

# ゲームグラフィックス特論

---

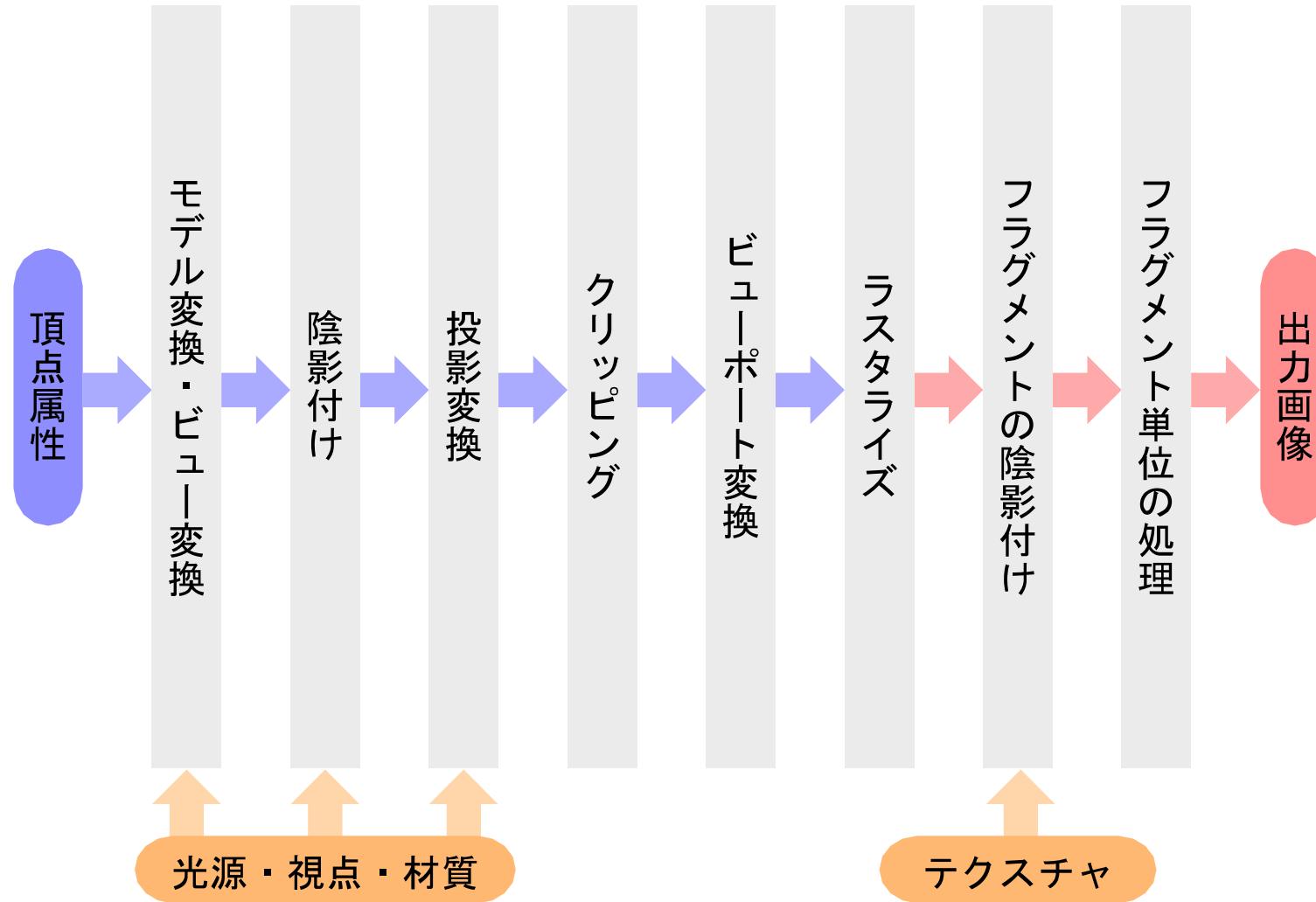
第2回 GPU (Graphics Processing Unit)

# 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

# レンダリングパイプラインは最初ソフトウェアで実装



つまり CPU で処理されていた



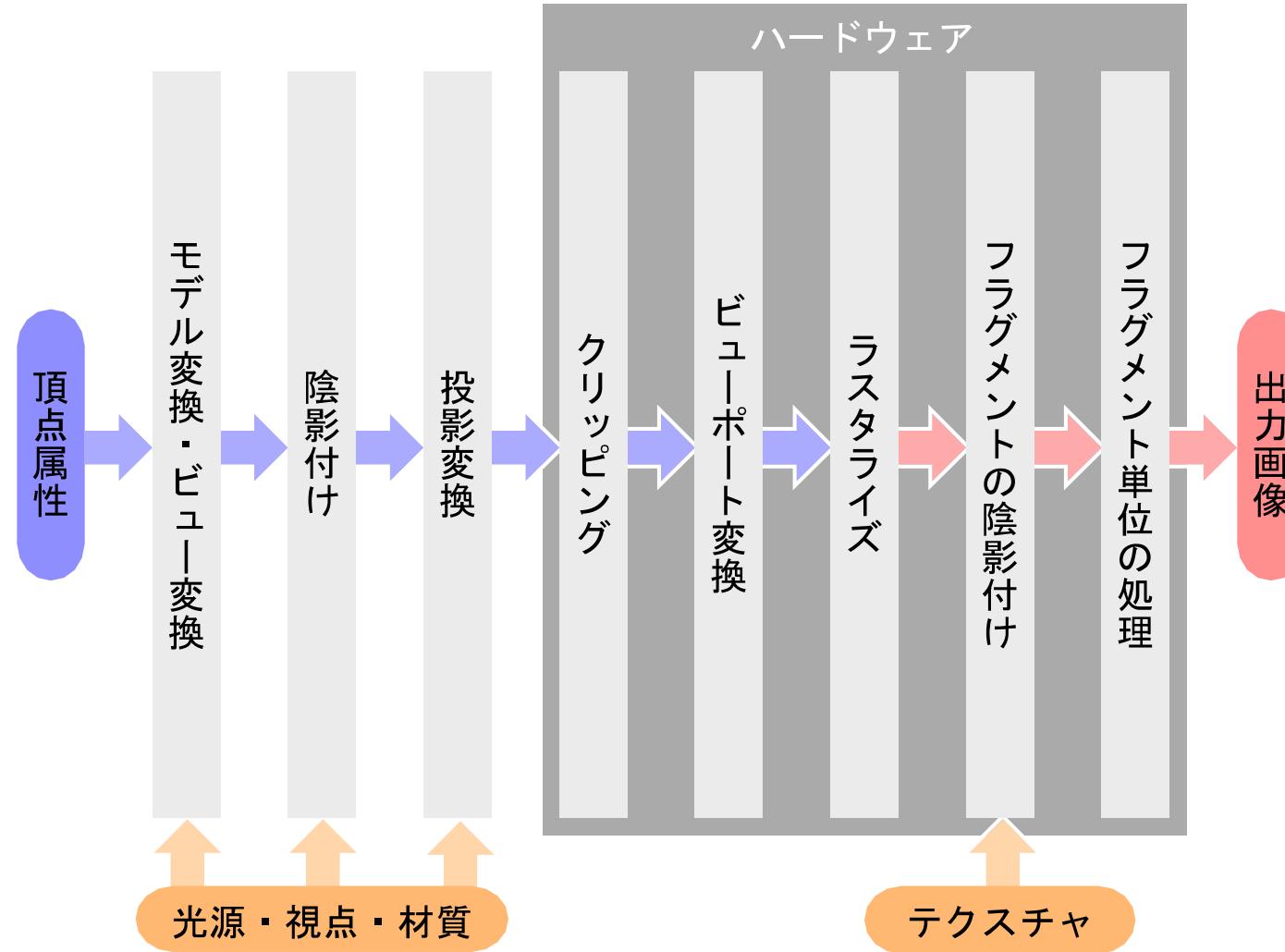
# レンダリングパイプラインのソフトウェア実装の問題

- ・単純な処理の大量に繰り返す
  - ・計算時間がかかる
- ・計算結果の格納先が CPU の外部にあるハードウェア
  - ・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

