情報可視化論 (2018 年)

Part 1: WebGL 1.0 入門

陰山 聡

神戸大学 システム情報学研究科 計算科学専攻

2018.04.10

まずは環境設定

- WebGL 環境オン
  - Safari 立ち上げ
  - 環境設定
  - → メニューバーに開発メニューを表示 にチェック
  - （開発メニューから “WebGL を有効”）
- WebGL 環境の確認
  - この講義のウェブページ
  - `http://tinyurl.com/kageyama2018v`
  - サンプルをクリック
- Safari でのソースコード表示方法：
  - 開発メニュー
  - → ページのソースを表示

WebGL 1.0 とは

WebBL = シェーダを使い、HTML5 の canvas に、JavaScript で 3D CG を書くための API

- スタンドアロンアプリからウェブアプリへの流れ
- クロスプラットフォーム
- オープンスタンダード
- Web で GPU を使ったレンダリングが可能
- 開発・利用が容易： プラグイン不要
- ソースコードが見える
- グラフィックス（OpenGL）と UI（ウィンドウ管理やイベント処理）の分離が明白

ガイダンス

講義計画

- Part 1
  - 担当：陰山（最初の二回）
  - WebGL 1.0 入門
  - 参考文献: “Professional WebGL Programming” by A. Anyuru
    『実践プログラミング WebGL』 吉川郁夫訳、翔泳社
- Part 2
  - 担当：坂本（三回目以降）

古い（3.0 より前の）OpenGL を知っている学生への注意

独学、実験・プロジェクト演習等で経験あり？
glBegin/glEnd, glVertex, glNormal... ⇒ もうなくなった！

ただし、CG（Computer Graphics）の基礎

- 正射影、透視射影
- OpenGL の照明モデル
- ...

は変わらず。

必要な数学はこの資料の Appendix (p. 394) にまとめた。適宜参照すること。

目標

- コンピュータグラッフィックス（CG）の基礎の習得
- 可視化の基本アルゴリズムの習得

WebGL 対応ブラウザ

- PC
  - Firefox
  - Google Chrome （9 からはデフォルトで ON）
  - Safari （8 からはデフォルトで ON）
  - Opera （15 からはデフォルトで ON）
  - Internet Explorer 11
- モバイル
  - Android ブラウザ
  - Firefox for Mobile
  - Internet Explorer Mobile
  - Opera Mobile
  - Safari
  - Tizen

ブラウザとバージョンによっては設定が必要かもしれない。

OpenGL の歴史

OpenGL とは

OpenGL 2.X は、OpenGL 1.X の機能を全て含む。

OpenGL 3.0 で上位互換性放棄

シェーダで置き換え可能な機能
OpenGL 3.0 非推奨機能
→ OpenGL 3.1 拡張機能
→ OpenGL 3.2 互換プロファイル

OpenGL Overview

`http://www.opengl.org`

OpenGL was first created as an open and reproducable alternative to Iris GL ...Silicon Graphics workstations... OpenGL 1.0 ...non-SGI 3rd party...

...

OpenGL 2.0...OpenGL Shading Language (also called GLSL), a C like language with which the transformation and fragment shading stages of the pipeline can be programmed.

OpenGL 3.0 adds the concept of deprecation (非推奨)... GL 3.1 removed most deprecated features, and GL 3.2 created the notion of core and compatibility OpenGL contexts.

Official versions of OpenGL released to date are 1.0, 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 2.0, 2.1, 3.0, 3.1, 3.2, 3.3, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5

OpenGL 関係

`http://www.khronos.org`

- OpenGL 4.5
- OpenGL SL 4.5 （シェーディング言語）

- OpenGL ES 3.1（組み込みシステム用） iOS, Android, Symbian

- WebGL 1.0 （≈ OpenGL ES 2.0）
- WebGL 2.0 （≈ OpenGL ES 3.0）← この講義には含まれない。

Shader-based OpenGL による OpenGL 教育

- OpenGL はグラフィックスの API として長年教育現場で教えられてきた。
- 仕様の変更も比較的少なかった。
- ところが OpenGL 3.0 から、OpenGL は大きく変化（Shader-base 化）。
- それに伴って教育内容も大改訂が必要となった。

