2] = 5.0f;
p[7][3] = 1.0f;

// 6 番目のデータ p[5] からデータを取り出す (access が GL_WRITE_ONLY 以外)
GLfloat x = p[5][0];
GLfloat y = p[5][1];
GLfloat z = p[5][2];
GLfloat w = p[5][3];

// アプリケーションのメモリ空間から頂点バッファオブジェクトを切り離す
glUnmapBuffer(GL_ARRAY_BUFFER);
```

インターリープな頂点属性

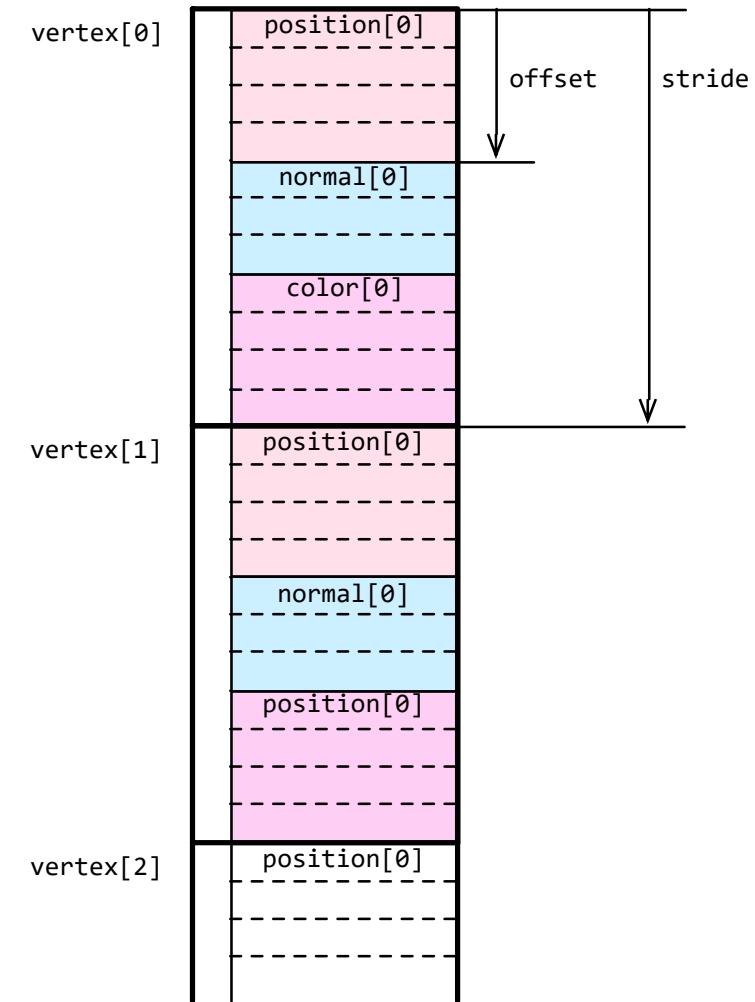
異なる頂点属性を一つのバッファに格納する

頂点属性をインターリープに配置する場合

```
struct Vertex
{
    GLfloat position[4]; // 位置
    GLfloat normal[3]; // 法線
    GLfloat color[4]; // 色
};
```

```
Vertex vertex[3];
```

- この頂点属性の stride は sizeof (Vertex)
- normal の offset は sizeof position
- color の offset は sizeof position + sizeof normal



この頂点属性を受け取るバーテックスシェーダの in 変数

```
#version 410

// シェーダの入力変数の宣言
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec4 color;

// バーテックスシェーダのエントリポイント
void main(void)
{
    gl_Position = position;
    ...
}
```

頂点配列オブジェクトの準備

```
// 頂点配列オブジェクトを作成する
GLuint vao;
 glGenVertexArrays(1, &vao);
 glBindVertexArray(vao);

// 頂点バッファオブジェクトを作成する
GLuint vbo;
 glGenBuffers(1, &vbo);
 glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof vertex, vertex, GL_STATIC_DRAW);
```

頂点オブジェクトを in 変数のインデックスに対応付ける

```
// 結合されている頂点バッファオブジェクトを in 変数から参照する

// この構造体のサイズ
constexpr GLsizei stride(static_cast<GLsizei>(sizeof (Vertex)));

// position
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, stride, 0));
glEnableVertexAttribArray(0);

// normal
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, stride, BUFFER_OFFSET(offsetof(Vertex, normal)));
glEnableVertexAttribArray(1);