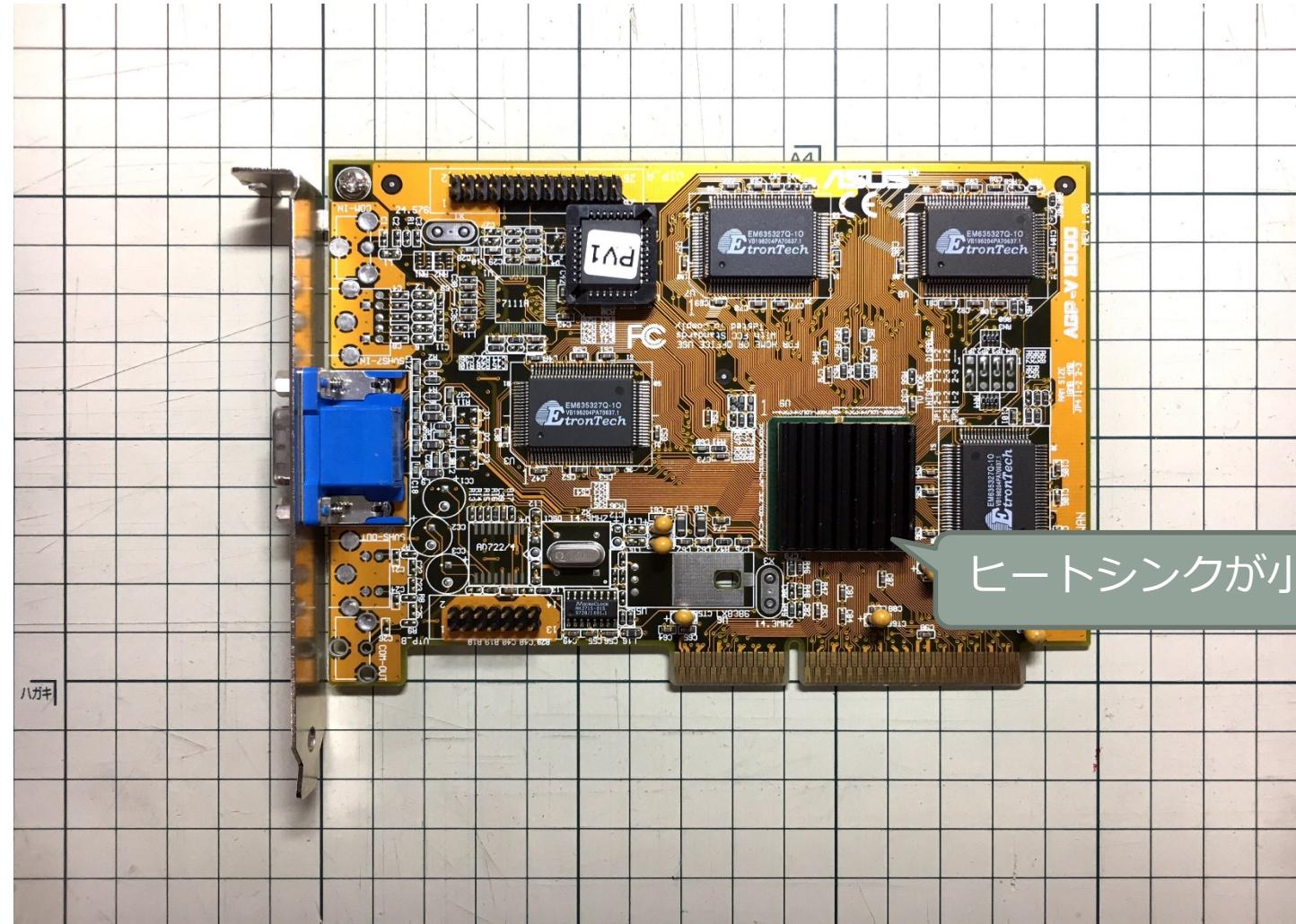


別のハードウェアに任せる

# 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)



ヒートシンクが小さい

# 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. 走査変換

- ・単純な整数の加減算とループ

## 4. 隠面消去

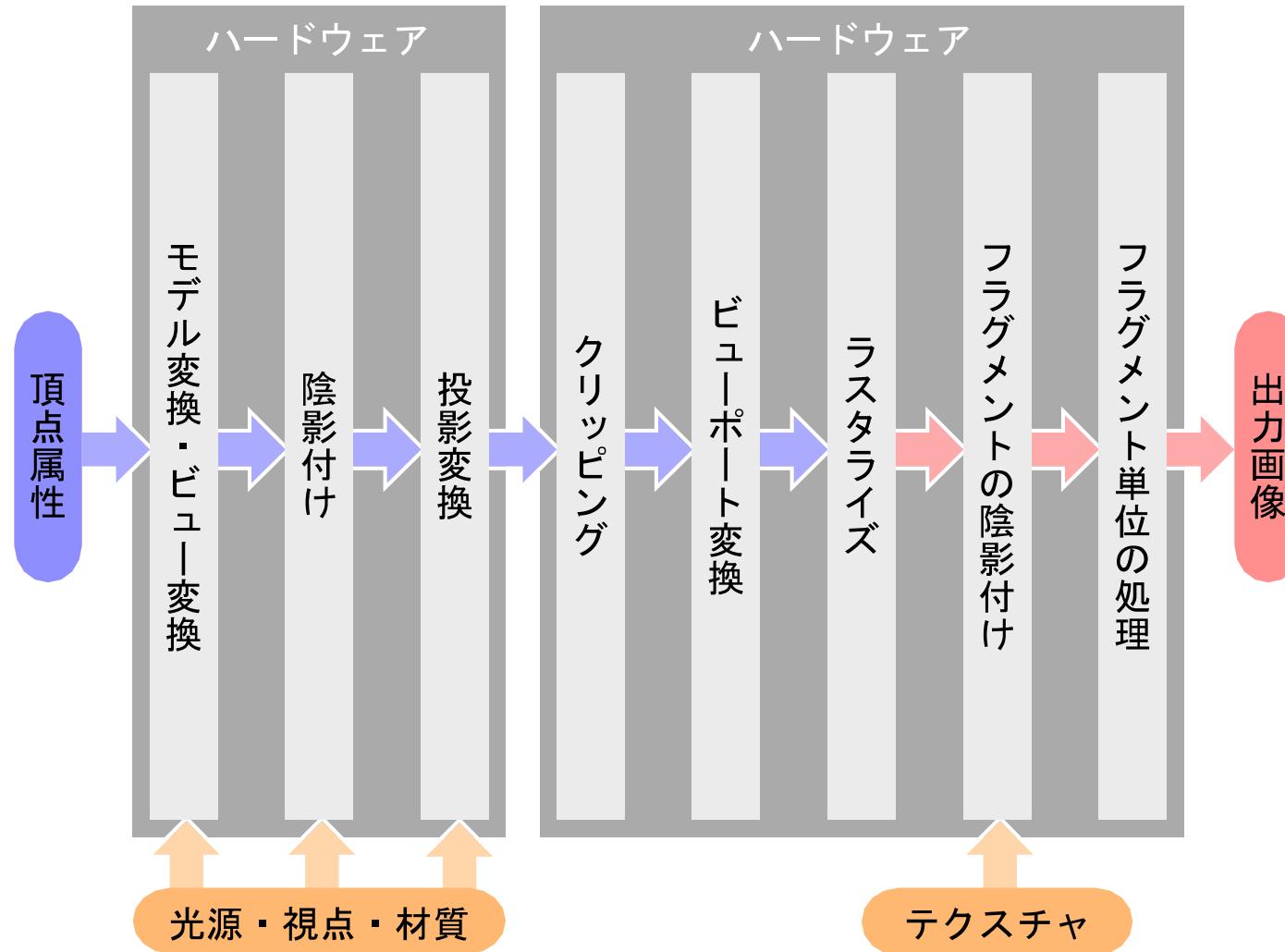
- ・デプス値の補間とデプスバッファとの比較による可視判定

## 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

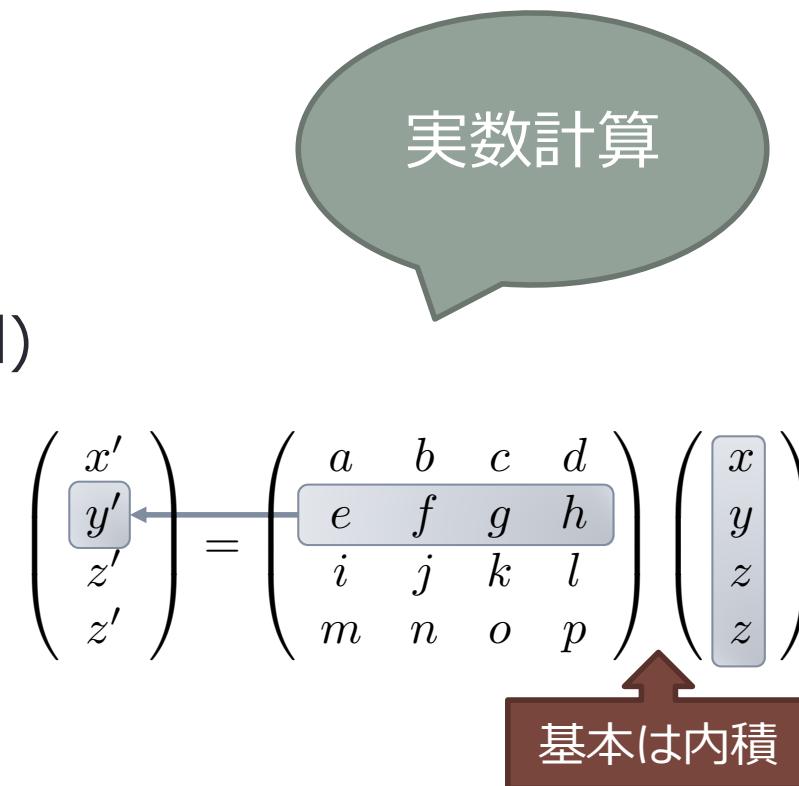
整数演算だけで  
実装可能

# ジオメトリ処理のハードウェア化

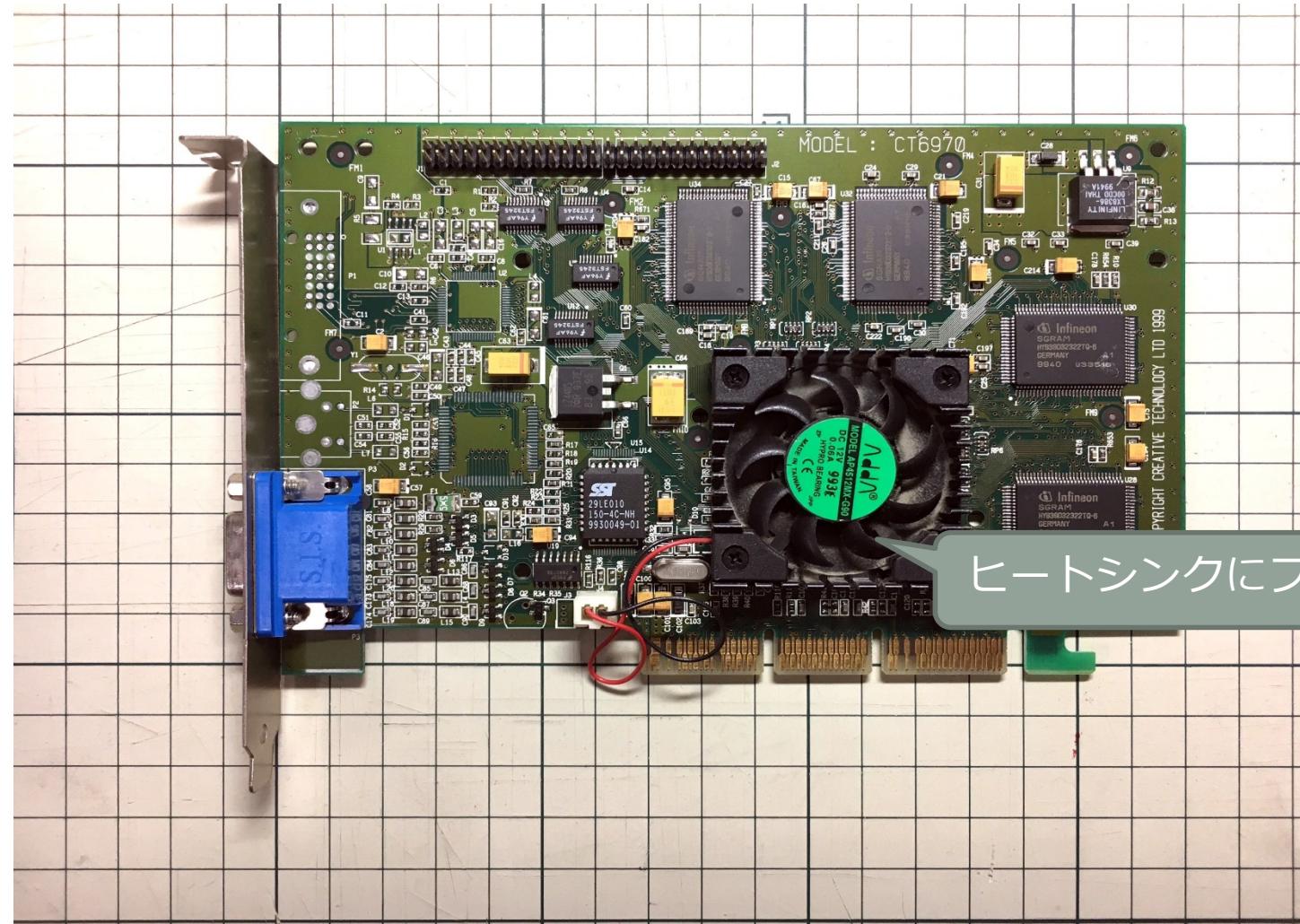


# ジオメトリ処理ハードウェア

- ・浮動小数点演算ハードウェア
  - ・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)
- ・ハードウェア T&L (Transform and Lighting)