OpenGL ver. 3.1 からの大きな変化

全てのアプリケーションは少なくとも一つの頂点シェーダと、少なくとも一つのフラグメントシェーダを与える必要がある。

OpenGL ver. 3.1 からの大きな変化

```
//
// Hello_World.c
//
// To compile on Mac.
// gcc this_source.c -framework GLUT -framework OpenGL
//

#include <GLUT/glut.h>

void display (void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_POLYGON);
   glVertex2f(-0.5, -0.5);
   glVertex2f(-0.5,  0.5);
```

```
    glVertex2f( 0.5,  0.5);
    glVertex2f( 0.5, -0.5);
    glEnd();
    glutSwapBuffers();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutCreateWindow("Hello World");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

Listing 1: 古い OpenGL コード

シェーダとシェーディング言語： GLSL

はじめに

OpenGL シェーディング言語（OpenGL SL, GLSL）4.0

GPU を使うための言語

CG ソースコード	＝	OpenGL ソースコード
	＋	GLSL(フラグメントシェーダ）ソースコード
	＋	GLSL(頂点シェーダ）ソースコード

OpenGL シェーディング言語

- GPU 内部で走る単体のプログラム
- 専用言語
- シェーディングプログラム（あるいは単にシェーダ）と呼ばれる
- 要するに数値演算プログラムの一種
- シェーディングに限らず様々な数値処理が可能（→ GPGPU）
- 並列処理

グラフィックス

- immediate-mode
  - 即時モード API
  - 描画のたびにシーン全体を GPU に送る。たとえ変更がなくても。
- retained-mode
  - 保持モード API
  - シーン全体をまず GPU に送る。その後は変更部分だけを送る。

WebGL は即時モード API

OpenGL プロファイル

OpenGL 3.0

deprecation（非推奨）を導入 → 不要な関数を段階的に取り除く

