第4章　GPU (Graphics Processing Unit)

4.1　レンダリングパイプラインのハードウェア化

4.1.1　フラグメント処理のハードウェア化

ジオメトリ処理のパイプラインとフラグメント処理のパイプラインを続けて書くと、図 63 のようになります。このパイプラインは、最初、すべてソフトウェアで実装されていました。

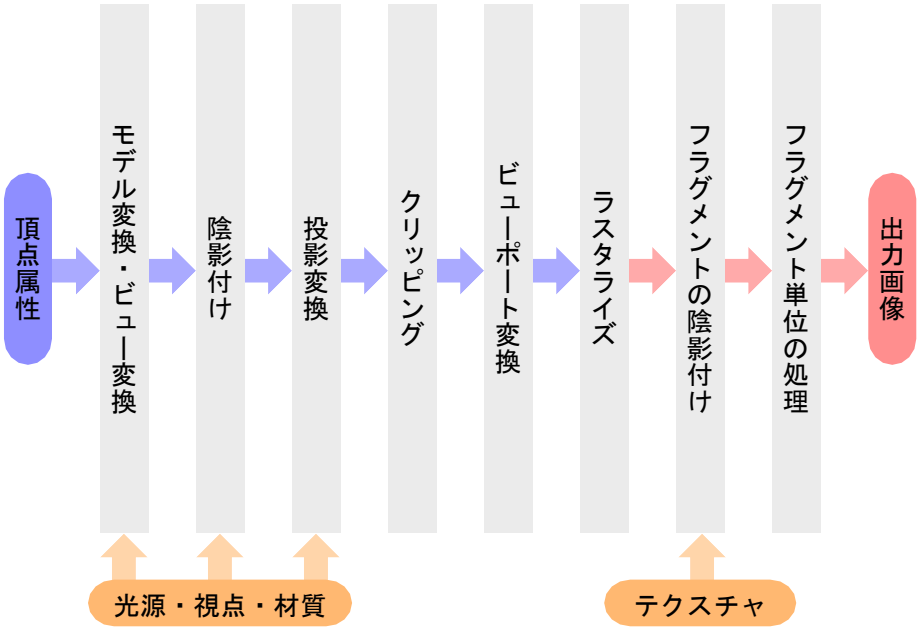

図 63 ソフトウェア中心のパイプライン

このようなソフトウェアによる処理は、それをハードウェアで置き換えることによって、処理速度を大幅に向上することができます。そこで、最初はクリッピングからラスタライズ以降の処

理がハードウェアで置き換えられました (図 64)。これは、このステージの処理がすべて整数で実行可能のためにハードウェアによる実装が容易であったことと、単純なくり返し処理が中心のためにハードウェアによる高速化の効果が大きいと考えられたからです。

● クリッピング

クリッピングでは標準ビューボリューム (クリッピング空間) の境界と線分あるいは三角形の辺との交差点を求め、その位置でこれらを分割します。このハードウェアはクリッピングディバイダと呼ばれ、二分法を用いてこの処理を実行します。

● ラスタライズ

ラスタライズの三角形セットアップのステージでは、三角形の辺の傾きなどを求め、それから操作変換のための係数を算出します。この処理には少数の整数演算と条件判断が用いられます。

走査変換のステージでは、三角形の内部にあるフラグメントを選択し、それを塗り潰します。これは単純な整数の加減算のくり返し処理になります。

● 隠面消去

他のものに隠されて見えない部分を描かないようにする隠面消去処理では、頂点のデプス値の処理対象のフラグメントにおける補間値を求め、その値とデプスバッファとの比較を行います。これも整数演算と条件判断が必要になります。

● フラグメントの陰影付け

頂点のテクスチャ座標の、処理対象のフラグメントにおける補間値を求め、その値を用いてテクスチャメモリから色をサンプリングします。これに、やはり頂点における陰影の処理対象のフラグメントにおける補間値を乗じて、フラグメントの色を決定します。

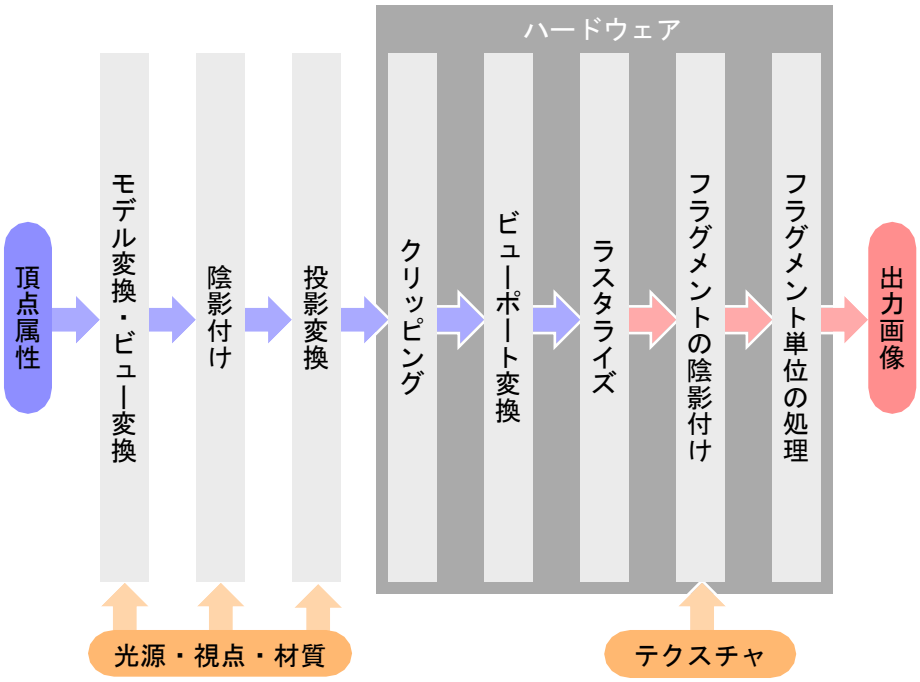

図 64 フラグメント処理のハードウェア化

4.1.2　ジオメトリ処理のハードウェア化

ジオメトリ処理のハードウェア化は、そのために必要な実数演算 (浮動小数点演算) のハードウェアのコストが高かったために、フラグメント処理のハードウェア化より若干遅れました。しかし、その後グラフィックスハードウェアに実数演算機能が追加され、この処理もグラフィックスハードウェア側で行われるようになりました。これによりレンダリングパイプライン全体のハードウェア化が達成されたことから、このハードウェアを GPU (Graphics Procssing Unit) と呼ぶようになりました。このハードウェアは主に座標変換 (Transform) と照明計算 (Lighting、陰影付け) を実行するので、この機能はハードウェア T & L (Transform and Lighting) と呼ばれます。

● **座標変換**

CG では座標値を 4 次元の同次座標で表し、座標変換には 4 行 4 列の正方行列を用います。そのために 4 次のベクトル同士の内積計算 (積和計算) を実行する演算器が用意されます。4 行 4 列の正方行列と 4 次のベクトルとの積は、この内積計算 4 回で実行できます。また、4 行 4 列の正方行列同士の積は、行列とベクトルの積 4 回あるいは内積計算 16 回で実行できます。

● **照明計算**

照明計算 (陰影付け) には、一般的な四則演算と逆数のほか、平方根の逆数、指数計算、および内積計算、外積計算、そして条件判断や値を指定した範囲内に収めるクランプなどが用いられます。外積計算はベクトルの要素の順序の入れ替え機能 (Swizzling) を組み合わせることで、ベクトルの要素ごとの積 2 回と和 1 回で実行できます。

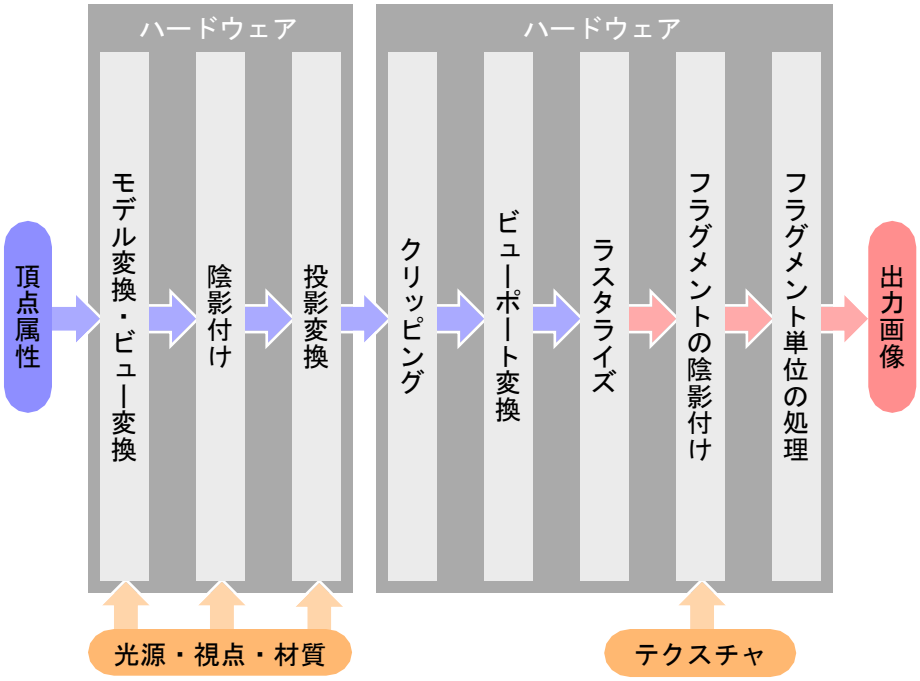

図 65 ジオメトリ処理のハードウェア化

4.1.3　プログラマブルシェーダによる置き換え

標準的なレンダリングパインプラインの構成はほぼ固定していたので、当初は前述のようなハ

ードウェアによる固定的な実装が行われました。その後、グラフィックスハードウェアの普及が進み、その応用や映像表現に対する要求の多様化に対応して、このレンダリングパイプラインには新しい機能がどんどん追加されていきました。

しかし、要求されるすべての機能をハードウェアで実装することは困難です。これはハードウェアを複雑にするだけでなく、ソフトウェア的にも使いこなすのが難しいものになってしまいます。そこで、それをソフトウェア的に実装できるように、GPU の一部の機能に (再び) CPU のようにプログラミングできるように改良が加えられました。この仕組みをシェーダといいます。

まず、ジオメトリ処理のステージが**バーテックスシェーダ**に、フラグメント単位の陰影付けのステージが**フラグメントシェーダ**に置き換えられました。**シェーディング**は陰影付けのことを指しますが、**シェーダ**は陰影付け以外に座標変換などのさまざまな処理を行います。この部分がプログラマブル化されたことにより、行列による座標変換以外の方法で頂点位置の決定を行ったり、画素ごとに精密な陰影付けを行ったりすることができるようになりました。

