

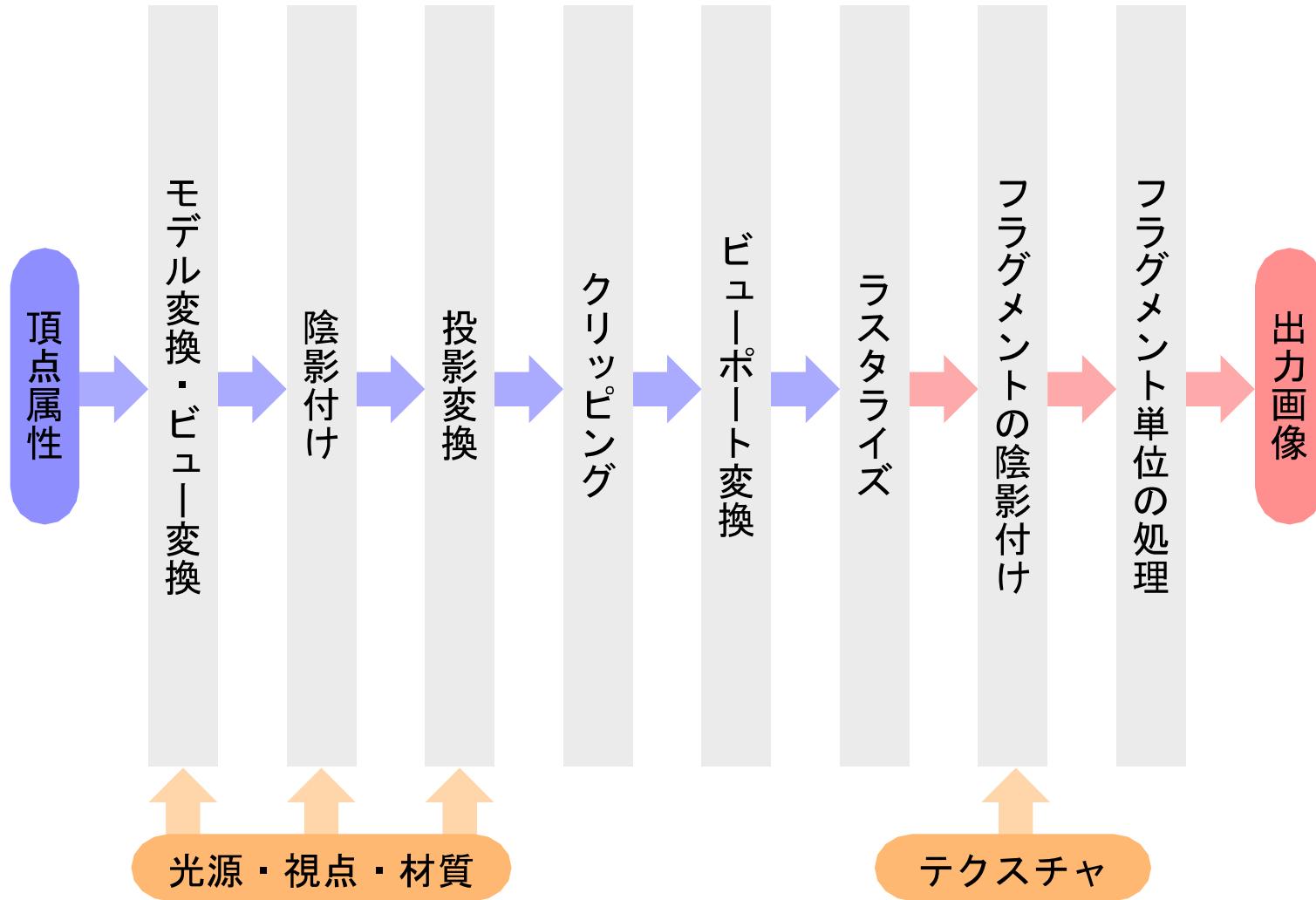
ゲームグラフィックス特論

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

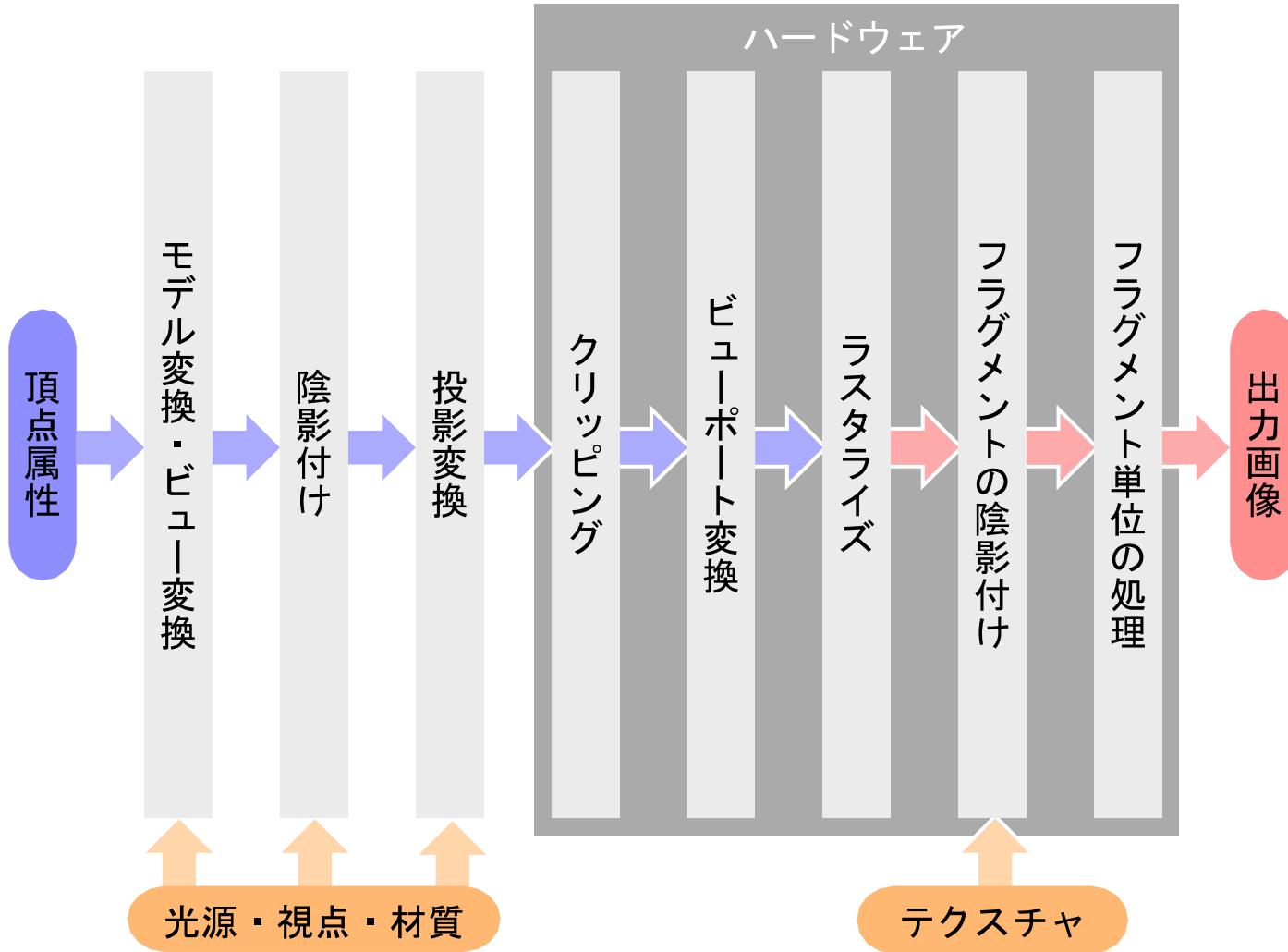
再びレンダリングパイプライン



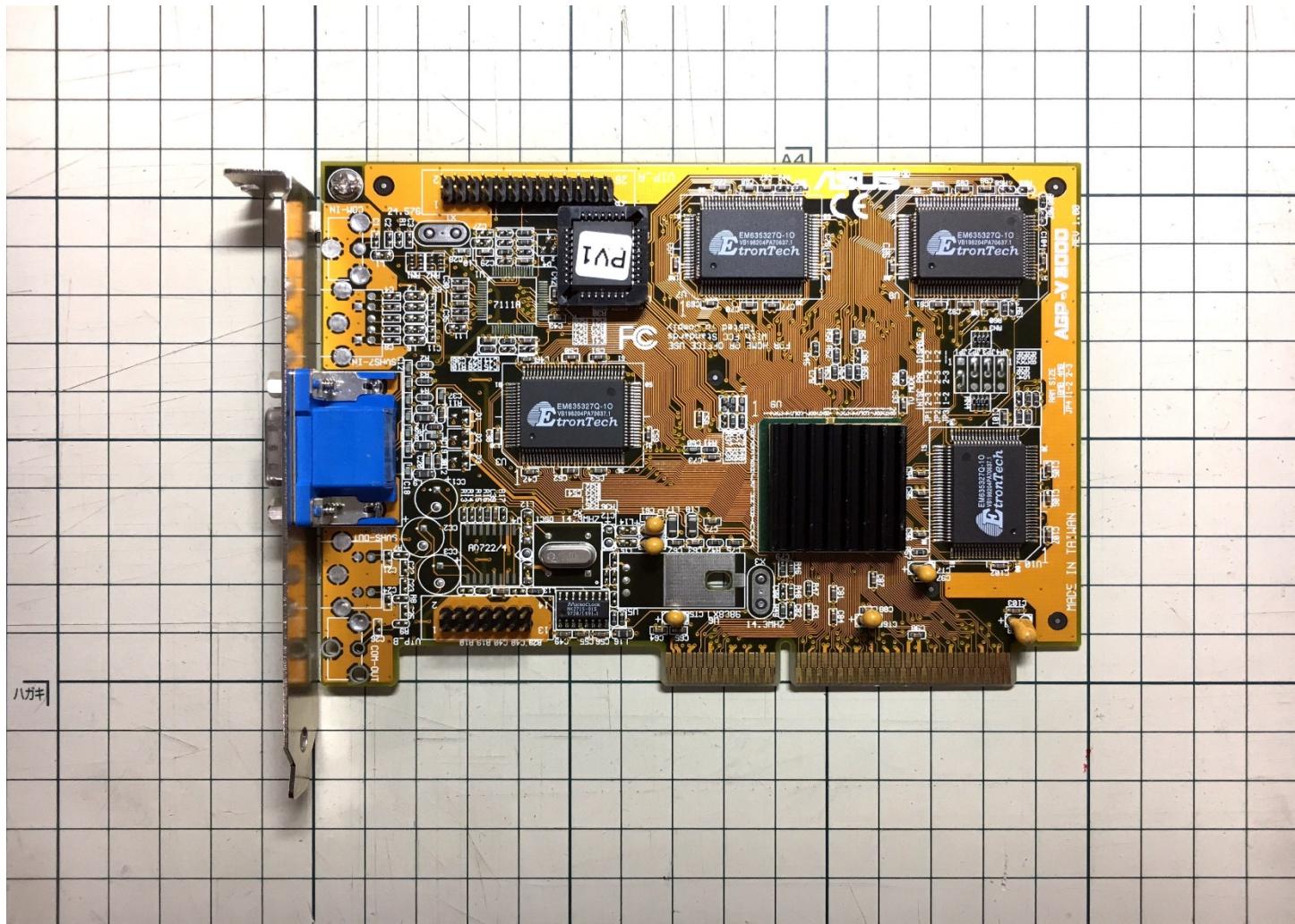
CPUで処理



フラグメント処理のハードウェア化



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. 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

