第2回 図形の描画

2.1. 画材としてのコンピュータ

2.1.1. フレームメモリ

デジタル画像において色を決定するのは、各画素に格納された色のデータです。ディスプレイに表示する場合、色のデータは数値化された明るさの値です。ディスプレイはこの値を光の強さに変換して色を再現します。つまり、デジタル画像は明るさの数値を絵の具に使って形を表現します。

そうなると、その絵の具を載せるカンバスは、データを格納する先のメモリだということになります。このメモリは２次元に配列されており、そこにデータを格納することによって、形を表現します（**エラー! 参照元が見つかりません。**）。このようなメモリをフレームメモリと言います。

フレームメモリは画像の記憶に用いられるメモリで、色のデータの格納先となるとともに、この内容を繰り返し読みだして、映像信号を生成することにも用いられます。このようにデータと映像信号の仲立ちをする役割を持つことから、これはフレームバッファとも呼ばれます。

2.1.2. ラスタグラフィックス

このフレームメモリを前提にした CG を、ラスタグラフィックスと呼ぶことがあります。一般的なディスプレイは、画面の左上隅から水平方向に

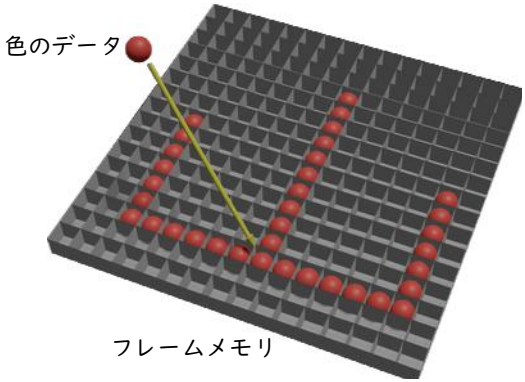


図 2.1 フレームメモリのイメージ

画素の色を更新してゆき、折り返しながら右下に到達します（図 2.2)。このような表示方式のディスプレイを、ラスタスキャンディスプレイと言います。ラスタ（Raster）はこのディスプレイ上の 1 本 1 本の水平線のことを言い、スキャンライン（Scan Line，走査線)とも言います。

これに対し、ベクタグラフィックス（ベクトルグラフィックス）というものも存在します。これは以前テレビでも使われていたブラウン管（CRT，陰極線管）やレーザなどを使って、輝点を動かして図形を描くものです。

テレビではこの輝点で画面全体を走査して濃淡のある映像を表示していましたが、ベクタグラフィックスでは輝点の位置を直接制御します。図

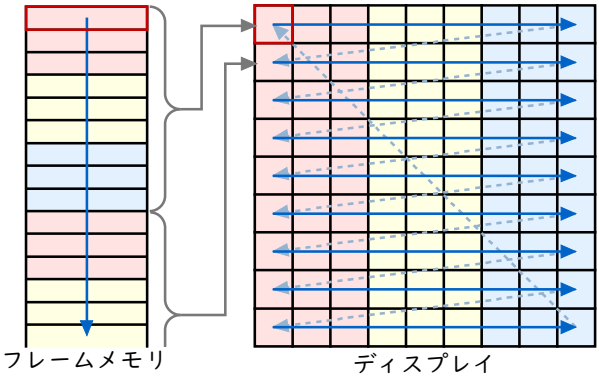


図 2.2 ラスタスキャンディスプレイ

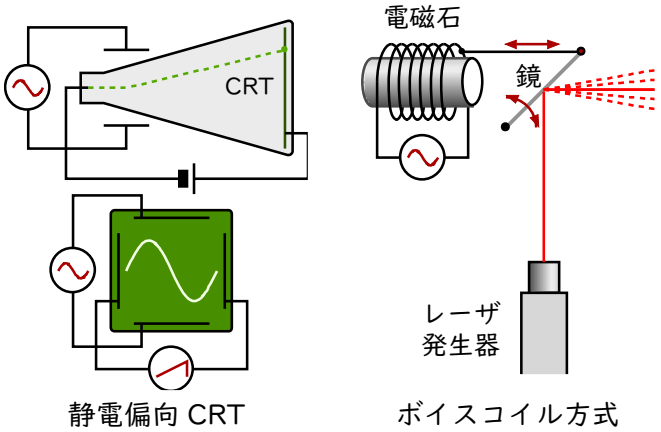


図 2.3 ベクタグラフィックスの例

2.3 の静電偏向 CRT 方式は、CRT 中を飛行する電子の向きを電極に与えた電位差によって変えることにより、CRT 管面上の輝点の位置を制御します。またボイスコイル方式はレーザ光を反射している鏡の向きを電磁石で変えることにより、反射光の輝点の位置を制御します。

最初の CG システムである SKETCHPAD は、MIT の TX-2 というコンピュータに接続されたベクタグラフィックス方式のディスプレイを使って動作していました（図 2.4)。これ以後、このディスプレイは初期の CG システムではよく使われていましたが、陰影を表示することが難しく、テレビ放送にもなじまなかったため、一般には使われなくなりました。

2.1.3. デジタル画像の種類

デジタル画像は、同時に使用する色の数によって、用途が異なります。ここでは、その色に数によって分類します。

(1) 2 値画像

画像を白・黒などの 2 値で表します（図 2.5)。白を 0，黒を 1 のように決めることで、1 画素を 1 bit で表すことができます。現在でもファクシミリ等で使用されていますが、濃淡は表現することはできません。濃淡のある白黒画像（グレイスケール画像）から２値画像を得る処理を２値化と言いますが、閾値を固定した２値化では元の濃淡が失われてしまうので、点の大きさや密度などを用いて濃淡を表現する面積階調法が用いられます。

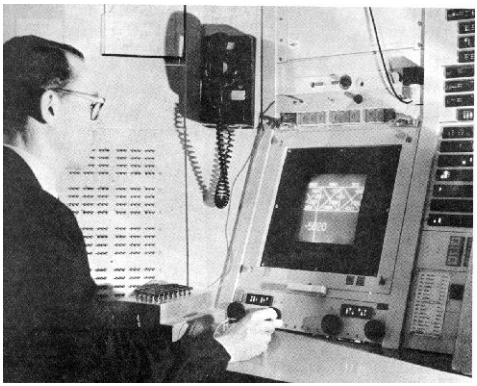


図 2.4 SKETCHPAD と開発者の Ivan Sutherland (WikiPedia)

a) パターンディザ法

パターンディザ法は 1 画素の濃淡を一定の面積（画素数）を持ったパターンに置き換える方法です。1 画素がパターンに置き換わるので、２値画像化後の解像度が低いとパターンの境界が見えてしまいます。

b) スクリーン（網点）

スクリーン（網点）はスクリーン印刷で使われる手法です。スクリーン印刷では、細かな穴をあけた版（スクリーン）にインクを載せ、穴から漏れ出たインクを版の下に置いた紙に転写します。このため、色の濃淡は穴

図 2.5 ２値画像

パターンディザ法

スクリーン（網点）

オーダードディザ法

誤差拡散法

図 2.6 面積階調法

の大きさで決まります。また、この濃淡の階調数は開ける穴の大きさのバリエーションで決まりますが、印刷可能な解像度[1]には上限があるため、階調数を増やすほど印刷可能な解像度が下がります。

この解像度は 1 インチ当たりに描ける線の数で表され、線数（スクリーン線数、単位 line per inch, lpi）と呼ばれます。線数はわら半紙や新聞紙など繊維の粗い紙では 60～80 線、通常の印刷物で 100～150 線、カタログ写真などのカラー印刷では 150～200 線、美術印刷のような用途では 300～350 線、もしくはそれ以上に設定されます。