# 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

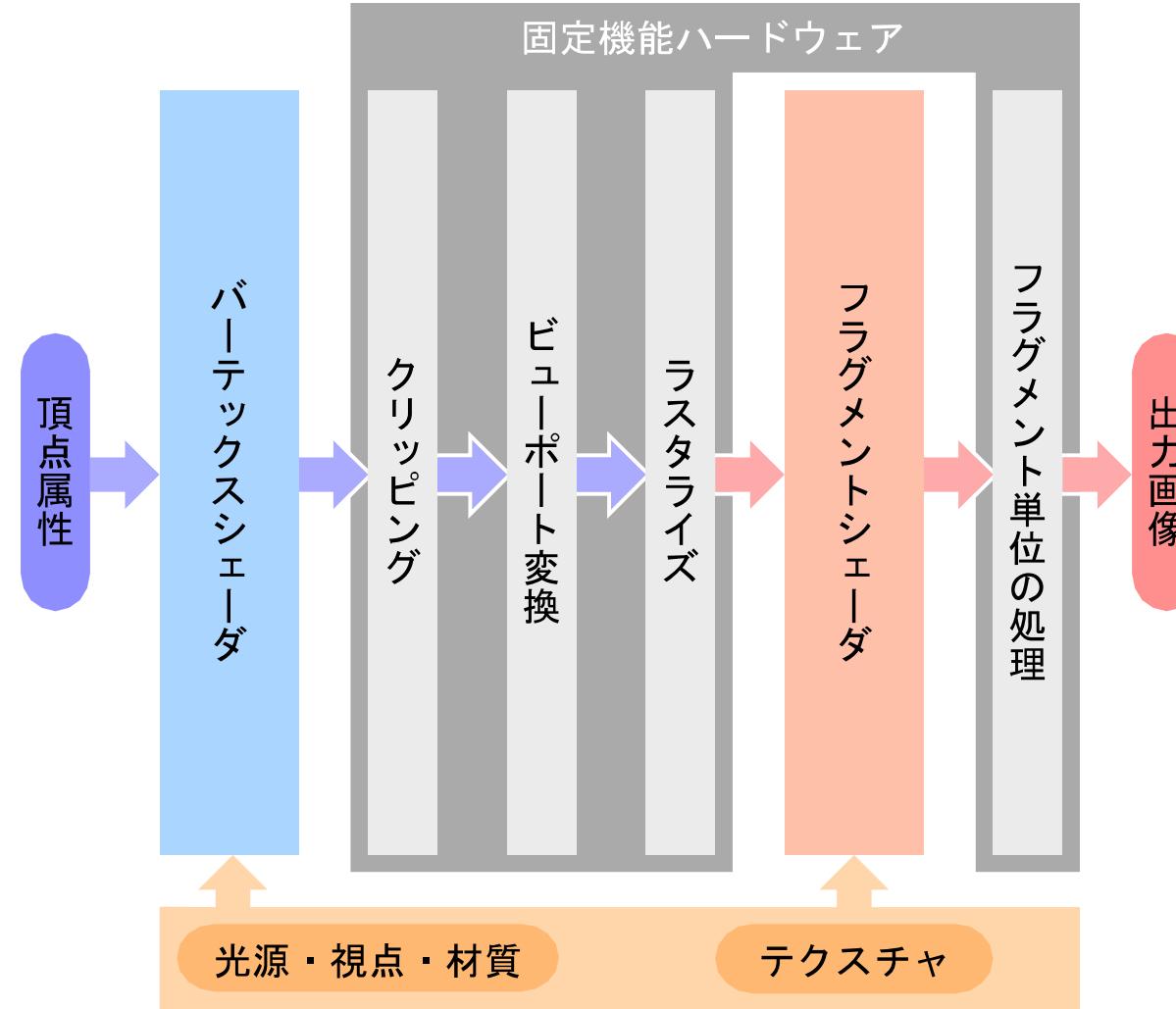
# ハードウェアによる機能追加の限界

- ・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
  - ・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

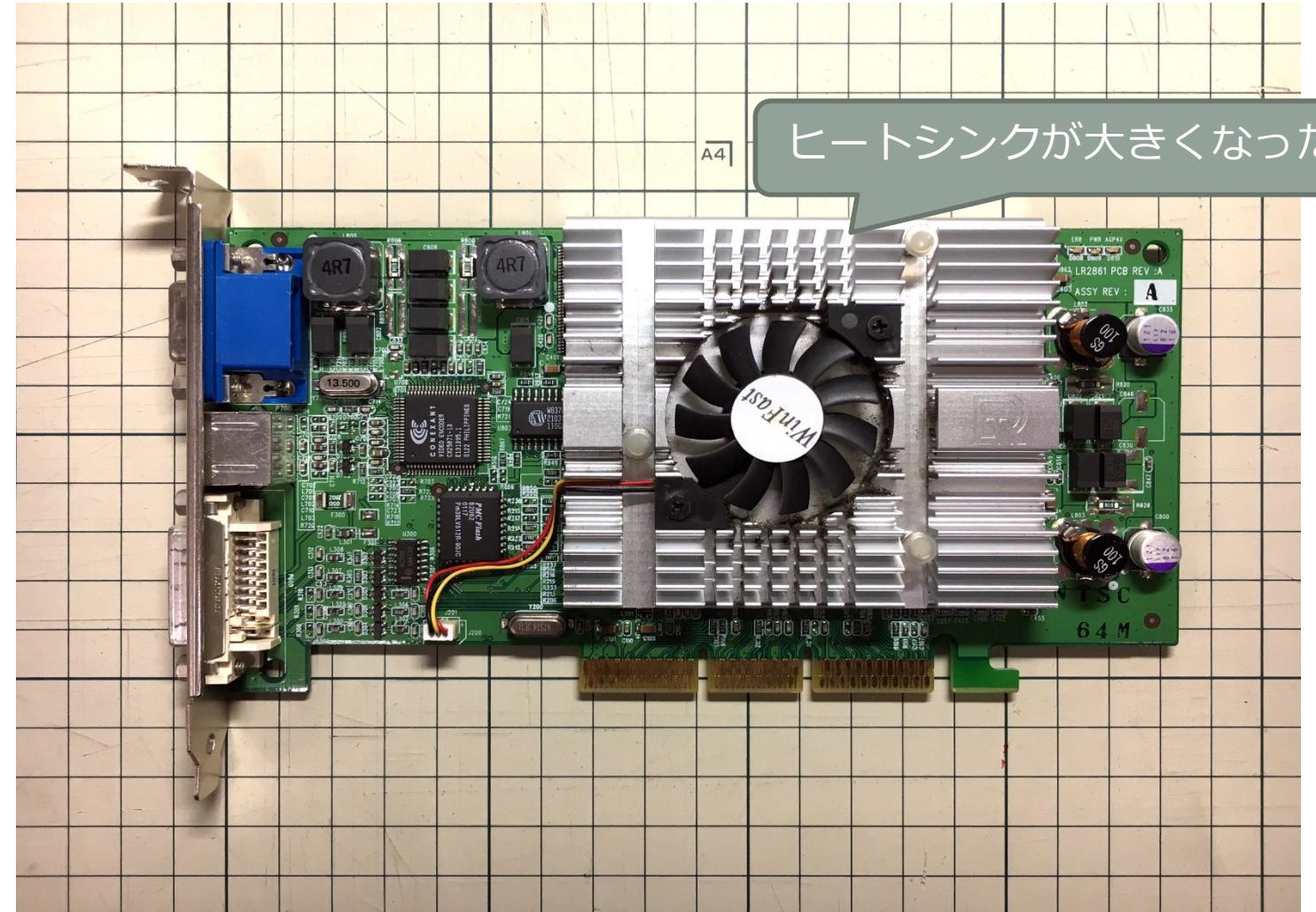


機能追加はユーザに任せる

# 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
  - ・入力された**頂点ごと**に実行される
  - ・座標変換
  - ・陰影付け
- ・フラグメントシェーダ
  - ・出力する**画素ごと**に実行される
  - ・画素の色の決定
    - ・テクスチャのサンプリング
    - ・テクスチャの合成 (**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

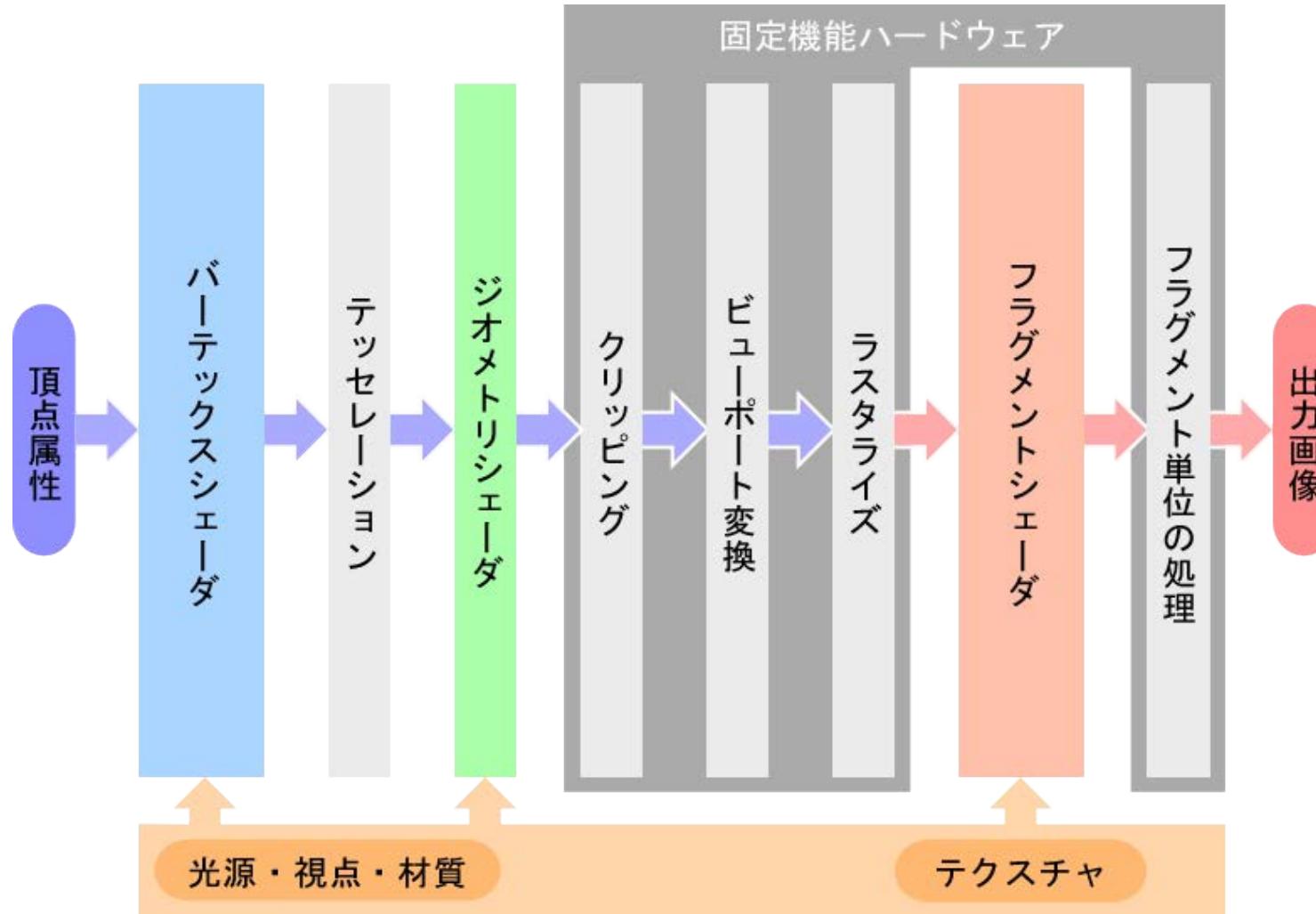


最初 (DirectX 8) は  
ループすらなかった

制御構造を備えて  
汎用プログラミング  
可能に (DirectX 9)