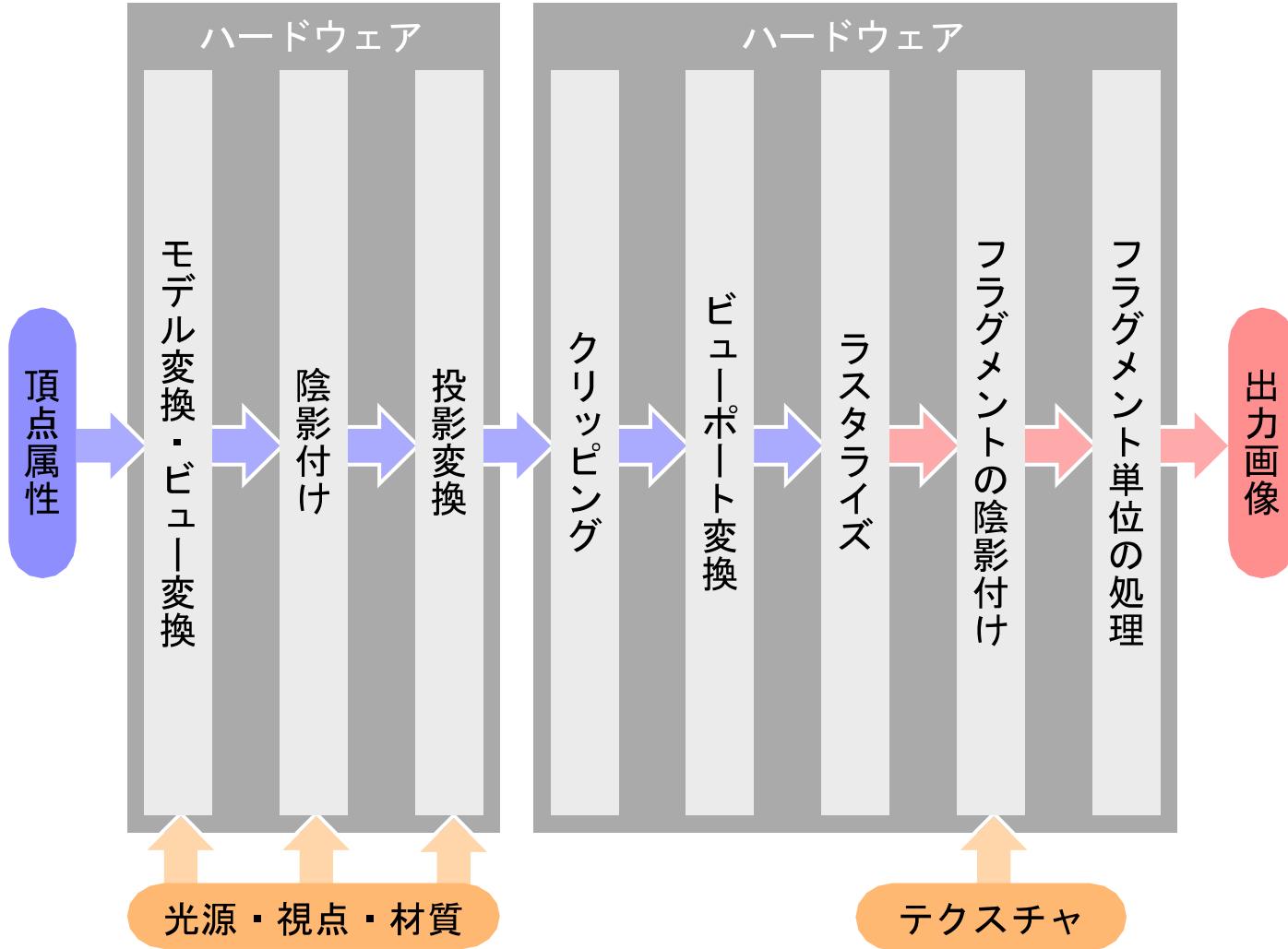
5. フラグメントの陰影付け

- ・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

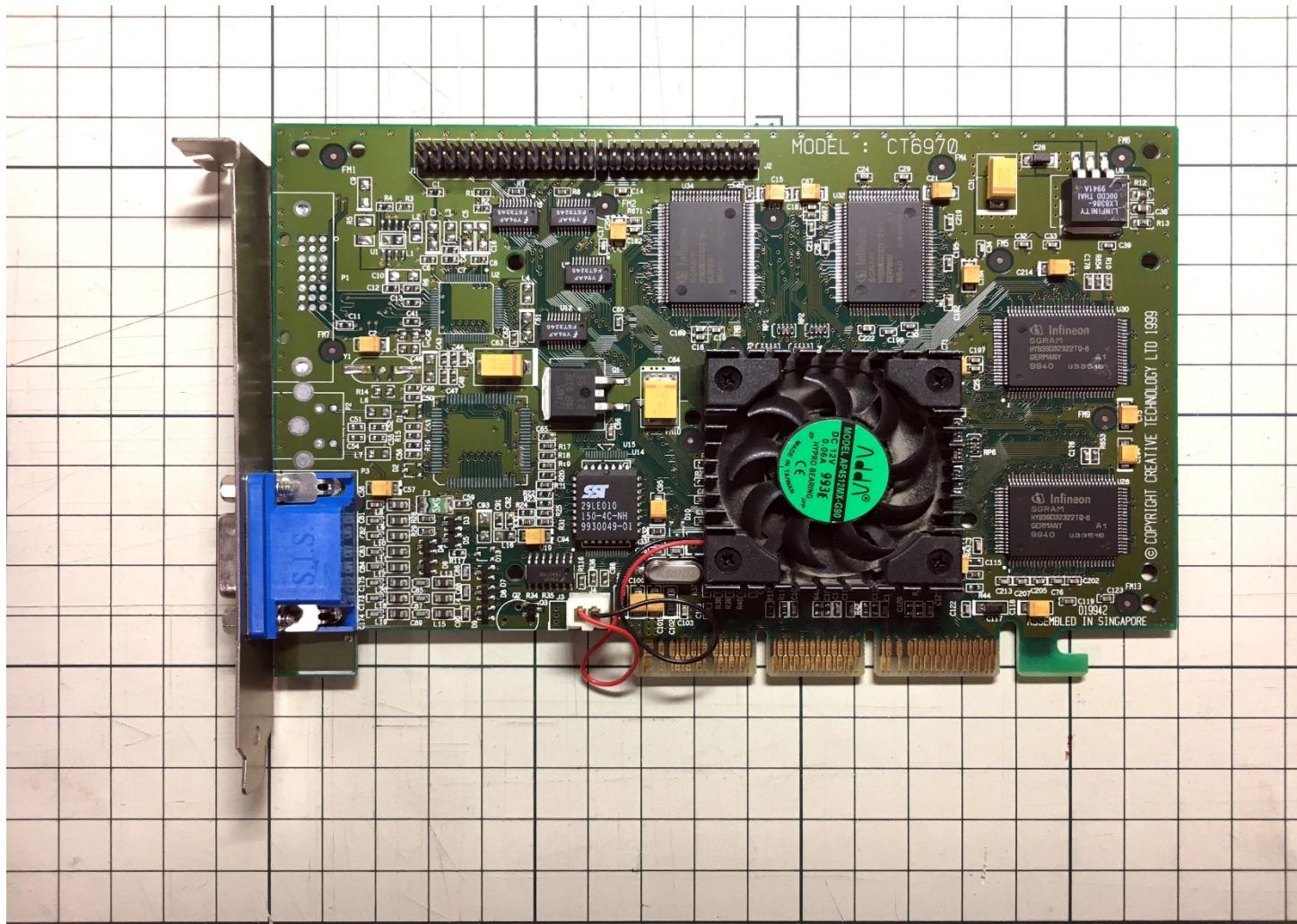


整数演算

ジオメトリ処理のハードウェア化

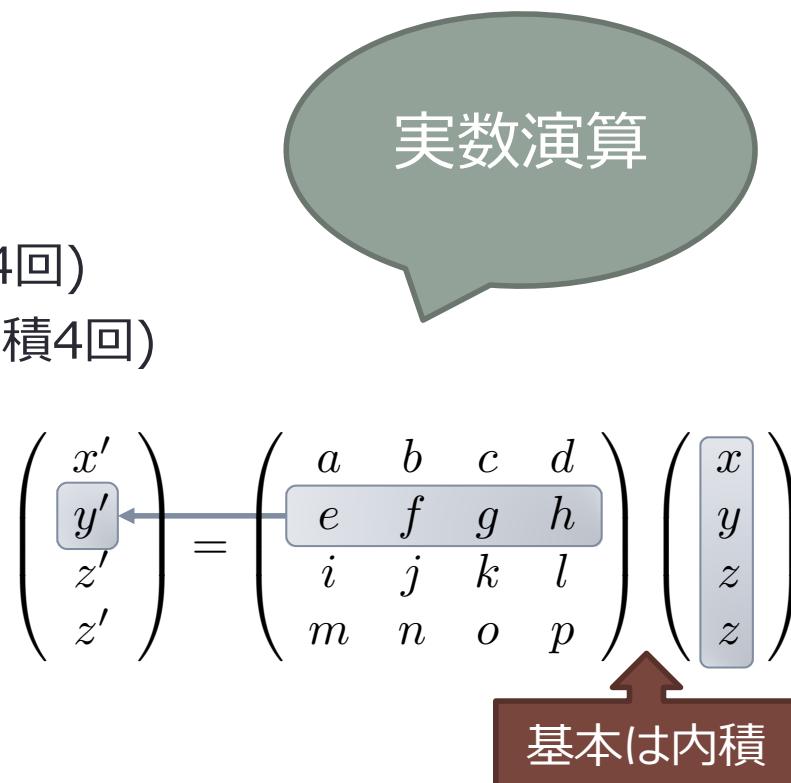


GeForce 256

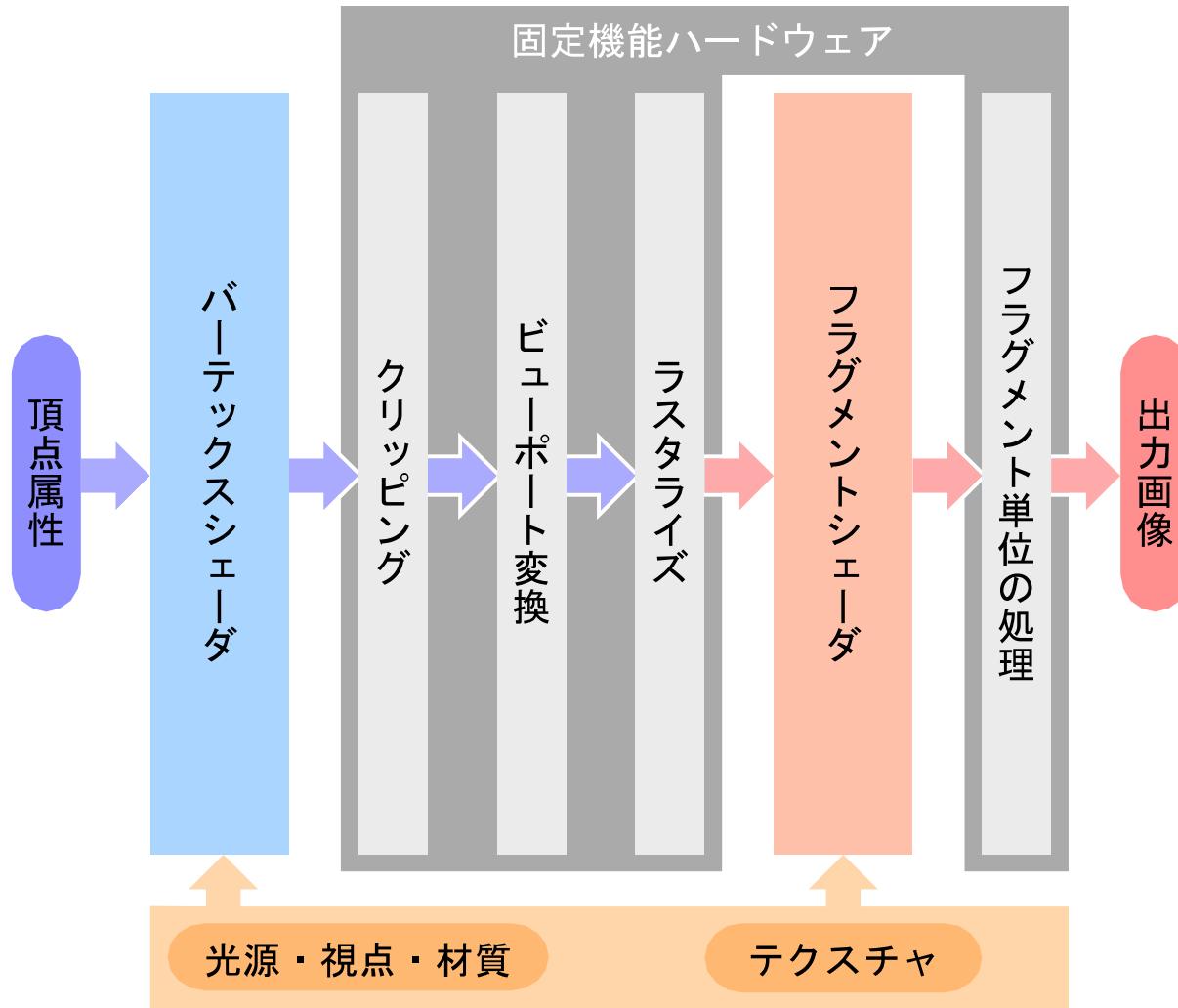


ジオメトリ処理ハードウェア

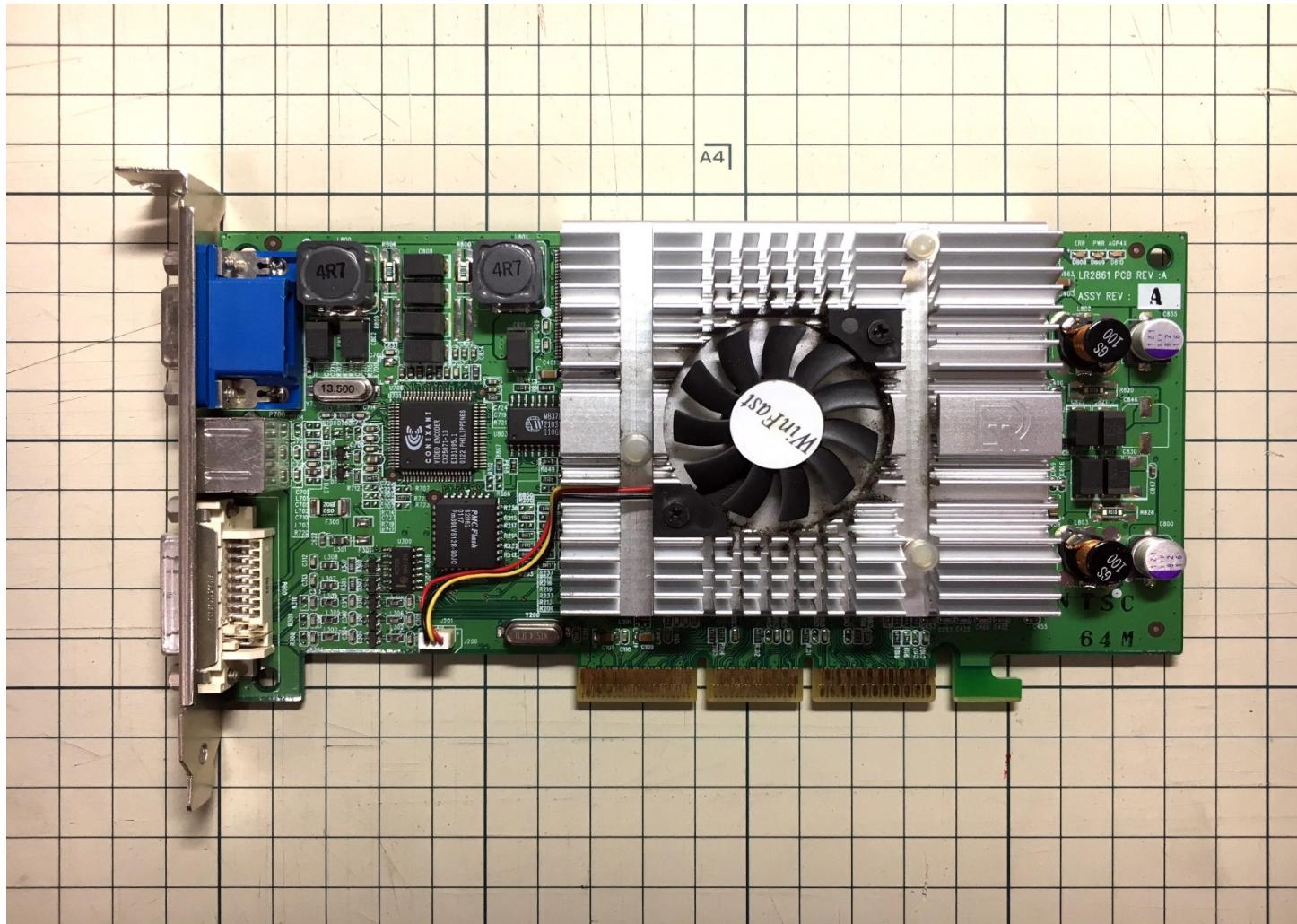
- 浮動小数点演算ハードウェア
 - 主として**積和計算**を実行する
- 座標変換 (Transform)
