WebGL 基礎 .01

WebGL の概要と 3D

Chapter.01

WebGL とはなにか

WebGL

WebGL は、JavaScript から OpenGL の機能を呼び出し、ブラウザ上でハイエンドなグラフィックス表現を行うことができる JavaScript の API である。

WebSocket や WebRTC などの、その他の次世代 API がそうであるように、WebGL はあくまでもブラウザの実装に依存する。このため、ブラウザごとに挙動が違ったり、そもそも動作しなかったり、といったことが起こる。

また、WebGL はデバイスの搭載している GPU によっても大きな影響を受ける。そういった意味では、他の API と比較して特に扱うことが難しいタイプの API であるとも言える。

WebGL の実行可能環境

WebGL は OpenGL ES と呼ばれるモバイル機器向けの軽量な OpenGL 実装を、ブラウザから制御できるようにしたものである。

つまり、WebGL が動作するためには、大前提として OpenGL ES 2.0 が動作するハードウェア要件が満たされている必要がある。

また、ブラウザも WebGL の API を備えていなければならないが、PC 用のブラウザでは主要ブラウザはほぼ全てが対応済み。モバイルでは iOS8 以降の iPhone や iPad に搭載された Safari、Android なら Chrome や Chromium ベースのブラウザで実行が可能。

WebGL の立ち位置

WebGL は OpenGL ES の API をブラウザから操作することができる、インターフェースのようなもの。つまり、WebGL を使いこなし完全に操るためには、必然的に OpenGL の知識が必要となる。

一見これは非常に大変なことのように感じられるが、実際には逆である。

C++ などを利用してプログラミングを行うことに比べると、ブラウザと JavaScript で実装できる WebGL はかなり簡単で気軽に始めることができる。3D プログラミングを始める上で、これほど障壁の少ない環境は珍しい。

Chapter.02

WebGL の開発環境

WebGL 開発を始めるには

他の言語での開発とは違い、通常の JavaScript を記述するための最低限の環境があれば、WebGL 開発を始めることができる。

強いて言うなら、3D 性能が高いマシンであるほど良いが、ラップトップなどのやや低いスペックのマシンでも、簡単なものなら十分に動作する。なにせ、モバイル端末でも動作するほどなのだから、あまり過度に心配する必要はない。

WebGL 開発を始めるには

また、開発や実行テストには、ローカルサーバ（あるいはそれを自力で起動できるスキルや環境）があると大変便利だが、これも必須ということではない。

特にこだわりがない、あるいは JavaScript での開発に慣れていないのであれば、NetBeans の利用を薦める。NetBeans はフリーで入手できる統合開発環境で、特に難しい設定をすることなく、簡単にローカルサーバを立てて動作を確認したりデバッグを行ったりすることが可能。

https://netbeans.org/downloads/?pagelang=ja

上記ページより「PHPバンドル版」をダウンロードすればよい

Chapter.03

グラフィックスパイプライン

画面にグラフィックスが表示されるまで

普段 PC やゲームで 3DCG を眺めているとき、大抵の人は、その映像がどのようにプログラムで生成され、どのような過程を経てモニターに映し出されるのか、そんなことは考えないだろう。

しかしハイエンドなグラフィックス開発を行うとなれば、その一連の流れを把握しておいて損はない。また、そういった知識を持っていることが、実装を行う際に理解をスムーズにすることも少なくない。