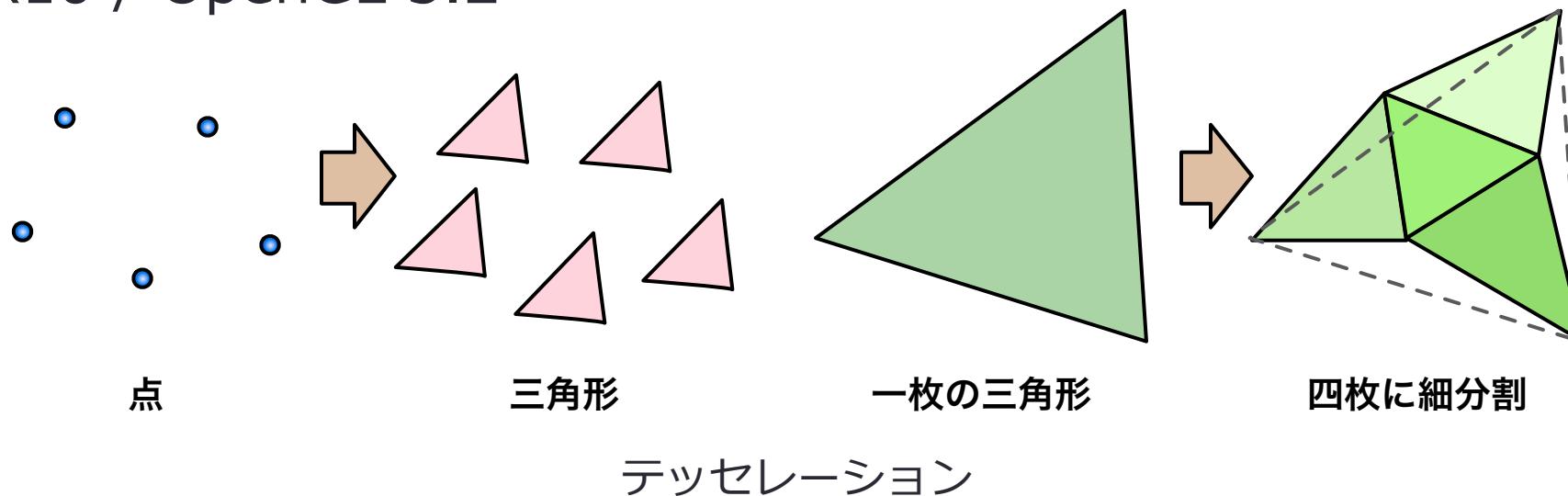
OpenGL 2.0

# ジオメトリシェーダの追加

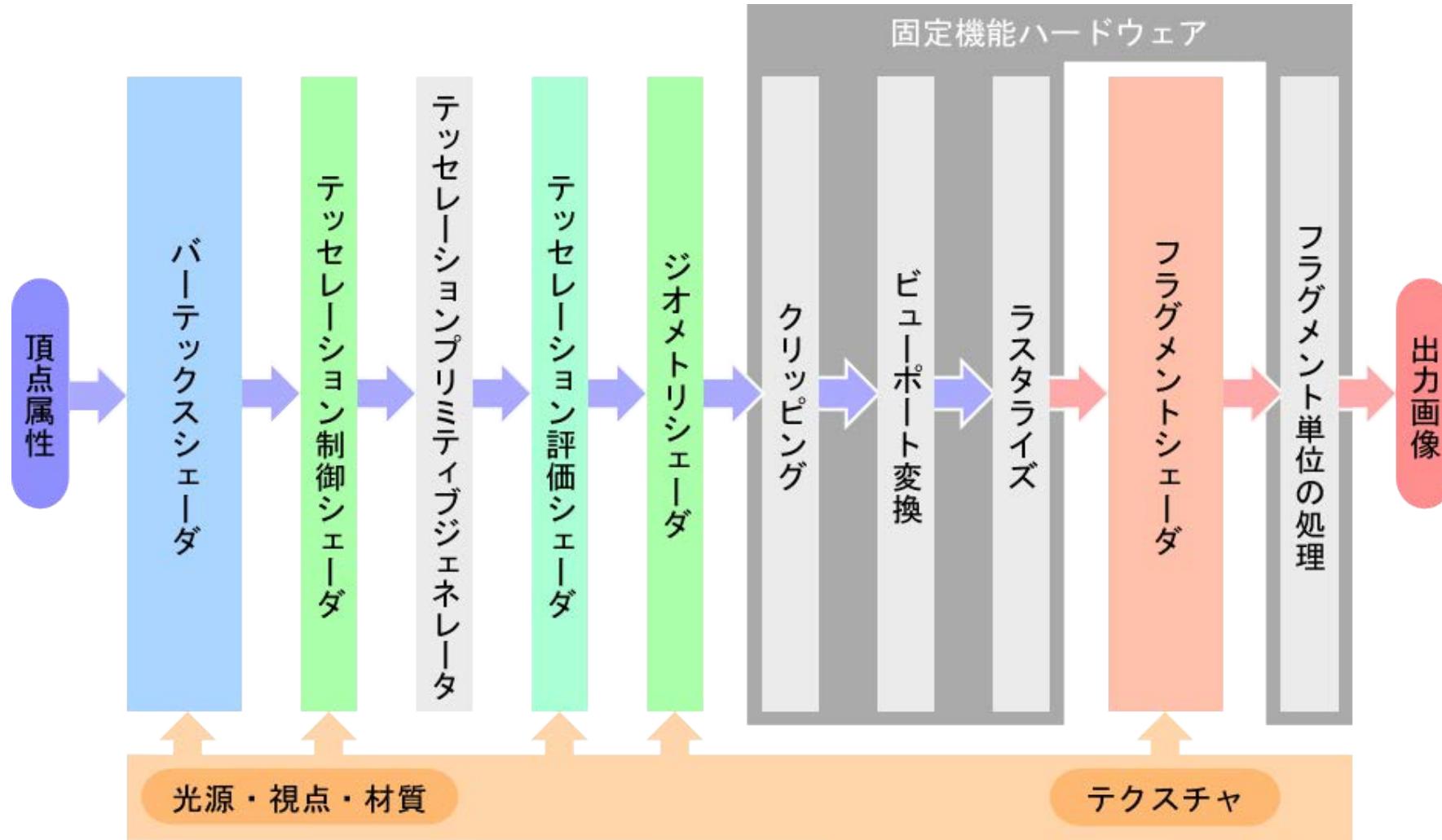


# ジオメトリシェーダ

- ・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細分化



# 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

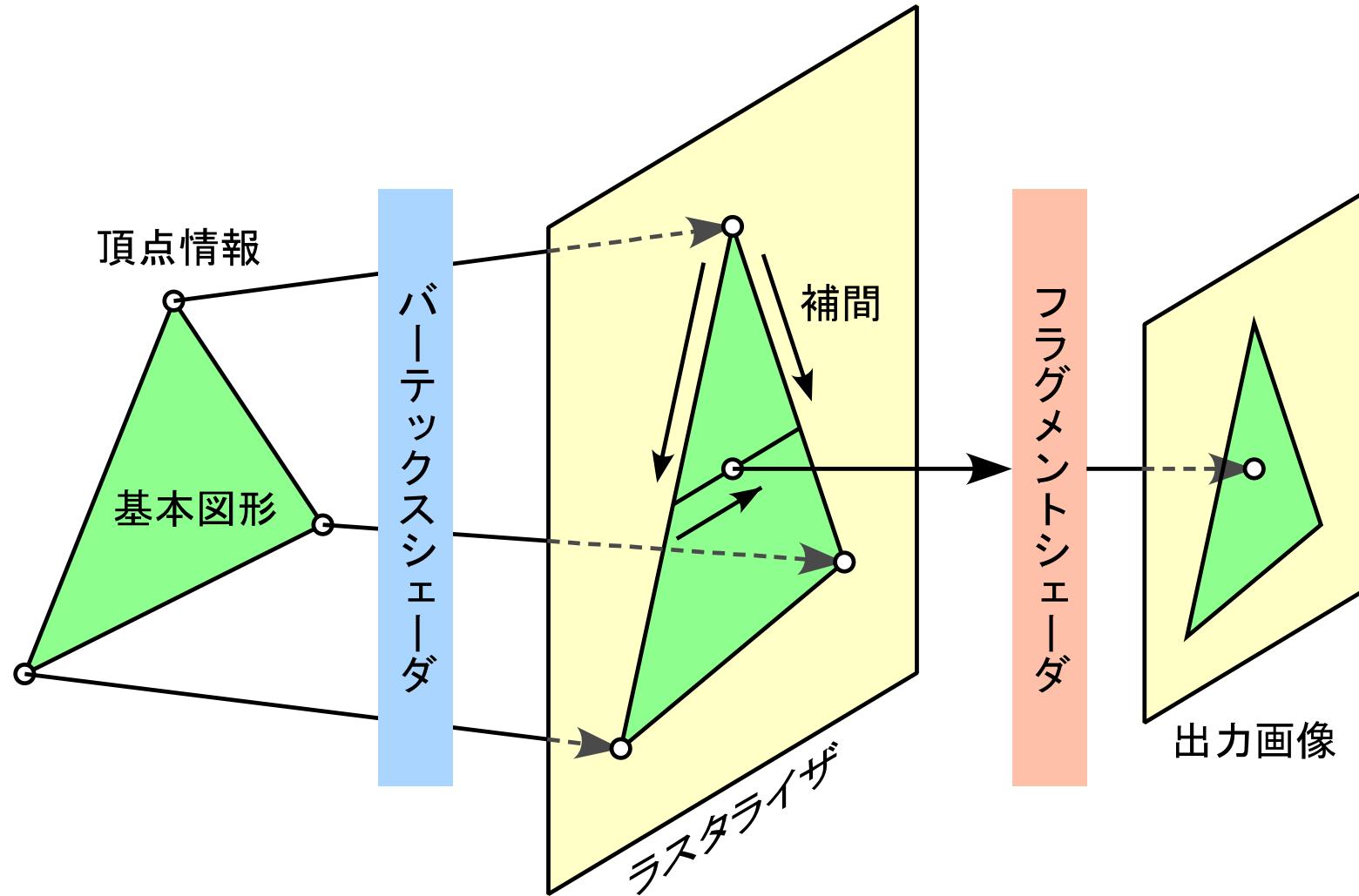
- **テッセレーション評価シェーダ**

- 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

---

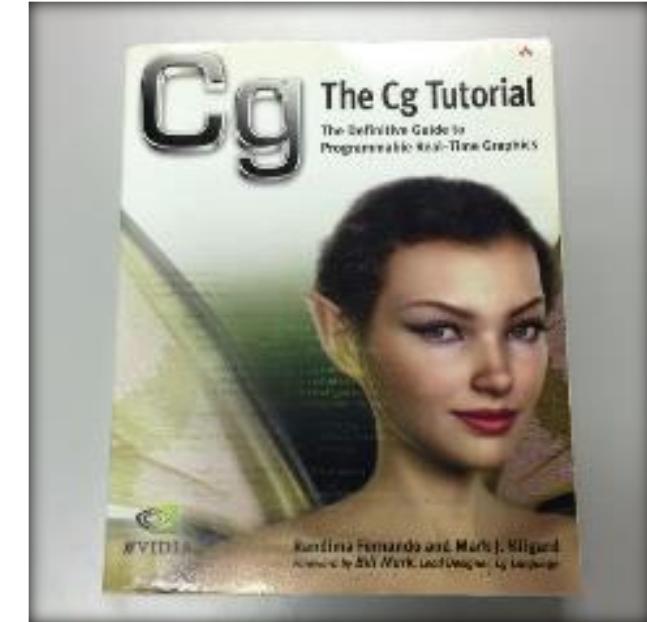
ソースプログラムのコンパイルとリンク

# 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によって処理される
  - ・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