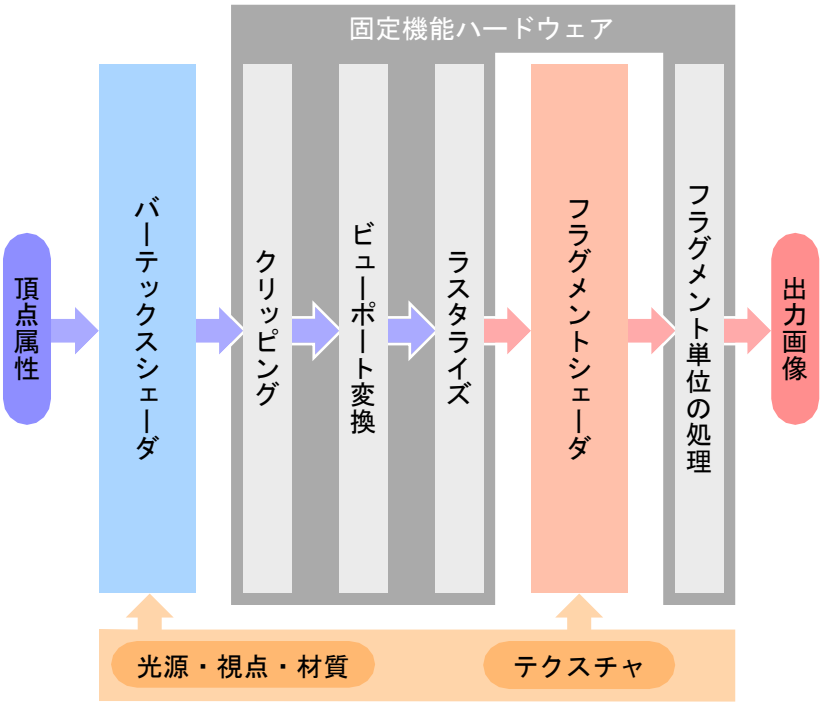

図 66 プログラマブルシェーダの導入

4.1.4 ジオメトリシェーダの追加

グラフィックスハードウェアは、ここまでアプリケーションから送られたジオメトリデータをそのまま画面表示することが仕事でした。したがって GPU といえども、その内部でジオメトリデータを生成することはありませんでした。

生成する映像の品質を上げるために表示するオブジェクトのディティールを向上しようとすれば、アプリケーションから GPU に送るジオメトリデータの量が増大します。ところが、今のところ GPU は、CPU から見れば (それが CPU に内蔵されていたとしても) 外部ハードウェア (peripheral) 扱いですから、CPU から GPU へのデータの送出にはデータのコピーが発生します。

このデータのコピーは、外付け (discrete) の GPU であれば PCI-Express などの外部バスを経由して行われますから、単位時間あたりのデータの転送量には限界があります。いくら GPU を増強してデータの処理性能を向上しても、そのままではデータの転送の待ち時間のために GPU の性能が発揮できないということが起こり得ます。

そこで、アプリケーションではオブジェクトの概形のジオメトリデータを生成し、シェーダプログラムによって、GPU の内部でディティールとなるジオメトリデータを生成するということが考えられました。そのためのシェーダが**ジオメトリシェーダ**です。

ジオメトリシェーダは、前段のバーテックスシェーダから送られてきたジオメトリデータ (基本図形と頂点属性) を受け取り、**テッセレーション** (基本図形の細分割) などの処理を行って、新しい基本図形と頂点属性を生成します。

なお、このシェーダはオプションであり、GPU でジオメトリデータを生成する必要が無ければ省略することができます。

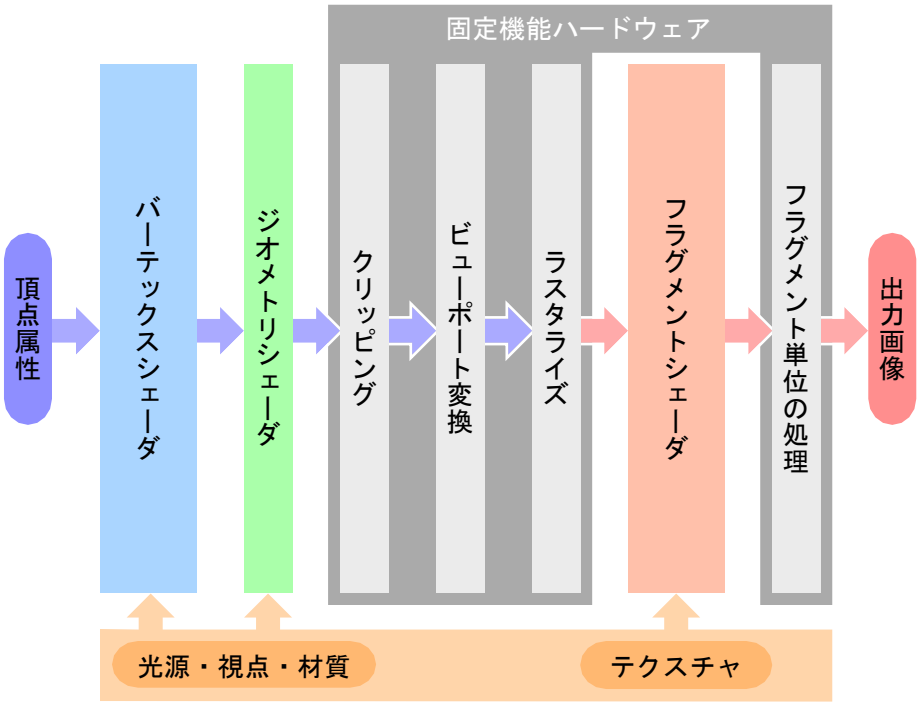

図 67 ジオメトリシェーダの追加

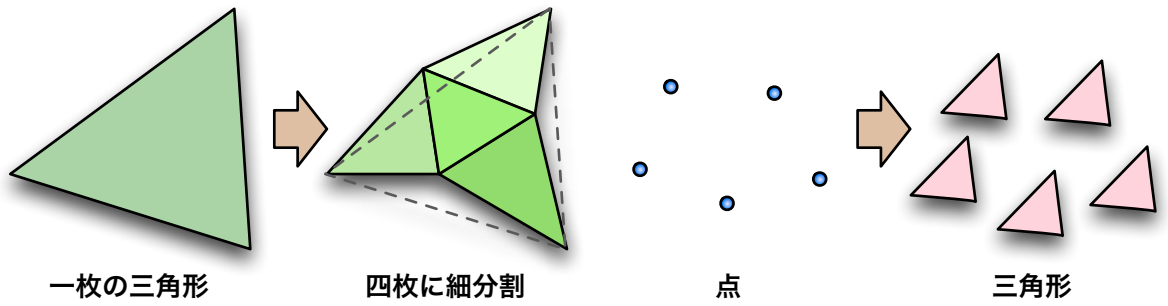

図 68 テッセレーションの例

4.1.5　ジオメトリシェーダの細分化

ジオメトリシェーダでは、頂点属性の設定をテッセレーションの制御と同時に行い、生成されたジオメトリデータはそのままラスタライザに出力します。しかし、生成されたジオメトリデータに対しても座標変換などのジオメトリ処理を加えたいこともあります。そこでジオメトリシェーダは、テッセレーションの方法を指定する**テッセレーション制御シェーダ**と、実際にテッセレーションを行う固定機能ハードウェアの**テッセレータ**、そして生成されたジオメトリデータに対してジオメトリ処理を行う (バーテックスシェーダに相当する) **テッセレーション評価シェーダ**に細分化されました。

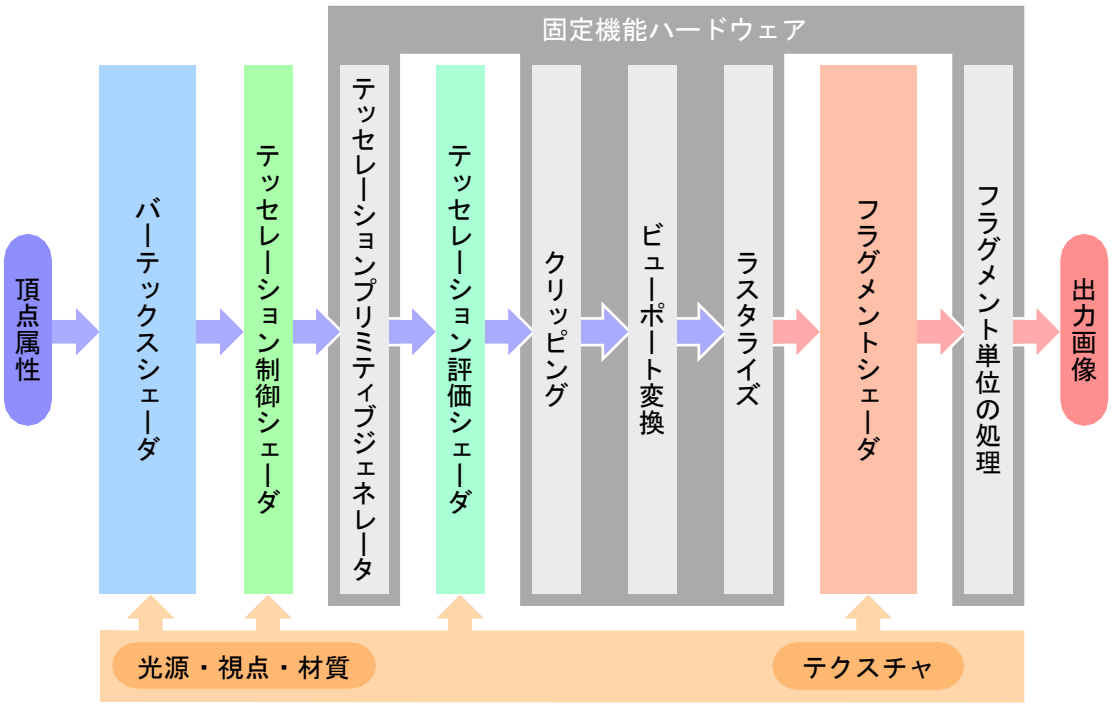

図 69 ジオメトリシェーダの細分化

現在は、これらの図形表示を目的としてレンダリングパイプラインに組み込まれたシェーダのほかに、計算処理を目的としてレンダリングパイプラインと独立して動作させることが可能な、**コンピュートシェーダ**も追加されています。

4.1.6　ラスタライザの役割

ラスタライズの前まではパイプラインにはジオメトリデータが流れており、ラスタライズ以降のパイプラインにはフラグメントデータが流れています。ラスタライザはジオメトリデータをフラグメントデータに変換する固定機能ハードウェアです。

ラスタライザは前段のバーテックスシェーダあるいはジオメトリシェーダ / テッセレーション評価シェーダからジオメトリデータを受け取り、基本図形の塗り潰し処理を行って、フラグメントの選択を行います。そして、選択した個々のフラグメントにフラグメントシェーダを割り当て、そのフラグメントにおける頂点属性 (頂点位置、頂点色、頂点の法線ベクトル、テクスチャ