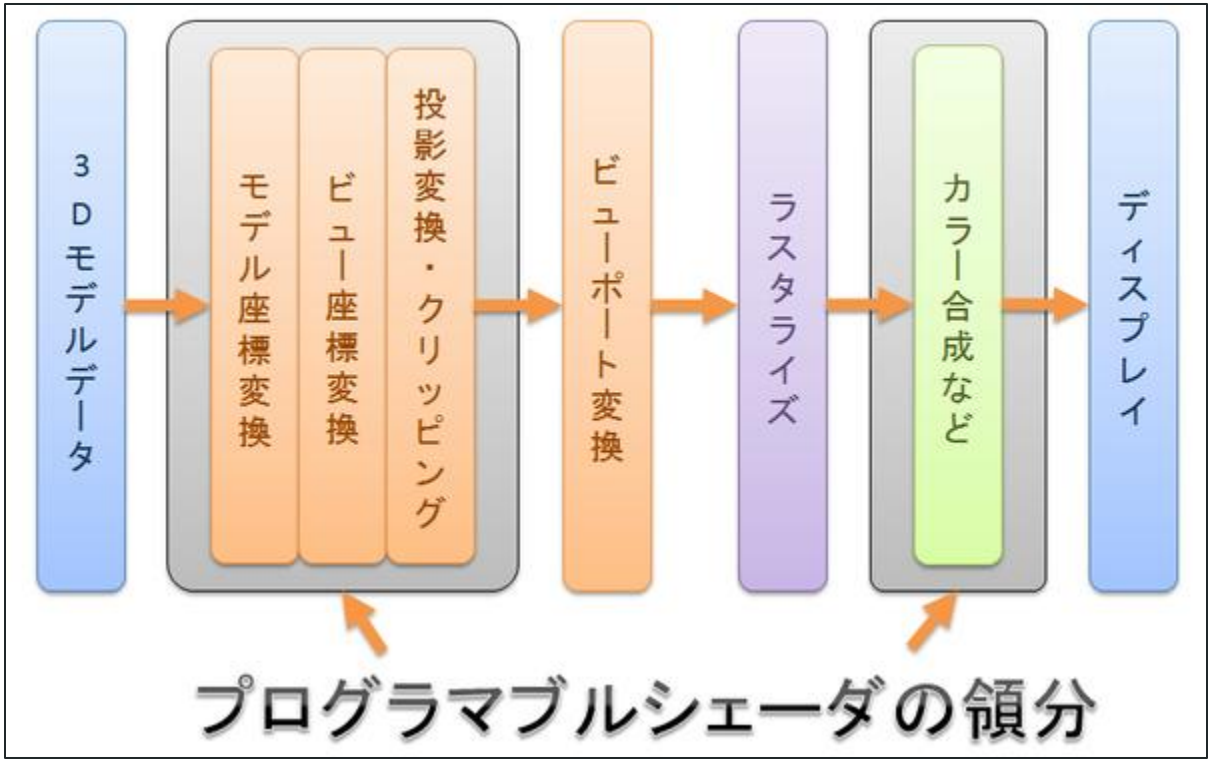
まずは、画面にグラフィックスが表示されるまでの大まかなプロセスを知ろう。

グラフィックスパイプライン

一般に、3D プログラミングではグラフィックスが画面に表示されるまでの一連の流れのことをグラフィックスパイプラインと呼ぶ。

このパイプラインに、C++ や JavaScript で書かれたアプリケーション・プログラムから様々なデータを流し込んでやると、パイプライン内で次々に処理が繰り返され、最終的にモニターに映像が出力される。

原則として、このグラフィックスパイプラインは GPU 上にあり、JavaScript の本来の機能だけでは、このようなハードウェア寄りの領域には干渉することはできない。WebGL の登場により、これが可能となった。



グラフィックスパイプラインの概念図（かなり大雑把だけど……）

Chapter.04

プログラマブルシェーダ

プログラマブルシェーダ

先ほどの概念図では、パイプラインの流れの一部に、プログラマブルシェーダの領分という箇所が、ふたつあった。

かつては、この部分は GPU が持つ固定的な機能が使われていたが、現在では、その部分をプログラマ側で柔軟に置き換える仕組みを用いるのが一般的になっている。

この、パイプライン上の一部の処理を置き換えるための仕組みが「プログラマブルシェーダ」である。

プログラマブルシェーダ

プログラマブルシェーダの仕組みを用いると、それまで固定機能による限定された表現しか許されなかった処理を、プログラマが自由に置き換えることができる。これにより、非常に高い表現力や、より柔軟性の高い制御が可能となった。