 - 4要素のベクトルどうしの**内積**
 - 4要素のベクトルと4×4行列の積 (内積4回)
 - 4×4行列どうしの積 (ベクトルと行列の積4回)
- 照明計算 (Lighting, 陰影付け)
 - 四則演算, 逆数
 - 平方根の逆数, 指数計算
 - 内積計算, 外積計算
 - 条件判断, クランプ (値の範囲の制限)
- ハードウェア T & L (Transform and Lighting)



プログラマブルシェーダによる置き換え



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

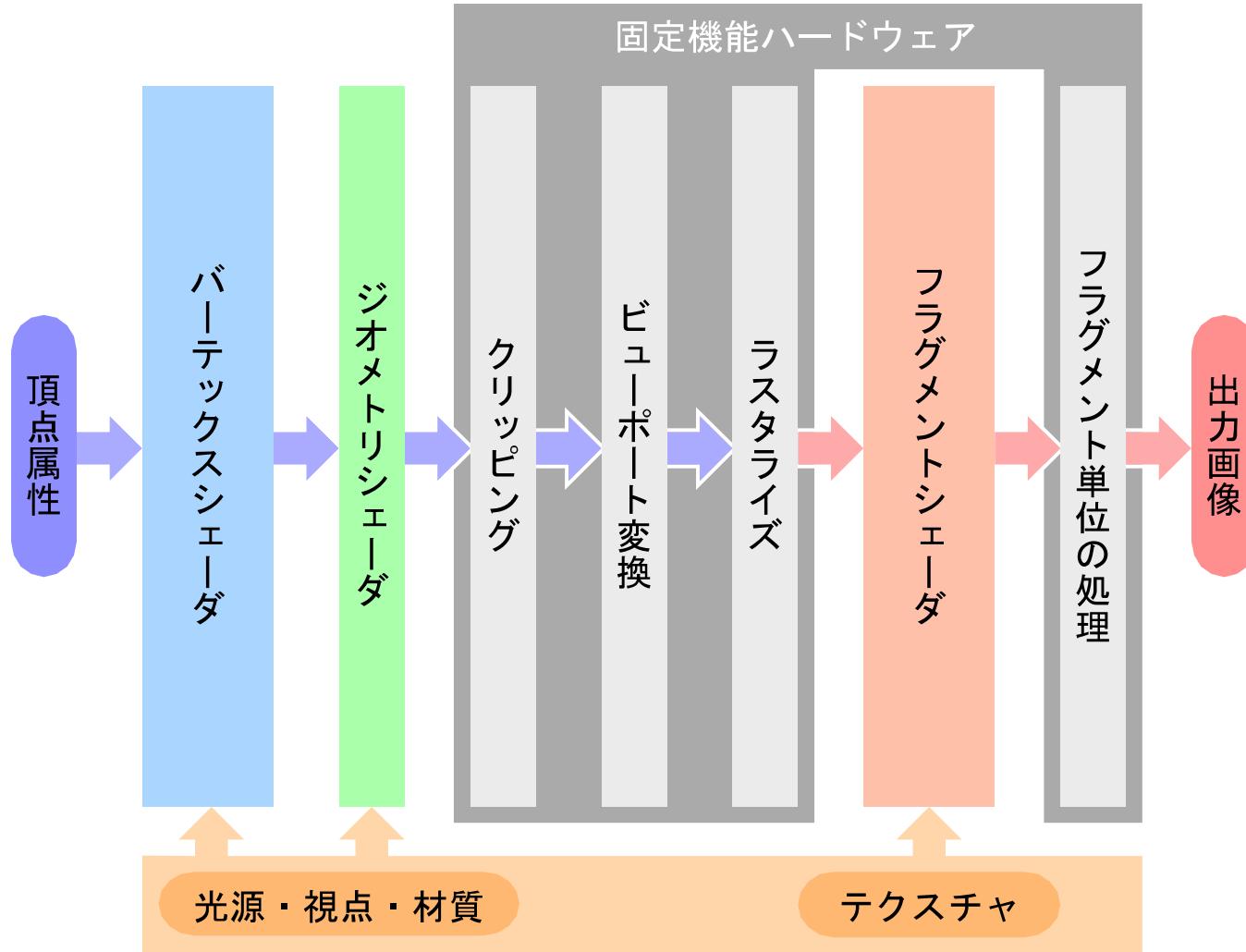


最初(DirectX 8)は
ループすらなかった

制御構造を備えて
汎用プログラミング
可能に(DirectX 9)