座標など) の補間値を準備して、フラグメントシェーダを起動します (図 70)。

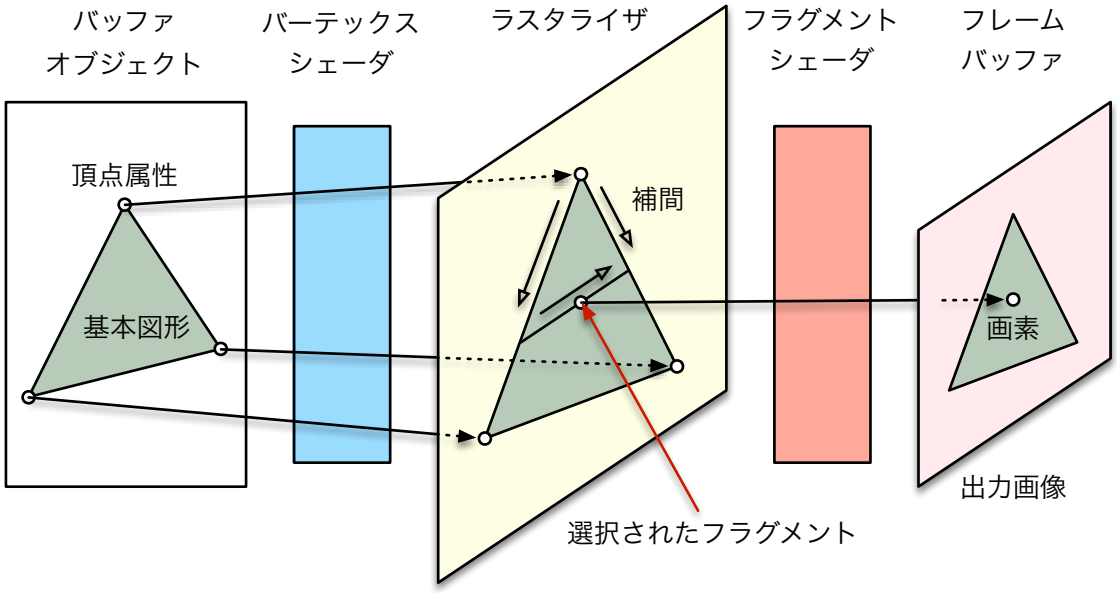

図 70 ラスタライザの役割

なお、この図の**バッファオブジェクト**は、ジオメトリデータを GPU 上に保持するメカニズムです。これについては後で解説します。

4.2 アプリケーションとのインタフェース

アプリケーション、すなわち CPU 側の処理では、あらかじめジオメトリ処理とフラグメント処理の内容をシェーダプログラムとして記述し、それぞれバーテックスシェーダとフラグメントシェーダに転送します。シーンの描き方はアプリケーションでも制御しますが、最近では多くの部分をシェーダで記述して、処理をできる限り GPU に移譲するようにします。

同様に、描画するシーンのデータもあらかじめ GPU に送ります。送るデータは頂点の位置のほか、法線ベクトル、テクスチャ座標などの**頂点属性** (attribute) です。これを GPU 側に確保した**バッファオブジェクト**と呼ばれるメモリに転送します。頂点属性を保持するバッファオブジェクトを、特に**頂点バッファオブジェクト** (Vertex Buffer Object, VBO) といいます。このほかに、描画に用いる頂点属性のインデックス (番号) を頂点属性とは別に保持する場合もあります。これは**インデックスバッファオブジェクト** (Index Buffer Object) と呼ばれます。

バッファオブジェクトに格納された頂点属性を用いて図形の描画を行うには、頂点属性の種類ごとに用意したバッファオブジェクトと、それらからバーテックスシェーダにデータを入力する方法を、**頂点配列オブジェクト** (Vertex Array Object) に登録する必要があります。

また前述のように、映像を生成する際には頂点属性の他に、様々な情報 (パラメータ) が必要になります。視点の位置は座標値ですが、これをもとに決定されるモデル変換やビュー変換、投影変換は行列で表されます。これらもあらかじめ GPU に送っておく必要があります。これらは描画命令の実行中には変更されない変数で、シェーダプログラムからは **uniform 変数**として参照されます。**テクスチャ**も同様に、あらかじめ GPU 側のテクスチャメモリに送ります。

前述のソフトウェアのパイプラインを実現するために、GPU のハードウェアのパイプラインは図 71 のような構成になっています。これは OpenGL 3.2 のハードウェアのパイプラインの概念を非常に簡略化したものです。OpenGL 4.x ではさらに要素が増えています。

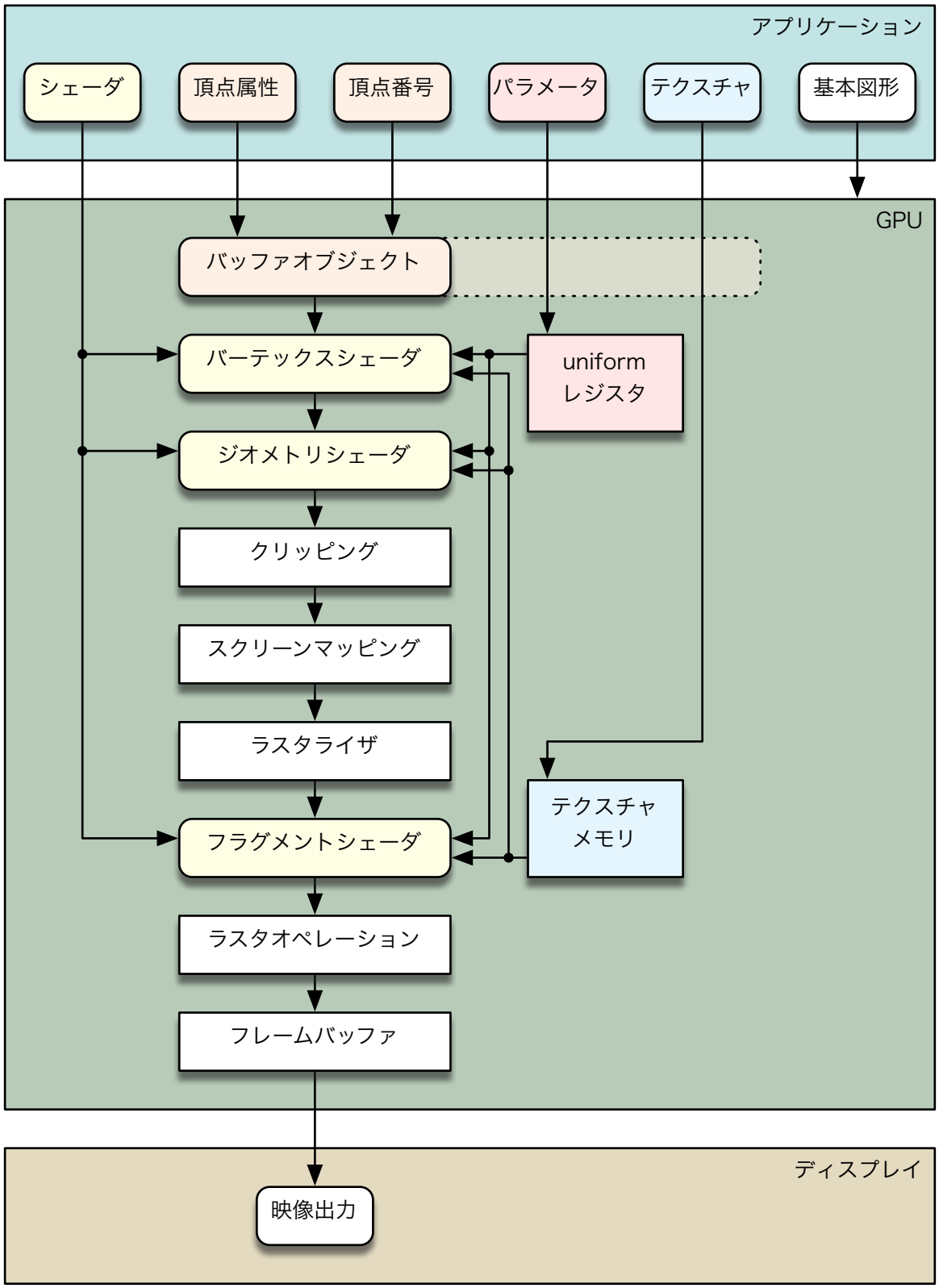

図 71 最近のグラフィックスハードウェアのパイプラインの概念図

シーンの描画は、テクスチャマッピングに使用する**テクスチャユニット**を指定し、それに使用するテクスチャオブジェクトを割り当てたあと、描画するシーンの情報を登録した頂点配列オブジェクトを指定し、描画に使用する線分や三角形などの**基本図形** (primitive) と、描画に用いる頂点属性のバッファオブジェクト上の範囲を指定して、描画命令を実行します。

第5章 シェーダプログラム

5.1 シェーディング言語

5.1.1 シェーディング言語の種類

シェーダのプログラミングに使われるプログラミング言語は**シェーディング言語**と呼ばれ、グラフィックスハードウェア用には現在の次のものがあります。いずれも C 言語に似ています。

- HLSL (High Level Shading Language)
  - DirectX に対応している
  - Microsoft が NVIDIA の協力を得て開発
- Cg (C for Graphics)
  - DirectX と OpenGL に対応している
  - NVIDIA により開発
- GLSL (OpenGL Shading Language, GLslang)
  - OpenGL 固有のシェーディング言語
  - OpenGL ARB (現在は Khronos Group) により開発

この講義では GLSL (OpenGL Shading Language) を使用します。

5.1.2 シェーダプログラムの種類

これまで述べた通り、グラフィックスパイプライン上には、それぞれ異なる役割を果たすシェーダが存在します。それらは OpenGL のバージョンによって異なります。

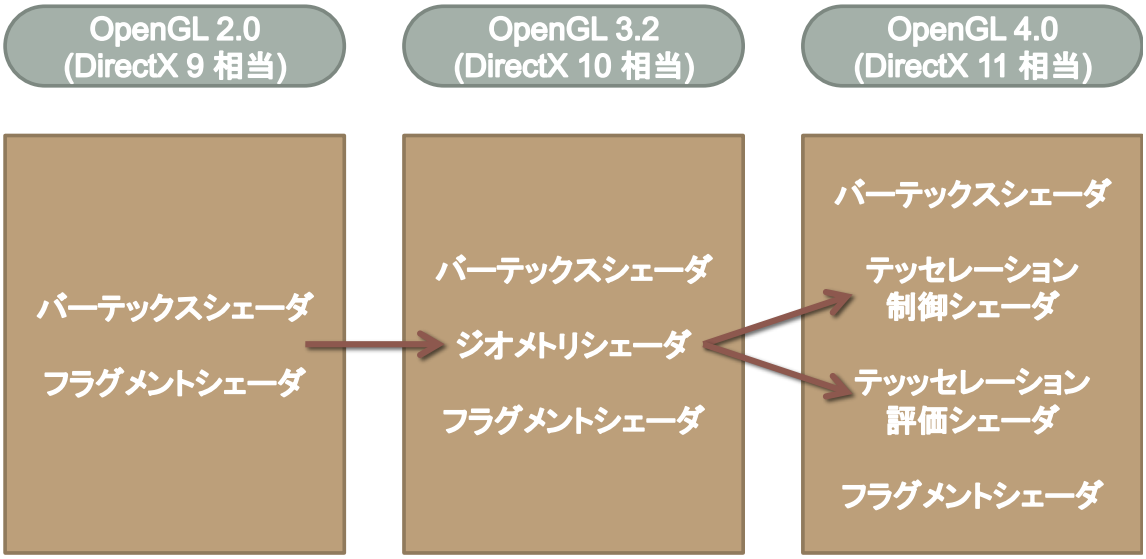

図 72 OpenGL のバージョンとシェーダプログラムの種類

OpenGL 2.0 は伝統的な固定機能のレンダリングパイプラインとの互換性を考慮しているため、そこからの移行には取り組みやすいものの、将来使われなくなる機能も含んでいます。この講義の時点における OpenGL の最新版は 2013 年の 7 月にリリースされた 4.4 ですが、現在使われているすべての GPU がこれをサポートしているわけではありません。

そこで、この講義では OpenGL 3.2 の Core Profile (将来の互換性を確保した仕様) を対象にします (というのは言い訳で実はキャッチアップできてないだけ)。これは Mac OS X 10.7 (Lion) 以降も対応しています。表 3 に OpenGL のバージョンと、それに対応した GLSL のバージョンを示します。括弧内の数字は GLSL のソースプログラムにおけるバージョンの表記です。

表 3 OpenGL のバージョンと GLSL のバージョン

OpenGL のバージョン	GLSL のバージョン
1.5	1.00 (100)
2.0	1.10 (110)
2.1	1.20 (120)
3.0	1.30 (130)
3.1	1.40 (140)
3.2	1.50 (150)
3.3	3.30 (330)
4.0	4.00 (400)
4.1	4.10 (410)
4.2	4.20 (420)
4.3	4.30 (430)
4.4	4.40 (440)