しかし、WebGL はこのプログラマブルシェーダを扱う都合上、どうしても JavaScript の他に、シェーダを記述するための専用の言語である「GLSL」を習得しなければならないため、学習コストが非常に大きくなりがちである。

GLSL（OpenGL Shading Language）

GLSL は、C 言語と基本的には似たような文法を持つ、シェーダを記述することに特化した言語。

OpenGL とその関連 API（OpenGL ES や WebGL）で利用されている。

シェーダを書くことだけに特化しているため、言語仕様としての難しい部分はあまりなく、覚えることもそれほど多くない。ただ、シェーダ自体が難しい概念を多く必要とするため、総じて難しい印象を持っている人が多い。（けど、基本的なことは意外と簡単）

GLSL の記述例

例えば、簡単な GLSL で記述されたシェーダのコードは次のようになる。

```
attribute vec3 position;
uniform   mat4 mvpMatrix;

void main(){
    gl_Position = mvpMatrix * vec4(position, 1.0);
}
```

GLSL の役割

おさらいすると、WebGL では、基本的な実装は JavaScript で行うものの、よりハードウェア（GPU）に近い部分の処理は、専用の言語で別途記述する必要があり、これが GLSL と呼ばれるもの。

GLSL を同時に習得するのは大変でもあるが、基本的な文法を覚えるだけならそれほど難しくはない。

※ むしろ大変なのは GLSL そのものというよりも、3D 数学などの普段はあまり利用しないような知識だったりする。

Chapter.05

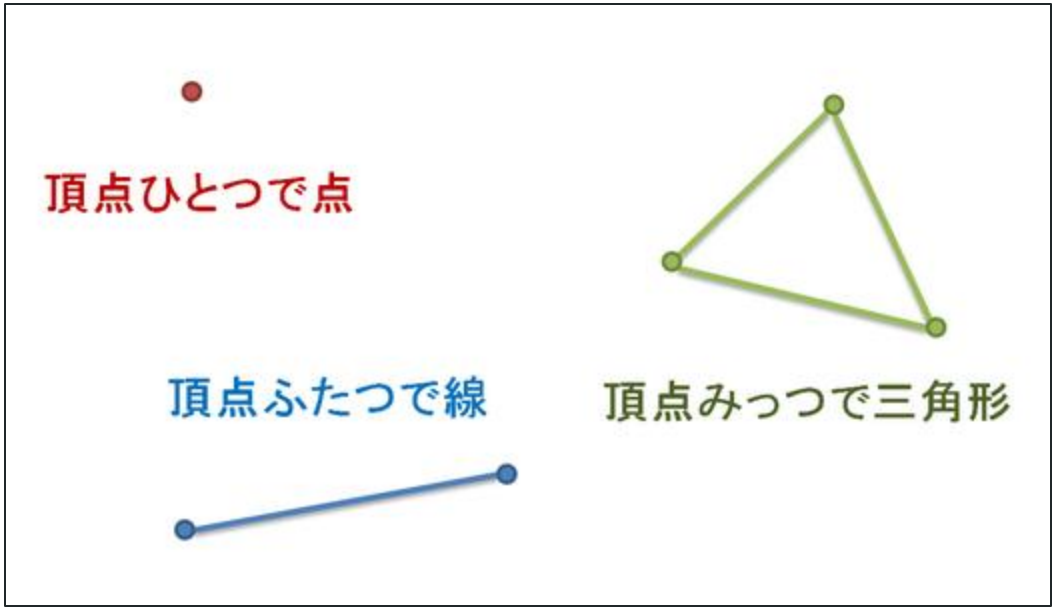
頂点

頂点がすべてのはじまり

3D プログラミングでは、Canvas2D とは違い、丸や四角形を命令ひとつで簡単に描画するということはできない。

その代わり「頂点」と呼ばれる点を繋いでいき、任意の形状を自由に表現できるようになっている。

WebGL で画面になにかを描き出したければ、最低でも頂点がひとつはなければ、なにも描画できない。頂点がふたつあれば線を表現することができるし、3つ以上あれば三角形のポリゴンを描画することもできる。