OpenGL 3.2 より

コア プロファイルと互換プロファイル

プロファイルはウィンドウシステム API で選択

glBegin/glEnd は非推奨

標準的な行列 GL_MODELVIEW, GL_PROJECTION もコアプロファイルから削除。

OpenGL ES 2.0

OpenGL for Embedded Systems （組み込みシステム用）

Immediate モード API

glBegin/glEnd なし。頂点配列を使う。

シェーダベース。シェーダプログラムが必要。

アプリケーションプログラム： C/C++, Objective-C, Java

OpenGL ES 2.0 ≈ WebGL

WebGL は JavaScript で呼ぶ。

HTML5 canvas

HTML5： 第 5 世代の HTML

canvas: JavaScript で描画できる長方形領域

→ 【HTML5 canvas サンプルプログラム】

コンピュータグラフィックス（CG）

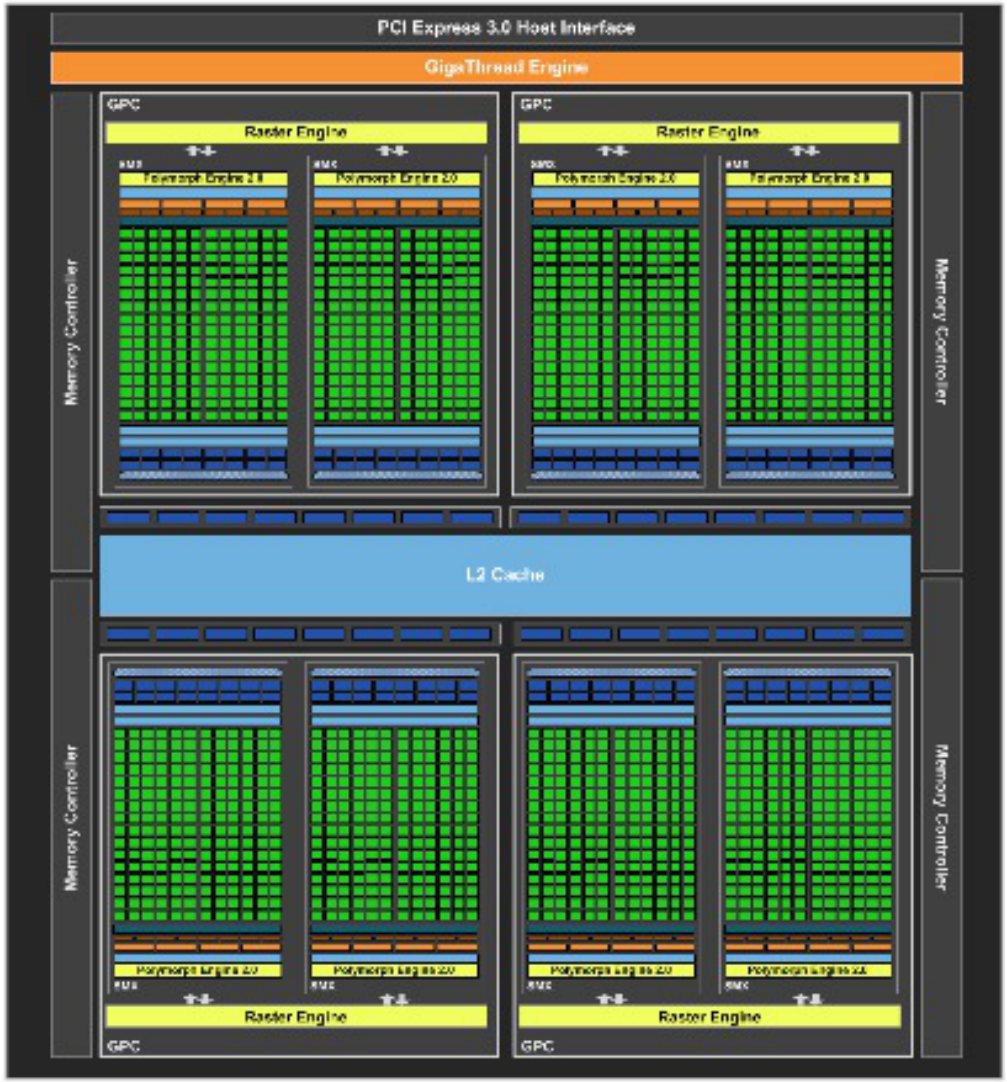
CG と並列計算

- 並列計算については学んできたはず。
- 空間を分割して、通信用のメモリを確保して、MPI で通信して ···