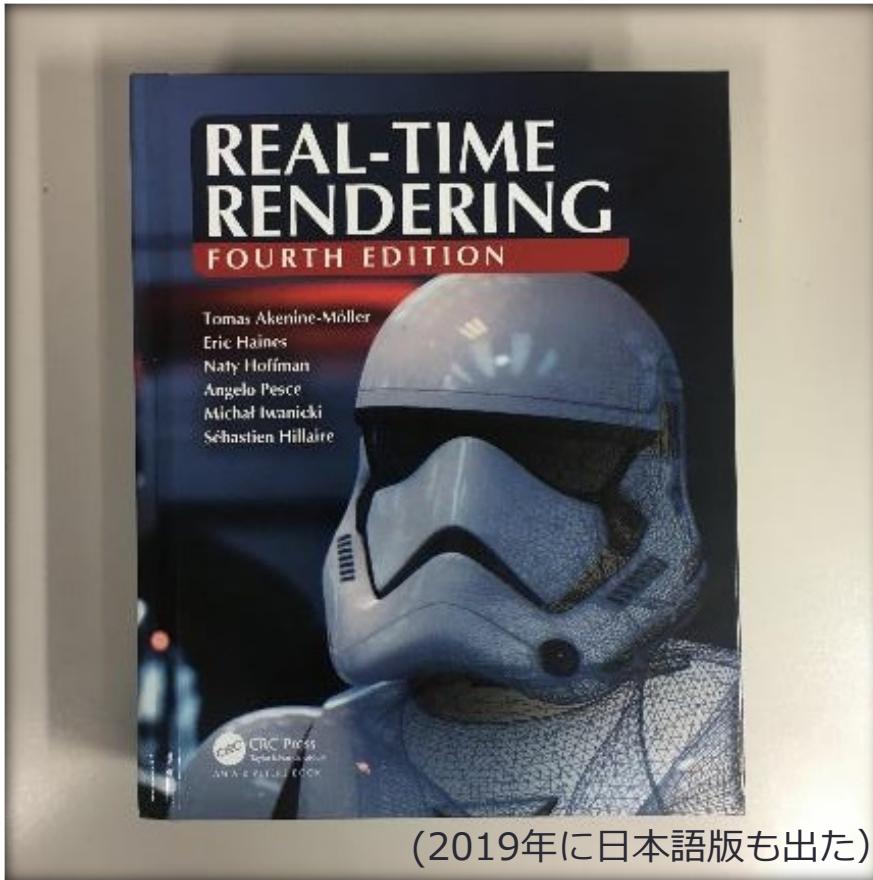
- ・シェーダのプログラミングに用いる
  - ・**Cg** (C for Graphics)
    - ・DirectX と OpenGL に対応
    - ・NVIDIA により開発
  - ・**HLSL** (High Level Shading Language)
    - ・DirectX に対応
    - ・Microsoft が NVIDIA の協力を得て開発
  - ・**GLSL** (OpenGL Shading Language, GLslang)
    - ・OpenGL に対応
    - ・OpenGL ARB (現在は Khronos Group) 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



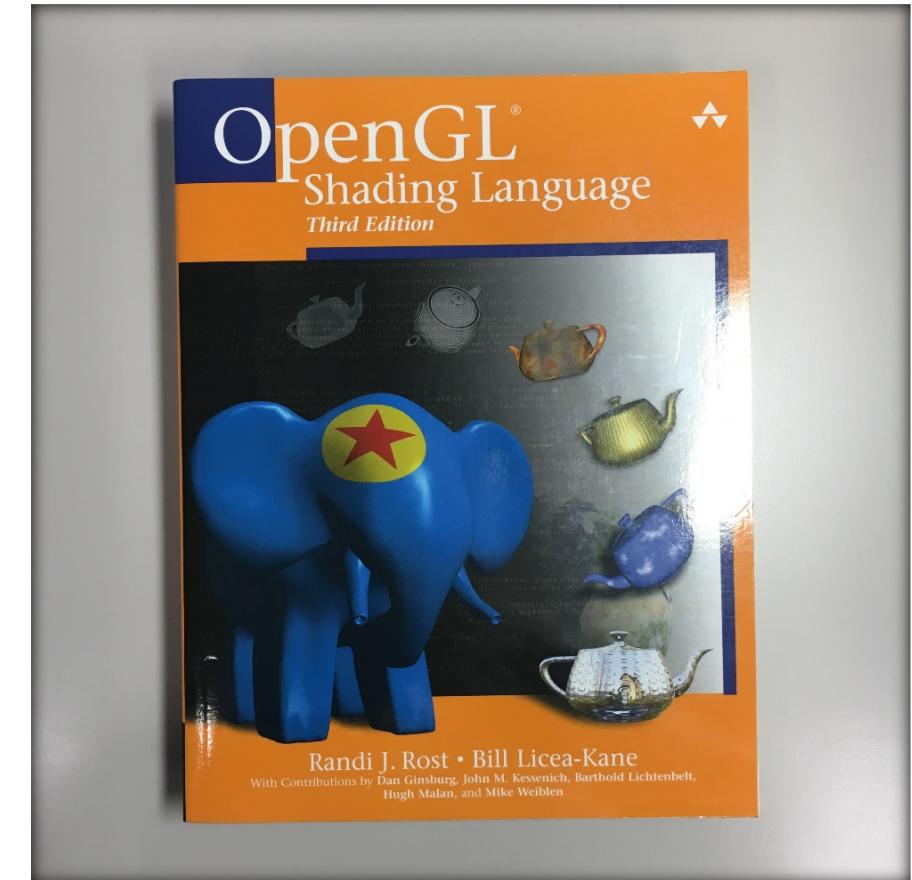
この講義では **GLSL** を採用します

# 参考書

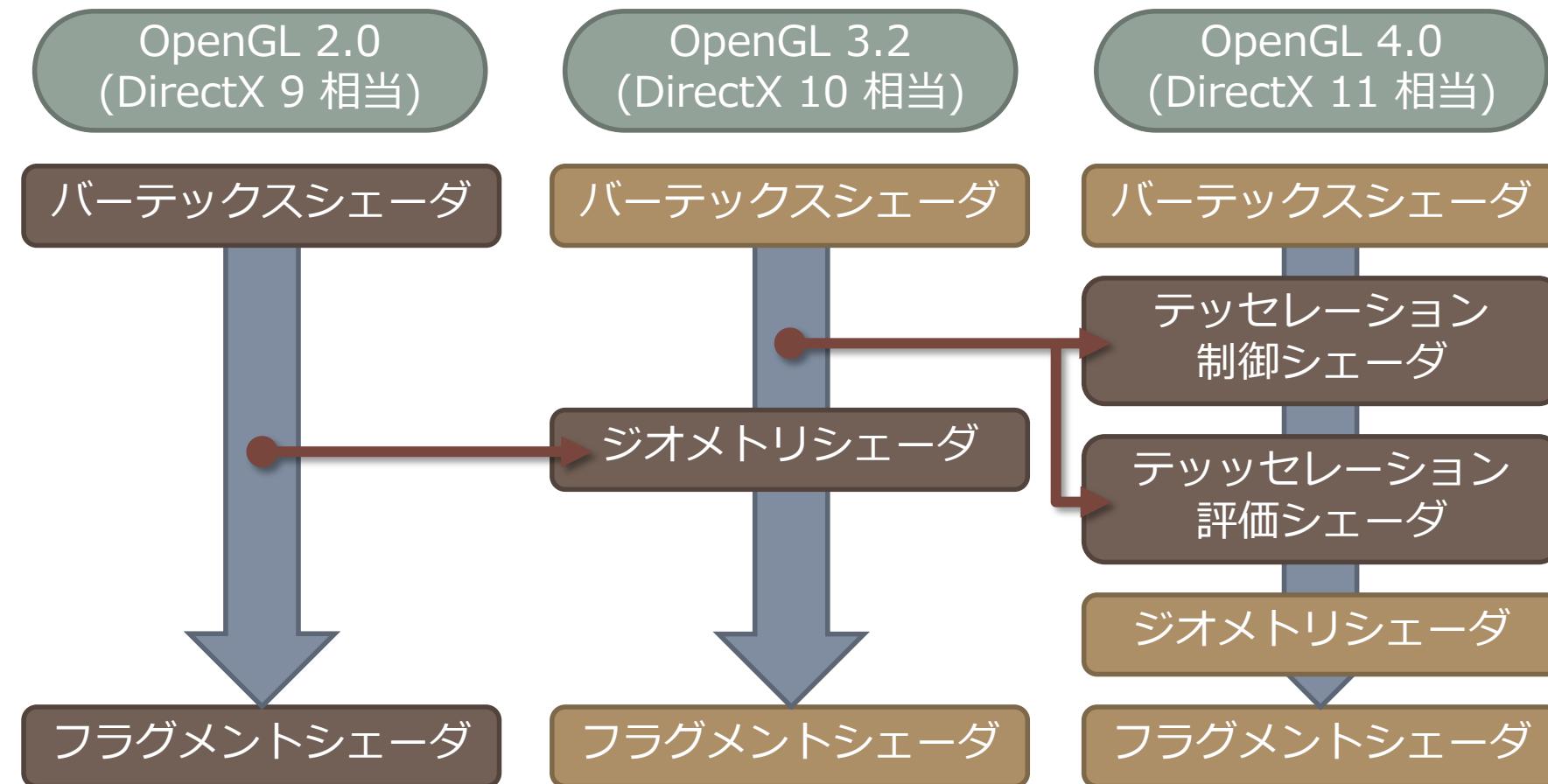
## Real-Time Rendering (4th Ed.)



## OpenGL Shading Language

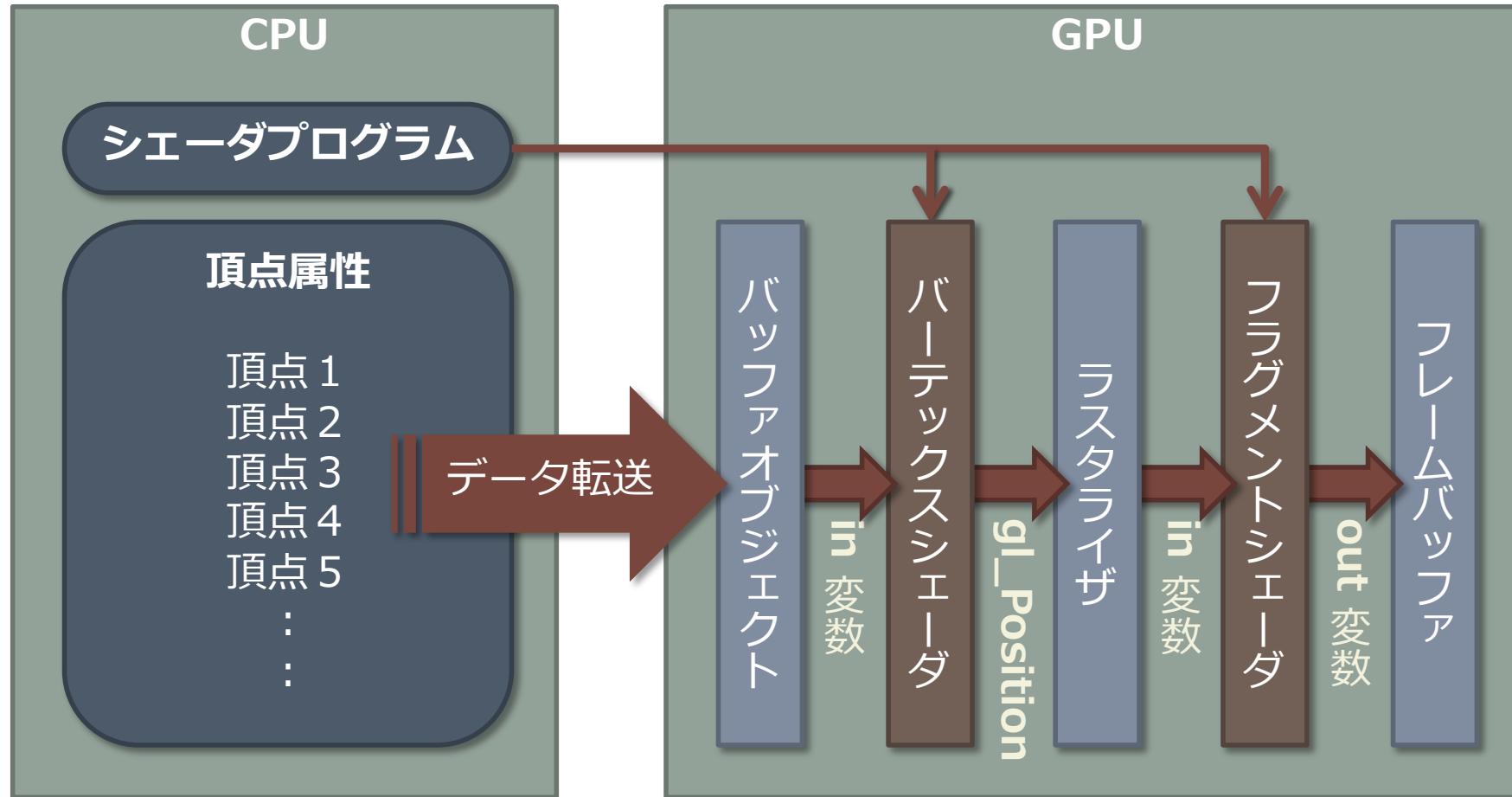


# シェーダプログラムの種類



Vulkan (OpenGL Next Generation) / DirectX 12 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
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