図 2.7 に画像処理ソフトウェアの PhotoShop を用いた、ハーフトーンスクリーンによる 2 階調化の例を示します。この元のグレイスケール画像

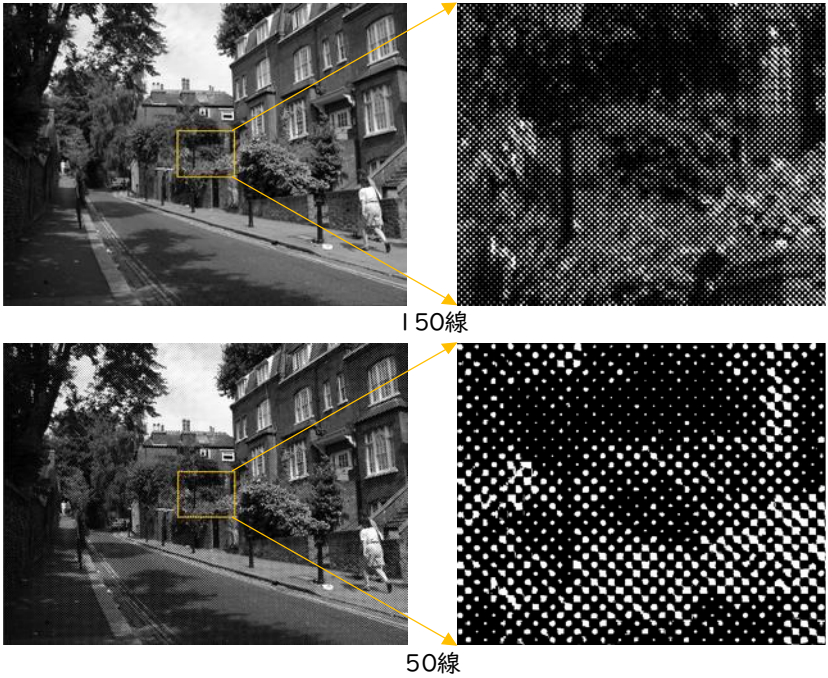


図 2.7 線数と解像度の違い

[1] 製版機の解像度（版に開けることが可能な最も小さな穴の密度）や印刷機の解像度（紙の種類によってインクのにじみによる印刷可能な最も小さい点の密度）。

の解像度は 1 インチ当たり 144 画素であり、それを 1 インチ当たり 1200 画素の解像度でモノクロ印刷するものとします。線数を 150 線とした場合は解像感が向上しますが、階調が少ないため場合によってはグラデーションの部分に疑似輪郭が発生する（階段状になる）ことがあります。線数を 50 線とした場合は階調が滑らかになりますが、解像度が低下します。

c) オーダードディザ法

画素ごとに濃淡値と乱数を比較して 2 値化を行うことにより、点の密度を確率で制御して濃淡を表す手法は、ランダムディザ法と言います。オーダードディザ法（組織的ディザ法）は、小領域ごとに対応する配列に格納された乱数と比較します。乱数には Bayer マトリクスが良く用いられます。

Bayer マトリクスは、次のようにして作ります。まず、2 行 2 列の Bayer マトリクスを、次のように与えます。

$$\mathbf{M}_2 = \frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \tag{2.1}$$

$2n$行$2n$列の Bayer マトリクスは、これから次の漸化式で求めることができます。

$$\mathbf{M}_{2n} = \frac{1}{(2n)^2}\begin{bmatrix} (2n)^2\mathbf{M}_n + 0 & (2n)^2\mathbf{M}_n + 2 \\ (2n)^2\mathbf{M}_n + 3 & (2n)^2\mathbf{M}_n + 1 \end{bmatrix} \tag{2.2}$$

これから、4 行 4 列の Bayer マトリクスは、次のようになります。

$$\begin{aligned} \mathbf{M}_4 &= \frac{1}{4^2}\begin{bmatrix} 4^2 \times \frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} + 0 & 4^2 \times \frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} + 2 \\ 4^2 \times \frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} + 3 & 4^2 \times \frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} + 1 \end{bmatrix} \\ &= \frac{1}{16}\begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix} \end{aligned} \tag{2.3}$$

d) 誤差拡散法

ランダムディザ法の一種ですが、2 値化の際に既に 2 値化が完了した近傍の画素における誤差（真の濃淡値と 2 値化後の濃淡値の差）を加味することにより、その近傍の領域全体での誤差を最小にしようとする方法です。

(2) グレイスケール画像

グレイスケール画像は 1 画素で濃淡（明るさ）を表現する画像です。白黒写真は色をグレイとして濃淡を変化させた場合です（図 2.8）。この濃淡は 8bit (1byte, 1octet[2]) の整数で表すことが多く、その場合は 0 が最も暗く、255 が最も明るくなります。

より多くの階調数（量子化数）を用いる場合もあり、たとえば画像処理ソフトウェアの PhotoShop には、16bit モードや 32bit モードが用意されています。また、高ダイナミックレンジ画像 (High Dynamic Range Image, HDRI) では実数が用いられます。

図 2.8 グレイスケール画像

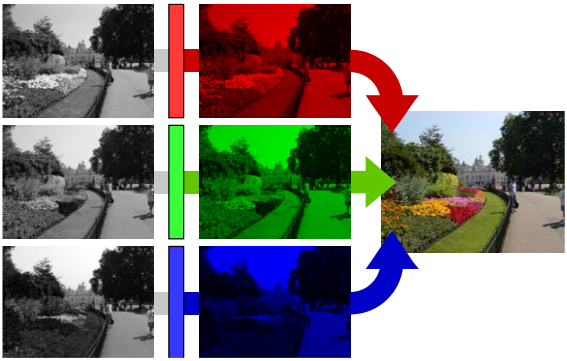


図 2.9 フルカラー画像

[2] 1 byte は 8 bit だとは限らないので、8 bit であることが重要なら octet という単位を使います。

(3) フルカラー画像

フルカラー画像は光の３原色（赤:R，緑:G，青:B）のそれぞれについて、グレイスケール画像で明度を表現するものです。ディスプレイなどでは、これらの色の画像を画素ごとに隣接させて表示することにより、人間の目には濃淡のあるフルカラー画像として知覚されます（図 2.9）。

本来、人間の視覚は赤、緑、青の３色しか知覚できません。ただし、それぞれの色を知覚する波長の範囲には幅があるので、中間の色は、その色のこの３色のそれぞれに対する刺激の強さの割合で知覚されます。例えば赤と緑の中間の波長の光は赤と緑の両方に知覚され、黄色だと知覚されます（図 2.10）。この３原色による色の合成を加法混色と言います（図 2.11）。

一方、印刷物の反射光の場合は、無色（白色・紙色）の部分が入射光に含まれるすべての波長の光を反射するのに対して、色のついた部分はインクがその色以外の波長の光を吸収しています。このため複数の印刷のイン

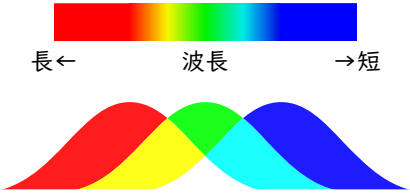
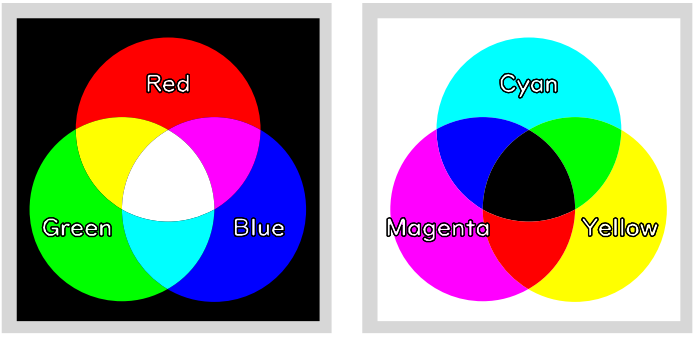