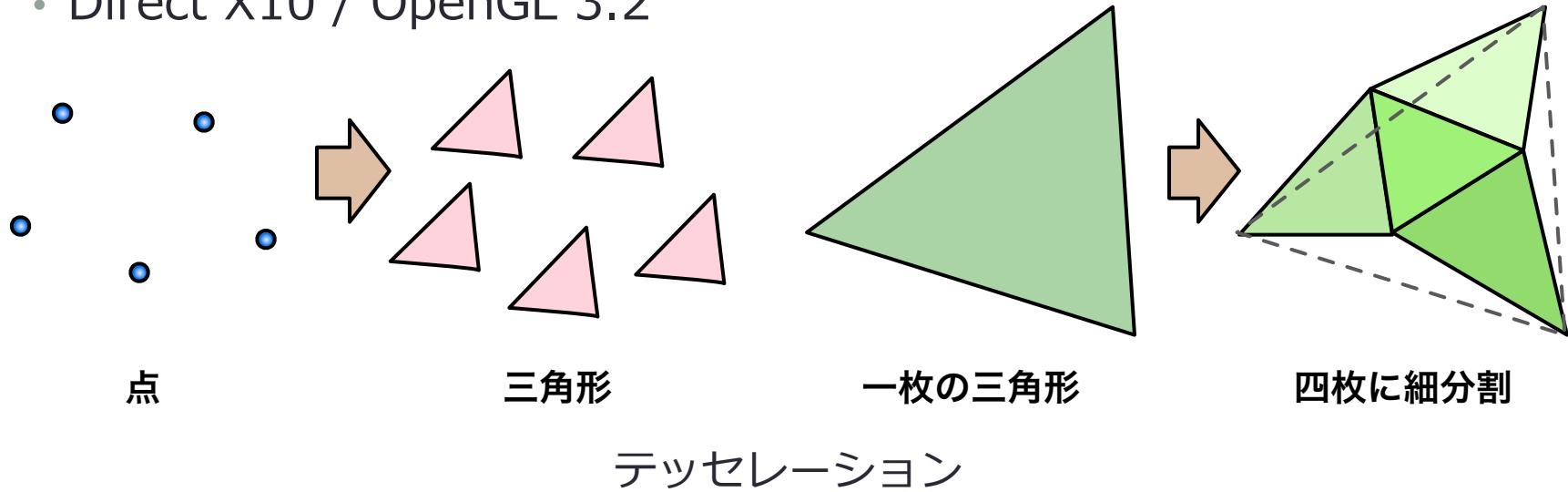
OpenGL 2.0

ジオメトリシェーダの追加

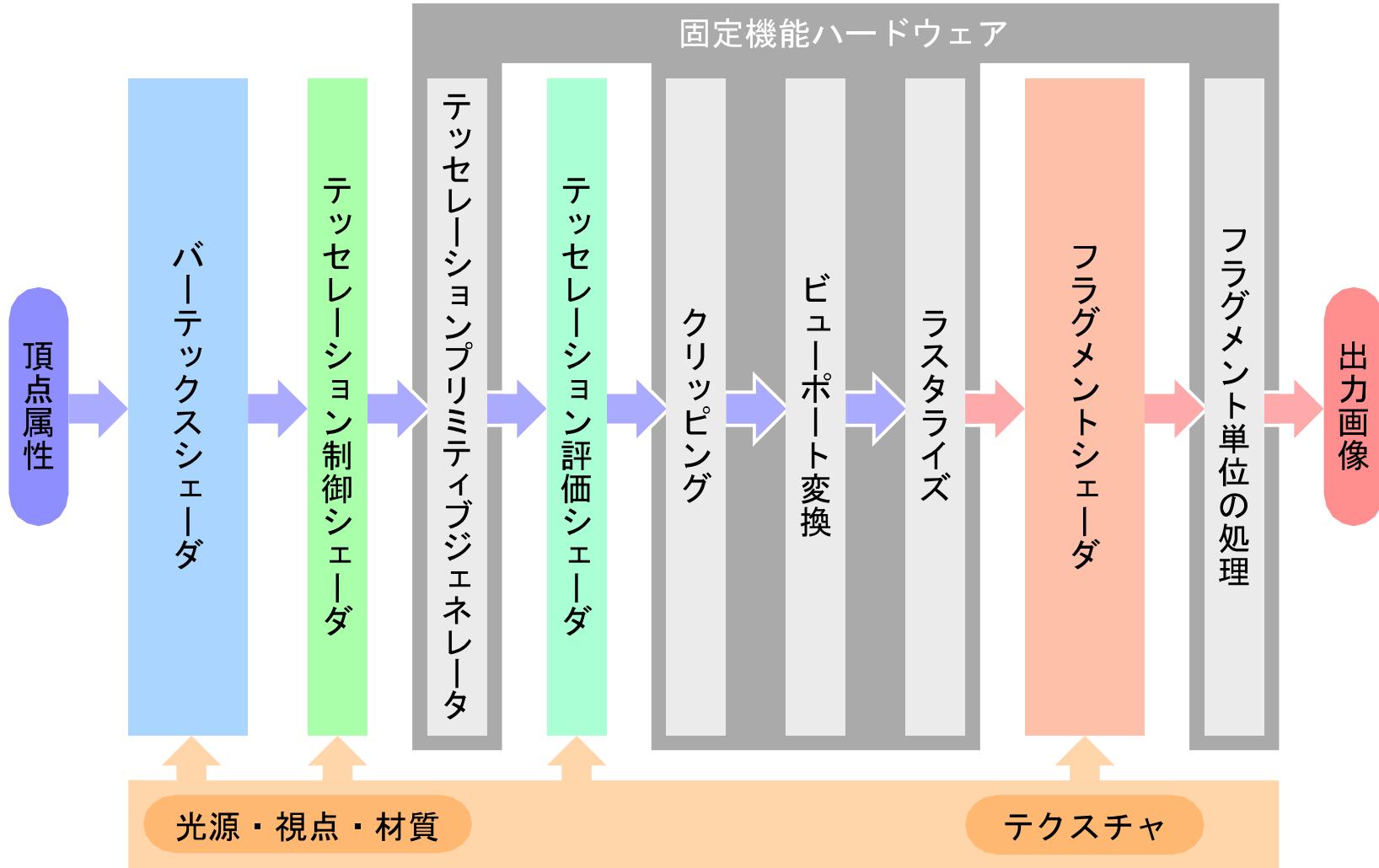


ジオメトリシェーダ

- ・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) という固定機能ハードウェアを制御する
 - ・オプション (使用しなくても良い)
 - ・Direct X10 / OpenGL 3.2



ジオメトリシェーダの細分化



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

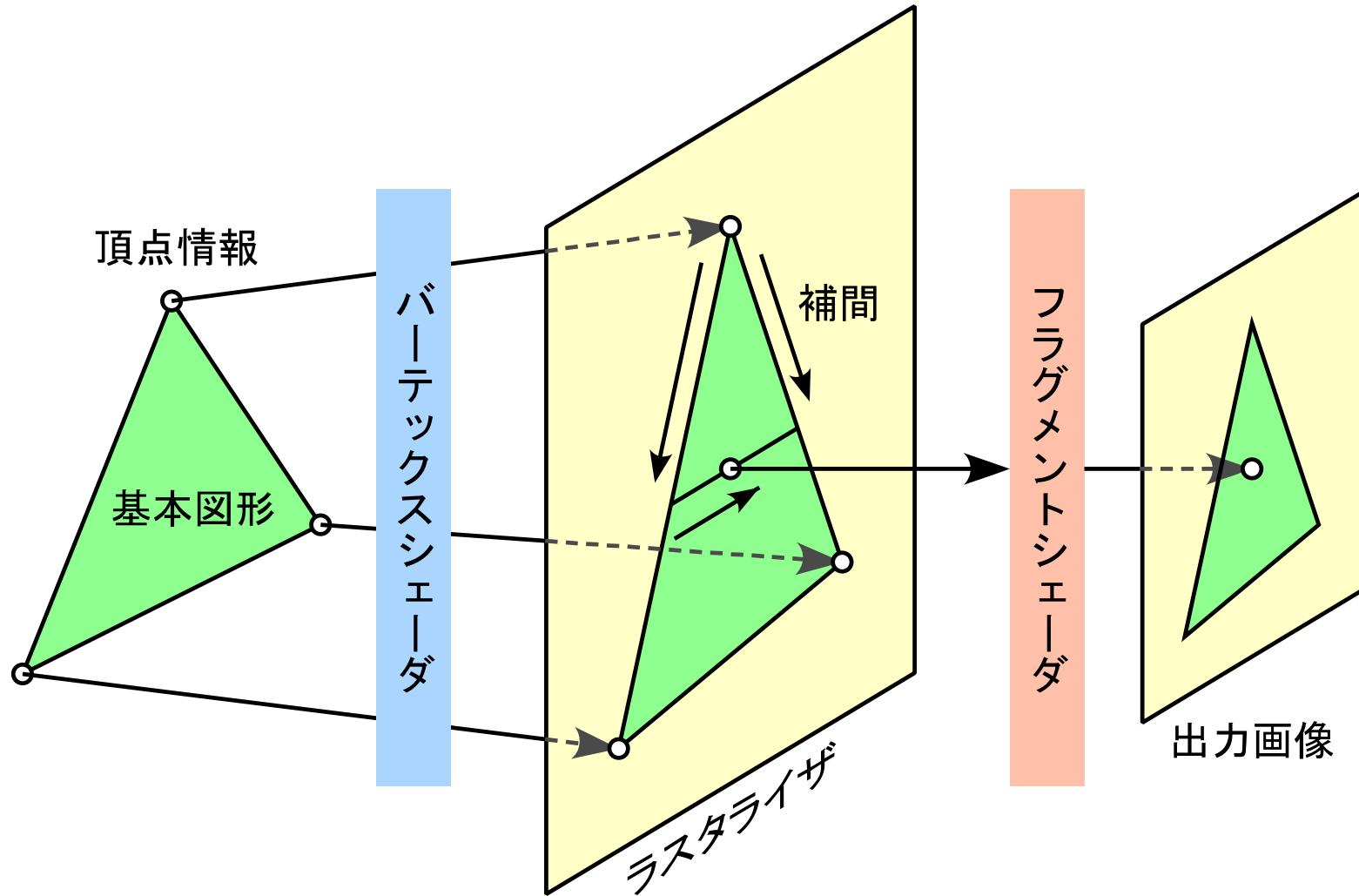
- **テッセレーション評価シェーダ**

- 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）

ラスタライザ



ラスタライザの役割

1. 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/テッセレーション評価シェーダ

2. 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

3. 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

4. フラグメントシェーダを起動する

シェーダプログラム

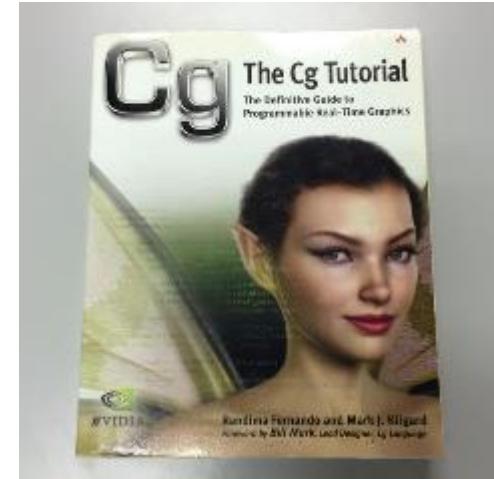
ソースプログラムのコンパイルとリンク

シェーダプログラムは GPU で実行される

- ・ビデオカードのドライバによる処理
 - ・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
- ・データの GPU 上のメモリへの転送
 - ・図形データ
 - ・画像データ
 - ・定数データ
- ・描画に使用する GPU 上のシェーダプログラムの指定
- ・図形の描画開始の指示
 - ・GPU 上のシェーダプログラムの実行

シェーディング言語

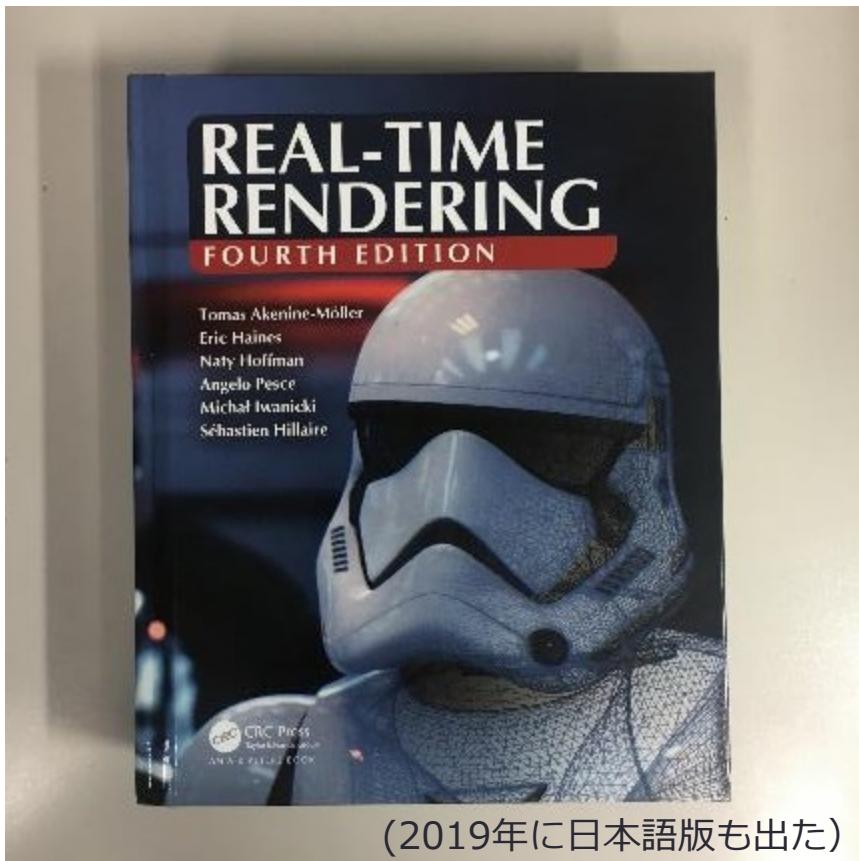
- ・シェーダのプログラミングに用いる
 - ・**Cg** (C for Graphics)
 - ・DirectX と OpenGL に対応
 - ・NVIDIA により開発
 - ・**HLSL** (High Level Shading Language)
 - ・DirectX に対応
 - ・Microsoft が NVIDIA の協力を得て開発
 - ・**GLSL** (OpenGL Shading Language, GLslang)
 - ・OpenGL に対応
 - ・OpenGL ARB (現在は Khronos Group) により開発
- ・いざれも**C言語**に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