- 並列プログラム作りは結構大変。← 結局同期が大変だから。
- 一方、同期について気にしなくてもよい問題もたくさんある。
- もともとのアルゴリズムが並列化可能なもの。→ “Embarrassingly” parallel problems

- CG の処理の多くが embarrassingly parallel
- 座標変換、テクスチャマップ、シェーディング、ラスタ化、etc.
- CG 専用並列計算機 → GPU (Graphics Processing Unit)

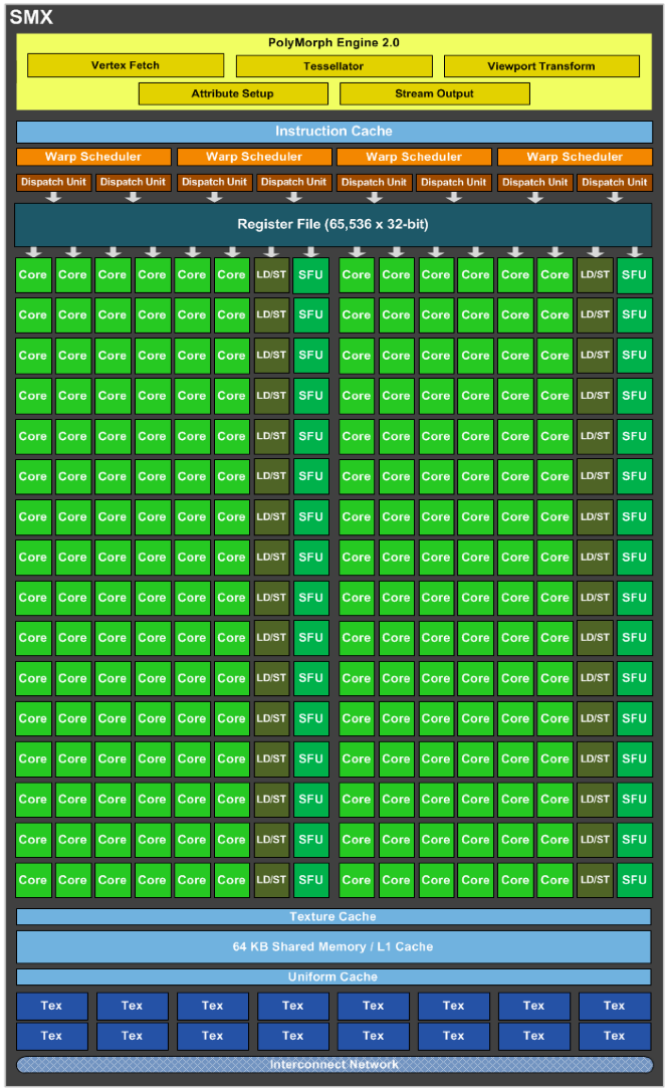
Kepler GeForce の構造

Nvidia white paper より引用



Kepler GeForce の構造

Nvidia white paper より引用

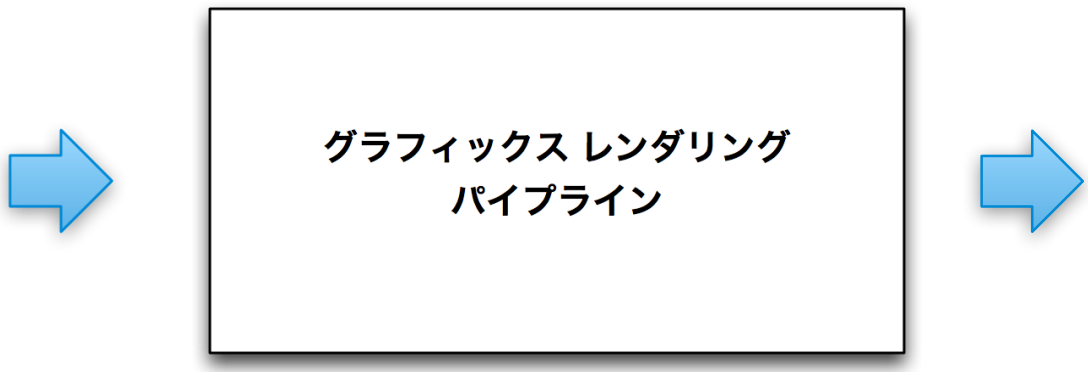


GPU のプロセッサ数

- NVIDIA TITAN Xp
- シェーダプロセッサ “CUDA Cores” 3840 個

グラフィックス レンダリング パイプライン

Graphics rendering pipeline 「パイプライン」