図 2.10 光の波長と色（イメージ）



加法混色（RGB）　　減法混色（CMY）

図 2.11 加法混色と減法混色

クを混ぜると、混ぜたインクのどれにも吸収されなかった波長の光しか反射しなくなります。

インクの３原色（シアン:C，マゼンタ:M，黄:Y）のうちシアンとマゼンタを混ぜると、そのどちらにも吸収されなかった青だけを反射するようになります。また、これらをすべて混ぜたときには、人間が知覚できる赤、緑、青のすべての波長の光を吸収してしまうため、黒に近くなります[3]。これは減法混色と言います。

フルカラー画像は複数のグレイスケール画像で構成されるため、そのデータ構造にもいくつかの種類があります。その一つに、１画素ごとのに３原色の色成分をすべて持たせる方法があります。これはパックドピクセル方式と呼ばれます。また、３枚のグレイスケール画像をそれぞれ保持する方法もあります。これはプレーナピクセル方式と呼ばれます（図 2.12）。

(4) インデックスカラー画像

インデックスカラー画像はルックアップテーブル（カラーパレットあるいはカラーマップと呼ぶこともあります）という表に色データを格納しておき、フレームメモリにはルックアップテーブルの色の番号を格納する方

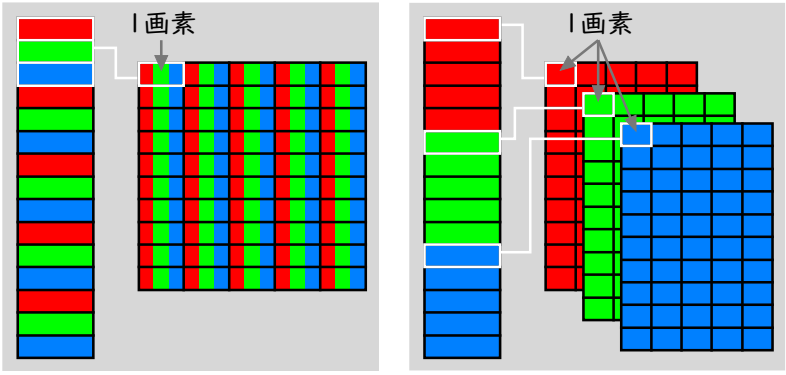


図 2.12 フルカラー画像のデータ構造の例

[3] 完全な黒になるとは限りません。こうして合成した黒色は、コンポジットブラックと呼ばれます。このため、印刷物では黒を別に用意するのが一般的です（CMYK）。

式です（図 2.13）。

この方式は少ないデータ量で多くの色を表示することができ、ルックアップテーブルを書き換えることにより色の変更も容易に行えますが、同時に表示できる色の数はルックアップテーブルの要素数に制限されます。GIF 画像や 8 bit の PNG 画像などは、この方式を採用しています。

2.2. デジタル画像の生成

2.2.1. 点を描く

ラスタグラフィックスにおける図形の描画は、フレームメモリに色のデータを格納することによって行われます。CPU や GPU から見たとき、メモリは 1 次元に配列されていますから、これをフレームメモリとして扱うには、2 次元の空間に置き換える必要があります。

たとえば、ディスプレイの表示領域の幅（横の画素数）を w、高さを h とし、原点をその左上隅とします。この表示領域の (x, y) の位置に点を打つためには、フレームメモリの先頭から $wy + x$ の位置のメモリにデータを格納します（図 2.14）。

この処理を次のプログラムで模倣してみます。表示領域の幅 `width` を `40`、高さ `height` を `30` とします。フレームメモリには 1 次元の配列変数を使い、要素数はこの画素数 `width`×`height` とします。

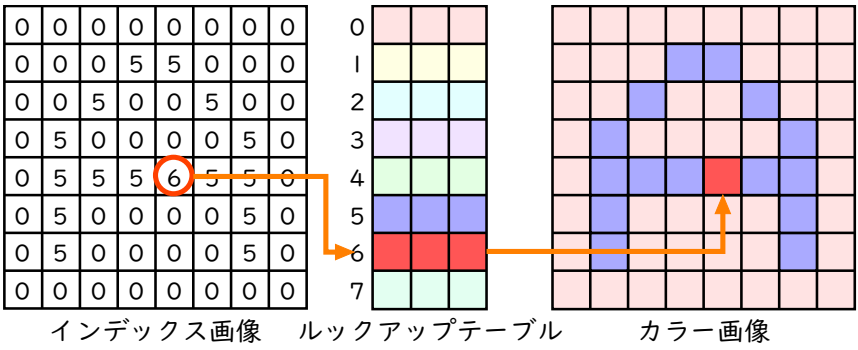


図 2.13 インデックスカラー画像

```
const width = 40
const height = 30
let frameMemory = Array(width * height)
```

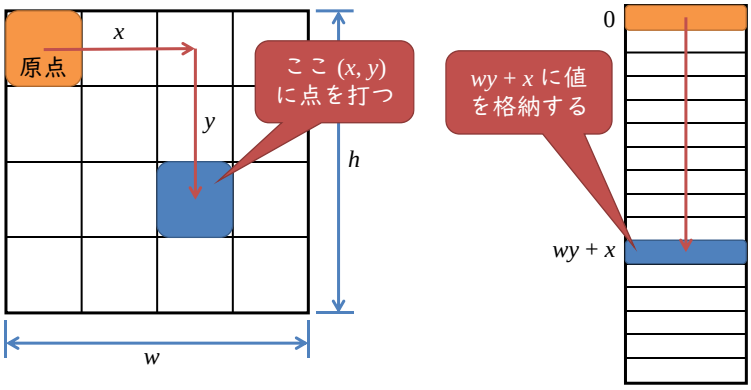
最初に、このすべての要素に空白文字‘ ’を入れておきます。

```
frameMemory.fill(' ')
```

次に、この表示領域の (x, y) の位置に点を打つ関数 point(x, y) を定義します。これはフレームメモリの $wy + x$ の位置にデータを格納します。配列の添え字には実数が使えないので、Math.round() で整数化します。データはアスタリスクの文字‘*’です。この文字幅はブラウザのフォントによって‘ ’と同じとは限らないので、適当に変えてください。

```
function point(x, y) {
  frameMemory[width * Math.round(y) + Math.round(x)] = '*'
}
```

また、画面表示を行う関数 print() を定義します。ここではフレームメモリの内容をブラウザ上に <pre>～</pre> はさんですべて出力します。1 行出力したら、document.writeln() によって表示を改行します。



ディスプレイの表示領域　　フレームメモリ

図 2.14 点の描画

```
function print() {
  document.write('<pre>')
  for (let y = 0; y < height; ++y) {
    for (let x = 0; x < width; ++x) {
      document.write(frameMemory[width * y + x])
    }
    document.writeln()
  }
  document.write('</pre>')
}
```

これで、例えば (15,25) に点を打つとすれば、次のようにします。

```
point(15, 25)
```

これでフレームメモリにデータが格納されたので、ブラウザ上に表示します。これで図 2.15 のような結果が出ます（破線は表示されません）。

```
print()
```