この講義では **GLSL** を採用します

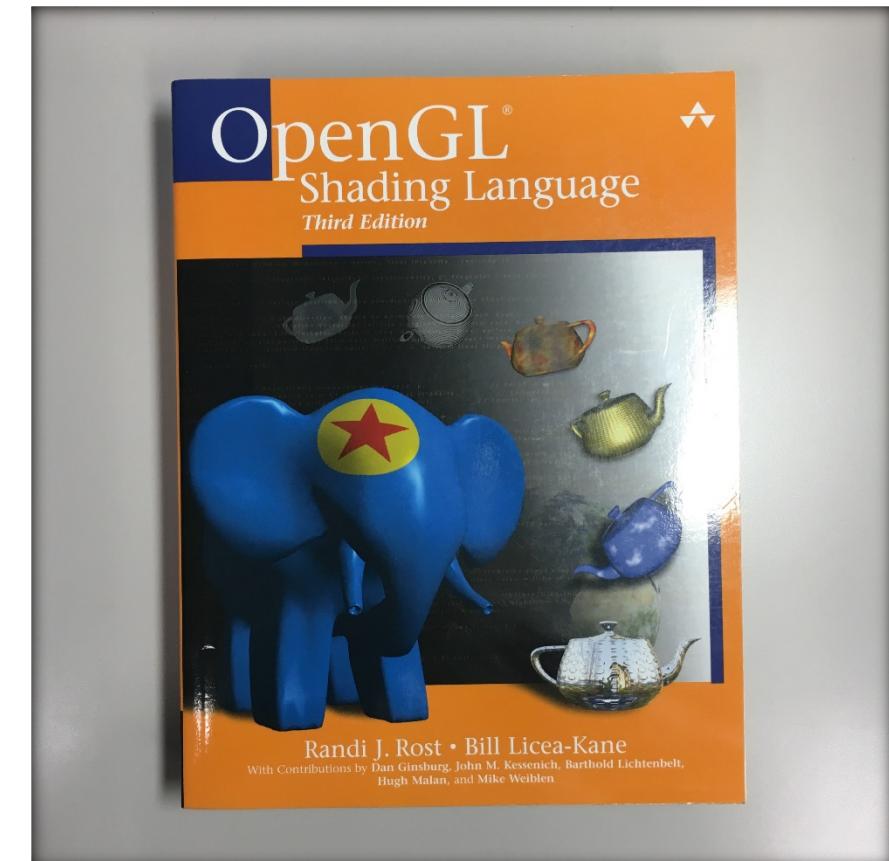
参考書

Real-Time Rendering (4th Ed.)

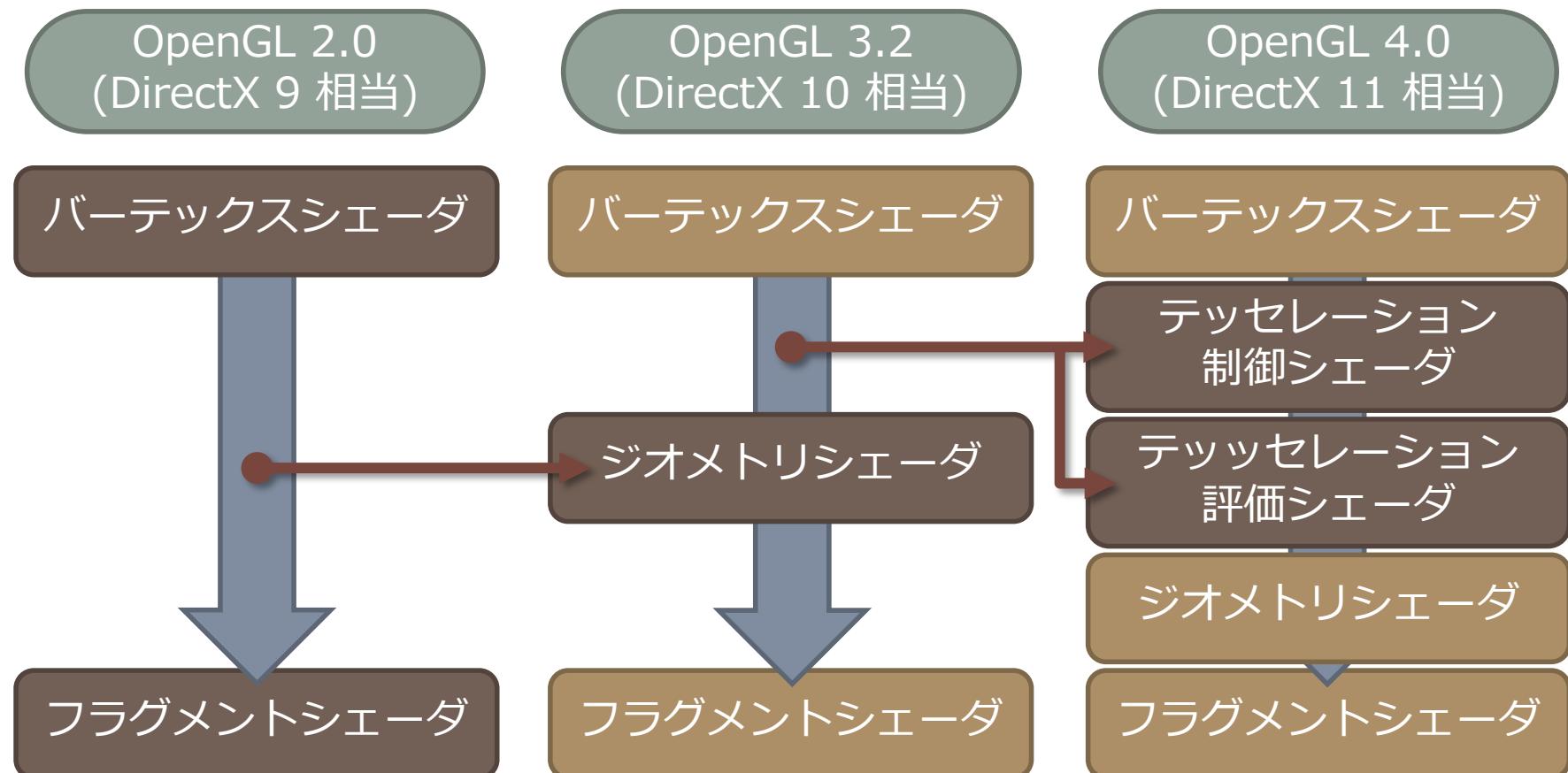


(2019年に日本語版も出た)