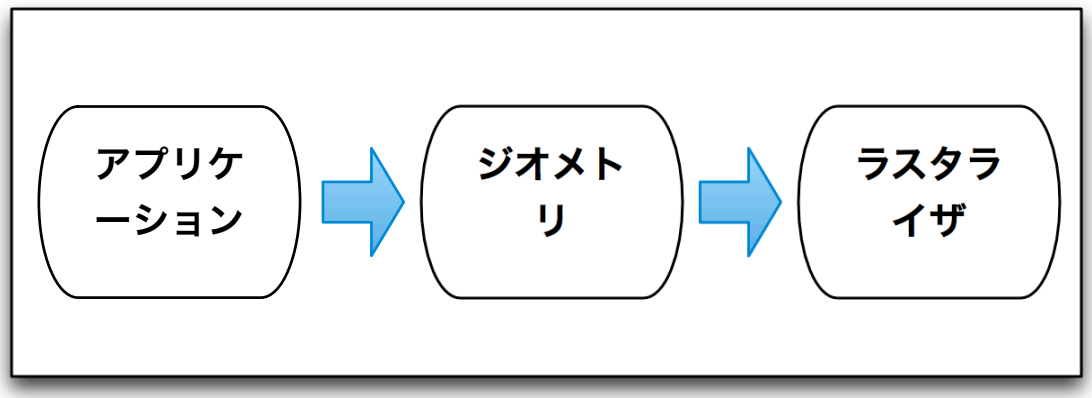


input: カメラ設定、3 次元オブジェクト、光源、ライティングモデル、テクスチャ

output: 2 次元画像

パイプラインの構造

アプリケーション ステージ → ジオメトリ ステージ→ ラスタライザ ステージ



アプリケーション ステージ

ソフトウェア

オブジェクトの生成、アニメーション、衝突検出、カリング、等々

ジオメトリ ステージ

ハードウェア

ビューボリューム

canonical view volume （標準ビューボリューム）

辺の長さ 2 の立方体

$$(-1,-1,-1) \leq (x,y,z) \leq (1,1,1)$$

標準ビューボリューム内の座標を正規化デバイス座標 （Normalized Device Coordinates, NDC）と呼ぶ。

ジオメトリステージの役割は、アプリケーションが設定した CG 世界をビューボリュームにマップすること。

射影

2 種類 :

・正射影 orthographic projection, 平行投影（parallel projection）

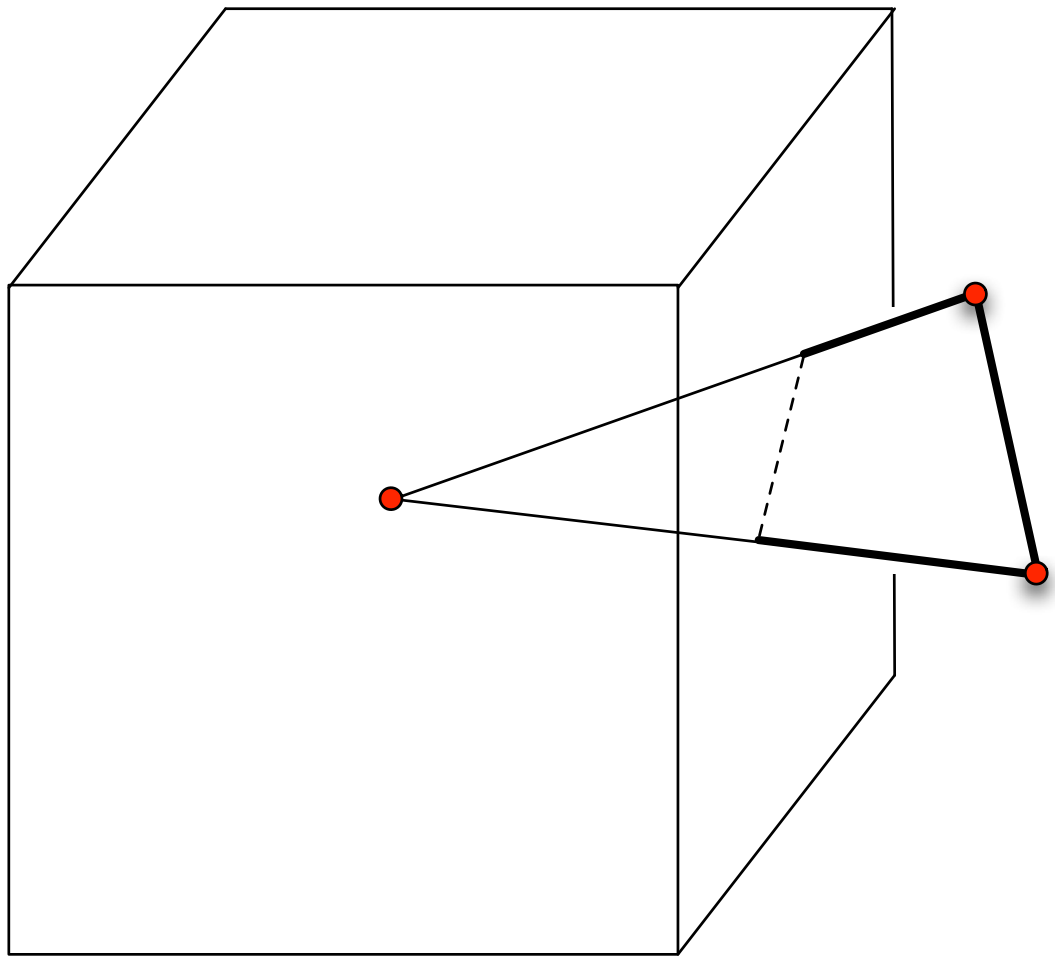
・透視射影 perspective projection, 遠近投影