5.2 図形の描画手順

図 73 に OpenGL 3.2 の Core Profile における図形の描画手順を示します。CPU 側で動作するアプリケーションプログラムでシェーダプログラムを準備して、それを GPU に転送しておきます。また、描画する図形の頂点属性は、あらかじめ GPU 側のメモリ上に確保しておいたバッファオブジェクトに転送ししておきます。そして、図形の描画に使うシェーダプログラムを指定し、頂点属性を格納したバッファオブジェクトと描画する基本図形を指定して、描画を実行します。

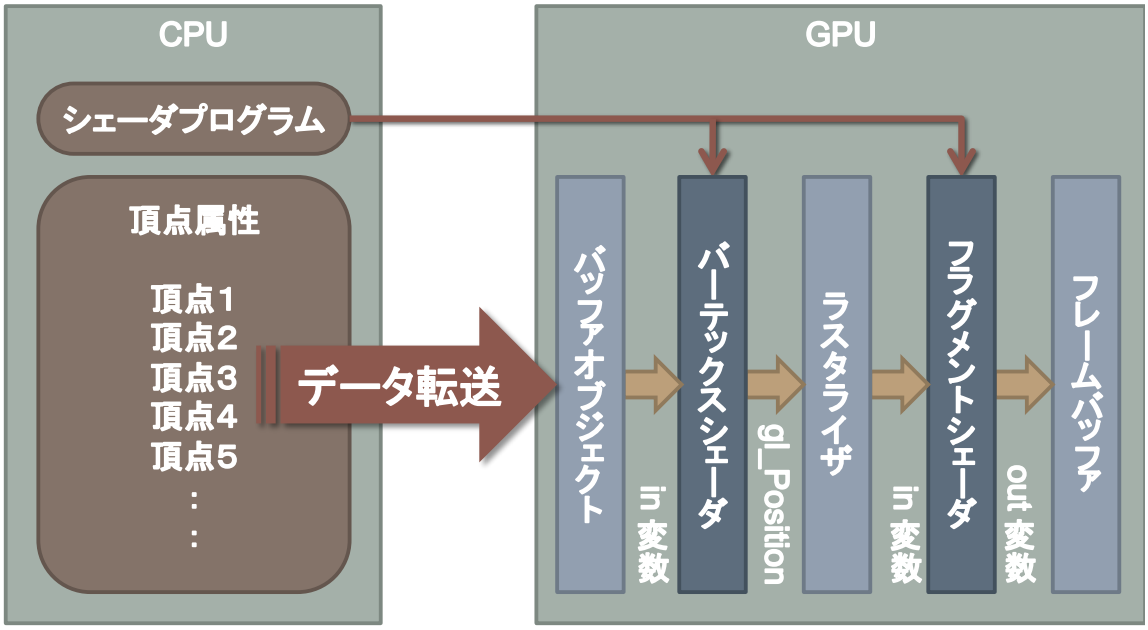

図 73 図形の描画手順

5.3 シェーダプログラムの作成

5.3.1 シェーダプログラムの作成手順

GLSL のシェーダプログラムを利用する手順は、次のようになります。

1. glCreateProgram() 関数によってプログラムオブジェクトを作成します。
2. glCreateShader() 関数によってバーテックスシェーダとフラグメントシェーダのシェーダオブジェクトを作成します。
3. glShaderSource() 関数によって、作成したそれぞれのシェーダオブジェクトに対してソースプログラムを読み込みます。
4. glCompileShader() 関数によって読み込んだソースプログラムをコンパイルします。
5. glAttachShader() 関数によってプログラムオブジェクトにシェーダオブジェクトを組み込みます。
6. glLinkProgram() 関数によってプログラムオブジェクトをリンクします。

以上の処理によりシェーダのプログラムオブジェクトが作成されますから、図形を描画する前に glUseProgram() 関数を実行して、図形の描画にこのシェーダプログラムを使うようにします。

5.3.2 シェーダのソースプログラム

● バーテックスシェーダのソースプログラム

バーテックスシェーダのソースプログラムとして、ここでは次のものを用います。このプログラムは、入力された図形データのひとつひとつの頂点に対して実行されます。

```
#version 150 core
in vec4 pv;
void main()
{
  gl_Position = pv;
}
```

この 1 行目の #version の行は、使用する GLSL のバージョンを指定します。150 は OpenGL 3.2 の GLSL のバージョンである 1.50 を表し、core は Core Profile であることを示します。

2 行目はこのシェーダプログラムの入力変数 pv の宣言です。in はこの変数が前のステージからデータを受け取ることを示します。バーテックスシェーダの in 変数には、CPU から送られた図形データの一つの頂点の頂点属性が設定されます。これは特に attribute 変数といいます。

vec4 はこの変数のデータ型が 32bit 浮動小数点型 (float 型) の 4 要素のデータからなるベクトル型であることを示します。ベクトル型には、このほかに 2 要素の vec2、3 要素の vec3 があります。ベクトルの各要素は、C 言語の構造体と同様にドット演算子 “.” を用いて変数名の後に { .x, .y, .z, .w }、{ .s, .t, .p, q } あるいは { .r, .g, .b, .a } を付けることにより、参照や代入を行うことができます (表 4)。

表 4 GLSL のデータ型

変数のデータ型	内容	ベクトル型の要素
float	単精度浮動小数点 1 個	.x .y .z .w
vec2	単精度浮動小数点 2 個	.r .g .b .a
vec3	単精度浮動小数点 3 個	.s .t .p .q
vec4	単精度浮動小数点 4 個	

凡例

vec4 pv;	vec4 型の変数宣言
pv.xy	pv の第1,2要素, vec2 型
pv.rgb	pv の第1,2,3要素, vec3 型
pv.q	pv の第4要素, float 型
pv.yx	pv の第1,2要素の順序を入れ替えたもの, vec2 型
pv.brg	pv の第1,2,3要素を3,1,2の順に並べ替え, vec3 型

シェーダプログラムは C や C++ 言語同様 main() 関数から実行を開始します。ただし、シェーダプログラムの main() 関数は、引数を持たず、値を戻すこともありません。したがって、関数の定義は void main() になります。

CPU から送られた頂点属性は、一旦 GPU が管理する頂点バッファと呼ばれるメモリに格納さ

れます。その後、CPU から描画命令が与えられると、GPU は頂点バッファから頂点ごとに頂点属性を取り出し、それを attribute 変数に格納して、バーテックスシェーダを起動します。

gl_Position は GLSL の組み込み変数で、この変数に代入した値がパイプラインの次のステージ (ラスタライザなど) に送られます。バーテックスシェーダは必ずこの変数に値を代入しなければなりません。このプログラムでは attribute 変数 pv をそのまま gl_Position に代入していますから、図形の描画に指定された頂点属性をそのまま次のステージのラスタライザに送ります。

● **フラグメントシェーダのソースプログラム**

フラグメントシェーダのソースプログラムには、次のものを用います。このプログラムは出力する画像のひとつひとつの画素に対して実行されます。

```
#version 150 core
out vec4 fc;
void main()
{
  fc = vec4(1.0, 0.0, 0.0, 1.0);
}
```

この 2 行目は、フラグメントの色の出力先として使う変数 fc の変数宣言です。out 宣言は、この変数が次のステージにデータを送る出力変数であることを示します。フラグメントシェーダでは、出力変数に代入した値がフレームバッファのカラーバッファに格納されます。

vec4() は四つの数値を vec4 型に変換 (キャスト) します。出力変数 fc には色を表す R (赤)、G (緑)、B (青)、および A (不透明度) の 4 要素のベクトルを代入します。もしフレームバッファに値を格納しない (画素を描かない) なら、フラグメントシェーダで discard 命令を実行します。

5.3.3 プログラムオブジェクトの作成

シェーダのソースプログラムを GPU で実行するには、それらをコンパイル・リンクして、シェーダのプログラムオブジェクトを作成する必要があります。まず、glCreateProgram() 関数で空のプログラムオブジェクト (program) を作成します。

```
const GLuint program(glCreateProgram());
```

GLuint glCreateProgram(void)

プログラムオブジェクトを作成します。戻り値は作成されたプログラムオブジェクト名 (番号) で、作成できれば 0 でない正の整数、作成できなければ 0 を返します。

5.3.4 シェーダオブジェクトの作成

次に、シェーダのソースプログラムをコンパイルして、シェーダオブジェクトを作成します。ここではバーテックスシェーダを例に説明します。フラグメントシェーダのシェーダオブジェクトも、同じ手順で作成します。

まずソースプログラムの各行を文字列にして、次のように配列に格納します。この 1 行目の #version の行のように、行頭が '#' の行の行末には '¥n' が必要です。

```
static const GLchar *vsrc[] =
{
  "#version 150 core¥n",
```

```
  "in vec4 pv;",
  "void main()",
  "{",
  "  gl_Position = pv;",
  "}",
};
```

そして以下の手順により、ソースプログラムからシェーダオブジェクト (vobj) を作成します。sizeof vsrc / sizeof vsrc[0] により配列変数 vsrc の要素数 (行数) を求めています。なお、これはバーテックスシェーダの場合です。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場合は、GL_VERTEX_SHADER を GL_FRAGMENT_SHADER に書き換えます。

```
const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
glShaderSource(vobj, sizeof vsrc / sizeof vsrc[0], vsrc, NULL);
glCompileShader(vobj);
```

シェーダのソースプログラムが単一の文字列なら、行数を 1 として、その文字列のポインタを格納した変数のポインタを用います。ただし、この書き方の場合は各行の行末に '¥n' を入れないと、エラーメッセージの行番号がどれも 2 になってしまいます。

```
static const GLchar *vsrc =
  "#version 150 core¥n"
  "in vec4 pv;¥n"
  "void main()¥n"
  "{¥n"
  "  gl_Position = pv;¥n"
  "}¥n";
```

この場合は次のようにしてソースプログラムの文字列を読み込みます。

```
const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
glShaderSource(vsrc, 1, &vsrc, NULL);
glCompileShader(vobj);
```

GLuint glCreateShader(GLenum shaderType)

シェーダオブジェクトを作成します。戻り値は作成されたシェーダオブジェクト名 (番号) で、作成できれば 0 でない正の整数、作成できなければ 0 を返します。

shaderType

作成するシェーダの種類を指定します。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場合は GL_VERTEX_SHADER、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場合は GL_FRAGMENT_SHADER を指定します。

void glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)

シェーダのソースプログラムを読み込みます。

shader

glCreateShader() 関数の戻り値として得たシェーダオブジェクト名。

count

引数 string に指定した配列の要素数。

string

シェーダのソースプログラムの文字列の配列。行頭が '#' の行の行末には '¥n' が必要で

す。

length

引数 length が NULL (0) でなければ、length の各要素には string の対応する要素の文字列の長さ (文字数) を格納します。また、この length の要素に負の値を格納したときは、string の対応する要素の文字列の終端がヌル文字 ('¥0') になっている必要があります。

void glCompileShader(GLuint shader)

シェーダオブジェクトに読み込まれたソースファイルをコンパイルします。

shader

コンパイルするシェーダオブジェクト名。

シェーダのソースプログラムのコンパイルに成功したなら、それをプログラムオブジェクトに組み込みます。プログラムオブジェクトに組み込んだシェーダオブジェクトは、他に使うあてがなければもう不要なので、ここで削除マークをつけておきます。

```
glAttachShader(program, vobj);
glDeleteShader(vobj);
```

void glAttachShader(GLuint program, GLuint shader)