2.2.2. 水平線を描く

次に、この point(15, 25) を書き換えて、表示領域の中央の高さに、左端から右端まで水平線を引くことを考えます。

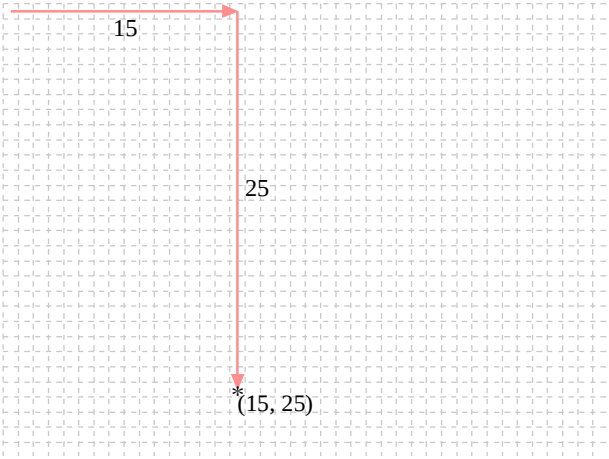


図 2.15 「点」を描画した結果

図 2.16 において、最初に変数 x を始点の位置 x_0 とし、x が終点の位置 x_1 以下の間 (x, y) に点を打ちます。そして x を 1 増して同じ処理を繰り返します。この処理をフローチャートで表現すると、図 2.17 のようになります。表示領域の端から端まで水平線を引く場合、始点の位置 x_0 は 0、終点の位置 x_1 は width - 1 です。また高さ y は height / 2 です。

このプログラムは次のようになります。繰り返しの終了条件は、上記によれば x <= width - 1 ですが、x は 1 ずつ増えるので、x < width としても x が width - 1 に達するまで繰り返されます。

```
let y =  height / 2
for (let x = 0; x < width; ++x) {
  point(x, y)
}
```

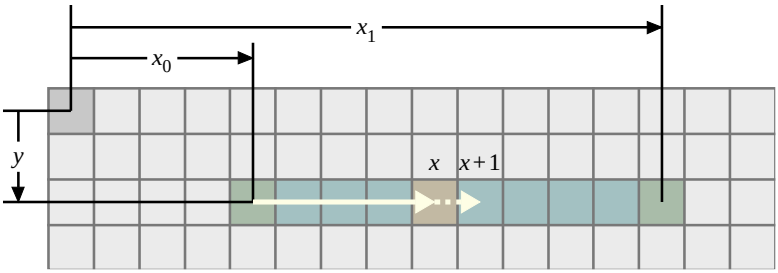


図 2.16 水平線を描く

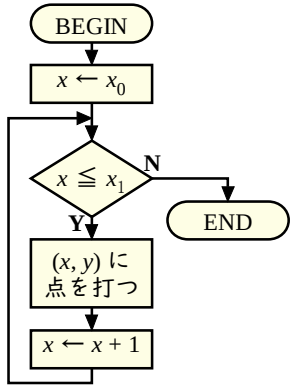


図 2.17 水平線を描くフローチャート

2.2.3. 領域を塗りつぶす

今度は、2 点 (x_0, y_0), (x_1, y_1) を対角線とする矩形の領域を塗りつぶす方法を考えます。ここで $x_0 \leq x_1$, $y_0 \leq y_1$ とします。その前に、前節で示した水平線を描くプログラムを関数に直します。この関数名は hline とし、y の高さの x_0 から x_1 に水平線を引くものとします。

```
function hline(x0, x1, y) {
  for (let x = x0; x <= x1; ++x) {
    point(x, y)
  }
}
```

これを高さ y について y_0 から y_1 まで繰り返します。ここでは関数名を fill にしています。

```
function fill(x0, y0, x1, y1) {
  for (let y = y0; y <= y1; ++y) {
    hline(x0, x1, y)
  }
}
```

なお、この関数は $x_0 \leq x_1$ かつ $y_0 \leq y_1$ という条件を確認していません。このプログラムを確実に動作するようにするには、条件を満たさないときに仮引数の内容を交換するなどの処理を追加する必要があります。

2.2.4. ラスタ化処理

このように指定された座標値から、その座標値が表す図形の形をフレームメモリに描く処理を、ラスタ化処理（ラスタライズ）と言います。この処理は現在のほとんどのグラフィックスハードウェアに組み込まれているラスタライザと呼ばれるハードウェアで実行されます。

ラスタ化処理のアルゴリズムは走査変換（スキャンコンバージョン）と呼ばれ、線分、円、台形・三角形、および任意の多角形を描くアルゴリズムなどが知られています。このうち一般的な GPU には、線分と三角形を描く機能が搭載されています。この場合、任意の多角形は三角形に分割されて描画されます。

(1) 線分の描画

- 「補助資料(1) 線分の描画」参照

(2) 三角形の描画

- 「補助資料(2) 三角形の描画」参照

2.3. WebGL

WebGL はブラウザ上でリアルタイム 3D グラフィックスの表示を行う機能です。もちろん、2D グラフィックスも描くことができます。

現在のパソコンやスマートフォンでは、グラフィックス表示を実際にアプリケーションプログラムが動作している CPU ではなく、CPU から指令を受けてグラフィックス表示を行う GPU が行っています。ところが前回示したスクリプトは、CPU 上のプログラム（ブラウザは CPU 上で動作しているアプリケーションプログラムです）によって、「どこからどこまで線を引け」というような命令をしているように見えます。

これは 2D グラフィックスの機能が、CPU が直接グラフィックス表示の処理を行っていた頃の API のデザインを反映したものになっている（と思われる）からです。現在のグラフィックスハードウェアでは GPU が CPU とは独立して自分自身でグラフィックス表示の処理を行いますが、このよ

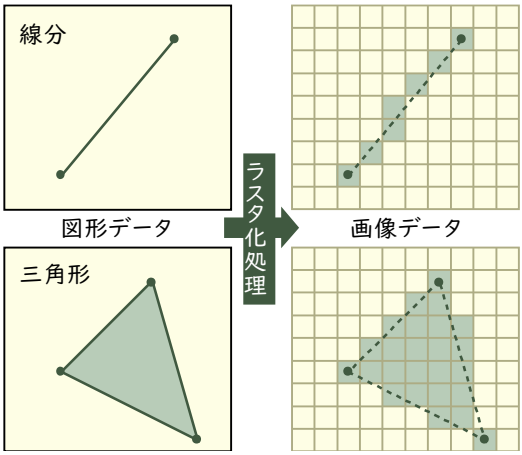


図 2.18 ラスタ化処理

うに CPU が直接グラフィックス表示を行う手順で現在のグラフィックスハードウェアを使っても、GPU の性能や機能を活かすことができません。

WebGL はグラフィックス API の標準規格である OpenGL ES を、Web ブラウザから使用する API です。OpenGL ES は GPU に対応したリアルタイム 3D グラフィックスライブラリです。WebGL 1.0 は OpenGL ES 2.0 から派生した仕様になっており、WebGL 2.0 は OpenGL ES 3.0 から派生した仕様になっています。この講義では WebGL 2.0 を使用します。

この WebGL によるプログラミングを行う際に念頭に置いて欲しいことは、これが CPU と GPU という異なる処理をする処理装置を組み合わせて使用するという点です。そのため WebGL では、CPU 用のプログラムと GPU 用のプログラムを別々に作ります。それぞれのプログラミング言語も、似てはいるものの、異なるものを使います。

2.3.1. レンダリングパイプライン

WebGL によるグラフィックス処理の流れは、レンダリングパイプラインという概念を用いて説明されます。パイプラインとは作業工程を細かな段階（ステージ）に分割することで、全体のスループット（単位時間あたりに処理できる量）を向上しようとする考え方です。例えば、ある処理が単一のステージで構成されていると、そのステージはタスクが完了するまで次のタスクを受け入れられません。処理を複数の小さなステージに分割すれば、個々のステージが完了したタスクを次のステージに送ることにより、ステージの数だけスループットを増すことができます（図 2.19）。