OpenGL Shading Language

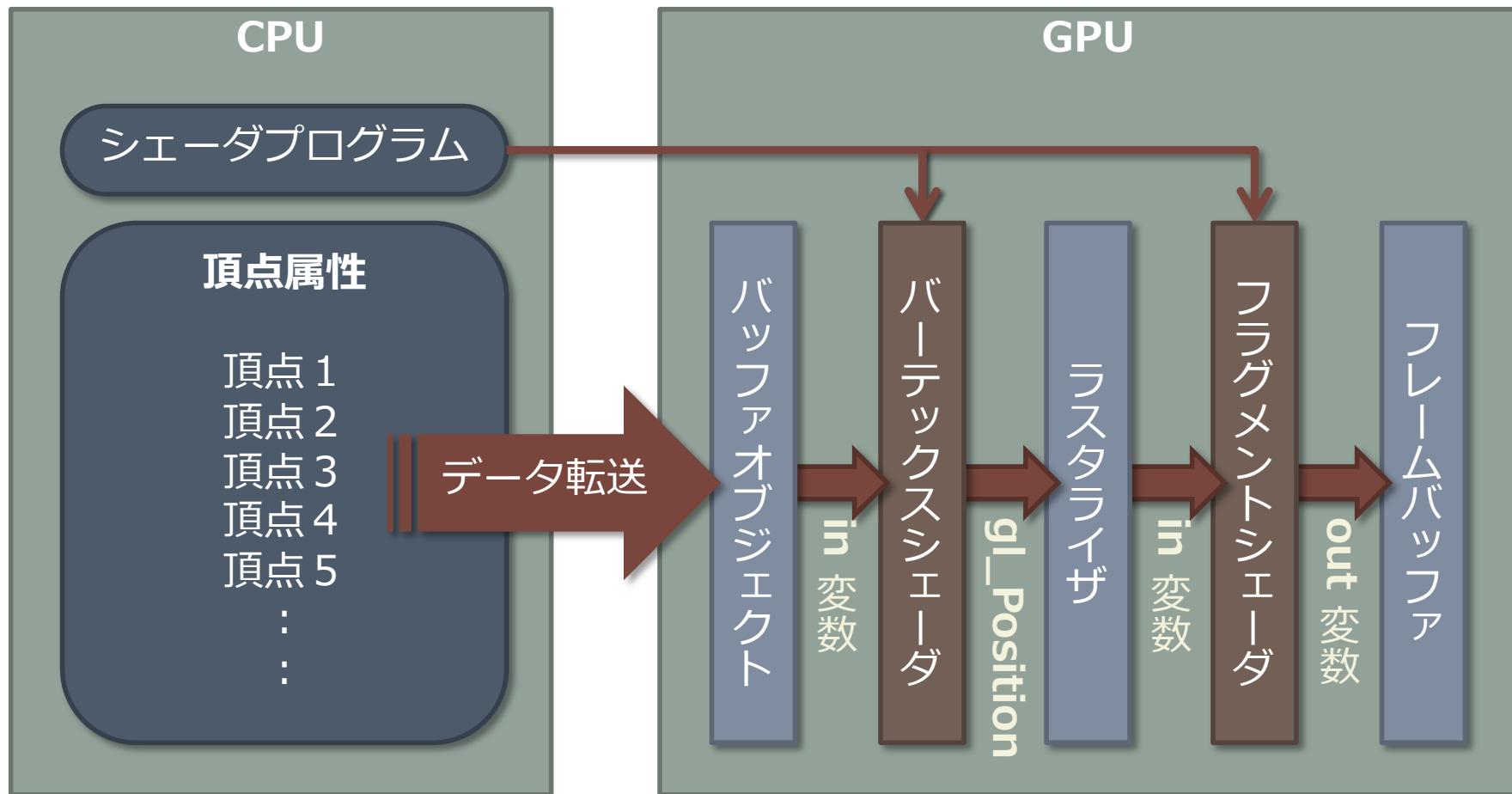


シェーダプログラムの種類

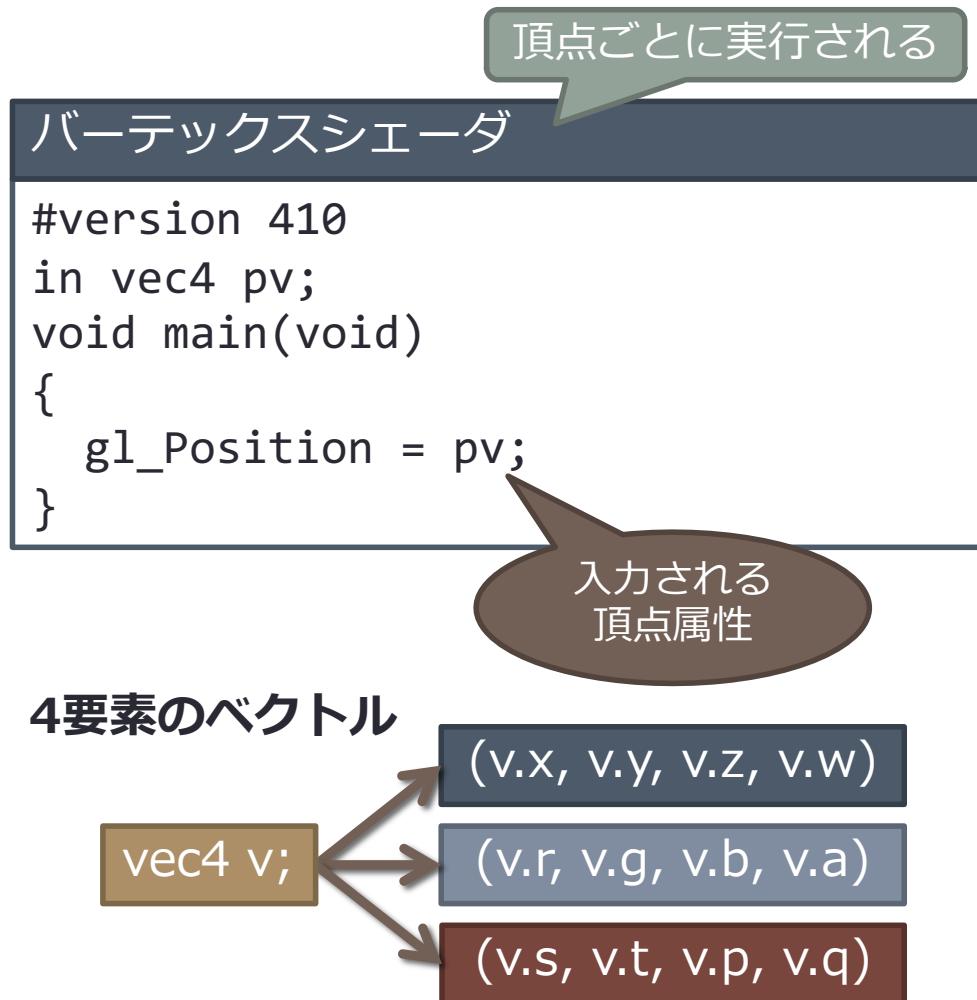


Vulkan (OpenGL Next Generation) / DirectX 12 は低レベルから作り直した

シェーダプログラムの呼び出し



バーテックスシェーダのソース



#version 410
GLSL version 4.1 プロファイルの宣言

in
シェーダの入力変数の宣言

vec4
32bit 浮動小数点 (float) 型の4要素のベクトルデータ型

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プログラムの出力先の組み込み変数

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から書き込み可能
 - 代入した値を読み出すことも可能
- クリッピングの対象になる
 - クリッピング空間にあるものだけが描画される

- **out 変数 (varying 変数)**

- 次のステージの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

画素ごとに実行される

フラグメントシェーダ

```
#version 410
out vec4 fc;
void main(void)
{
    fc = vec4(1.0, 0.0, 0.0, 1.0);
}
```

カラーバッファに結合した変数

出力する画素の色

out
シェーダの出力変数の宣言

fc
フラグメントシェーダプログラムの出力先（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
 - バーテックスシェーダから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), *read only*
- **vec4 gl_FragDepth**
 - 画素のデプス値, 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
- **out** 変数
 - 次のステージに送るデータ
 - フレームバッファのカラーバッファに結合

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

- ・バーテックスシェーダプログラムオブジェクトの作成
 - ・GLuint `vertShader` = **glCreateShader(GL_VERTEX_SHADER);**
- ・バーテックスシェーダソースプログラムの読み込み
 - ・**glShaderSource(vertShader, lines, source, &length);**
- ・バーテックスシェーダソースプログラムのコンパイル
 - ・**glCompileShader(vertShader);**
- ・フラグメントシェーダプログラムオブジェクトの作成
 - ・GLuint `fragShader` = **glCreateShader(GL_FRAGMENT_SHADER);**
- ・フラグメントシェーダソースプログラムの読み込み
 - ・**glShaderSource(fragShader, lines, source, &length);**
- ・フラグメントシェーダソースプログラムのコンパイル
 - ・**glCompileShader(fragShader);**

シェーダプログラムのリンク

- プログラムオブジェクトの作成
 - `GLuint program = glCreateProgram();`
- シェーダオブジェクトの取り付け
 - `glAttachShader(program, vertShader);`
 - `glAttachShader(program, fragShader);`
- シェーダオブジェクトの削除
 - `glDeleteShader(vertShader);`
 - `glDeleteShader(fragShader);`
- シェーダプログラムのリンク
 - `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

- ・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 = glBindAttribLocation(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 変数

- ・ フラグメントシェーダの out 変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

- 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;
...
```

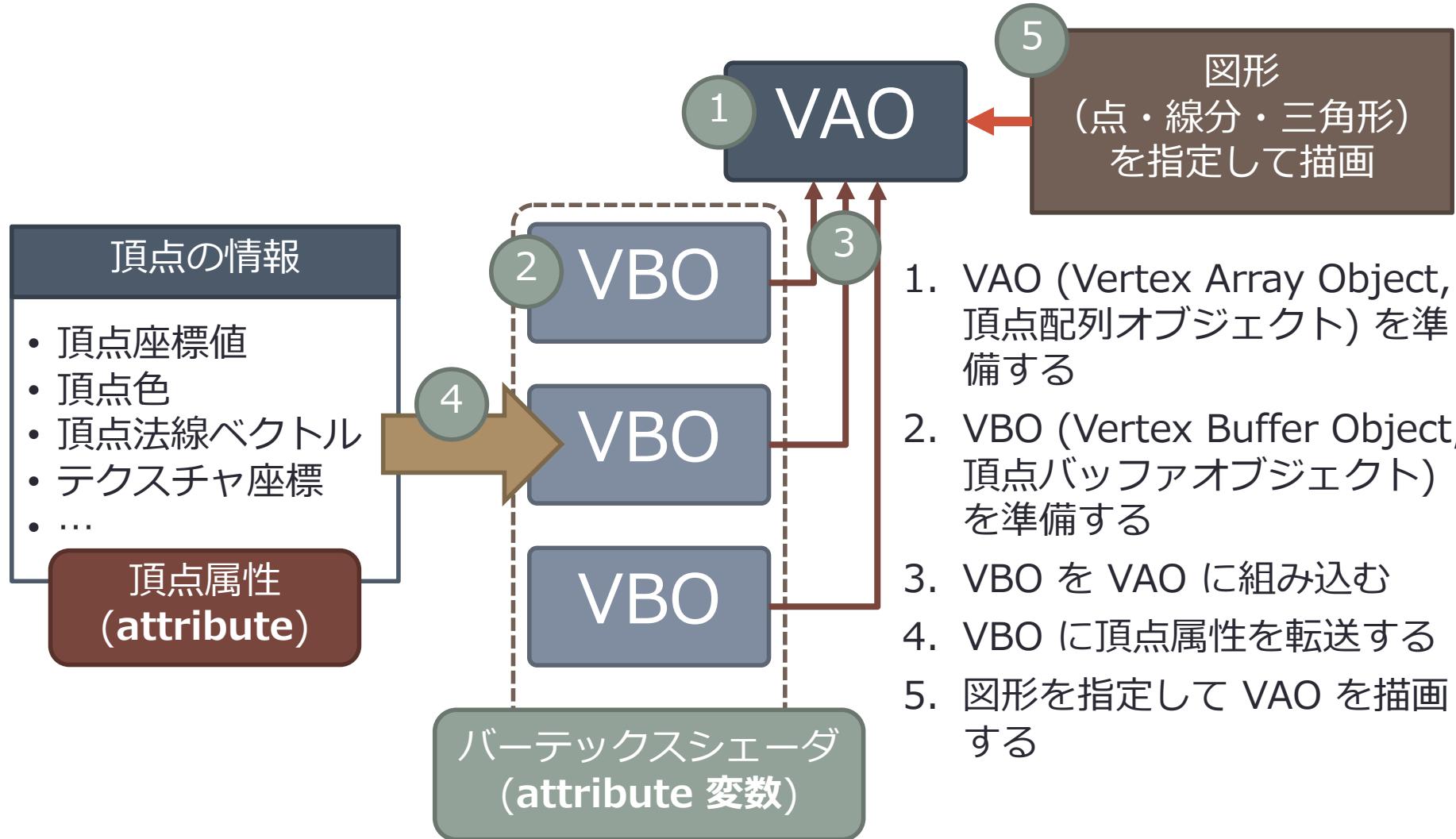
フラグメントシェーダ

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

図形描画

基本図形と頂点情報

OpenGL の図形描画の手順



図形描画の手順

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 (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

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

VBO (Vertex Buffer Object) の準備

- 頂点属性を GPU 側のメモリに保持する
 - 頂点属性をGPU 側のメモリにあらかじめ転送しておく
- N 個の VBO を作成する
 - GLuint `vbo[N];`
 - **glGenBuffers(N, vbo);**
- 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);**
- 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);**
- 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 Buffer Object の大きさ
 - 配列 *p* を全部送るなら `sizeof p`
- *data*
 - 確保した Buffer Object 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 (Buffer Object の確保のみ行う)

glBufferData() の引数 *usage*

- Buffer Object 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** – Buffer Object へのアクセスの頻度
 - STATIC
 - データは1回の変更に対して何回も使用される (つまり, あんまり変更されない)
 - 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 (つまり, 描画のたびに変更される)
 - DYNAMIC
 - データは何回も変更され頻繁に使用される (つまり, 描画より高い頻度で変更される)
- **YYYY** – Buffer Object へのアクセスの性質
 - DRAW
 - データはアプリケーション (CPU側) から変更され描画に使用される
 - READ
 - データはアプリケーション側からの問い合わせによるGPU側からの読み出により変更される
 - COPY
 - データはGPU側からの読み出により変更され描画に使用される (CPU はあまり関与しない)

VBO と attribute 変数の対応付け

- 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)
- バーテックスシェーダの `in` 変数の `index` を有効にする
 - glEnableVertexAttribArray(index);**

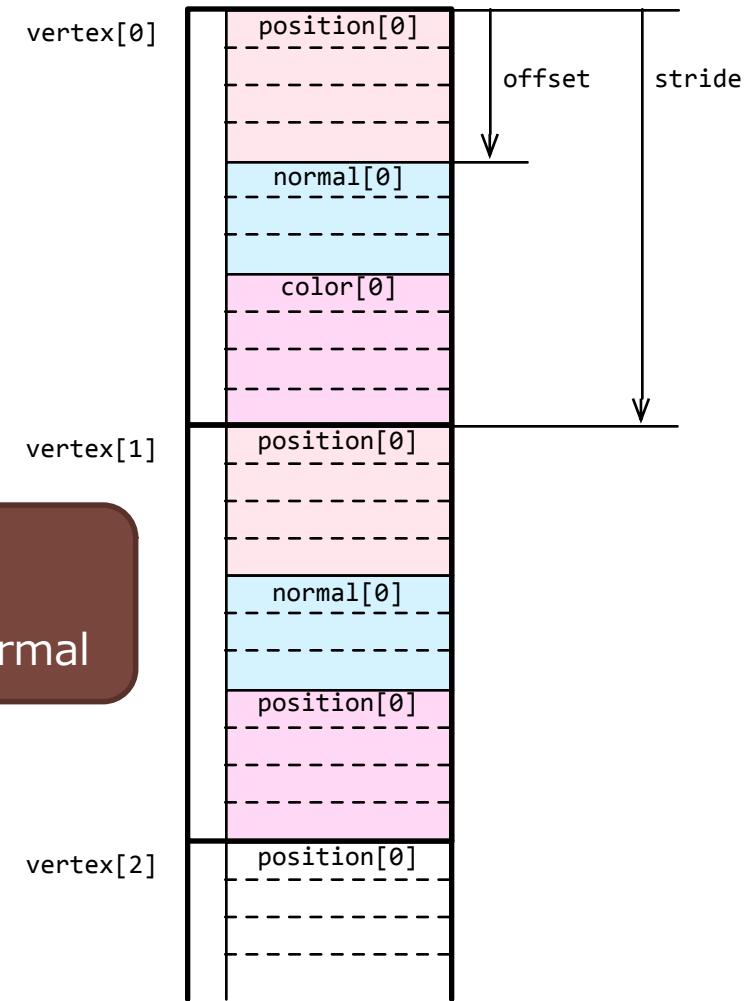
BUFFER_OFFSET(*bytes*) マクロ

- 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
- 確保した Buffer Object の先頭から使うなら `bytes = 0`

頂点データのインター／リーブな配置

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



Buffer Object のデータの更新

- 既存の Buffer Object にデータを転送する
 - glBufferSubData**(*target*, *offset*, *size*, *data*);
 - target*: GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER
 - offset*: 転送先の Buffer Object 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
- Buffer Object からアプリケーションにデータを取得する
 - 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する
 - void *glMapBuffer**(*target*, *access*);
 - target*: 同上
 - access*: GL_READ_ONLY, GL_WRITE_ONLY, GL_READ_WRITE

glMapBuffer()