## 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>bvec<math>n</math></code>	論理型、true / false
int	<code>ivec<math>n</math></code>	符号付き 32bit 整数
uint	<code>uvec<math>n</math></code>	符号なし 32bit 整数
float	<code>vec<math>n</math></code>	单精度実数
double	<code>dvec<math>n</math></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**



- 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プログラム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プログラム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ぬので削除する

- **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 をシェーダ側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

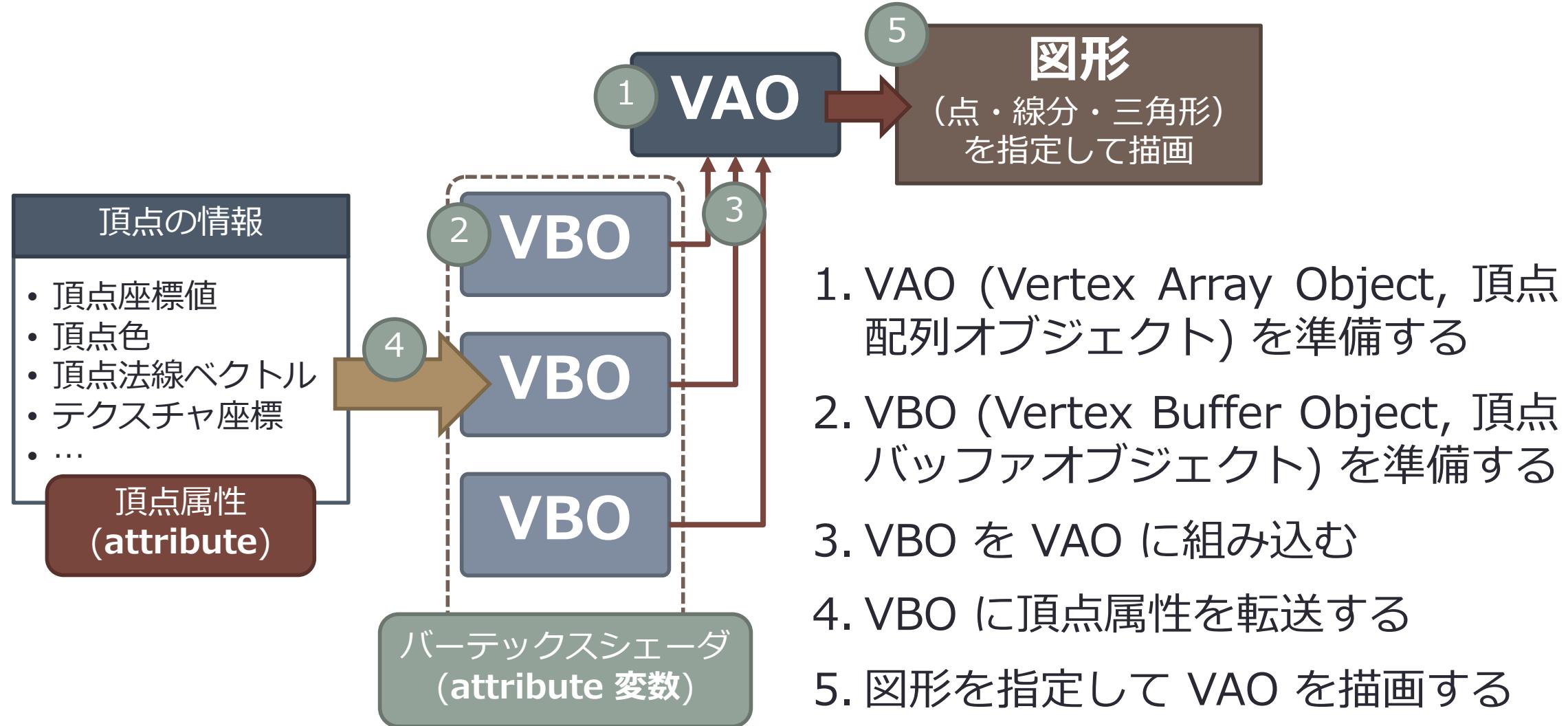
```
#version 150 core
#extension GL_ARB_explicit_attrib_location: enable
layout (location = 0) out vec4 fc;
layout (location = 1) out vec4 fv;
...
```

# 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
  - i. VAO を作成する
  - ii. 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
  - i. VBO を作成する
  - ii. 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**)
  - iii. VBO のメモリを確保し頂点属性を転送する
  - iv. バーテックスシェーダの `in` 変数の `index` に VBO を結合する
  - v. バーテックスシェーダの `in` 変数の `index` を有効にする
3. 使用するシェーダプログラムを指定する
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
  - 0 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

# 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`);

# 図形の描画

1. 使用するシェーダプログラムを選ぶ

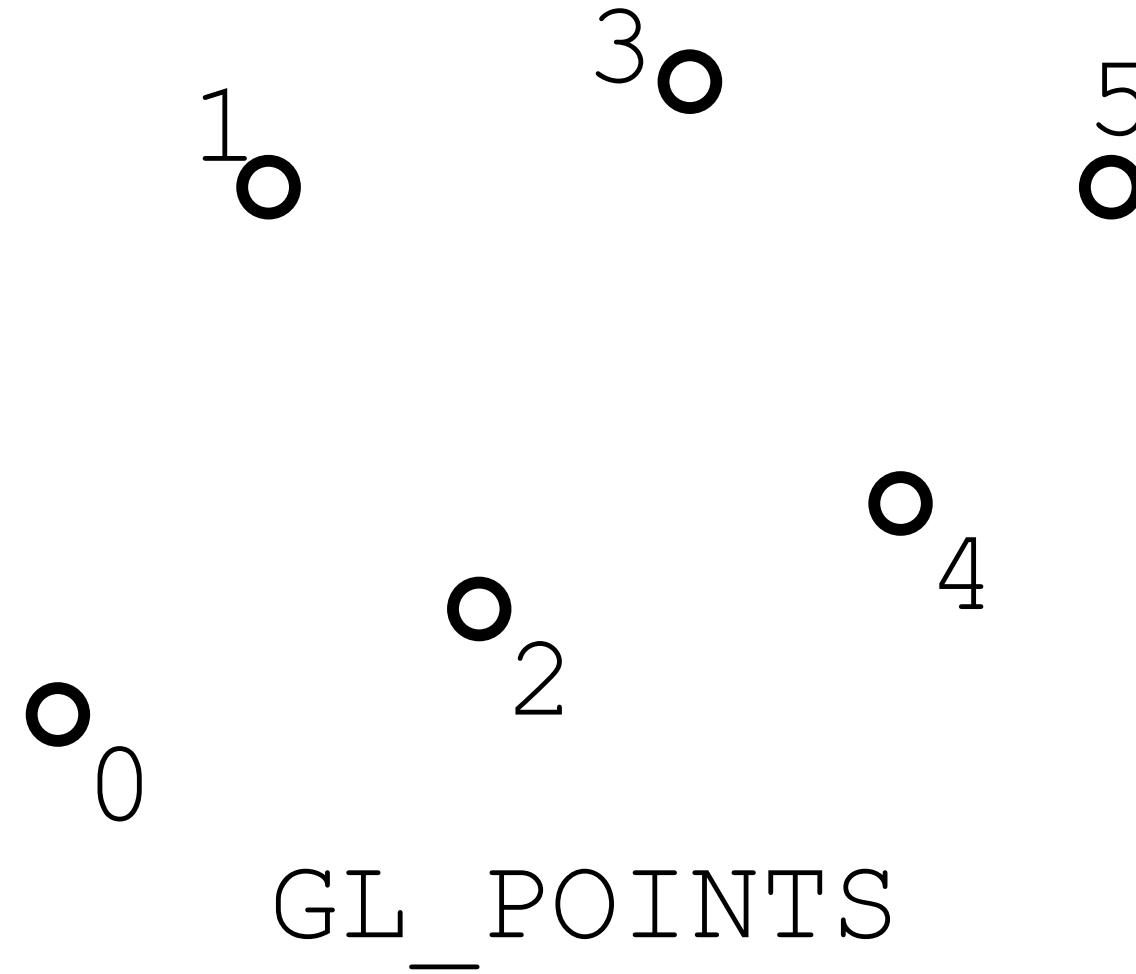
- **glUseProgram(program);**

2.  $i$  番目の VAO を図形を描画する

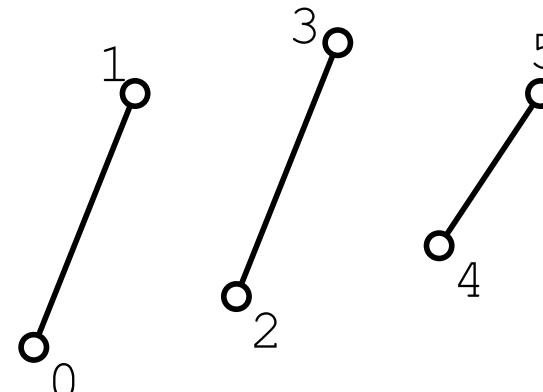
- **glBindVertexArray(vao[i]);**
- **glDrawArrays(mode, first, count);**