視錐台 view frustum

どちらの射影も 4 行 4 列の行列演算で書ける（後述）。

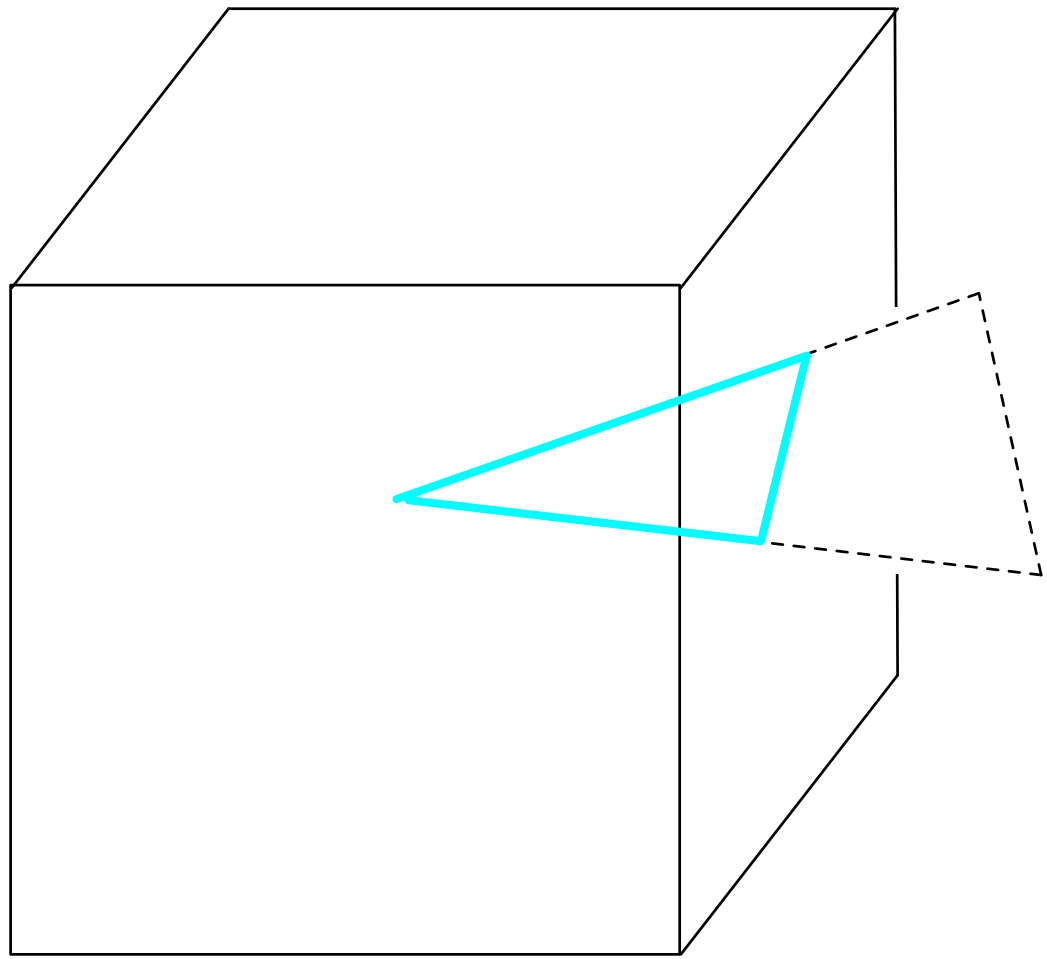
クリップ処理

ビューボリュームの外部は描かない。



クリップ処理

ビューボリュームの表面に新しい辺が自動的にできる。

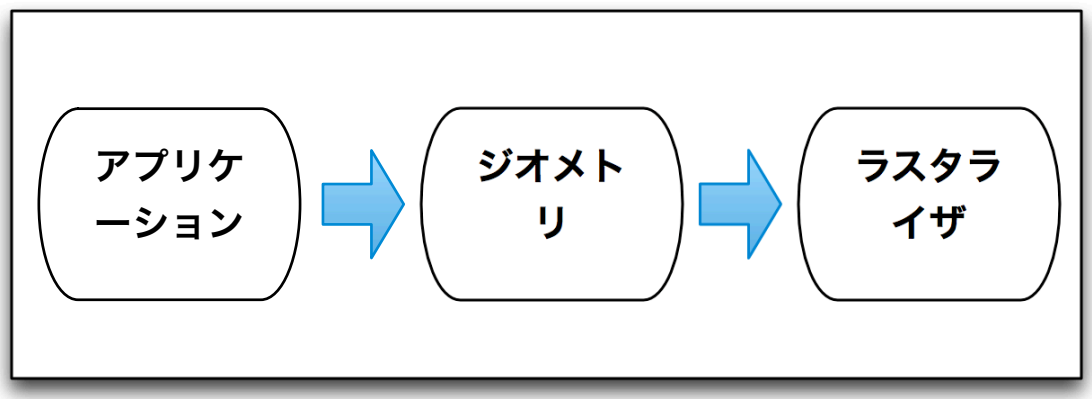


スクリーンマッピング

最終的な画像の大きさ（width と height）に合わせて、立方体のビューボリュームを x 方向と y 方向にスケール変換＋平行移動させる。z 方向には何もしない（$-1 \leq z \leq 1$）

パイプラインの構造

再掲

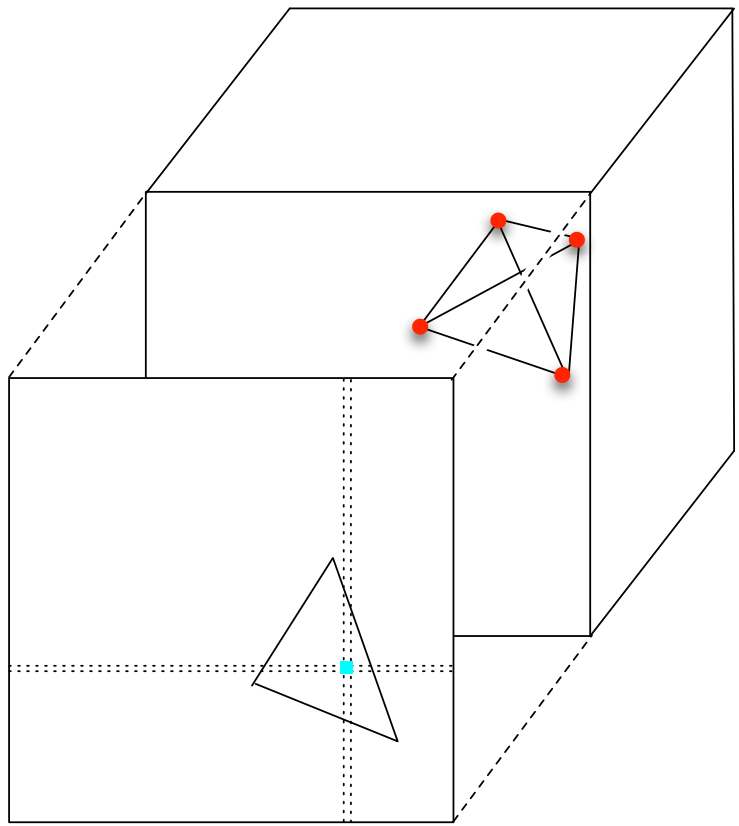


ラスタライザステージ

ビューボリューム内のデータ（3D）＋テクスチャデータ（2D） → 画像（2D）

頂点のデータ → ピクセルの色

簡単のため、スクリーンマッピングを省略すると



ラスタライザステージ

- デプステスト。後述。
- アルファテスト
- ステンシルテスト
- テクスチャマッピング

フレームバッファ

- カラーバッファ
- Z バッファ