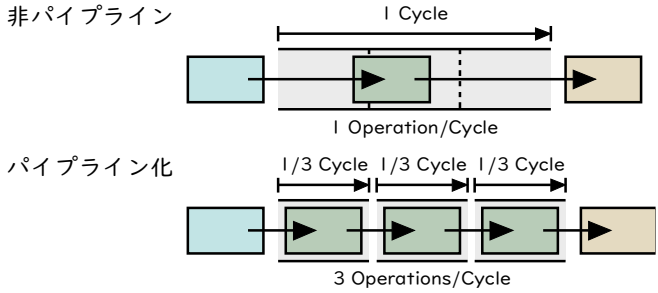


図 2.19 パイプライン処理

図 2.20 にグラフィックスハードウェアのレンダリングパイプラインの概略を示します。このレンダリングパイプラインから見れば、前のステージは CPU であり、グラフィックスハードウェアはそこから頂点の位置などの頂点属性を受けとります。受け取った頂点属性は、頂点バッファというグラフィックスハードウェア上のメモリに保持します。

頂点バッファが保持している頂点属性は、GPU が CPU からの描画命令を受け取ったときに頂点ごとに取り出され、個々にバーテックスシェーダに入力されます。バーテックスシェーダは入力された頂点属性に対して座標変換などの頂点ごとの処理を実行し、結果を（描画する図形に応じて頂点属性を組み合わせた後）次のステージであるラスタライザに送ります。

ラスタライザは受け取った頂点属性をもとにラスタ化処理を行います。ラスタ化処理は描画する図形の形に塗りつぶすフラグメント（画素）を選択し、個々のフラグメントに対してフラグメントシェーダを起動します。フラグメントシェーダはラスタライザから受け取った頂点属性の補間値などをもとに画素の色などを決定し、フレームバッファに描き込みます。

2.3.2. WebGL によるグラフィックスプログラミング

この WebGL を使ってグラフィックスプログラミングを行うための、基本的な手順について説明します。

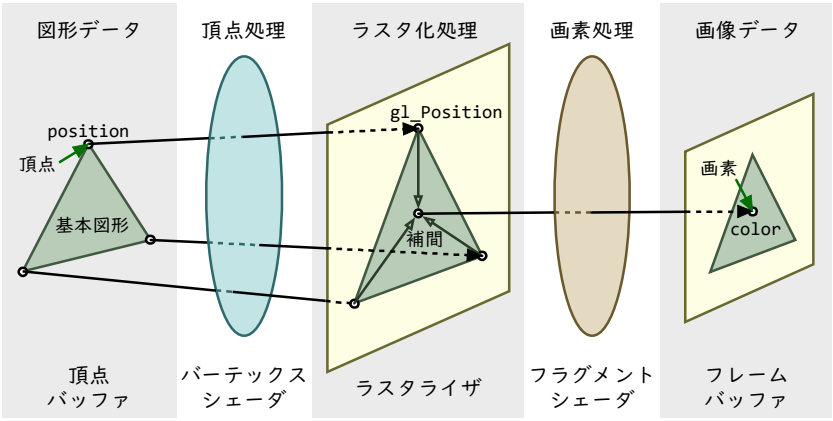


図 2.20 レンダリングパイプライン

(1) WebGL 2.0 のコンテキストを作る

最初に 2 次元グラフィックスのコンテキスト('2d')の代わりに WebGL 2.0 のコンテキスト（'webgl2'）を得ます。これで WebGL の API が使えるようになりますから、それを使って canvas 要素全体をビューポート（図形の表示領域）に設定し、canvas 要素を消去するときの色を設定します。

```
<body>
  <canvas id="CG2" width="400" height="300" style="border: inset;">
    HTML5 の canvas が使えません
  </canvas>
  <script>
    const canvas = document.getElementById('CG2')
    const gl = canvas.getContext('webgl2')

    gl.viewport(0, 0, canvas.width, canvas.height)
    gl.clearColor(0.2, 0.3, 0.4, 1.0)

    gl.clear(gl.COLOR_BUFFER_BIT)
  </script>
</body>
```

void gl.viewport(x, y, w, h)

デバイス座標系上にビューポート（図形を描く領域）を設定します。図形のビューポートからはみ出た部分は切り取られて表示されません。

x, y

ビューポートの左下隅の位置を指定します。ウィンドウの左下隅にあるデバイス座標系の原点からの相対位置を画素数で指定します。

w, h

ビューポートの幅と高さを画素数で指定します。w に負の数を指定すると、描画する図形の左右が反転します。h に負の数を指定すると、描画する図形の上下が反転します。

void gl.clearColor(R, G, B, A)

gl.clear(gl.COLOR_BUFFER_BIT) でウィンドウを塗り潰す色を指定します。

R, G, B

それぞれ赤色、緑色、青色の成分の強さを示す 0～1 の値を指定します。1 が最も明るく、それぞれ 0, 0, 0 を指定すれば黒色、1, 1, 1 を指定すれば白色になります (図 2.21)。

A

アルファ値と呼ばれ、OpenGL ES では不透明度として扱われます (0 で透明、1 で不透明)。

void gl.clear(mask)

描画するフレームバッファ全体を塗り潰します（図 2.22)。

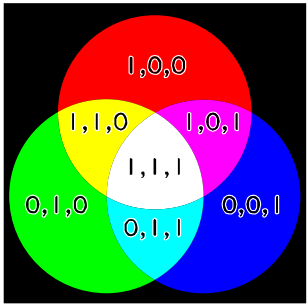


図 2.21 R, G, B の値と色

図 2.22 カンバスを gl.clear() で消去した結果

mask

塗り潰すバッファを指定します。図形の描画を行うメモリをフレームバッファといいます。これは色を格納するカラーバッファのほか、隠面消去処理に使うデプスバッファ、図形の型抜きを行うステンシルバッファなどの複数のバッファで構成されています(図 2.23)。mask に gl.COLOR_BUFFER_BIT を指定したときは、カラーバッファだけを gl.clearColor() で指定した色で塗り潰します。

(2) シェーダを準備する

GPU が実行するプログラムのことをシェーダと言います。OpenGL ES のシェーダのソースプログラムは、OpenGL Shading Language (GLSL) というプログラミング言語で記述します。また、これは CPU から送られてきた頂点属性を処理するバーテックスシェーダと、フレームバッファに描き込む画素を処理するフラグメントシェーダの 2 つを用意する必要があります。これらを CPU 側で文字列として用意して、OpenGL ES の API を使ってコンパイルとリンクを実行します。こうして GPU で実行可能なプログラムオブジェクトを作成し、GPU に送ります。

a) バーテックスシェーダ

バーテックスシェーダ (Vertex Shader) は図形の頂点ごとに実行されるプログラムで、頂点属性をもとに座標変換などを行います。このソース

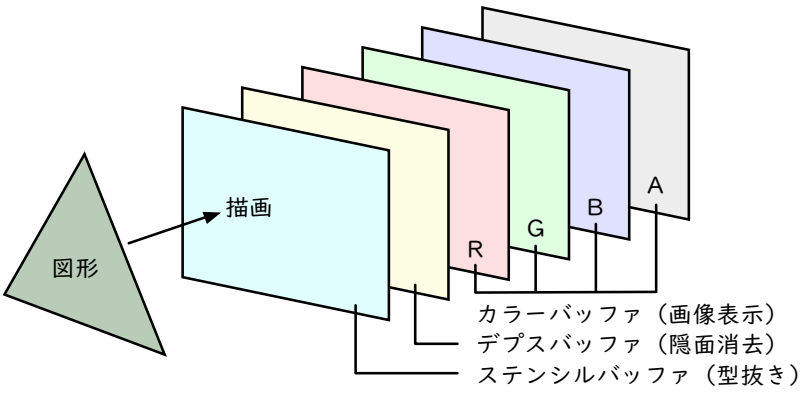


図 2.23 フレームバッファの構成

プログラムは、現時点では次のようにしています。

```
#version 300 es

layout (location = 0) in vec4 position;

void main()
{
  gl_Position = position;
}
```

この 1 行目の「#version 300 es」は、使用する GLSL のバージョンを表します。この「300 es」は OpenGL ES バージョン 3.0 とともに策定された GLSL ES バージョン 3.0 を表します。この行は GLSL のソースプログラムの 1 行目に置く必要があります。