- $mode$ : 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

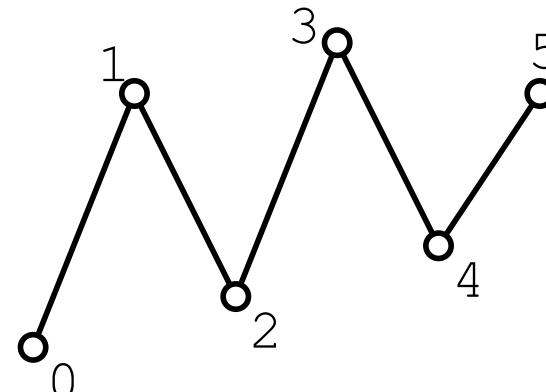
## 基本図形 - 点



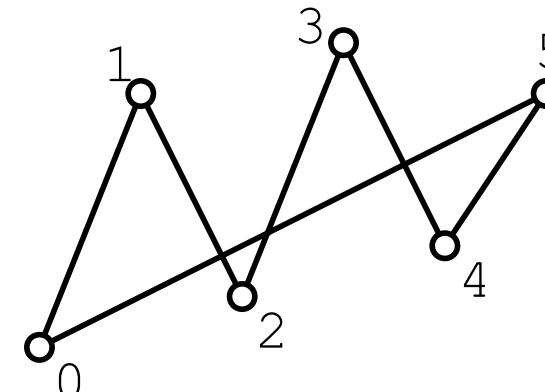
# 基本図形 – 線分と三角形



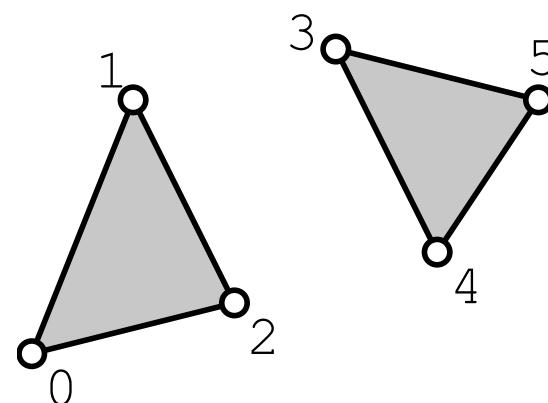
GL\_LINES



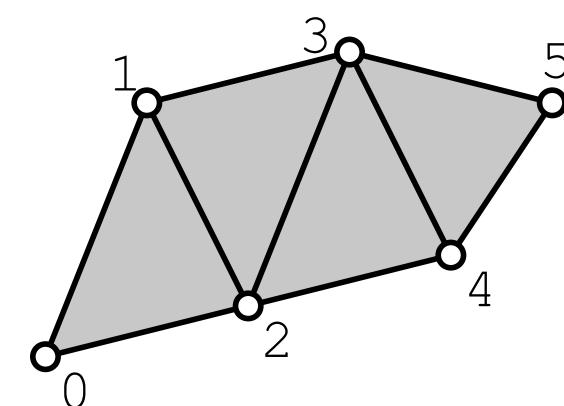
GL\_LINE\_STRIP



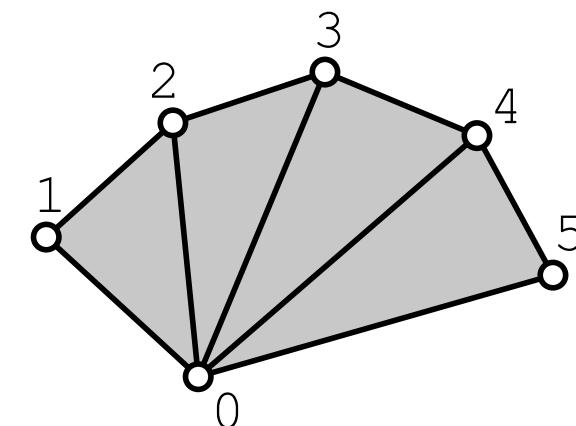
GL\_LINE\_LOOP



GL\_TRIANGLES

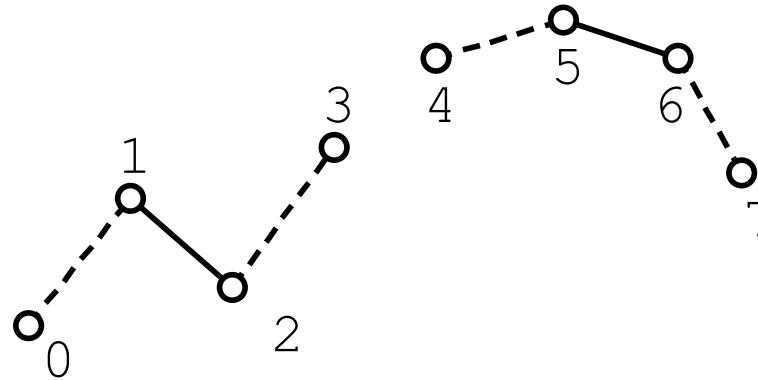


GL\_TRIANGLE\_STRIP

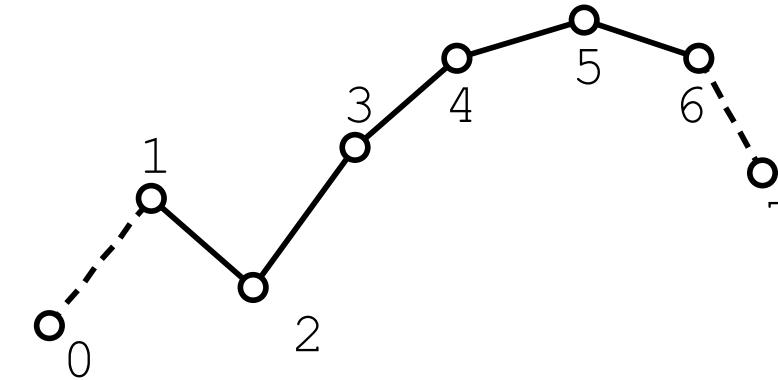


GL\_TRIANGLE\_FAN

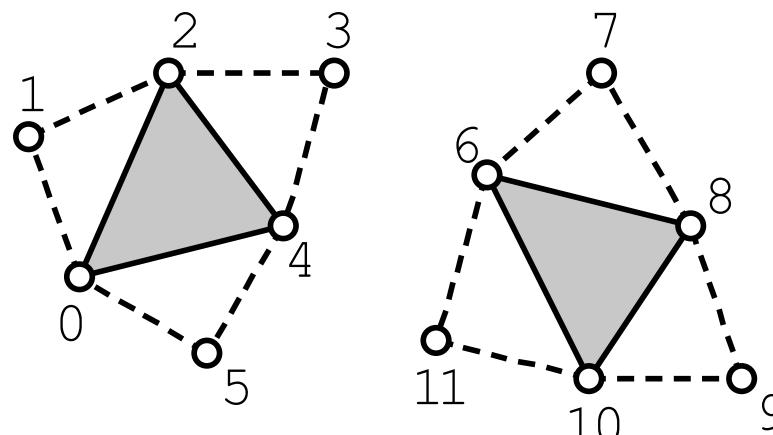
## ジオメトリシェーダで追加されたもの



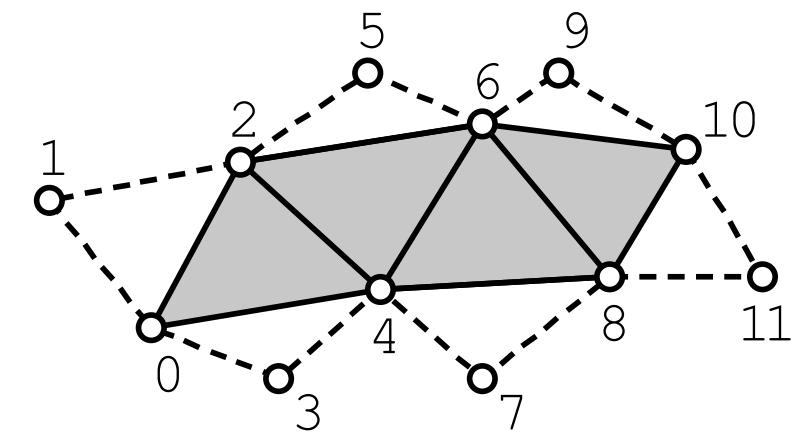
`GL_LINES_ADJACENCY`



`GL_LINE_STRIP_ADJACENCY`



`GL_TRIANGLES_ADJACENCY`



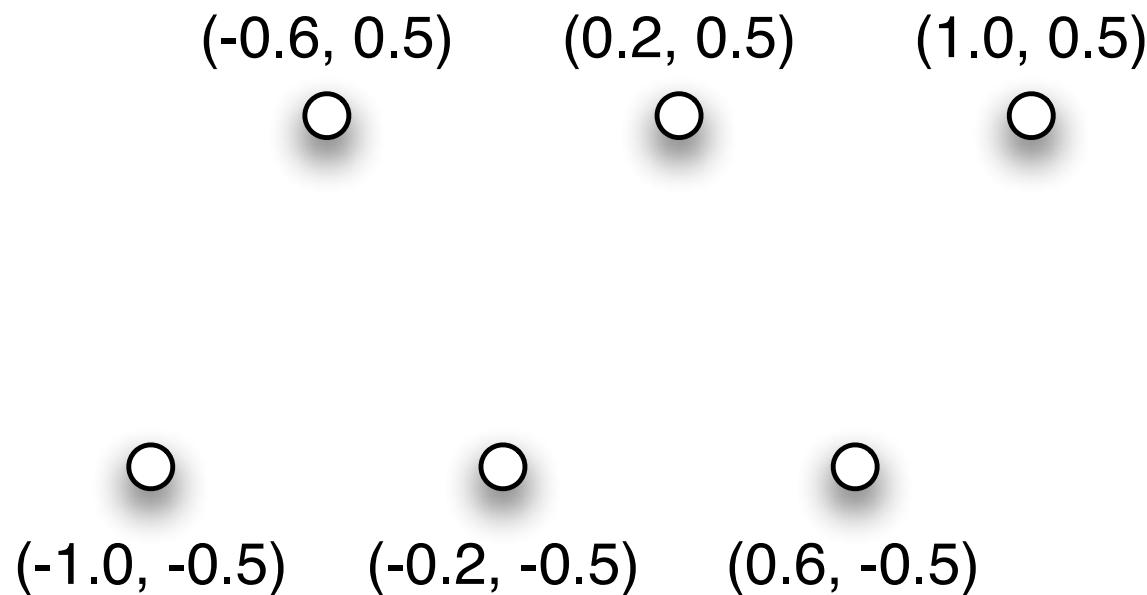
`GL_TRIANGLE_STRIP_ADJACENCY`

# 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

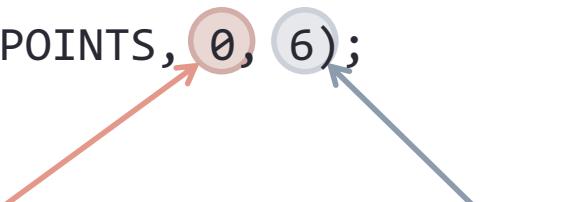
1 頂点あたりのデータ数 (次)