// color
glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, stride, BUFFER_OFFSET(offsetof(Vertex, color)));
glEnableVertexAttribArray(2);
```

position は Vertex の先頭にあるので

offsetof() を使う場合は #include <cstddef>

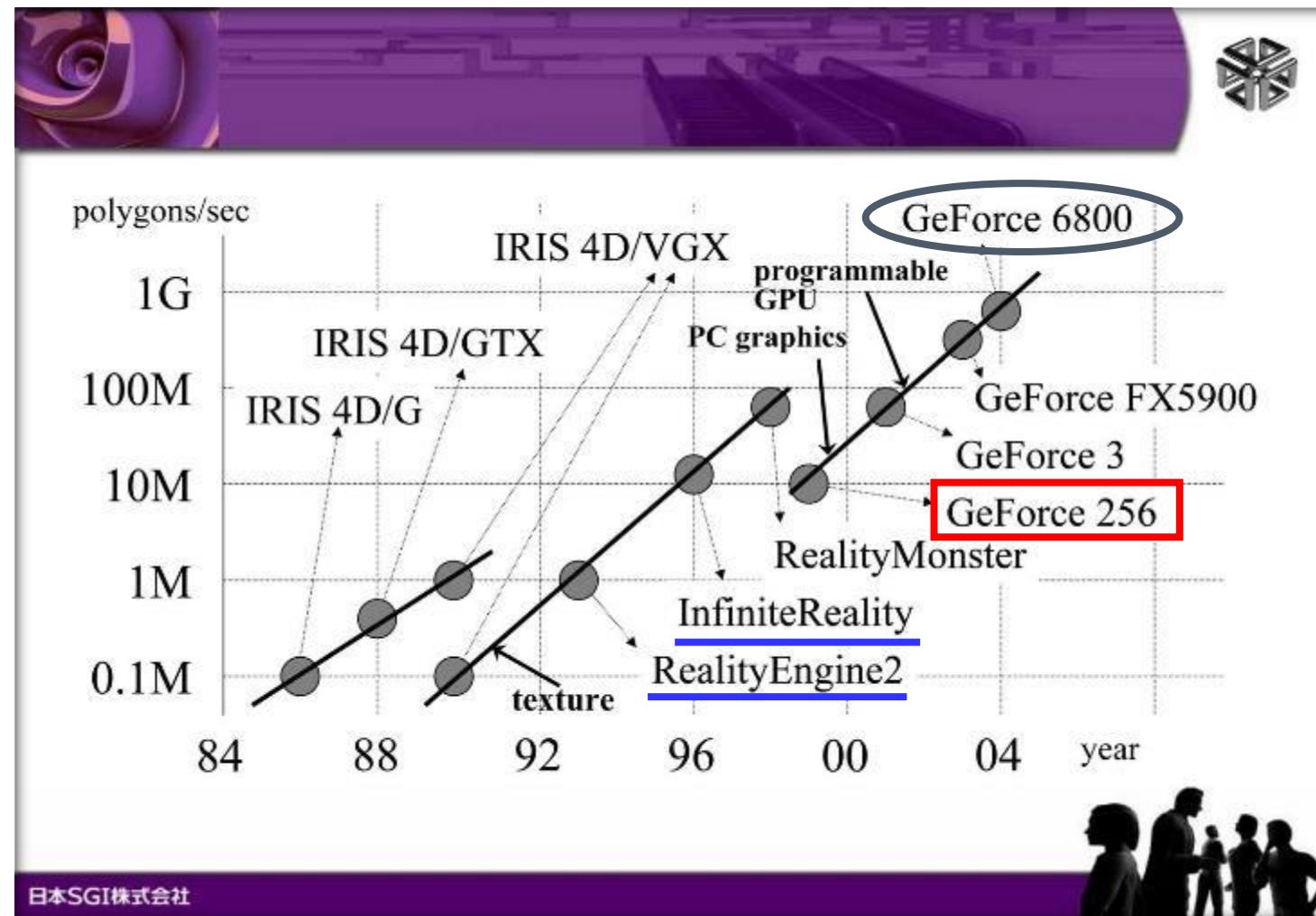
BUFFER_OFFSET(*bytes*) マクロ

- VBO の場合メモリは **GPU 側** にある
 - `glVertexAttribPointer(..., pointer);` の引数 `pointer` は CPU 側のメモリのポインタではない
 - `pointer` には `glBufferData()` で確保した GPU 上のメモリ領域の先頭からのオフセットを指定する必要がある
 - 引数 `bytes` をポインタと見なして `pointer` に渡す必要がある
- 引数 `bytes` の値をそのままポインタに変換する
 - `#define BUFFER_OFFSET(bytes) ((GLubyte *)0 + (bytes))`
 - 0 すなわち `NULL` を `GLubyte` 型のポインタに型変換 (キャスト)
 - それに整数値 `bytes` を足せば `byte` の値のポインタになる
- バッファオブジェクトの先頭から使うなら `bytes = 0`

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

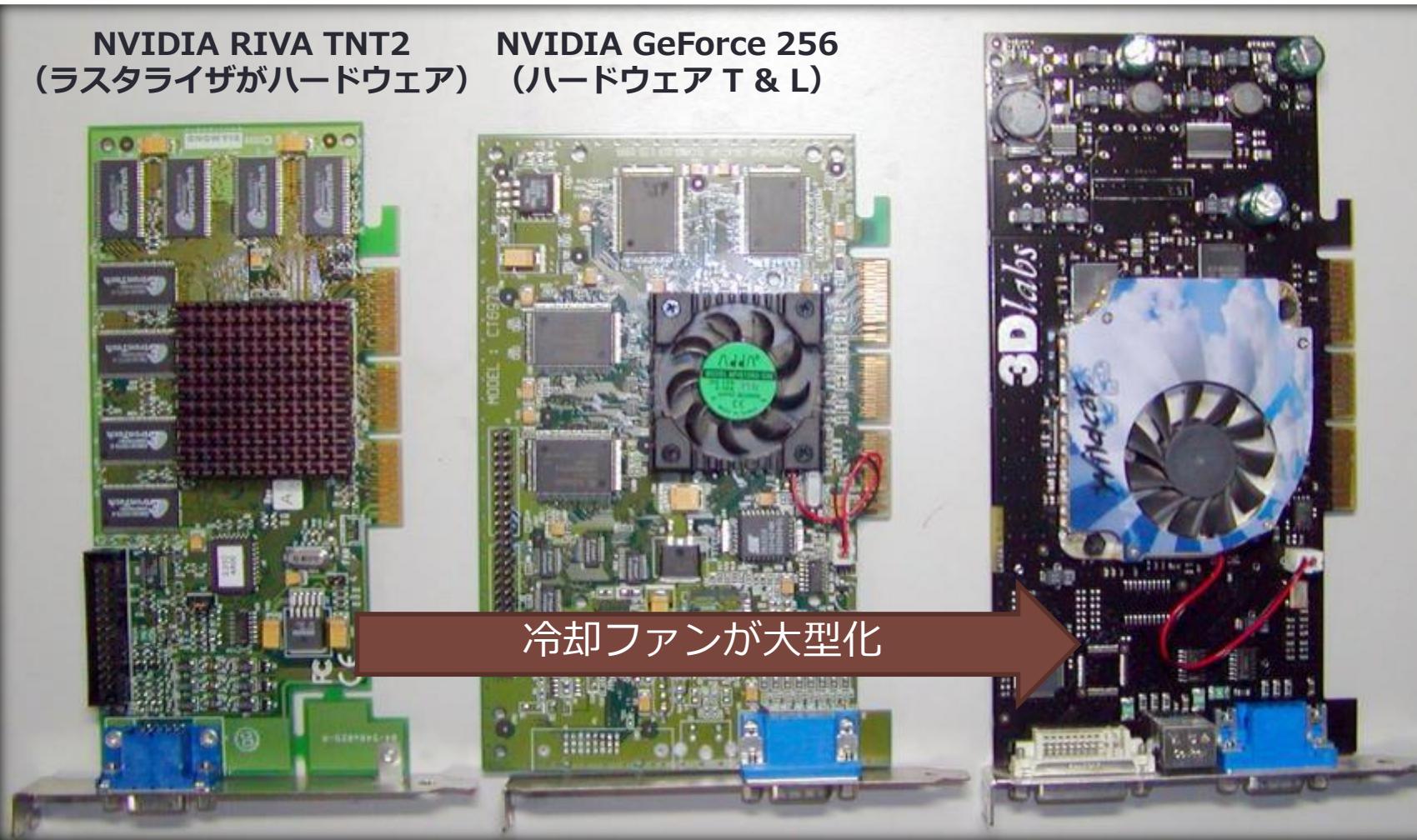


SGI Onyx



ビデオカード

3DLabs WildCat VP870
(プログラマブルシェーダ)



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
- これを **NVIDIA** は **GPU** (**Graphics Processing Unit**) と名付けた