「in vec4 position」は vec4 型の変数 position を、このシェーダの入力変数 (in) に使うという宣言です。vec4 は float 型の 4 つの実数からなるベクトルのデータ型です。OpenGL ES では 3 次元の座標値 (x^*, y^*, z^*) を 4 次の同次座標 (x^*w, y^*w, z^*w, w) で表します。同次座標については、次回以降に解説します。

バーテックスシェーダの入力は頂点バッファに保持されている頂点属性すなわち Vertex Attribute なので、バーテックスシェーダの入力変数は、特に attribute 変数と呼ばれます。「layout (location = 0)」は、この attribute 変数によりインデックスが 0 の頂点バッファオブジェクトからデータを取り出されることを示す修飾子 (qualifier) です。頂点バッファオブジェクトおよびそのインデックスについては後述します。

main() は C 言語などと同様、このプログラムのエントリポイント（実行開始点）の関数名です。ただしシェーダは値を返さないので、この戻り値のデータ型は void です。

gl_Position は GLSL ES に最初から用意されている組み込み変数で、これに代入した値が次のステージに送られます。このプログラムでは、頂点バッファオブジェクトから取り出した座標値を、そのまま gl_Position

に代入しています。

なお、gl_Position に代入した 4 次の同次座標 (x, y, z, w) は、3 次元空間上の $(x/w, y/w, z/w)$ の位置に投影されます。このとき、$-1 \leq x/w, y/w, z/w \leq 1$ の範囲からはみ出た図形の部分は刈り取られます（図 2.24）。これはクリッピング処理と言います。このことから、gl_Position の空間はクリッピング空間と呼ばれます。

このクリッピング空間に描かれた図形の、クリピング空間の xy 平面上への投影像が、ディスプレイの表示領域のビューポートに表示されます。

b) フラグメントシェーダ

バーテックスシェーダで処理された頂点属性は、描画命令で指定された図形要素（線分、三角形など）の頂点として組み合わせられた後、ラスタ化処理（ラスタライザ）に入力されます。ラスタ化処理は頂点属性で表された図形データから、画面に表示するための画像データを生成します。

フラグメントシェーダ (Fragment Shader) は、ラスタ化処理によって図形を塗りつぶす際に画素ごとに実行され、画素データを生成します。このソースプログラムは、現時点では次のようにしています。

```
#version 300 es

layout (location = 0) out vec4 color;
```

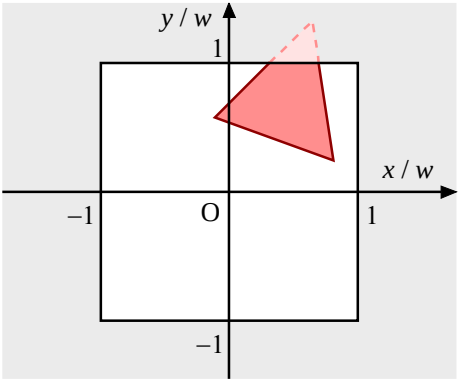


図 2.24 クリッピング

```
void main()
{
  color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

「out vec4 color」は vec4 型の変数 color を、このシェーダの出力変数 (out) に使うという宣言です。「layout (location = 0)」は、この変数に代入した値をフレームバッファの 0 番のアタッチメントに格納することを示す修飾子です。0 番のアタッチメントは、デフォルトでは画面に表示されるカラーバッファになっています。

「vec4(1.0, 1.0, 0.0, 1.0)」は 1.0、1.0、0.0、1.0 の 4 つの実数値を vec4 型の 1 つのベクトルに型変換するキャストです。このプログラムの場合、それぞれの値は左から順に赤 (Red)、緑 (Green)、青 (Blue) の強さ、および不透明度 (Alpha) を 0〜1 の値で指定します。この場合は赤と緑、および不透明度が 1 なので、図形は黄色で描画されます。

c) シェーダのソースプログラムの埋め込み

シェーダのソースプログラムは、次のようにして HTML ファイルに埋め込むことができます。これは実際にプログラムオブジェクトを作成するより前に置きます。

```
<body>
  <script id="vertex-shader" type="x-shader/x-vertex">
    #version 300 es
    precision mediump float;

    layout (location = 0) in vec4 position;

    void main()
    {
      gl_Position = position;
    }
  </script>
  <script id="fragment-shader" type="x-shader/x-fragment">
    #version 300 es
    precision mediump float;
```

```
    layout (location = 0) out vec4 color;

    void main()
    {
      color = vec4(1.0, 1.0, 0.0, 1.0);
    }
  </script>
  <canvas id="CG2" width="400" height="300">
    HTML5 の canvas が使えません
  </canvas>
  <script>
    const canvas = document.getElementById('CG2')
    const gl = canvas.getContext('webgl2')
    gl.viewport(0, 0, canvas.width, canvas.height)
    gl.clearColor(0.2, 0.3, 0.4, 1.0)
    gl.clear(gl.COLOR_BUFFER_BIT)
  </script>
</body>
```

d) プログラムオブジェクトの作成

プログラムオブジェクトの作成手順は少々長いので、関数にまとめます。以下の内容を先ほど作成したシェーダのソースプログラムのスクリプトと canvas 要素の間に追加します。ここでは関数名を loadShader() にしています。また、プログラムミスを発見しやすいように、最初に 'use strict' を入れて文法チェックを「厳格モード」にしておきます。

```
  <script>
    'use strict'

    //
    // シェーダのソースプログラムを読み込んでコンパイルする
    //
    function compileShader(id) {

      // シェーダのソースプログラムを読み込む
      const element = document.getElementById(id)
      const source = element.text.trim()

      // シェーダの種類
```

```
  const type = {
    "x-shader/x-vertex": gl.VERTEX_SHADER,
    "x-shader/x-fragment": gl.FRAGMENT_SHADER
  }

  // シェーダのソースプログラムをコンパイルする
  const shader = gl.createShader(type[element.type])
  gl.shaderSource(shader, source)
  gl.compileShader(shader)

  // コンパイルエラーが発生していたらエラーメッセージを表示する
  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    console.error(element.type + ':\n'
      + gl.getShaderInfoLog(shader))
    gl.deleteShader(shader);
    return null
  }

  return shader;
}

//
// プログラムオブジェクトを作成する
//
function loadShader(vertId, fragId) {

  // バーテックスシェーダをコンパイルする
  const vertShader = compileShader(vertId)

  // フラグメントシェーダをコンパイルする
  const fragShader = compileShader(fragId)

  // エラーが発生していればプログラムオブジェクトを作成しない
  if (!vertShader || !fragShader) return null

  // プログラムオブジェクトを作成する
  const program = gl.createProgram()

  // バーテックスシェーダとフラグメントシェーダをリンクする
  gl.attachShader(program, vertShader)
  gl.deleteShader(vertShader)
  gl.attachShader(program, fragShader)
  gl.deleteShader(fragShader)
```

```
    gl.linkProgram(program)

    // リンクエラーが発生していたらエラーメッセージを表示する
    if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
      console.error('Link: ' + gl.getProgramInfoLog(program))
      gl.deleteProgram(program)
      return null
    }

    // 作成されたプログラムオブジェクトを返す
    return program
  }
</script>
```

(3) 図形を準備する

図形の作成手順も結構長いので、関数にまとめます。ここでは関数名を createObject() にしています。仮引数の position は、頂点属性を格納した配列を受け取ります。この関数を先ほど定義した loadShader() の後ろに続けて定義します。