```
// GPU 上の Buffer Object 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;

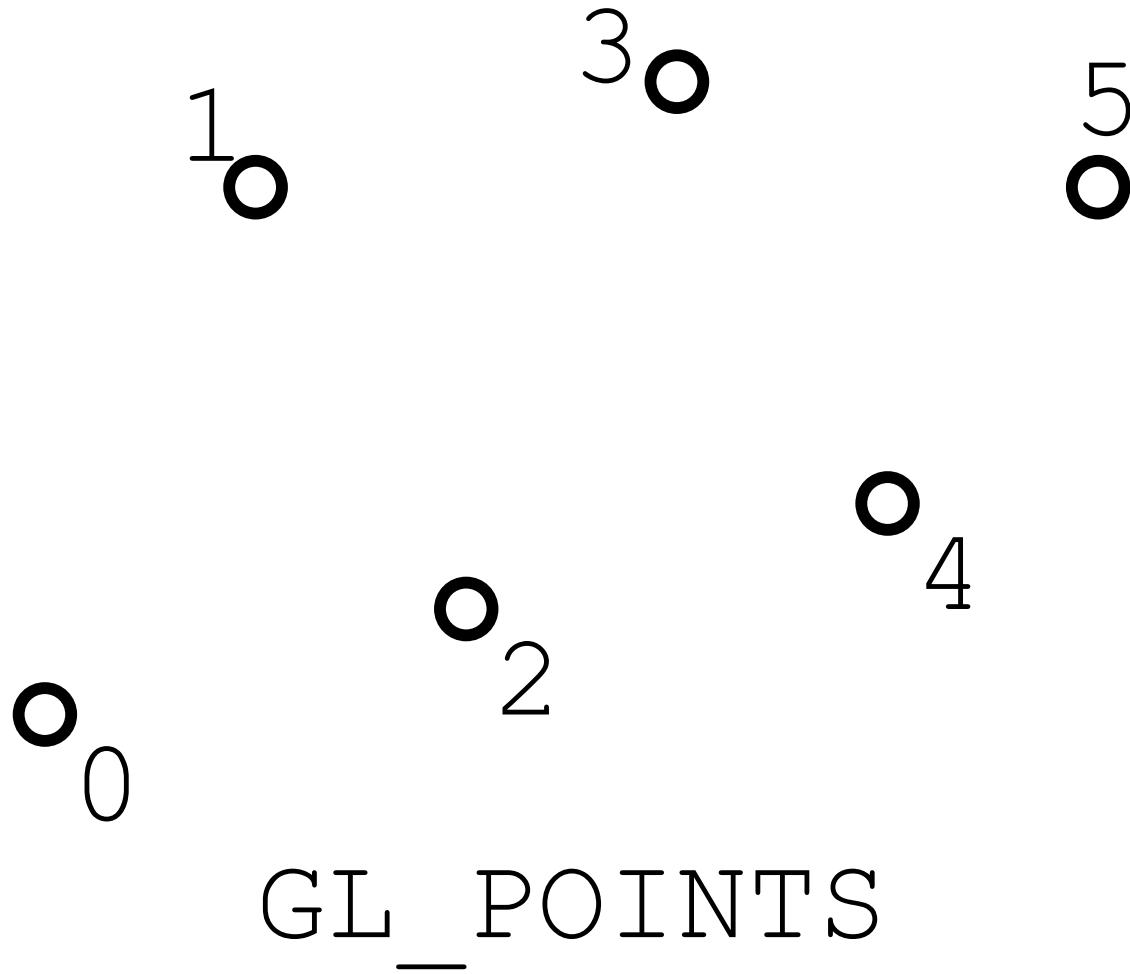
// 6 番目のデータ p[5] からデータを取り出す (access が GL_WRITE_ONLY 以外)
GLfloat x = p[5][0];
GLfloat y = p[5][1];
GLfloat z = p[5][2];
GLfloat w = p[5][3];

// アプリケーションのメモリ空間から Buffer Object を切り離す
glUnmapBuffer(GL_ARRAY_BUFFER);
```

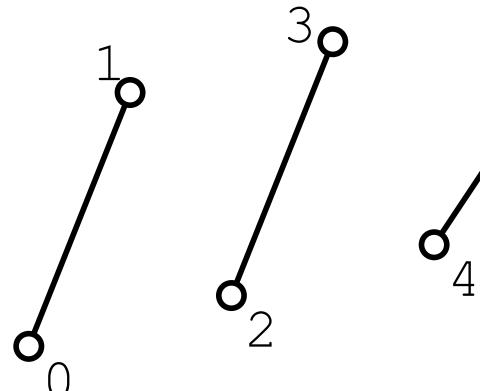
図形の描画

- 使用するシェーダプログラムを選ぶ
 - **glUseProgram(program);**
- 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

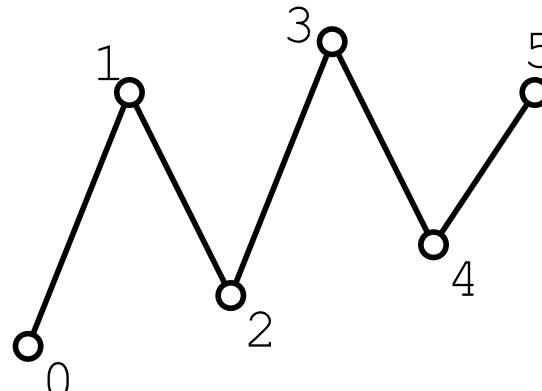
基本图形 - 点



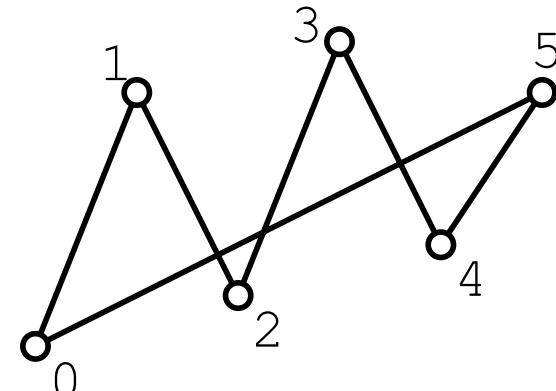
基本図形 – 線分と三角形



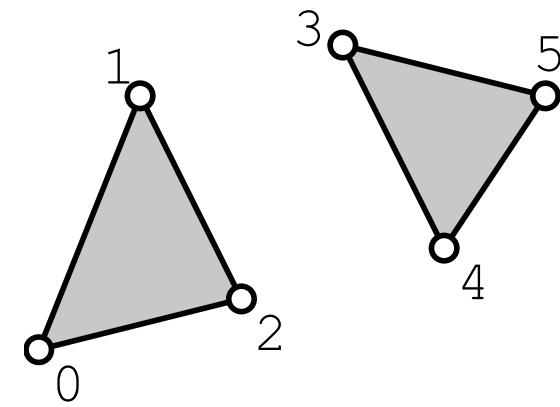
GL_LINES



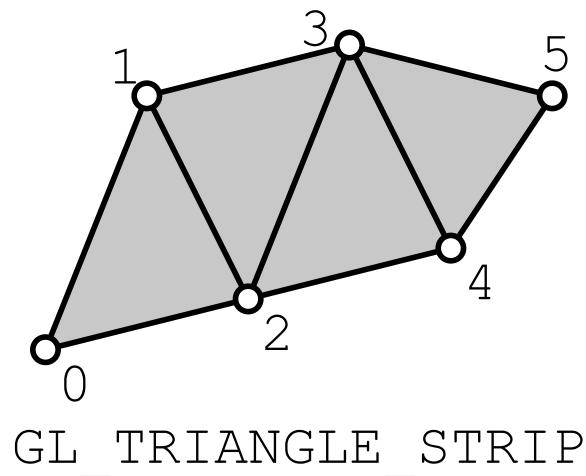
GL_LINE_STRIP



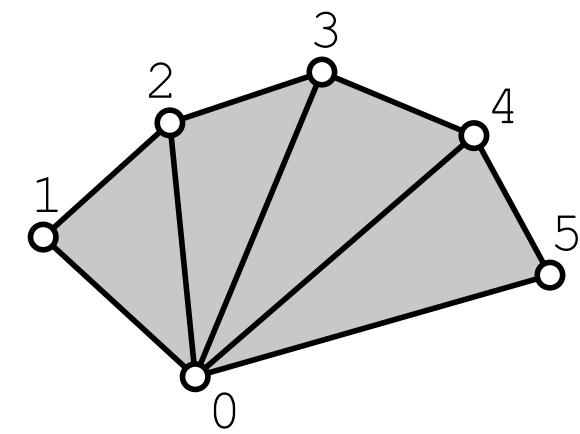
GL_LINE_LOOP



GL_TRIANGLES

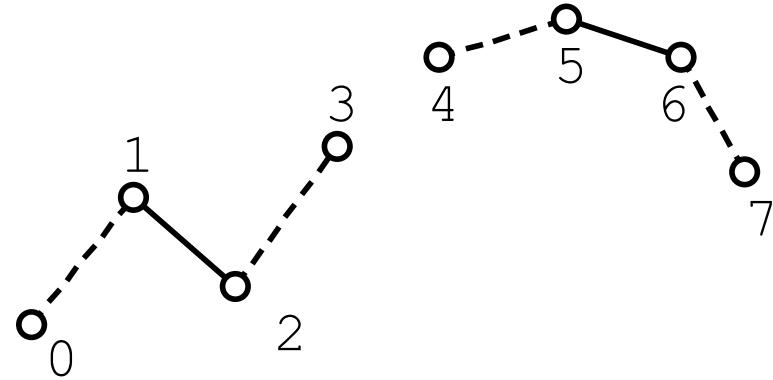


GL_TRIANGLE_STRIP

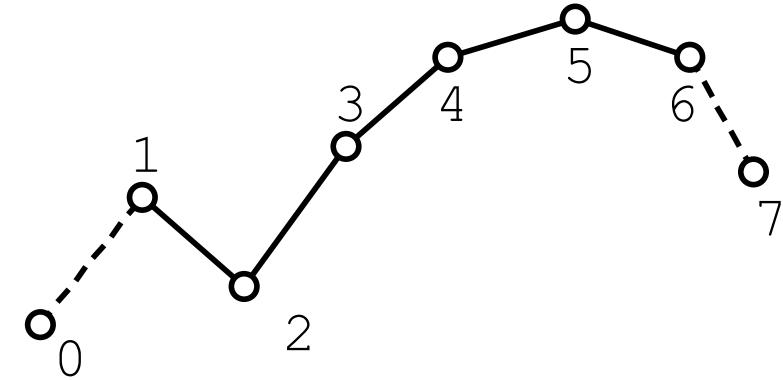


GL_TRIANGLE_FAN

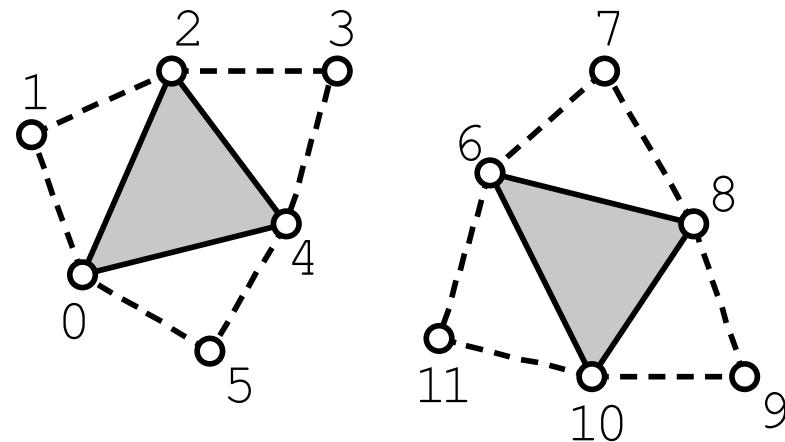
ジオメトリシェーダで追加されたもの



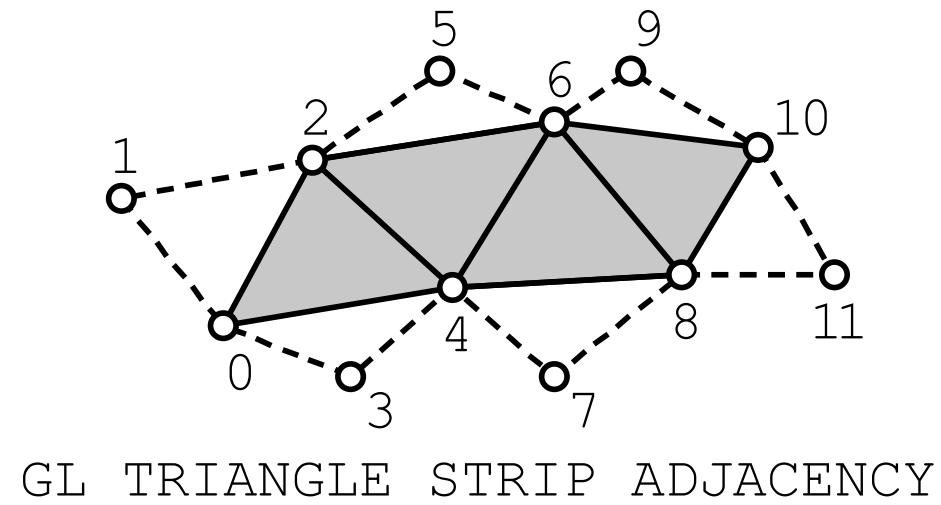
GL_LINES_ADJACENCY



GL_LINE_STRIP_ADJACENCY

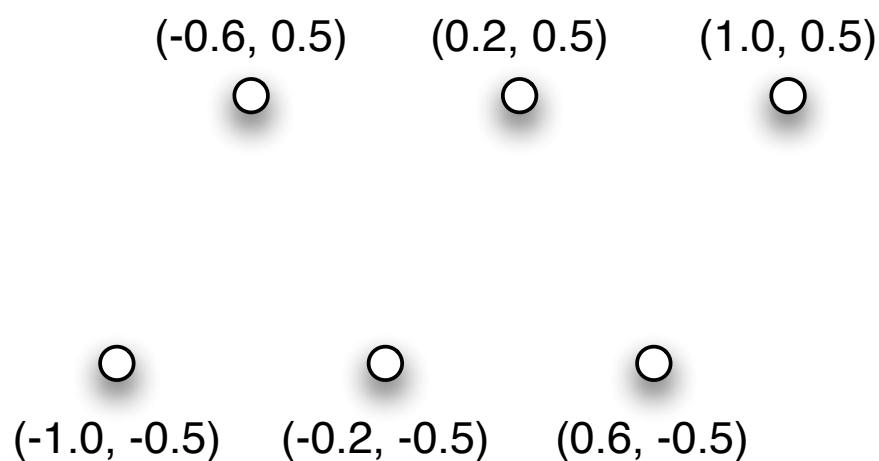


GL_TRIANGLES_ADJACENCY



GL_TRIANGLE_STRIP_ADJACENCY

頂点属性データ