頂点によって形成される形状

頂点属性

頂点には、それぞれの点ひとつひとつに異なる情報を持たせることができる。

この頂点に持たせる情報のことを「頂点属性」と呼び、どのような頂点であっても最低限「座標位置」だけは絶対に持たせてやる必要がある。

座標がわからなければ、どこに点を置けばいいのかがわからないからである。

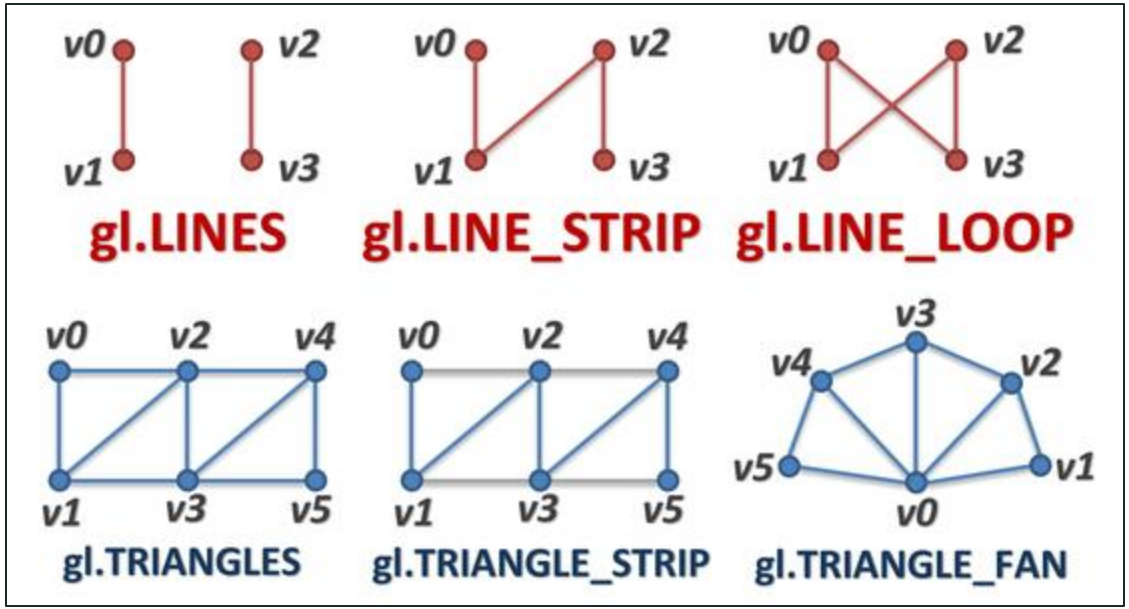
代表的な頂点属性には、他に頂点の色などがある。頂点属性はプログラマが自由に頂点に付加できるものなので、頂点に個別に持たせたい情報があれば、頂点属性を追加すればよい。

頂点のプリミティブ

頂点がひとつなら点、ふたつで線、3つあれば三角形が表現できる。

実際には、もっと細かく頂点の結び方は決められており、この結び方のルールのことを「頂点プリミティブ」などと呼ぶことが多い。

WebGL で利用できるプリミティブの形状には、最大でも三角形を表現するものしか存在しないが、本家 OpenGL には四角形や多角形を表現するプリミティブ形状もある。



WebGL で利用可能なプリミティブ形状

Chapter.06

頂点バッファ

頂点バッファ

頂点に関する情報は、JavaScript から直接 GPU（シェーダ）に流し込むことはできず、一度専用のデータの入れ物に格納してやる必要がある。

このデータの入れ物のことを「頂点バッファ」と呼ぶ。

頂点バッファは英語表記では Vertex Buffer Object となり、頭文字を取って、アルファベットの略語「VBO」と呼ばれることも多い。

バッファのバインド

WebGL では、バッファと呼ばれるオブジェクトがいくつも登場する。それらに共通するのは、なにかしらのデータの入れ物である、ということ。

WebGL はこういったデータの入れ物を、同じ種類のものは、同時にひとつしか手に持つことができない上に、今現在手に持っているものに対してしか処理を行うことができない。