プログラムオブジェクトにシェーダオブジェクトを組み込みます。

program

シェーダオブジェクトを組み込むプログラムオブジェクト名。

shader

組み込むシェーダオブジェクト名。

void glDetachShader(GLuint program, GLuint shader)

プログラムオブジェクトからシェーダオブジェクトを取り外します。

program

シェーダオブジェクトを取り外すプログラムオブジェクト名。

shader

取り外すシェーダオブジェクト名。

void glDeleteShader(GLuint shader)

シェーダオブジェクトに削除マークを付けます。

shader

削除マークを付けるシェーダオブジェクト名。

void glDeleteProgram(GLuint program)

プログラムオブジェクトに削除マークを付けます。

program

削除マークを付けるプログラムオブジェクト名。

glDeleteShader() 関数で削除マークを付けたシェーダオブジェクトは、そのシェーダオブジェクトが全てのプログラムオブジェクトから glDetachShader() 関数によって取り外されたときに削

除されます。またプログラムオブジェクトが削除されるときには、それに組み込まれているシェーダオブジェクトは自動的に取り外されます。したがって、削除マークが付けられたシェーダオブジェクトは、それを組み込んだプログラムオブジェクトが全て削除された時点で削除されます。

プログラムオブジェクトに削除マークを付けるには、glDeleteProgram() 関数を用います。削除マークを付けたプログラムオブジェクトは、どのレンダリングコンテキストでも使用されなくなったときに削除されます。

5.3.5 プログラムオブジェクトのリンク

プログラムオブジェクトに必要なシェーダオブジェクトを組み込んだら、プログラムオブジェクトのリンクを行います。その前に、図形の頂点のデータ (頂点属性, attribute) をバーテックスシェーダから参照するために用いる変数 (attribute 変数、ここでは変数名を pv) の場所 (index) と、フラグメントシェーダから出力するデータを格納する変数 (ここでは変数名を fc とします) の出力先 (colorNumber) を指定しておきます。これらはいずれも 0 以上の整数です。

CPU 側のプログラムと GPU 側のシェーダプログラムの間のデータの受け渡しは、このような番号を介して行います。これは GPU のハードウェアのレジスタ番号に相当します。

```
glBindAttribLocation(program, 0, "pv");
glBindFragDataLocation(program, 0, "fc");
glLinkProgram(program);
```

void glBindAttribLocation(GLuint program, GLuint index, const GLchar *name)

バーテックスシェーダのソースプログラム中の attribute 変数の場所を指定します。attribute 変数は、バーテックスシェーダのソースプログラム内では in 変数として宣言されます。

program

attribute 変数の場所を調べるプログラムオブジェクト名。

index

引数 name に指定した in 変数の場所 (番号) を、0 以上の整数で指定します。デフォルトでは 0 が設定されています。

name

バーテックスシェーダのソースプログラム中の in 変数の変数名の文字列。

void glBindFragDataLocation(GLuint program, GLuint colorNumber, const char *name)

フラグメントシェーダのソースプログラム中の out 変数にカラーバッファを割り当てます。

program

フラグメントの出力変数 (out 変数) の場所を調べるプログラムオブジェクト名。

index

引数 name に指定した out 変数の場所 (番号) を、0 以上の整数で指定します。デフォルトでは 0 が設定されています。0 は標準のフレームバッファのカラーバッファを示します。

name

フラグメントシェーダのソースプログラム中の out 変数の変数名の文字列。

void glLinkProgram(GLuint program)

program に指定したプログラムオブジェクトをリンクします。これが成功すれば、シェーダプログラムが完成します。

program

リンクするプログラムオブジェクト名。

5.3.6 プログラムオブジェクトを作成する手続き

前節までで説明した手順を次のように関数にまとめます。ソースプログラムの文字列が NULL の時は、シェーダオブジェクトを作成しないようにします。

```
// プログラムオブジェクトを作成する
//   vsrc: バーテックスシェーダのソースプログラムの文字列
//   pv: バーテックスシェーダのソースプログラム中の in 変数名の文字列
//   fsrc: フラグメントシェーダのソースプログラムの文字列
//   fc: フラグメントシェーダのソースプログラム中の out 変数名の文字列
static GLuint createProgram(const char *vsrc, const char *pv,
  const char *fsrc, const char *fc)
{
  // 空のプログラムオブジェクトを作成する
  const GLuint program(glCreateProgram());

  if (vsrc != NULL)
  {
    // 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
    const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
    glShaderSource(vobj, 1, &vsrc, NULL);
    glCompileShader(vobj);

    // バーテックスシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    glAttachShader(program, vobj);
    glDeleteShader(vobj);
  }

  if (fsrc != NULL)
  {
    // 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
    const GLuint fobj(glCreateShader(GL_FRAGMENT_SHADER));
    glShaderSource(fobj, 1, &fsrc, NULL);
    glCompileShader(fobj);

    // フラグメントシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    glAttachShader(program, fobj);
    glDeleteShader(fobj);
  }

  // プログラムオブジェクトをリンクする
  glBindAttribLocation(program, 0, pv);
  glBindFragDataLocation(program, 0, fc);
  glLinkProgram(program);

  // 作成したプログラムオブジェクトを返す
  return program;
}
```

5.3.7　シェーダプログラムの使用

図形の描画を行う前に、glUseProgram() 関数で使用するプログラムオブジェクトを指定します。

```
glUseProgram(program);
```

void glUseProgram(GLuint program)

図形の描画に使用するプログラムオブジェクトを指定します。

program

図形の描画に使用するプログラムオブジェクト名。0 を指定すると、どのプログラムオブジェクトも使用されなくなります。

● ソースプログラムの変更点

main() 関数の定義のより前で前述のプログラムオブジェクトを作成する関数 createProgram() 関数を定義し、GLEW の初期化を行った後にこれを実行します。そして図形の描画を行う前に、この関数の戻り値として得たプログラムオブジェクト名を glUseProgram() 関数の引数に指定して、このシェーダプログラムの使用を開始します。

```
#include <cstdlib>
#include <iostream>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

// プログラム終了時の処理
static void cleanup()
{
  // GLFW の終了処理
  glfwTerminate();
}

// プログラムオブジェクトを作成する
//   vsrc: バーテックスシェーダのソースプログラムの文字列
//   pv: バーテックスシェーダのソースプログラム中の in 変数名の文字列
//   fsrc: フラグメントシェーダのソースプログラムの文字列
//   fc: フラグメントシェーダのソースプログラム中の out 変数名の文字列
static GLuint createProgram(const char *vsrc, const char *pv,
  const char *fsrc, const char *fc)
{
  《省略》
}

《省略》

int main()
{
  《省略》

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // バーテックスシェーダのソースプログラム
  static const GLchar vsrc[] =
    "#version 150 core¥n"
    "in vec4 pv;¥n"
```

```
    "void main()¥n"
    "{¥n"
    "  gl_Position = pv;¥n"
    "}¥n";

  // フラグメントシェーダのソースプログラム
  static const GLchar fsrc[] =
    "#version 150 core¥n"
    "out vec4 fc;¥n"
    "void main()¥n"
    "{¥n"
    "  fc = vec4(1.0, 0.0, 0.0, 1.0);¥n"
    "}¥n";

  // プログラムオブジェクトを作成する
  const GLuint program(createProgram(vsrc, "pv", fsrc, "fc"));

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);

    //
    // ここで描画処理を行う
    //

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

● 補足：glUseProgram() 関数を実行する位置

glUseProgram(program); は、使用するシェーダプログラムがひとつしかなければ while ループの外 (前) に置いて構いません。シェーダプログラムを図形 (あるいは描画単位) ごとに切り替えて使う場合は、ループの中に置きます。また、シェーダプログラムが不要になれば glUseProgram(0); を実行しますが、シェーダプログラムは使いっぱなしでも構いません。

5.4 エラーメッセージの表示

ここまではシェーダのコンパイルやリンクの時にエラーチェックを行っていませんでした。GLSL といえどもプログラミング言語なので、書き間違えればエラーが発生します。その時、エラーメッセージがわからなければ、間違いを見つけることが難しくなります。そこで、コンパイルの際にエラーの発生をチェックし、エラーメッセージを表示する手続きを追加します。この関数の戻り値は、エラーが発生しなければ GL_TRUE、発生すれば GL_FALSE にします。

```
// シェーダオブジェクトのコンパイル結果を表示する
//   shader: シェーダオブジェクト名
```

```
//   str: コンパイルエラーが発生した場所を示す文字列
static GLboolean printShaderInfoLog(GLuint shader, const char *str)
{
  // コンパイル結果を取得する
  GLint status;
  glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
  if (status == GL_FALSE) std::cerr << "Compile Error in " << str << std::endl;

  // シェーダのコンパイル時のログの長さを取得する
  GLsizei bufSize;
  glGetShaderiv(shader, GL_INFO_LOG_LENGTH , &bufSize);

  if (bufSize > 1)
  {
    // シェーダのコンパイル時のログの内容を取得する
    std::vector<GLchar> infoLog(bufSize);
    GLsizei length;
    glGetShaderInfoLog(shader, bufSize, &length, &infoLog[0]);
    std::cerr << &infoLog[0] << std::endl;
  }

  return static_cast<GLboolean>(status);
}
```

void glGetShaderiv(GLuint shader, GLenum pname, GLint *params)

シェーダオブジェクトの情報を取り出します。

shader

情報を取り出すシェーダオブジェクト。

pname

シェーダオブジェクトから取り出す情報の種類。以下のものが指定できます。

GL_SHADER_TYPE

shader に指定したシェーダオブジェクトのシェーダの種類 (GL_VERTEX_SHADER, GL_FRAGMENT_SHADER) を調べて *params に格納します。

GL_DELETE_STATUS

shader に指定したシェーダオブジェクトに glDeleteShader() 関数によって削除マークが付けられているかどうかを調べて、削除マークがついていれば GL_TRUE、ついていなければ GL_FALSE を *params に格納します。

GL_COMPILE_STATUS

shader に指定したシェーダオブジェクトのコンパイルが成功したかどうかを調べて、成功していれば GL_TRUE、失敗していれば GL_FALSE を *params に格納します。

GL_INFO_LOG_LENGTH

shader に指定したシェーダオブジェクトのコンパイル時に生成されたログの長さを調べて *params に格納します。ログがなければ 0 を格納します。

GL_SHADER_SOURCE_LENGTH

shader に指定したシェーダオブジェクトのソースプログラムの長さを調べて *params に格納します。ソースプログラムがなければ 0 を格納します。

params

取り出した情報の格納先。

void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length, GLchar *infoLog)

シェーダオブジェクトのコンパイル時のログを取り出します。

shader

ログを取り出すシェーダオブジェクト。

maxLength

取り出すログの最大の長さ。引数 infoLog に指定するログの格納先の大きさは、これより小さくなければなりません。

length

取り出したログの実際の長さの格納先。

infoLog

取り出したログの格納先。

同様に、リンクの際にエラーの発生をチェックし、エラーメッセージを表示する手続きを追加します。戻り値は、エラーが発生しなければ GL_TRUE、発生すれば GL_FALSE にします。

```
// プログラムオブジェクトのリンク結果を表示する
//   program: プログラムオブジェクト名
static GLboolean printProgramInfoLog(GLuint program)
{
  // リンク結果を取得する
  GLint status;
  glGetProgramiv(program, GL_LINK_STATUS, &status);
  if (status == GL_FALSE) std::cerr << "Link Error." << std::endl;