```
// 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);
```

in 変数の

1 頂点あたりのデータ数 (次)

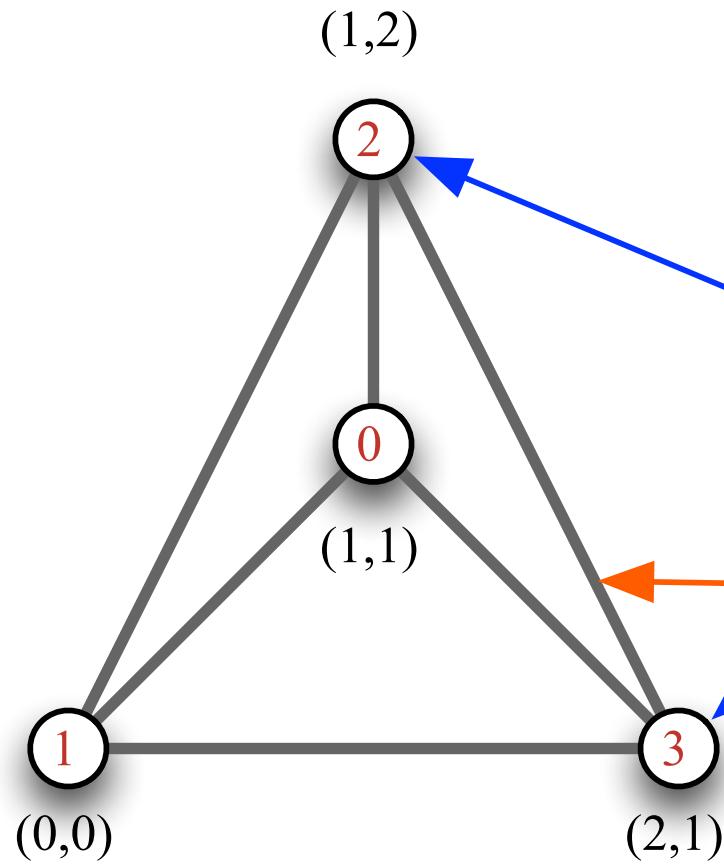
glDrawArrays() による描画

```
// 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

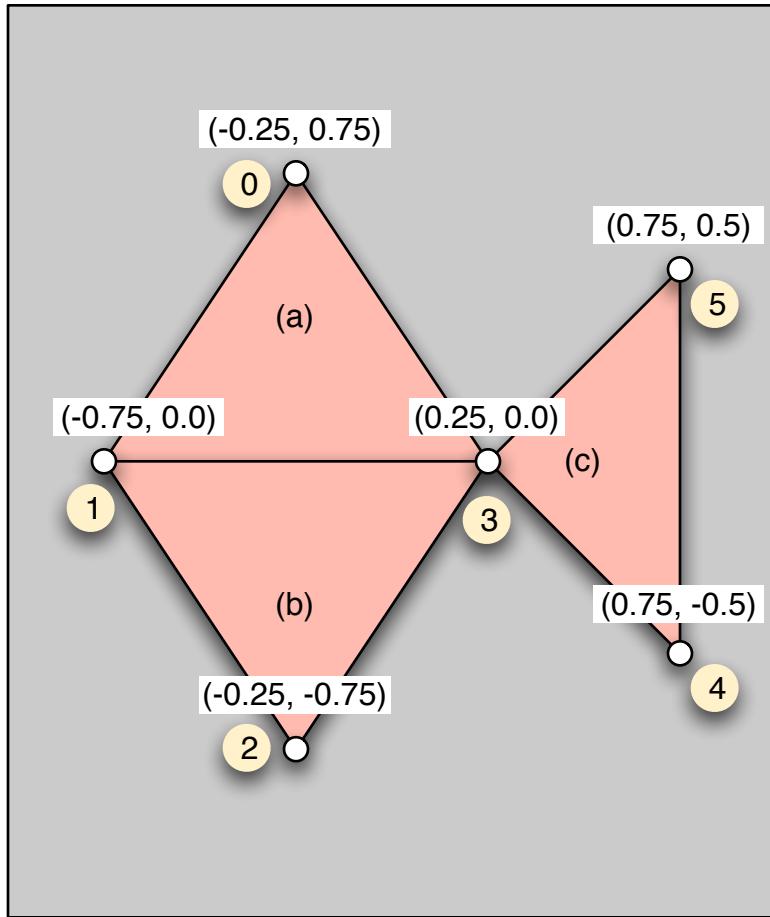
頂点インデックスの Buffer Object

- 頂点インデックスも GPU に送る必要がある
 - もうひとつ Buffer Object 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 Buffer Object に転送する
 - `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[j]);`
 - `glBufferData(GL_ELEMENT_ARRAY_BUFFER,
sizeof edge, edge, usage);`
- VAO に VBO を登録する際に一緒に登録する

頂点インデックスを使った図形の描画

- 使用するシェーダプログラムを選ぶ
 - `glUseProgram(program);`
- 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);
```

さつき
と同じ

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);
```



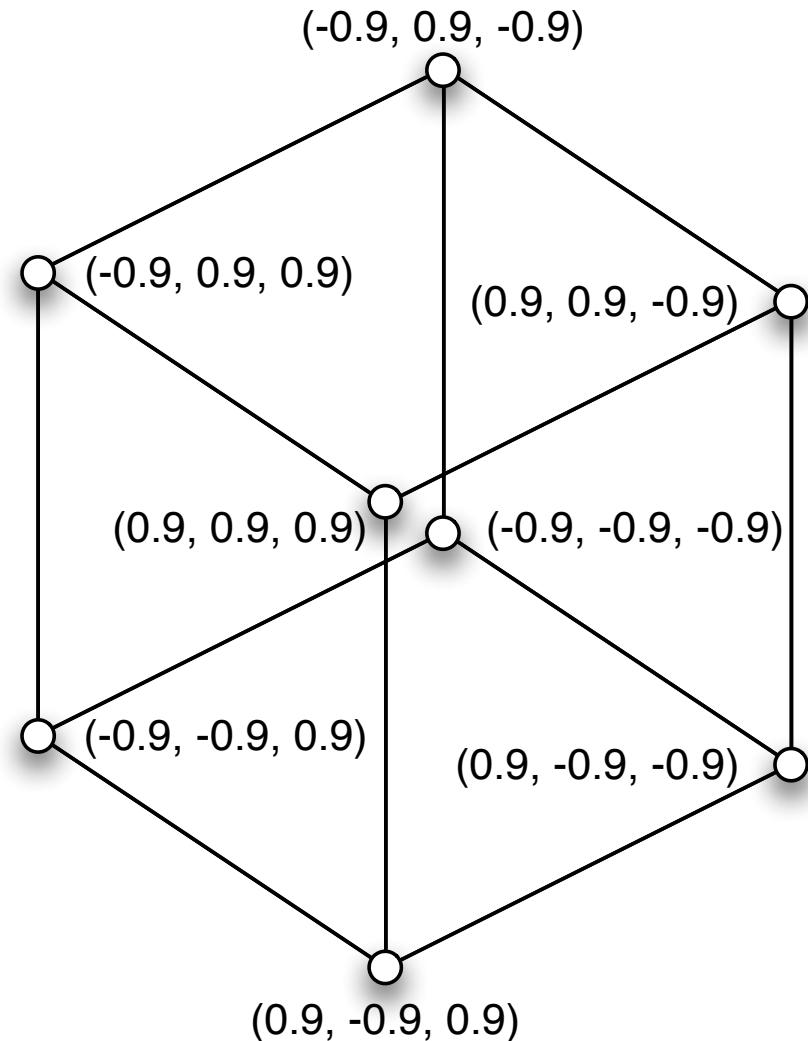
インデックス
配列の要素数

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

課題

- 右の図を **GL_LINES** で描くための点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



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
 - “学生番号.txt” というファイル名を使うのは第1回の宿題だけです
 - アップロード先
 - <https://www.wakayama-u.ac.jp/~tokoi/lecture/gg/upload/>

結果

このような画像が表示されれば、多分、正解です