このような、バッファなどを「これから使う入れ物として捕捉する（手に持つ）こと」を、「バインドする」というふうに表現する。WebGL では様々なシーンで、オブジェクトのバインドが登場するので覚えておくこと。

VBO の生成

VBO を生成する際、これの元となるのは JavaScript の純粋の配列であるが、その行程をよく見てみると TypedArray が使われているのがわかる。

WebGL では原則として、まず JavaScript で配列を作り、それを TypedArray に変換したものを、WebGL の持つメソッドによって VBO へと変換する。

VBO は頂点の持つ情報をシェーダへと渡すためのものなので、頂点属性が増えた場合には VBO を正しく追加し、シェーダへと送ってやる必要がある。違う言い方をすれば、VBO を生成しておきバインドすることによって、初めてそのデータを利用した描画が行える、ということである。

VBO の生成

VBO を生成する関数を見ると、型付き配列が使われているのがわかる。

```
var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(data), gl.STATIC_DRAW);
```

第一引数は、これが VBO に対する処理であることを表す ARRAY_BUFFER を、第二引数に TypedArray に変換した配列データを渡す。

第三引数はこのバッファを更新する頻度などを指定するが、基本的に VBO は、生成後に中身を変化させることが少なく、特に理由がなければ STATIC_DRAW を指定すればよい。

シンプルな WebGL 実装サンプル

サンプル.w01 は最もシンプルな、WebGL 実装のサンプルになっている。

ここでは全体的な流れを見つつ、修正を加えるなどして実行結果にどのような変化が起こるのかを確認する。

やってみよう

頂点定義を行っている箇所を JavaScript のソースコードから見つけ出して、頂点の座標を手動で変更してみよう。

頂点属性についての理解

サンプル.w02 では、頂点属性が追加されている。

頂点属性（attribute）について思い出しながら、GLSL と JavaScript でどのように連携を取ればいいのかをしっかり把握する。

やってみよう

頂点色を変更してみたり、頂点色を複数持たせ、シェーダ内で合成したりしてみよう。

Chapter.07

プログラムオブジェクトと

シェーダオブジェクト

シェーダオブジェクト

シェーダオブジェクトは、GLSL で書かれたソースコードが割り当てられるオブジェクト。

WebGL にはシェーダオブジェクトを生成するメソッドや、ソースコードをコンパイルするメソッドなどがあり、もしもソースコードに文法間違いなどのミスや間違いがあれば、コンパイルが失敗する。

コンパイルが失敗してもエラーなどが発生するわけではなく、JavaScript の処理は何事もなかったかのように進んでいくため、エラーが起こっていないかチェックする方法が用意されている。

シェーダコンパイルとチェック

シェーダオブジェクトを生成し、ソースコードを割当て、コンパイルし、結果をチェックする、というような手順で処理するのが一般的。

```
var shader = gl.createShader(gl.VERTEX_SHADER);     // 頂点シェーダの場合
gl.shaderSource(shader, source);                     // ソースコード割当て
gl.compileShader(shader);                            // シェーダのコンパイル
gl.getShaderParameter(shader, gl.COMPILE_STATUS);    // 結果のチェック
```

最後のチェックの部分は真偽値を返してくるので、コンパイルが成功したかどうかわかる。
失敗していた場合は、なぜ失敗したのかを調べることもでき、その場合以下を使う。

```
gl.getShaderInfoLog(shader);
```

プログラムオブジェクト

プログラムオブジェクトはシェーダを取りまとめる司令塔となる特殊なオブジェクトで、JavaScript とシェーダとのやりとりも、このプログラムオブジェクトがハブとなって行ってくれる。

先にシェーダオブジェクトを、頂点シェーダとフラグメントシェーダそれぞれにコンパイルしておき、これをプログラムオブジェクトにアタッチ（紐付け）する。すると、以降はプログラムオブジェクトがハブになってやりとりを一元管理してくれるので、プログラムオブジェクトに対して命令をするだけでシェーダに対する処理をうまく行ってくれる。