  // シェーダのリンク時のログの長さを取得する
  GLsizei bufSize;
  glGetProgramiv(program, GL_INFO_LOG_LENGTH, &bufSize);

  if (bufSize > 1)
  {
    // シェーダのリンク時のログの内容を取得する
    std::vector<GLchar> infoLog(bufSize);
    GLsizei length;
    glGetProgramInfoLog(program, bufSize, &length, &infoLog[0]);
    std::cerr << &infoLog[0] << std::endl;
  }

  return static_cast<GLboolean>(status);
}
```

void glGetProgramiv(GLuint program, GLenum pname, GLint *params)

プログラムオブジェクトの情報を取り出します。

program

情報を取り出すプログラムオブジェクト。

pname

プログラムオブジェクトから取り出す情報の種類。以下のものが指定できます。これら以外にもありますが、ここでは割愛します。

GL_DELETE_STATUS

program に指定したプログラムオブジェクトに glDeleteProgram() 関数によって削除マークが付けられているかどうかを調べて、削除マークがついていれば GL_TRUE、ついていなければ GL_FALSE を *params に格納します。

GL_LINK_STATUS

program に指定したプログラムオブジェクトのリンクが成功したかどうかを調べて、成功していれば GL_TRUE、失敗していれば GL_FALSE を *params に格納します。

GL_VALIDATE_STATUS

program に指定したプログラムオブジェクトに対する glValidateProgram() 関数による検証結果を、*params に格納します。*params にはプログラムオブジェクトが現在の OpenGL の状態で実行可能なら GL_TRUE、実行できなければ GL_FALSE が格納されます。

GL_INFO_LOG_LENGTH

program に指定したプログラムオブジェクトのリンク時に生成されたログの長さを調べて *params に格納します。ログがなければ 0 を格納します。

GL_ATTACHED_SHADERS

program に指定したプログラムオブジェクトに組み込まれているシェーダオブジェクトの数を調べて *params に格納します。

params

取り出した情報の格納先。

void glValidateProgram(GLuint program)

プログラムオブジェクトが現在の OpenGL の状態で実行可能かどうかを検証します。結果は glGetProgramiv() 関数の引数 pname に GL_VALIDATE_STATUS を指定して取り出します。

program

検証するプログラムオブジェクト。

void glGetProgramInfoLog(GLuint program, GLsizei maxLength, GLsizei *length, GLchar *infoLog)

シェーダオブジェクトのコンパイル時のログを取り出します。

program

ログを取り出すプログラムオブジェクト。

maxLength

取り出すログの最大の長さ。引数 infoLog に指定した格納先の大きさを超えてはなりません。

length

取り出したログの実際の長さ (infoLog に格納された長さ) の格納先。

infoLog

ログの格納先。格納先の大きさは引数 maxLength よりも大きくなければなりません。

● **ソースプログラムの変更点**

これらの関数を、プログラムオブジェクトを作成する関数 createProgram() 関数の定義より前

で定義し、シェーダオブジェクトのコンパイルの後やプログラムオブジェクトのコンパイルの後に実行します。なお、これらの関数では C++ の標準ライブラリに含まれる vector を使用しているので、ソースプログラムの先頭で vector を #include します。

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

// プログラム終了時の処理
static void cleanup()
{
  // GLFW の終了処理
  glfwTerminate();
}

// シェーダオブジェクトのコンパイル結果を表示する
//   shader: シェーダオブジェクト名
//   str: コンパイルエラーが発生した場所を示す文字列
static GLboolean printShaderInfoLog(GLuint shader, const char *str)
{
  《省略》
}

// プログラムオブジェクトのリンク結果を表示する
//   program: プログラムオブジェクト名
static GLboolean printProgramInfoLog(GLuint program)
{
  《省略》
}

// プログラムオブジェクトを作成する
//   vsrc: バーテックスシェーダのソースプログラムの文字列
//   pv: バーテックスシェーダのソースプログラム中の in 変数名の文字列
//   fsrc: フラグメントシェーダのソースプログラムの文字列
//   fc: フラグメントシェーダのソースプログラム中の out 変数名の文字列
static GLuint createProgram(const char *vsrc, const char *pv,
  const char *fsrc, const char *fc)
{
  // 空のプログラムオブジェクトを作成する
  const GLuint program(glCreateProgram());

  if (vsrc != NULL)
  {
    // 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
    const GLuint vobj(glCreateShader(GL_VERTEX_SHADER));
    glShaderSource(vobj, 1, &vsrc, NULL);
    glCompileShader(vobj);

    // バーテックスシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    if (printShaderInfoLog(vobj, "vertex shader"))
      glAttachShader(program, vobj);
    glDeleteShader(vobj);
  }

  if (fsrc != NULL)
```

```
  {
    // 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
    const GLuint fobj(glCreateShader(GL_FRAGMENT_SHADER));
    glShaderSource(fobj, 1, &fsrc, NULL);
    glCompileShader(fobj);

    // フラグメントシェーダのシェーダオブジェクトをプログラムオブジェクトに組み込む
    if (printShaderInfoLog(fobj, "fragment shader"))
      glAttachShader(program, fobj);
    glDeleteShader(fobj);
  }

  // プログラムオブジェクトをリンクする
  glBindAttribLocation(program, 0, pv);
  glBindFragDataLocation(program, 0, fc);
  glLinkProgram(program);

  // 作成したプログラムオブジェクトを返す
  if (printProgramInfoLog(program))
    return program;

  // プログラムオブジェクトが作成できなければ 0 を返す
  glDeleteProgram(program);
  return 0;
}

《省略》
```

5.5 シェーダのソースプログラムを別のファイルから読み込む

前のプログラムでは、GLSL で書いたシェーダのソースプログラムを文字列の形で C++ のソースプログラムに埋め込みました。この方法はシェーダのソースプログラムと C++ のソースプログラムをひとつのファイルにまとめることができるので便利と言えば便利ですが、シェーダのソースプログラムを文字列で表さなければならないのは、やっぱり面倒です。

そこで、シェーダのソースプログラムを別のファイルにして、C++ のプログラムから実行時に読み込むようにします。こうすると、シェーダのソースプログラムに変更を加えたときに C++ のプログラムをコンパイルし直さずにすみます。以下の関数 readShaderSource() を追加します。

```
// シェーダのソースファイルを読み込んだメモリを返す
//   name: シェーダのソースファイル名
static GLchar *readShaderSource(const char *name)
{
  // ファイル名が NULL なら NULL を返す
  if (name == NULL) return NULL;

  // ソースファイルを開く
  std::ifstream file(name, std::ios::binary);
  if (file.fail())
  {
    // 開けなかった
    std::cerr << "Error: Can't open source file: " << name << std::endl;
    return NULL;
  }

  // ファイルの末尾に移動し現在位置（＝ファイルサイズ）を得る
  file.seekg(0L, std::ios::end);
```

```
  const GLsizei length(static_cast<GLsizei>(file.tellg()));

  // ファイルサイズ＋文字列の終端文字（'¥0'）分のメモリを確保
  GLchar *buffer(new(std::nothrow) GLchar[length + 1]);
  if (buffer == NULL)
  {
    // メモリが足らなかった
    std::cerr << "Error: Too large file: " << name << std::endl;
    file.close();
    return NULL;
  }

  // ファイルを先頭から読み込む
  file.seekg(0L, std::ios::beg);
  file.read(buffer, length);
  buffer[length] = '¥0';

  if (file.bad())
  {
    // うまく読み込めなかった
    std::cerr << "Error: Could not read souce file: " << name << std::endl;
    delete buffer;
    buffer = NULL;
  }
  file.close();

  //ソースプログラムを読み込んだメモリのポインタを返す
  return buffer;
}
```

さらに、この関数を使ってシェーダのソースファイルを読み込み、createProgram() 関数を使ってプログラムオブジェクトを作成する関数 loadProgram() を追加します。

```
// シェーダのソースファイルを読み込んでプログラムオブジェクトを作成する
//   vert: バーテックスシェーダのソースファイル名
//   pv: バーテックスシェーダのソースプログラム中の in 変数名の文字列
//   frag: フラグメントシェーダのソースファイル名
//   fc: フラグメントシェーダのソースプログラム中の out 変数名の文字列
static GLuint loadProgram(const char *vert, const char *pv,
  const char *frag, const char *fc)
{
  // シェーダのソースファイルを読み込む
  const GLchar *vsrc(readShaderSource(vert));
  const GLchar *fsrc(readShaderSource(frag));

  // プログラムオブジェクトを作成する
  const GLuint program(createProgram(vsrc, pv, fsrc, fc));

  // 読み込みに使ったメモリを解放する
  delete vsrc;
  delete fsrc;

  // 作成したプログラムオブジェクトを返す
  return program;
}
```

● ソースプログラムの変更点

関数 createProgram() の後に readShaderSource() と loadProgram() を追加します。また main() 関数では、バーテックスシェーダのソースプログラムの文字列 vsrc とフラグメントシェーダのソースプログラムの文字列 fsrc を削除し、createProgram() を loadProgram() に置き換えます。

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include <GL/glew.h>
#include <GLFW/glfw3.h>

《省略》

// プログラムオブジェクトを作成する
//   vsrc: バーテックスシェーダのソースプログラムの文字列
//   pv: バーテックスシェーダのソースプログラム中の in 変数名の文字列
//   fsrc: フラグメントシェーダのソースプログラムの文字列
//   fc: フラグメントシェーダのソースプログラム中の out 変数名の文字列
static GLuint createProgram(const char *vsrc, const char *pv,
  const char *fsrc, const char *fc)
{
  《省略》
}

// シェーダのソースファイルを読み込んだメモリを返す
//   name: シェーダのソースファイル名
static GLchar *readShaderSource(const char *name)
{
  《省略》
}

// シェーダのソースファイルを読み込んでプログラムオブジェクトを作成する
//   vert: バーテックスシェーダのソースファイル名
//   pv: バーテックスシェーダのソースプログラム中の in 変数名の文字列
//   frag: フラグメントシェーダのソースファイル名
//   fc: フラグメントシェーダのソースプログラム中の out 変数名の文字列
static GLuint loadProgram(const char *vert, const char *pv,
  const char *frag, const char *fc)
{
  《省略》
}

《省略》

int main()
{
  《省略》

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // バーテックスシェーダのソースプログラムの文字列 vsrc と
  // フラグメントシェーダのソースプログラムの文字列 fsrc の宣言は削除します。

  // プログラムオブジェクトを作成する
```

```
  const GLuint program(loadProgram("point.vert", "pv", "point.frag", "fc"));

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    《省略》
  }
}
```

シェーダのソースファイルは、C++ のソースファイルとは別に作成します。バーテックスシェーダのソースファイルのファイル名は point.vert、フラグメントシェーダのソースファイルのファイル名は point.frag にします (それぞれの拡張子は適当です)。これらを C++ のソースファイルと同じディレクトリ (フォルダ) に保存してください。

● シェーダのソースファイル

バーテックスシェーダのソースファイル point.vert

```
#version 150 core
in vec4 pv;
void main()
{
  gl_Position = pv;
}
```

フラグメントシェーダのソースファイル point.frag