```
  //
  // 頂点配列オブジェクトを作成する
  //
  function createObject(position) {
```

a) 頂点配列オブジェクトの作成

頂点属性は図形の一つ一つの頂点に設定されるデータです。一つの頂点には位置や色、法線ベクトル、テクスチャ座標など、必要に応じて様々なものが設定されます。そのため、一つの図形を複数の頂点属性の組み合わせで表現すれば、頂点バッファオブジェクトも複数必要になります。

そこで頂点配列オブジェクト (Vertex Array Object, VAO) を作成して、それにこれらを登録することにより、これらをひとまとめにして取り扱えるようにします。図形の描画などの処理は、この VAO を結合 (bind, 処理の対象に指定) して行います。

```
    // 頂点配列オブジェクトを作成する
    const vao = gl.createVertexArray()
```

```
// 頂点配列オブジェクトを結合する
gl.bindVertexArray(vao)
```

WebGLVertexArrayObject gl.createVertexArray()

頂点配列オブジェクトを作成します。

戻り値

作成した頂点配列オブジェクト。

void gl.bindVertexArray(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。null なら現在の結合を解除します。

b) 頂点バッファオブジェクトの作成

GPU で図形を描画するには、先に描画する図形の頂点属性をグラフィックスハードウェアのメモリに転送します。そのため、このメモリ上に頂点属性のデータを保持する領域を、あらかじめ確保しておく必要があります。このメモリの領域のことを、頂点バッファといいます。

一方、バッファオブジェクトは GPU がメモリの管理に使う識別子で、これにバッファ、すなわちメモリ上の領域を割り当てます。バッファオブジェクトを削除すると、割り当てられたバッファも解放されます。

頂点バッファオブジェクト (Vertex Buffer Object, VBO) はバッファオブジェクトとして管理されている頂点バッファです。これをを作成するには、まずバッファオブジェクトを作成して、それを頂点バッファとして結合し、それにメモリの領域（バッファ）を割り当てます。VAO を結合している状態で VBO を結合すれば、その VAO に VBO が組み込まれます。

```
// バッファオブジェクトを作成する
const vbo = gl.createBuffer()

// 作成したバッファオブジェクトを頂点バッファとして結合する
gl.bindBuffer(gl.ARRAY_BUFFER, vbo)
```

WebGLBuffer gl.createBuffer()

バッファオブジェクトを作成します。

戻り値

作成したバッファオブジェクト。

void gl.bindBuffer(target, buffer)

バッファオブジェクトを結合して使用可能にします。

target

バッファオブジェクトを結合する対象を指定します。バッファオブジェクトの用途によって、結合する対象は異なります。

gl.ARRAY_BUFFER

頂点属性を保持するバッファ（頂点バッファ）。

gl.ELEMENT_ARRAY_BUFFER

頂点のインデックスを保持するバッファ。

gl.COPY_READ_BUFFER

バッファオブジェクトを他にコピーするとき読み出し側として使うバッファ。

gl.COPY_WRITE_BUFFER

バッファオブジェクトを他にコピーするとき書き込み側として使うバッファ。

gl.TRANSFORM_FEEDBACK_BUFFER

Transform Feedback 処理（GPU で計算した結果を再びバッファオブジェクトに書き戻す機能）に用いるバッファ。

gl.UNIFORM_BUFFER

uniform ブロックに使うバッファ。

gl.PIXEL_PACK_BUFFER

画素データを（CPU のメモリなどから）詰め込む先のバッファ。

gl.PIXEL_UNPACK_BUFFER

画素データを（CPU のメモリなどへ）取り出す元のバッファ。

buffer

結合するバッファオブジェクト。null なら現在の結合を解除します。

c) メモリの確保とデータの転送

現在 VBO として結合されているバッファオブジェクトにバッファを割り当て、そこにデータを転送します。

頂点バッファオブジェクトのバッファの確保と、そこへのデータの転送は、次の手順で行います。OpenGL ES の場合、転送元のデータは連続したメモリ上に並んでいる必要があります。しかし、いま作成している関数 createObject() の引数で与えられた配列 position は、要素の追加・挿入や削除を行いやすくするために連続したメモリ上にありません。

そこで、Float32Array() を使ってプラットフォームのバイト順による 32bit の実数（C や C++ 言語の float 型に相当します）の配列のオブジェクトを生成します。

```
// 頂点バッファオブジェクトのバッファを確保してデータを転送する
gl.bufferData(gl.ARRAY_BUFFER,
  new Float32Array(position), gl.STATIC_DRAW)
```

void gl.bufferData(target, data, usage)

現在結合されているバッファオブジェクトにバッファ（メモリの領域）を確保し、そこにデータを転送します。バッファオブジェクトには頂点バッファオブジェクト以外のものもあります。頂点バッファオブジェクト（gl.ARRAY_BUFFER）を指定したときは、頂点属性を転送します。

data

バッファに転送するデータが格納されている CPU 側の配列。

usage

このバッファの使われ方。gl.STREAM_DRAW、gl.STREAM_READ、g.STREAM_COPY 、 gl._STATIC_DRAW 、 g.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

このバッファは一度だけ書き込まれ、高々数回使用されます。

STATIC

このバッファは一度だけ書き込まれ、何度も使用されます。

DYNAMIC

このバッファは繰り返し書き込まれ、何度も使用されます。

DRAW

このバッファには CPU 側から書き込まれ、描画のためのデータとして用いられます。

READ

このバッファの内容は CPU 側からの問い合わせによって、OpenGL ES (GPU) 側から読み出されます。

COPY

このバッファの内容は CPU 側からの指令によって OpenGL ES (GPU) 側から読み出され、描画するデータとして用いられます。

d) 頂点バッファオブジェクトのインデックスの設定

また、この頂点バッファオブジェクトにインデックス（番号）を付けておきます。このインデックスはシェーダの実行を開始した GPU が頂点属性を取り出すバッファの識別子として用います。

```
// 頂点バッファオブジェクトの先頭のインデックスを 0 に設定する
gl.vertexAttribPointer(0, 2, gl.FLOAT, false, 0, 0)

// 0 番のインデックスを有効にする
gl.enableVertexAttribArray(0)
```

void gl.vertexAttribPointer(index, size, type, normalized, stride, offset)

図形の描画時に attribute 変数（バーテックスシェーダの in 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

attribute 変数に layout 修飾子で指定した location の値を指定します。このプログラムでは attribute 変数 position に location = 0 を指定していますから、それに合わせてこれに 0 を指定します。

size

attribute 変数が受け取る 1 つの頂点属性の要素数を指定します。1、2、3、4 の値が指定できます。頂点属性が 2 次元 (x, y) なら 2、3 次元 (x, y, z) なら 3 を指定します。このプログラムの図形データは 2 次元なので、ここでは 2 を指定します。

type

attribute 変数が受け取るデータ型を指定します。gl.BYTE（符号付き 8bit 整数）、gl.UNSIGNED_BYTE（符号なし 8bit 整数）、gl.SHORT（符号付き 16 ビット整数）、gl.UNSIGNED_SHORT（符号なし 16bit 整数）、gl.FLOAT（32bit 実数）、gl.HALF_FLOAT（16bit 実数）が指定できます。このプログラムの図形データは実数なので、ここでは gl.FLOAT を指定します。

normalized

これが true のとき、type が gl.BYTE か gl. SHORT なら頂点バッファ内の値を $[-1,1]$ に正規化し、type が gl. UNSIGNED_BYTE か gl.UNSIGNED_SHORT ならバッファ内の値を $[0,1]$ に正規化します。これが false か、あるいは type が gl.FLOAT か gl.HALF_FLOAT のときは、バッファ内の値がそのまま用いられます。このプログラムでは type が gl.FLOAT なのでどちらでも構わないのですが、正規化を行わないことには変わりはないので false を指定します。

stride

attribute 変数が受け取る頂点属性の配列の、要素間の間隔を指定します。0 なら頂点属性の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 を指定します。

offset

attribute 変数が受け取る頂点属性のデータが格納されている場所を、頂点バッファの先頭からバイト単位で指定します。ここでは頂点バッファの先頭からデータを取り出すので、0 を指定します。

void gl.enableVertexAttribArray(index)

頂点バッファオブジェクトのインデックスを有効にします。

index

有効にする頂点バッファオブジェクトのインデックス。

e) 頂点バッファオブジェクトの結合解除

頂点バッファオブジェクトの設定が終わったら、結合を解除します。