プログラムオブジェクトの生成とアタッチ

プログラムオブジェクトを生成してシェーダをアタッチ、そしてリンクするという一連の作業が必要。

```
var prg = gl.createProgram();           // プログラムオブジェクトの生成
gl.attachShader(prg, vertexShader);     // 頂点シェーダをアタッチ
gl.attachShader(prg, fragmentShader); // フラグメントシェーダをアタッチ
gl.linkProgram(prg);                    // シェーダをリンク！
```

シェーダオブジェクトのコンパイルのときと同じように、正しくリンクが行われたかチェックしたり、失敗した場合にはその理由を調べたりすることができる。

```
gl.getProgramParameter(prg, gl.LINK_STATUS);
gl.getProgramInfoLog(prg);
```

プログラムオブジェクト単位で選択する

WebGL では、一度に複数のシェーダを使った実装を行うことは可能だが、描画命令を発行するタイミングで、同時に複数のシェーダを一緒に動かす、ということはできない。

利用するシェーダは、常に一組しか指定できないようになっているわけである。

どのシェーダを利用するのかは自由に決めることができ、基本的にいつでも切り替えることは可能だが、このとき、シェーダオブジェクトを直接操作することはなく、アタッチしてあるプログラムオブジェクトを選択すれば、自動的にリンク済みのシェーダのセットが使われる。

コンパイル失敗時の挙動

サンプル.w02 を使って、あえてコンパイルを失敗させてみるとどうなるか、確認してみる。

またプログラムオブジェクトがシェーダのリンクに失敗するというのはどういうことか、についても理解しておくとよい。

やってみよう

わざと GLSL の記述に間違った部分を入れて、コンパイルを失敗させてみよう。

またその場合のエラーメッセージがどのような内容になっているのかについても理解しておこう。

Chapter.08

シェーダとのやり取り

シェーダとのやり取り

JavaScript で定義したデータをシェーダへと送る場合は、先述のとおりプログラムオブジェクトがハブの役割を担ってくれる。

しかし実際には、目的ごとにメソッドが用意されているので、そのメソッドの引数に、該当するプログラムオブジェクトを指定するという方法を用いることが多い。

JavaScript からシェーダに送る情報には、attribute と uniform の大別してふたつの種類があるが、それぞれデータの送り方は全く違う方法を用いるため、混乱しないように注意。

頂点データ（attribute）

頂点のデータをシェーダへと送るためには、VBO を用意しておくこと、さらにシェーダ内のポインタを事前に取得しておくこと、のふたつが必要。

```
// attribute 変数の名前を指定してポインタ（ロケーションと呼ぶ）を取得
// コードを見るとわかるとおり、事前にプログラムオブジェクトの処理まで完了している前提
var loc = gl.getAttribLocation(prg, 'position');

// ロケーションの有効化
gl.enableVertexAttribArray(loc);

// ロケーションを指定してデータを登録
gl.vertexAttribPointer(loc, str, gl.FLOAT, false, 0, 0);
```

uniform データの送信

VBO（頂点の情報）は、バインドされているバッファの情報がシェーダへと送られる仕組みになっているが、uniform の場合は、やり方が少々異なる。

ただし、シェーダ内のポインタ（ロケーション情報）を取得しておかなくてはならない点は attribute の場合と同じである。

```
// ロケーション情報を事前に取得しておく
var uloc = gl.getUniformLocation(prg, 'mvpMatrix');
```