```
#version 150 core
out vec4 fc;
void main()
{
  fc = vec4(1.0, 0.0, 0.0, 1.0);
}
```

● 補足：Xcode の場合

Mac OS X の統合開発環境である Xcode では、ビルドして作成した実行ファイルが C++ のソースファイルとは異なる場所に作成されます。また、これを Xcode 内で実行したときの作業ディレクトリも、そこに設定されます。このため、シェーダのソースファイルを C++ のソースファイルと同じディレクトリに置いた場合は、シェーダのソースファイルのファイル名にそのディレクトリのパスを含めないと読み込むことができません。

そこで、プログラムを Xcode 内で実行したときの作業ディレクトリが、シェーダや C++ のソースファイルを置いたディレクトリになるよう、Xcode の設定を変更します。Xcode の左上の “Set Active Scheme” から “Edit Scheme” を選んでください (図 74)。

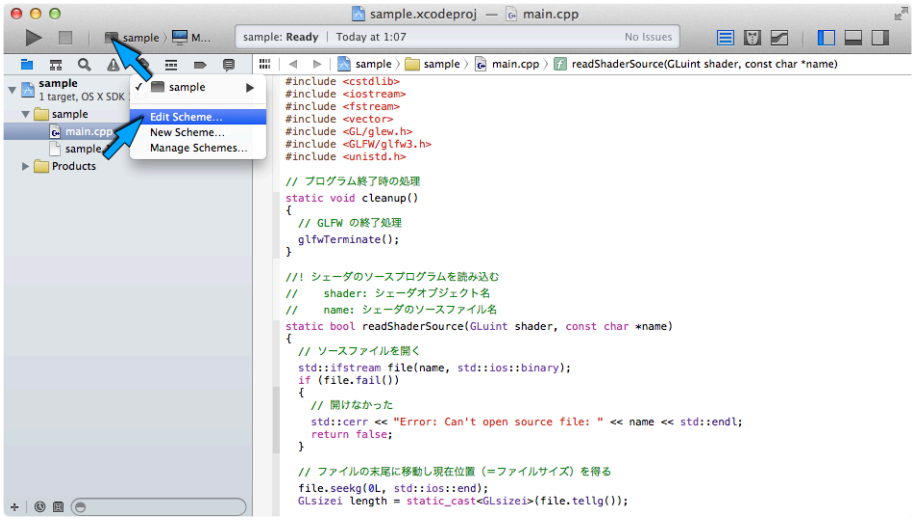

図 74 スキームの設定

左のペインの “Run プロジェクト名” を選び、上の “Options” を選択して “Working Directory” にチェック☑を入れてください。その後、その下の欄にシェーダや C++ のソースファイルを置いたディレクトリを設定してください。右側のフォルダのアイコンをクリックすれば、フォルダ選択のダイアログが現れます。最後に “OK” をクリックしてください (図 75)。

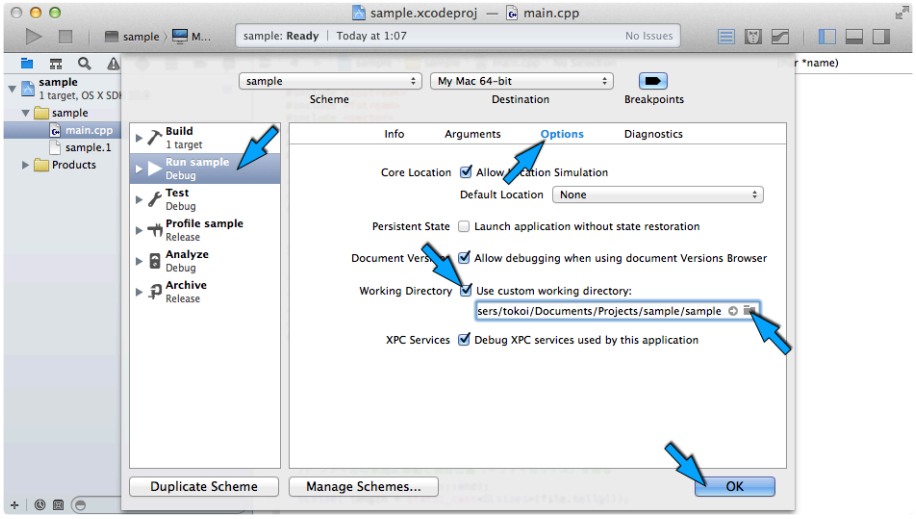

図 75 作業ディレクトリ (Working Directory) の設定

第6章 図形の描画

6.1 OpenGL のデータ型

グラフィックスハードウェア (GPU) が内部的に使用しているデータの書式がプラットホーム (CPU) と一致しているとは限らないので、OpenGL で取り扱うデータには独自のデータ型が割り当てられています (表 5)。

しかし、基本的には、そのデータ型はプラットホーム、あるいは使用するプログラミング言語のものに対応付けられています。たとえば GLubyte 型は C 言語あるいは C++ 言語の unsigned char 型に対応し、GLfloat 型は float に対応しています。また GLclampf は float 型のデータに対応しますが、OpenGL で使用されるときには、負の値は 0 に、1 を超える正の値は 1 として扱われます。

なお、GLhalf 型に対応するプラットホーム側のデータ型は存在しないため、プラットホーム側でこの値をそのまま計算などに用いることはできませんが、グラフィックスハードウェアから取り出してプラットホーム側で保持しておくことは可能です。

6.2 OpenGL の図形データ

OpenGL では、図形のデータは頂点の位置や色、法線ベクトルなどの情報で表されます。これを頂点属性 (attribute) と呼びます。図形を描画するには、この頂点属性を一旦 GPU 側のメモリに転送します。この GPU 側のメモリを頂点バッファといいます。また、この頂点バッファを管理する機構を、頂点バッファオブジェクト (Vertex Buffer Object, VBO) といいます。

図形の描画に用いる頂点属性には、位置や色のほか、法線ベクトルやテクスチャ座標など、必要に応じて様々なものが与えられます。したがって、実際に図形を描画するには、複数の頂点属性、すなわち頂点バッファオブジェクトを組み合わせる必要があります。そのため、この組み合わせを管理するために、頂点配列オブジェクト (Vertex Array Object, VAO) を用います。

OpenGL における図形の描画では、まず描画に用いる頂点属性を保持する頂点バッファオブジ

ェクトを作成し、頂点属性をそれに転送します。そして頂点配列オブジェクトにこの頂点バッファオブジェクトを組み込み、この頂点配列オブジェクトを指定して描画を実行します。

表 5 OpenGL のデータ型

OpenGL のデータ型	最小ビット数	説明
GLboolean	1	論理値
GLbyte	8	符号付き二進整数（二の補数表現）
GLubyte	8	符号なし二進整数
GLchar	8	文字列中の文字
GLshort	16	符号付き二進整数（二の補数表現）
GLushort	16	符号なし二進整数
GLint	32	符号付き二進整数（二の補数表現）
GLuint	32	符号なし二進整数
GLint64	64	符号付き二進整数（二の補数表現）
GLuint64	64	符号なし二進整数
GLsizei	32	非負の二進整数で表したサイズ
GLenum	32	二進整数で表した列挙子
GLintptr	※	符号付き二進整数（二の補数表現）
GLsizeiptr	※	非負の二進整数で表したサイズ
GLsync	※	同期オブジェクトのハンドル
GLbitfield	32	ビットフィールド
GLhalf	16	符号なしの値に符号化された半精度浮動小数点数
GLfloat	32	単線度浮動小数点数
GLclampf	32	[0, 1] にクランプされた単精度浮動小数点数
GLdouble	64	倍線度浮動小数点数
GLclampd	64	[0, 1] にクランプされた倍精度浮動小数点数

※ ポインタ（アドレス）を保持するのに必要なビット数

6.3 頂点配列オブジェクト

頂点配列オブジェクトは、glGenVertexArrays() 関数を使って作成します。作成した頂点配列オブジェクトを使用するときは、glBindVertexArray() 関数を実行します。

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

void glGenVertexArrays(GLsizei n, GLuint *arrays)

頂点配列オブジェクトを作成します。

n

作成する頂点配列オブジェクトの数。

arrays

作成した頂点配列オブジェクト名 (番号) を格納する配列。少なくとも引数 n 個以上の要素数を持つ必要があります。

void glBindVertexArray(GLuint array)

頂点配列オブジェクトを結合して使用可能にします。

array

結合する頂点配列オブジェクト名。0 なら現在結合されている頂点配列オブジェクトの結合を解除します。

6.4 頂点バッファオブジェクト

6.4.1 頂点バッファオブジェクトの作成

頂点配列オブジェクトに図形のデータを登録します。図形は以下の 4 点を頂点とする線図形にします。変数 vertices は頂点の数です。

```
// 図形データ
static const GLfloat position[][2] =
{
  { -0.5f, -0.5f },
  {  0.5f, -0.5f },
  {  0.5f,  0.5f },
  { -0.5f,  0.5f }
};

// 頂点の数
static const GLuint vertices(sizeof position / sizeof position[0]);
```

頂点バッファオブジェクトを作成し、このデータをそこに送ります。頂点バッファオブジェクトは GPU 側に確保したデータの保存領域を管理します。転送するデータのサイズは、GLfloat 型の二次元の位置データが vertices 個なので、sizeof (GLfloat) * 2 * vertices になります。

```
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER,
  sizeof (GLfloat) * 2 * vertices, position, GL_STATIC_DRAW);
```

void glGenBuffers(GLsizei n, GLuint *buffers)

頂点バッファオブジェクトを作成します。

n

作成する頂点バッファオブジェクトの数。

arrays

作成した頂点バッファオブジェクト名 (番号) を格納する配列。少なくとも引数 n 個以上の要素数を持つ必要があります。

void glBindBuffer(GLenum target, GLuint buffer)

頂点バッファオブジェクトを結合して使用可能にします。

target

頂点バッファオブジェクトの結合対象。GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER,

GL_PIXEL_UNPACK_BUFFER, GL_TEXTURE_BUFFE, GL_UNIFORM_BUFFE, GL_TRANSFORM_FEEDBACK_BUFFER が指定できます。頂点バッファの内容を描画に使う頂点属性として使う場合は、GL_ARRAY_BUFFER を指定します。

array

結合する頂点バッファオブジェクト名。0 なら現在結合されている頂点バッファオブジェクトの結合を解除します。

void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage)

頂点バッファオブジェクトのメモリを確保し、そこにデータ (頂点属性) を転送します。

target

データを転送する頂点バッファオブジェクトの結合対象。

size

GPU 側に確保する頂点バッファのサイズ。

data

頂点バッファに転送するデータが格納されている CPU 側のデータのポインタ。データが格納されているメモリ (配列変数等) のサイズは少なくとも引数 size バイトある必要がある。なお、これが 0 なら GPU 側に頂点バッファの確保だけを行い、データの転送は行わない。

usage

この頂点バッファがどのような使われ方をするのかを指定します。GL_STREAM_DRAW、GL_STREAM_READ、GL_STREAM_COPY、GL_STATIC_DRAW、GL_STATIC_READ、GL_STATIC_COPY 、 GL_DYNAMIC_DRAW 、 GL_DYNAMIC_READ 、 GL_DYNAMIC_COPY が指定できます。これは GPU の動作を最適化するためのヒントとして利用されます。

STREAM

データの保存領域には一度だけ書き込まれ、高々数回使用されます。

STATIC

データの保存領域には一度だけ書き込まれ、何度も使用されます。

DYNAMIC

データの保存領域には繰り返し書き込まれ、何度も使用されます。

DRAW

データの保存領域の内容はアプリケーションによって書き込まれ、描画のためのデータとして用いられます。

READ

データの保存領域の内容はアプリケーションからの問い合わせによって OpenGL (GPU) 側から読み出され、アプリケーション側に返されます。

COPY

データの保存領域の内容はアプリケーションからの指令によって OpenGL (GPU) 側から読み出され、描画のためのデータとして用いられます。

6.4.2 頂点バッファオブジェクトと attribute 変数

頂点バッファオブジェクトに格納されている頂点属性は、バーテックスシェーダの in 変数を使って取り出します。これを attribute 変数といいます。ある in 変数がどの頂点バッファオブジェクトからデータを取り出すのかを指定するには、glVertexAttribPointer() 関数を用います。そして glEnableVertexAttribArray() 関数によって、この in 変数を有効にします。

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

void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid *pointer);

図形の描画時に attribute 変数が受け取るデータの格納場所と書式を指定します。

index

シェーダプログラムのリンク時に glBindAttribLocation() 関数で指定した、データを受け取る attribute 変数の場所 (index)。このシェーダプログラムでは pv の index に 0 を指定していますから、ここでは 0 を指定しています。

size

attribute 変数が受け取る一個のデータのサイズを指定します。1、2、3、4 の値が指定できます。一個のデータが二次元 (x, y) なら 2、三次元 (x, y, z) なら 3 を指定します。このプログラムの形状データは二次元なので、ここでは 2 を指定しています。

type

attribute 変数が受け取る (pointer で示された先の) データ型を指定します。GL_BYTE、GL_UNSIGNED_BYTE 、 GL_SHORT 、 GL_UNSIGNED_SHORT 、 GL_INT 、GL_UNSIGNED_INT、GL_FLOAT、GL_DOUBLE が指定できます。このプログラムの形状データは GLfloat 型なので、ここでは GL_FLOAT を指定しています。

normalized

GL_TRUE なら type が固定小数点型 (GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT) のとき、その値をそのデータ型で表現可能な最大値で正規化します。GL_FALSE なら正規化しません。ここでは正規化を行わないので、GL_FALSE を指定しています。

stride

attribute 変数が受け取る (pointer で示された先の) データの配列の、要素間の間隔を指定します。0 ならデータは密に並んでいるものとして扱います。ここでは 0 を指定しています。

pointer

attribute 変数が受け取るデータが格納されている場所を指定します。バイト単位のオフセットをポインタにキャストして渡します。頂点バッファの先頭から取り出すなら 0 を指定します。なお、バイト単位のオフセットが int 型の変数 offset に格納されているとき、それをポインタに直すには、static_cast<char *>(0) + offset あるいは (char *)0 + offset とします。

void glEnableVertexAttribArray(GLuint index)

attribute 変数を有効にします。

index

有効にする attribute 変数の場所。

頂点配列オブジェクトの作成が完了したら、一旦頂点バッファオブジェクトの結合を解除し、頂点配列オブジェクトの結合を解除します。

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

なお、この例では glGenVertexArray() 関数で頂点配列オブジェクトを作成した後に頂点バッファオブジェクトを作成していますが、先に頂点バッファオブジェクトを作成しても構いません。頂点バッファオブジェクトは別のところで作っておいて、glVertexAttribPointer() 関数を実行する直前に glBindBuffer() 関数で結合する、という書き方もできます。

● **頂点配列オブジェクトを作成する手続き**

以上の手続きを関数にまとめると、次のようになります。