```
    // 頂点バッファオブジェクトの結合を解除する
    gl.bindBuffer(gl.ARRAY_BUFFER, null)
```

f) 頂点配列オブジェクトの結合解除

全ての頂点バッファオブジェクトの設定が終わって図形の定義が完了したら、頂点配列オブジェクトの結合を解除します。

```
    // 頂点配列オブジェクトの結合を解除する
    gl.bindVertexArray(null)
```

最後に、作成した頂点配列オブジェクト（vao）と頂点数（count）をプロパティにもつオブジェクトを戻り値として返します。頂点数は position の要素数を次元で割ったもの（2 次元なら x, y の 2 つで 1 頂点）です。

```
    // 作成した頂点配列オブジェクトと頂点数を返す
    return { vao: vao, count: Math.floor(position.length / 2) }
}
```

(4) 図形を描画する

ディスプレイに図形を描画するには、使用するシェーダ（プログラムオブジェクト）を指定し、描画する図形（頂点配列オブジェクト）を指定して、描画命令を実行します。このうち、図形を指定して描画命令を実行す

る手順を関数にまとめます。ここでは関数名を draw() にしています。これも createObject() の後ろに続けて定義します。

この仮引数の object は、先ほど定義した createObject() の戻り値を受けとります。また、この関数の最初で描画する先の canvas 要素を消去（gl.clearColor() で指定した背景色で塗りつぶし）します。

```
//
// 描画する
//
function draw(object) {

  //カンバス内の表示を消去する
  gl.clear(gl.COLOR_BUFFER_BIT)
```

a) 図形の指定

描画する図形の頂点配列オブジェクトを指定します。仮引数 object のプロパティ vao には描画する図形の頂点配列オブジェクトが入っています。

```
  // 描画する頂点配列オブジェクト（図形）を指定する
  gl.bindVertexArray(object.vao)
```

b) 図形の描画

図形の描画には gl.drawArrays() を使います。このような命令は、ドローコール (Draw Call) と呼ばれます。ドローコールにより GPU が図形の描画処理を開始します。シェーダはこのときに実行されます。

gl.drawArrays() は頂点属性のみを使って図形を描画します。ここでは描画する基本図形に gl.LINE_STRIP を指定して、頂点バッファに格納されている 0 番目（先頭）の頂点属性から頂点数分を描画に使います。仮引数 object のプロパティ count には描画する頂点数が入っています。

```
  // 図形を描画する
  gl.drawArrays(gl.LINE_STRIP, 0, object.count)
```

void gl.drawArrays(mode, first, count)

頂点配列を用いて図形を描画します。

mode

描画する基本図形の種類。図 2.25 に示す基本図形があります。

first

描画する頂点の先頭の番号。頂点バッファの先頭の頂点から描画するなら 0 にします。

count

描画する頂点の数。4 角形なら 2 次元でも 3 次元でも 4 になります。

c) 図形の指定解除

描画する図形の指定を解除します。描く図形が他にあるなら、ここでその頂点配列オブジェクトを指定しても構いません。

```
    // 頂点配列オブジェクトの指定を解除する
    gl.bindVertexArray(null)
}
```

(5) 実行する

これまでに定義した関数を使って、canvas 要素に対して実際に描画を実行します。gl.clear(gl.COLOR_BUFFER_BIT) は draw() の中に入っているので、これを以降の**太字**の内容で書き換えます。

描画する図形の頂点属性は、図 2.26 に示す 2 次元平面上の 4 点の座標値です。OpenGL ES の内部では、これは $z = 0$、$w = 1$ の同次座標とし

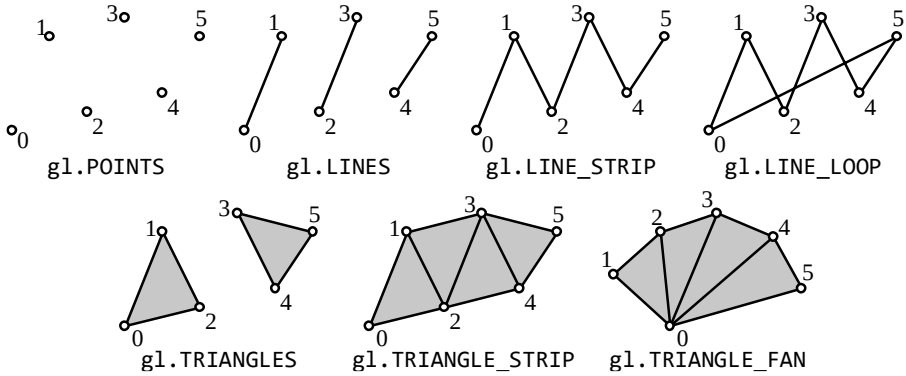


図 2.25 基本図形

て扱われます。これを配列変数 position に格納しています。これは図形を準備する関数 createObject() の引数に使います。

loadShader() の実引数 'vertex-shader' は、バーテックスシェーダのソースプログラムのスクリプトの id、'fragment-shader' はフラグメントシェーダのソースプログラムのスクリプトの id です。loadShader() は document.getElementById() を使って、それぞれの script 要素を取り出しています。

```
draw() で図形を描画する前に、gl.useProgram() を使って描画に使用するシェーダを指定します。同じシェーダを使って複数の図形を描画する場合は、描画終了のたびに引数に null を指定する必要はありません。  <canvas id="CG2" width="400" height="300" style="border: inset;">
    HTML5 の canvas が使えません
  </canvas>
  <script>
    const canvas = document.getElementById('CG2')
    const gl = canvas.getContext('webgl2')

    gl.viewport(0, 0, canvas.width, canvas.height)
    gl.clearColor(0.2, 0.3, 0.4, 1.0)

    // 座標データ
    const position = [
      -0.5, -0.5,
       0.5, -0.5,
       0.5,  0.5,
      -0.5,  0.5
    ]

    const shader = loadShader('vertex-shader', 'fragment-shader')
    const object = createObject(position)
    gl.useProgram(shader)
    draw(object)
    gl.useProgram(null)
  </script>
</body>
```

void gl.useProgram(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。null 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。

この結果は図 2.27 のようになります。

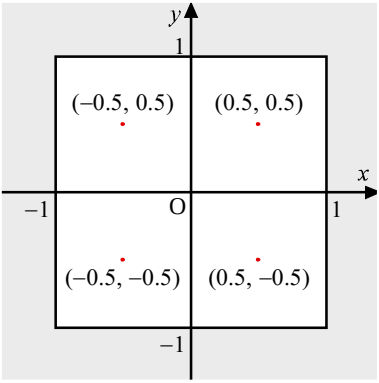


図 2.26 描画する頂点属性

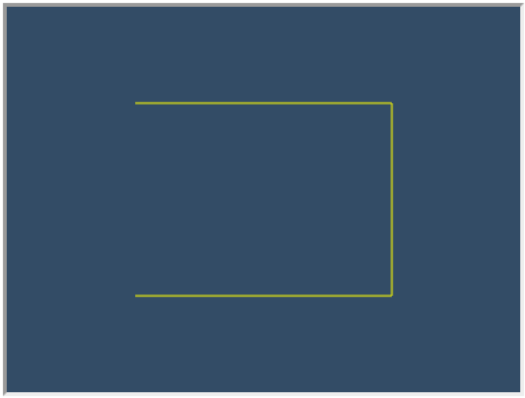


図 2.27 WebGL で描画した折れ線