汎用データ（uniform）

uniform 変数のデータは、単なる汎用グローバル変数としてシェーダ内で使われるため、事前にバッファを生成する頂点データとは違い、値を直接シェーダへ送る。ただし、送るためのメソッドが複数あり紛らわしいので注意。

```
// vec2 なら、浮動小数点を配列で送信するので …… 「2fv」
gl.uniform2fv(uloc, [0.0, 0.0]);
// vec4 なら、浮動小数点を配列で送信するので …… 「4fv」
gl.uniform4fv(uloc, [0.0, 0.0, 0.0, 0.0]);
// float なら、浮動小数点を単体で送信するので …… 「1f」
gl.uniform1f(uloc, 0.0);
// 行列なら名前には matrix というキーワードを使い、引数が３つになる
gl.uniformMatrix4fv(uloc, false, matrix);
```

Chapter.09

描画命令

描画命令

これまで見てきた様々な手順は、WebGL で描画を行うための、あくまでも「事前準備」である。事前にバッファを用意したり、それをシェーダに送ったりしていただけで、まだ画面には何も映しだされてはいない。

実際に画面に何かが描画されるのは、描画命令が発行され、それが実行されたあとなる。

スクリーンのクリア

Canvas2D の clearRect のように、描画領域をリセットするためのメソッドがある。先に色などを指定しておき、クリアする。

また、WebGL では色が書き込まれるカラーバッファの他に、特殊なバッファとして深度バッファと呼ばれるバッファが存在するので、サンプルでは一緒にこちらもクリアするようにしている。（詳細はいずれ）

```
gl.clearColor(0.0, 0.0, 0.0, 1.0); // クリアする色を不透明な黒に指定
gl.clearDepth(1.0);                // クリアする深度を 1.0 に指定

// あらかじめ指定してあった条件で描画領域と深度バッファをクリアする
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

ビューポートの設定

描画対象となる領域を設定するために、ビューポートの設定も行っておく。

ビューポート設定を行わなくても、もともとの Canvas エレメントの大きさが自動的に初期値のビューポートとして設定されるが、ただしく設定する癖をつけたほうがよい。

```
gl.viewport(0, 0, width, height);

// ビューポートには、左下を原点として、矩形の始点と幅、高さ、の４つを指定する
```

描画命令

いよいよ描画命令を発行する。

描画命令にはいくつか種類があるが、まずは下記に記したように drawArrays を利用して頂点を描画する。

```
gl.drawArrays(gl.TRIANGLES, 0, 3);

// 第一引数は利用する頂点プリミティブを指定する
// 第二引数は頂点データの何番目から描画するか、第三引数がいくつの頂点を描画するか
```

仕上げ処理

描画命令は、原則何度でも繰り返し呼び出すことができる。むしろ、複数の頂点データを描画するには、繰り返しバインドする頂点データを変更したりしながら、複数回の描画命令を呼び出す。

全ての描画命令の呼び出しが完了したら、画面を更新する最後の仕上げとして、次のように flush を呼ぶ。

```
gl.flush();

// gl.flush が実行されると確実に全ての描画命令が実行される
```

Chapter.10

インデックスバッファ

インデックスバッファ

これは余談なのであまり深く考えなくてもよいが、VBO 以外に、バッファと名の付くオブジェクトの代表的なものとして IBO、Index Buffer Object がある。

その名の通り、インデックスを専門に格納するためのバッファで、このバッファは頂点をどのように結んで描画を行うのか、というインデックス情報を扱うために使われる。

IBO の生成とデータの格納

インデックスデータは整数で、頂点の結んでいく順番を指定する。また、バッファをバインドする際に、gl.ELEMENT_ARRAY_BUFFER を指定する。

```
var index = [0, 1, 2, 2, 1, 3];

var ibo = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, ibo);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Int16Array(index), gl.STATIC_DRAW);

// バッファの種類のほか、bufferData の第二引数に与える配列データも、
// Float32 ではなく Int16 になっている点が VBO とは異なる
```

描画命令

インデックスバッファを使った描画では、描画命令も変更する必要がある。

頂点を直接描画する drawArrays ではなく、gl.drawElements を使わなくてはならない。

```
gl.drawElements(gl.TRIANGLES, index.length, gl.UNSIGNED_SHORT, 0);
```

付属サンプル

- w01（最も基本）
- w02（頂点色あり版）
- w03（インデックスバッファ版）