# 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

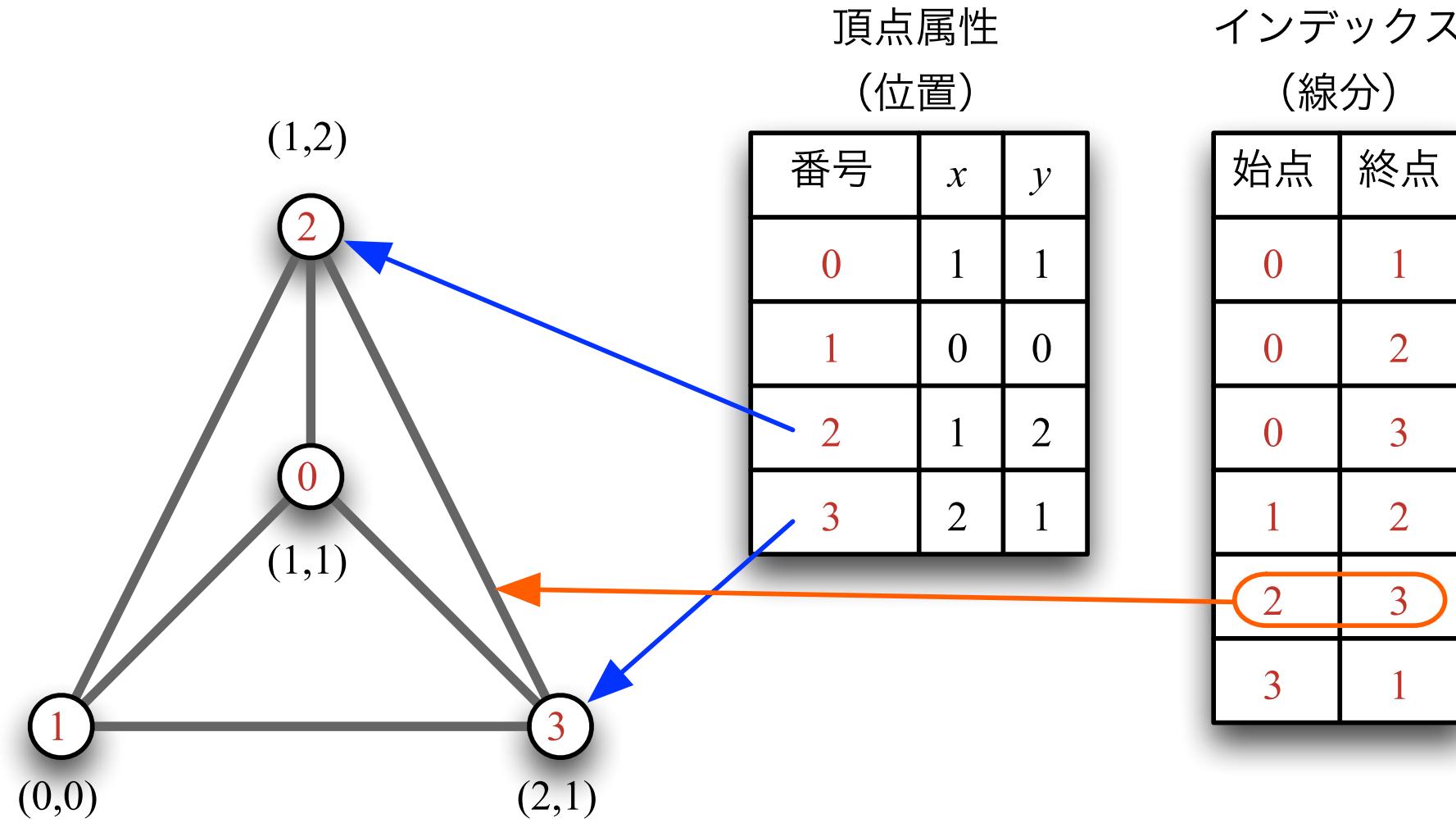


# 頂点インデックスを用いた図形描画

---

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

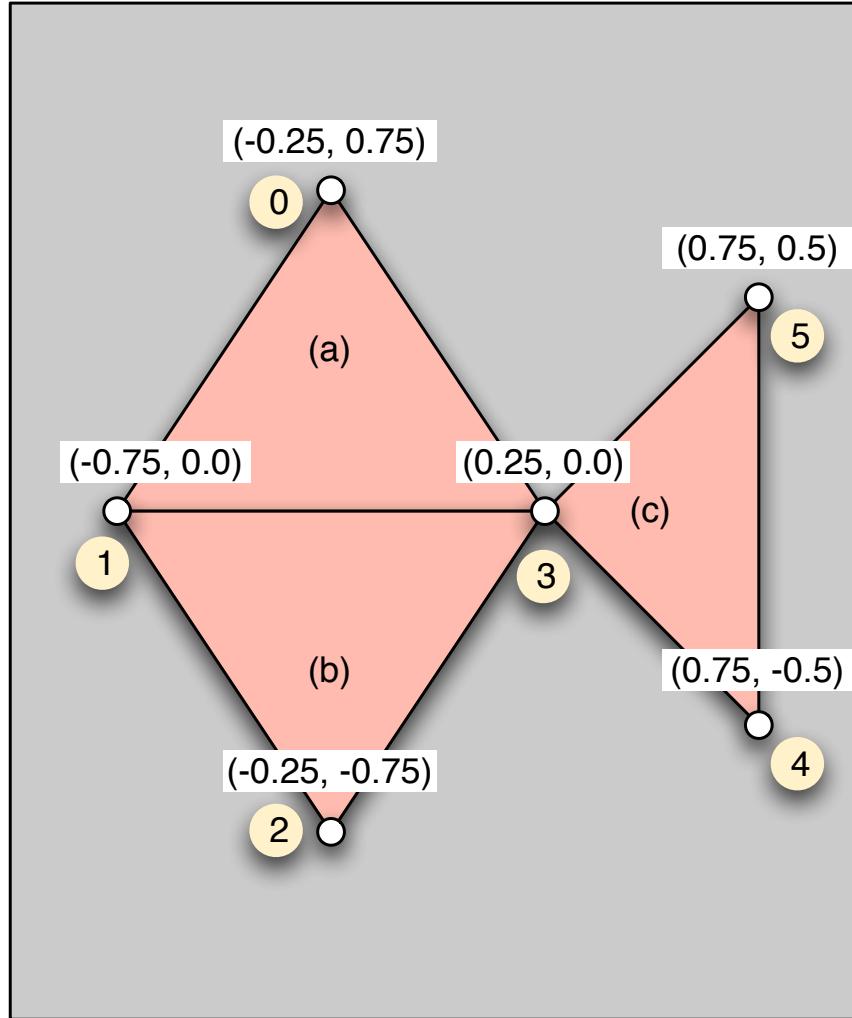
## 1. 使用するシェーダプログラムを選ぶ

- `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

---

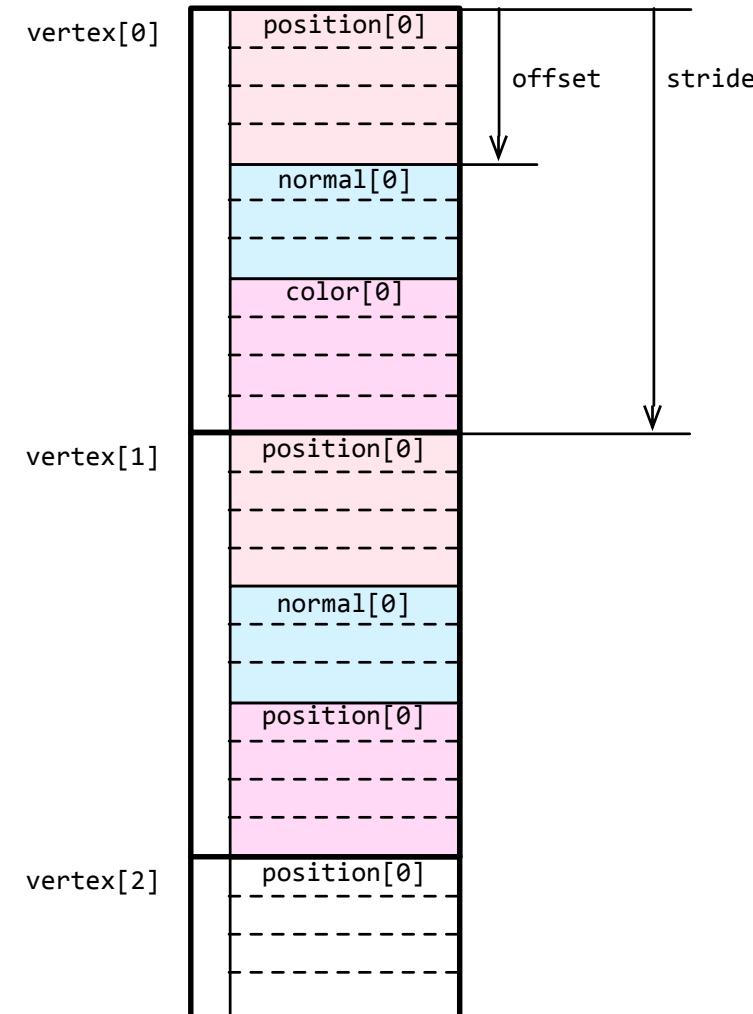
異なる頂点属性を一つのバッファに格納する

# 頂点属性をインターリーブに配置する場合

```
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, BUFFER_OFFSET(offsetof(Vertex, position)));
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);
```

offsetof() を使う場合は #include <cstddef>

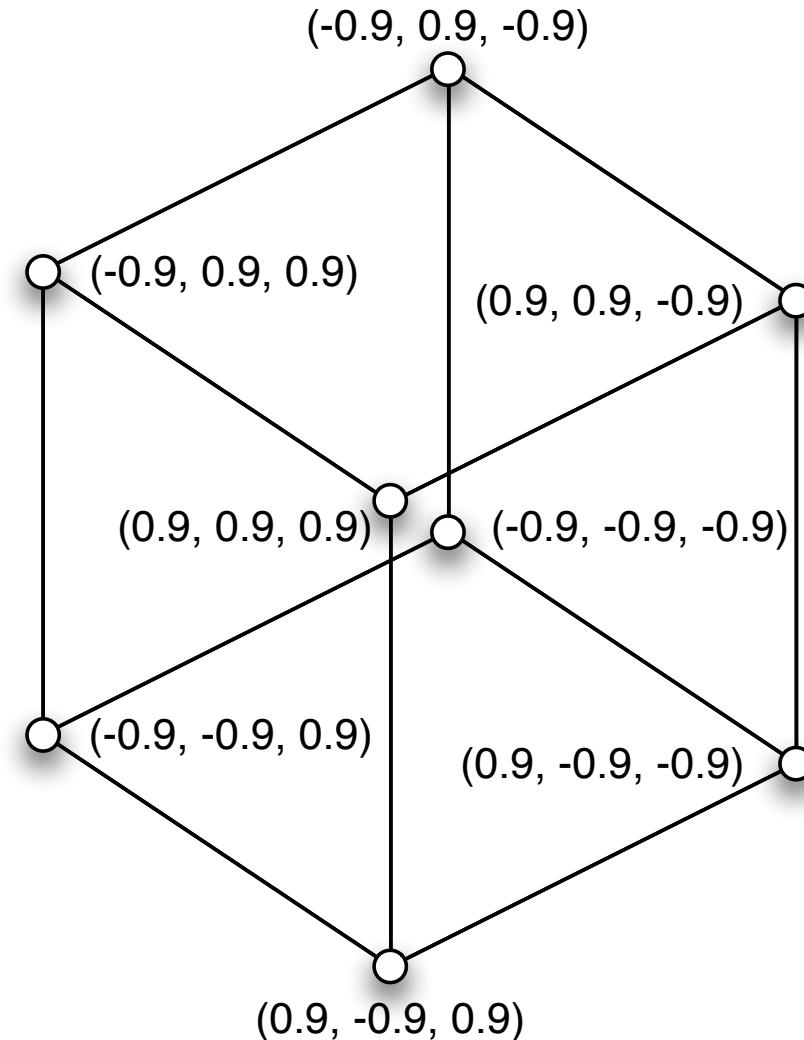
position に関しては 0 でもよい

# 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`

## 課題

- 右の図を **GL\_LINES** で描くための頂点の位置データの配列変数とインデックスの配列変数の内容を答えなさい
- 回答は Moodle にお願いします



## 宿題

- 宿題のひな形を変更して課題の図形の正面図 (xy 平面への直交投影図) をシェーダを使って描いてください
  - 宿題のひな形は GitHub にあります
    - <https://github.com/tokoik/ggsample02>
    - 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/>
  - 三次元なので頂点データの要素数が 3 になっています
    - glVertexAttribPointer() の第 2 引数 size をそれに合わせてください
  - ggsample02.cpp をアップロードしてください
    - “ggsample02.cpp” というファイル名を変更しないでください

## 結果

このような図形が表示されれば、  
多分、正解です