```
// 頂点配列オブジェクトの作成
//   vertices: 頂点の数
//   position: 頂点の位置を格納した配列
static GLuint createObject(GLuint vertices, const GLfloat (*position)[2])
{
  // 頂点配列オブジェクト
  GLuint vao;
  glGenVertexArrays(1, &vao);
  glBindVertexArray(vao);

  // 頂点バッファオブジェクト
  GLuint vbo;
  glGenBuffers(1, &vbo);
  glBindBuffer(GL_ARRAY_BUFFER, vbo);
  glBufferData(GL_ARRAY_BUFFER,
    sizeof (GLfloat) * 2 * vertices, 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 変数から参照できるようにする
  glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
  glEnableVertexAttribArray(0);

  // 頂点バッファオブジェクトと頂点配列オブジェクトの結合を解除する
  glBindBuffer(GL_ARRAY_BUFFER, 0);
  glBindVertexArray(0);

  return vao;
}
```

● **図形データの作成**

createObject() 関数を使って矩形のデータを作成する createRectangle() という関数を定義します。図形の描画には頂点配列オブジェクト名と頂点の数が必要ですから、この二つを保持する構造体 (“Object” とします) を定義しておきます。

```
// 形状データ
struct Object
{
  // 頂点配列オブジェクト名
  GLuint vao;

  // データの要素数
  GLsizei count;
};
```

関数 createObject() は、この型のデータを返すようにします。

```
// 矩形のデータを作成する
static Object createRectangle()
{
  // 頂点の位置データ
  static const GLfloat position[][2] =
  {
    { -0.5f, -0.5f },
    {  0.5f, -0.5f },
    {  0.5f,  0.5f },
    { -0.5f,  0.5f }
  };

  // 頂点の数
  static const int vertices(sizeof position / sizeof position[0]);

  // 頂点配列オブジェクトの作成
  Object object;
  object.vao = createObject(vertices, position);
  object.count = vertices;

  return object;
}
```

6.5 描画の実行

描画する図形データを保持した頂点配列オブジェクトを glBindVertexArray() 関数で指定し、glDrawArrays() 関数で基本図形の種類 (図 76) を指定して描画します。

```
glBindVertexArray(vao);
glDrawArrays(GL_LINE_LOOP, 0, vertices);
```

void glDrawArrays(GLenum mode, GLint first, GLsizei count)

頂点配列による図形の描画を行います。

mode

描画する基本図形の種類。図 76 に示すものに加えて、OpenGL 4.0 以降では GL_PATCHES が指定できます。

first

描画する頂点属性の要素の先頭の番号。頂点バッファの先頭の頂点から描画するなら 0。

count

描画する頂点属性の要素の数。例えば四角形なら 4。

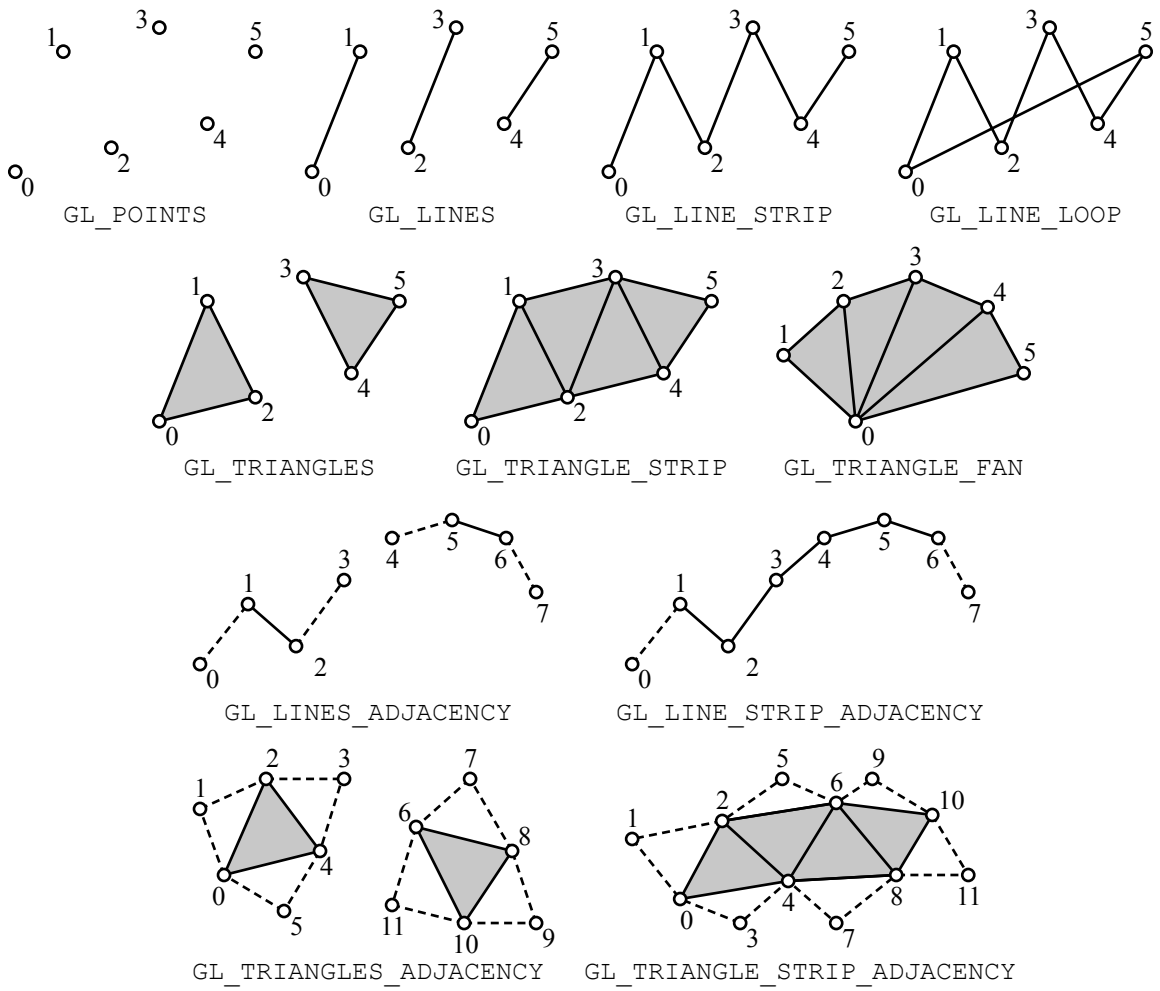

図 76 OpenGL の基本図形

● ソースプログラムの変更点

《省略》

```
// 頂点配列オブジェクトの作成
//   vertices: 頂点の数
//   position: 頂点の位置を格納した配列
static GLuint createObject(GLuint vertices, const GLfloat (*position)[2])
{
  《省略》
}

// 形状データ
struct Object
{
  《省略》
};

// 矩形のデータを作成する
static Object createRectangle()
{
  《省略》
}
```

```
int main()
{
  《省略》

  // 背景色を指定する
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);

  // プログラムオブジェクトを作成する
  const GLuint program(loadProgram("point.vert", "pv", "point.frag", "fc"));

  // 形状データを作成する
  const Object object(createRectangle());

  // ウィンドウが開いている間繰り返す
  while (glfwWindowShouldClose(window) == GL_FALSE)
  {
    // ウィンドウを消去する
    glClear(GL_COLOR_BUFFER_BIT);

    // シェーダプログラムの使用開始
    glUseProgram(program);

    // 図形を描画する
    glBindVertexArray(object.vao);
    glDrawArrays(GL_LINE_LOOP, 0, object.count);

    // カラーバッファを入れ替える
    glfwSwapBuffers(window);

    // イベントを取り出す
    glfwWaitEvents();
  }
}
```

- 実行結果

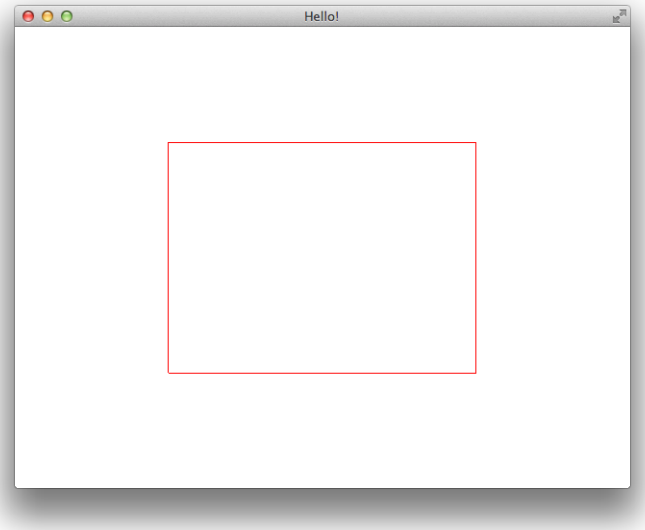

図 77 実行結果