 - 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** (**Visual Processing Unit**) と名付けたが普及しなかった

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
 - ・プログラム可能な高性能の画像処理装置
- ・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機械学習にも用いられている

最近の GPU

NVIDIA TITAN RTX



(298,000円, 280W)

高い
でかい
電気を食う

AMD Radeon VII



(90,000～100,000円, 300W)

最近の GPU の性能



Release date ≤ Q2 2014.

XT 6800 GS GT

Nvidia vs AMD - Top 5 Games

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



¥ 275,047

Release date ≈ Q4 2018.

Titan Black X Pascal Xp V RTX

Nvidia vs AMD - Top 5 Games

+547,402%

5,475倍以上

最近の GPU の性能



最新の GPU

NVIDIA RTX 3090



(293,700円, 350W)

AMD Radeon RX 6900 XT



(194,700円, 300W)

やっぱり高い
更にでかい
電気を食う

NVIDIA H100

GDEP Advance with you
株式会社ジーデップ・アドバンス

NVIDIA ELITE PARTNER

ご購入前の
お問い合わせ
受付時間：平日9:00～17:00

03-6803-0620

サイト内検索

ラインナップ ご購入ガイド 導入事例 評価用貸出機 カタログ 新着情報 セミナーイベント 企業情報 お問い合わせ



NVIDIA® H100 80GB

合計お見積金額
¥4,745,950

ご注文台数： 台

本体価格：	¥4,313,000
消費税：	¥431,300
配送料：	¥1,650
保証体制：	3年標準センドバック

正式見積もりを依頼

概算見積もりを印刷

※ジーデップならレンタルもお得です