- ステンシルバッファ

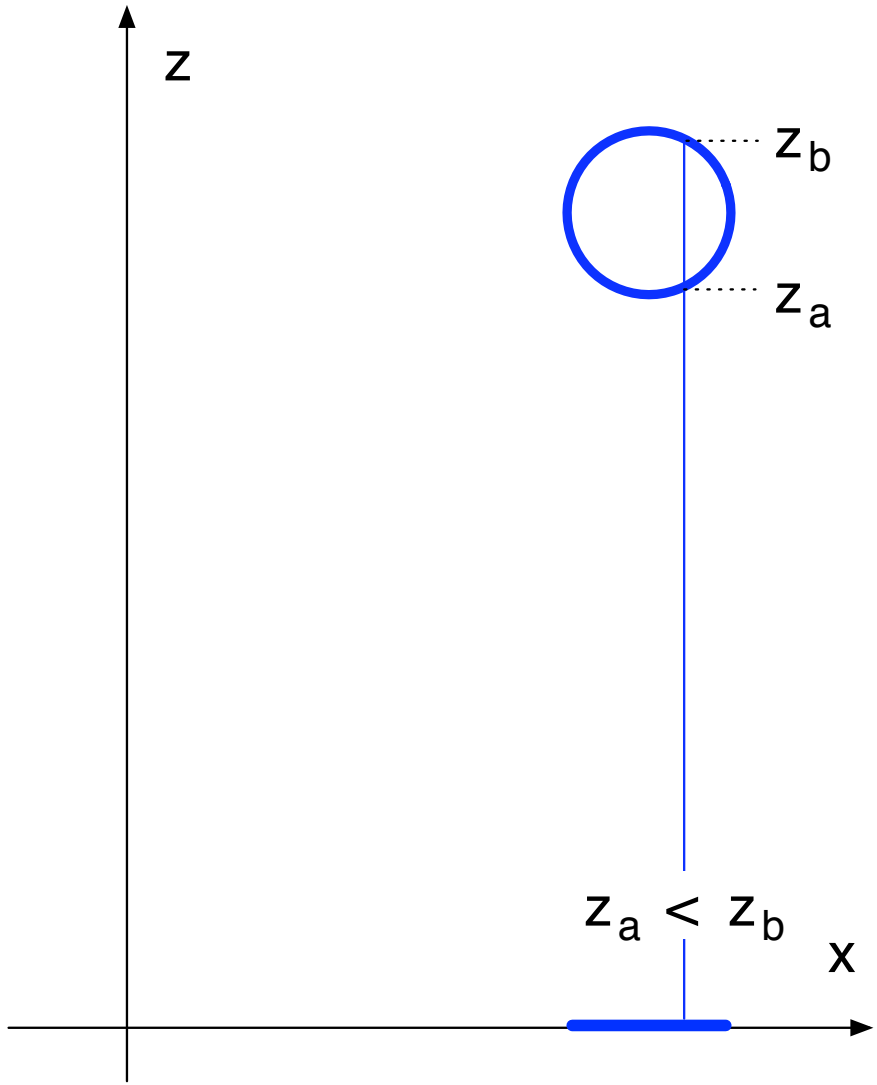
カラーバッファ

- ピクセル RGBA
- 16 bits, 23 bits, 32 bits
- RGBA32 = R8 + G8 + B8 + A8

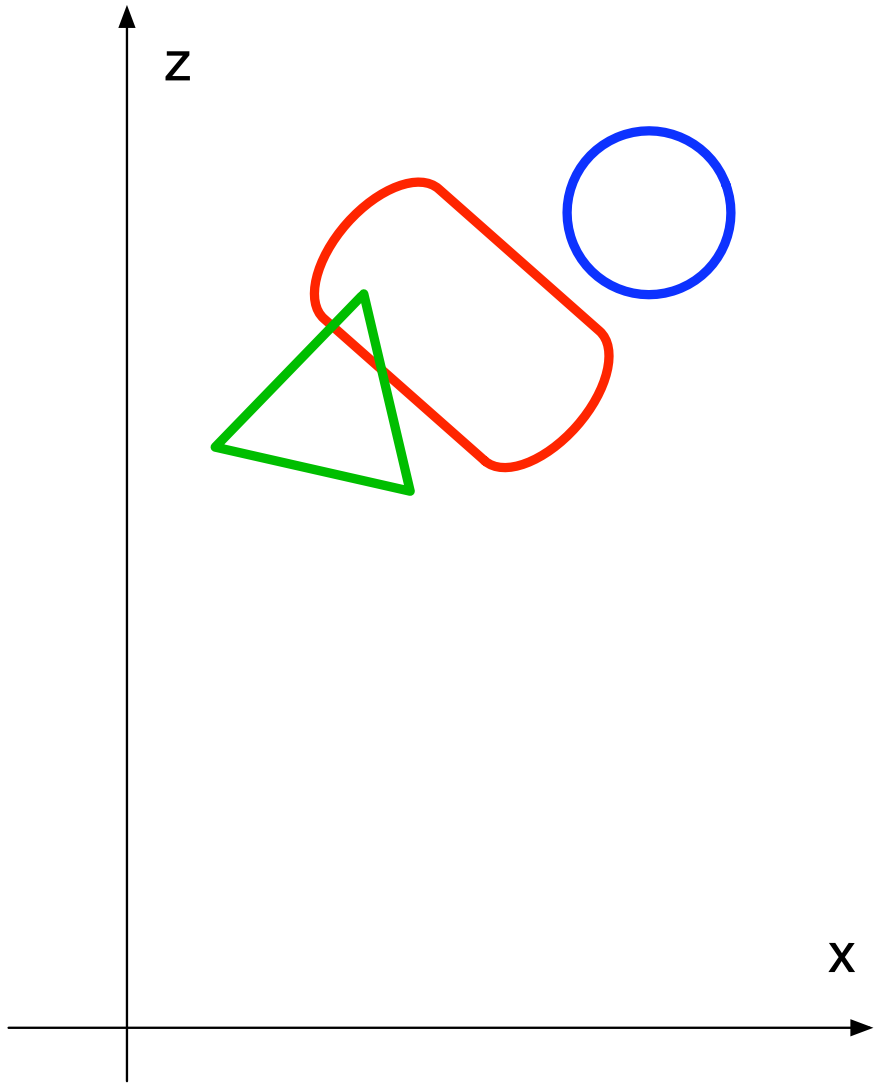
Z バッファ

- Z-buffer
- デプステスト
- Edwin Catmull が考案。
- 博士論文（1974 年）。
- RenderMan 開発（アカデミー賞）
- ルーカスフィルム → ピクサー設立。
- ウォルト・ディズニー・アニメーション・スタジオ社長

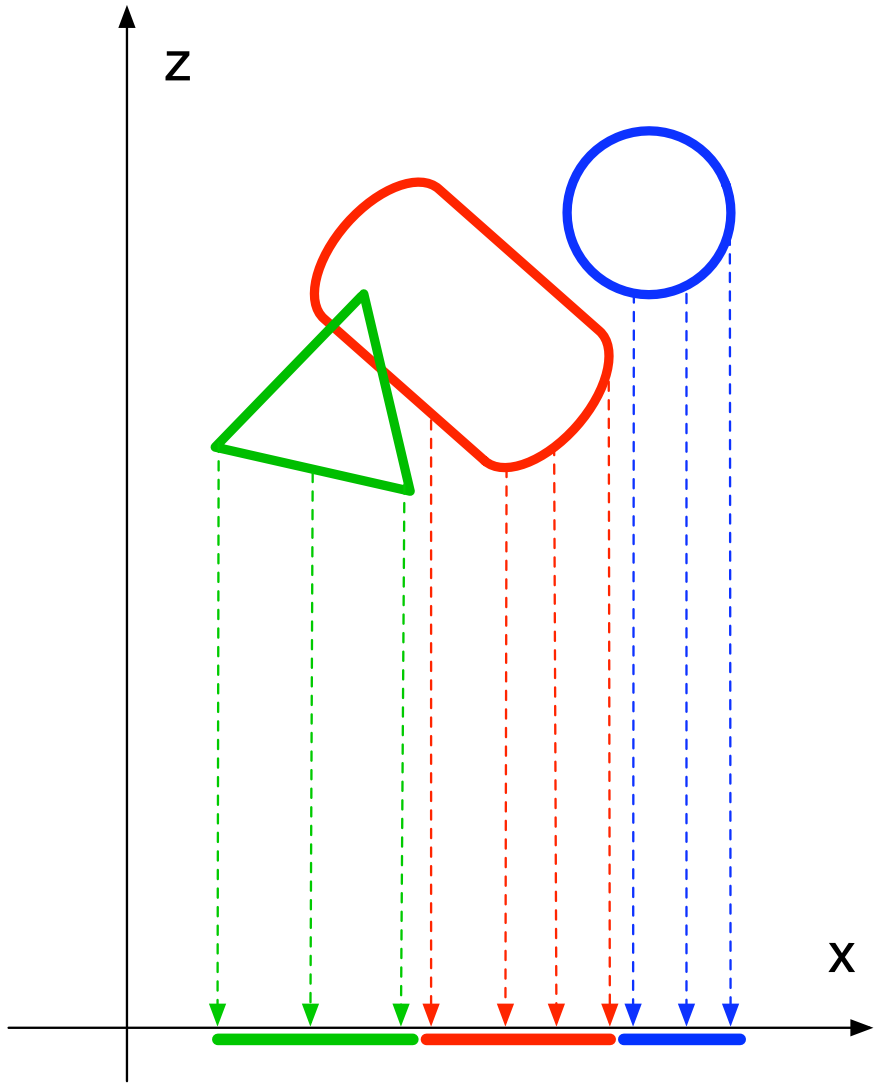
Z-buffer



Z-buffer

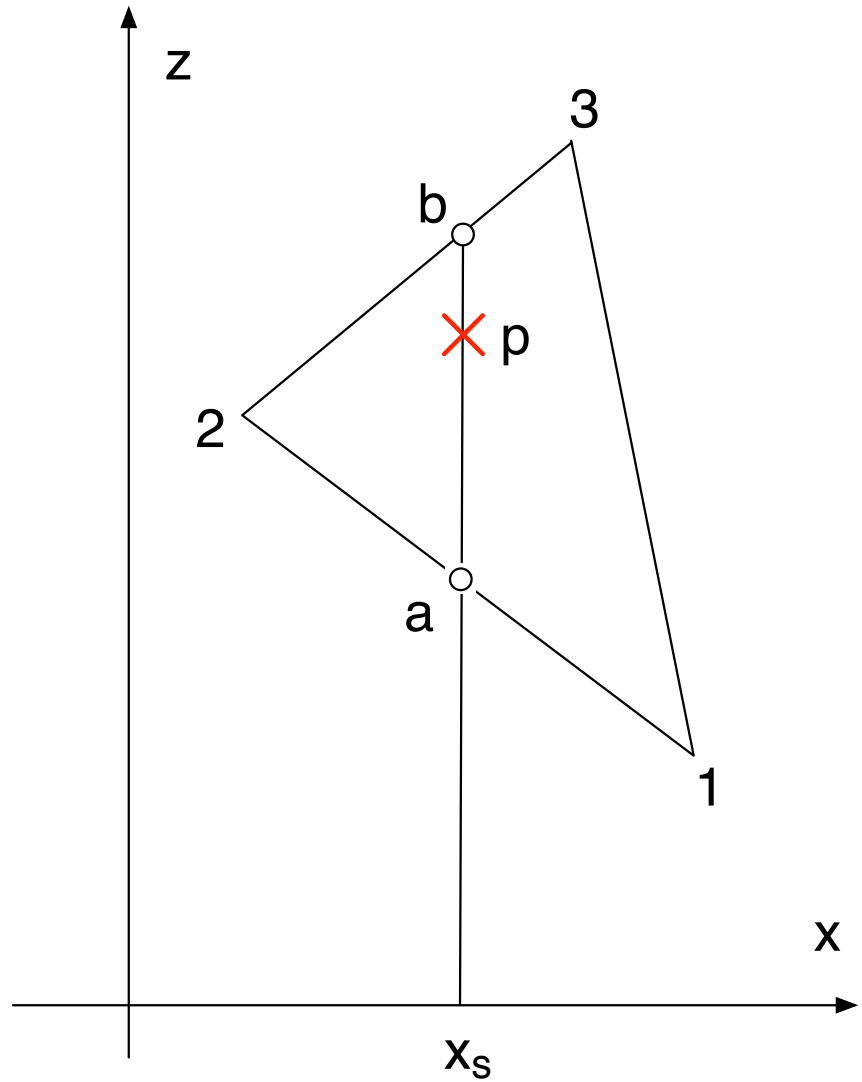


Z-buffer



Z-buffer

$$
\begin{aligned}
z_a &= z_1 + (z_2 - z_1)\frac{x_s - x_1}{x_2 - x_1} \\
z_b &= z_2 + (z_3 - z_2)\frac{x_s - x_2}{x_3 - x_2} \\
z_p &= z_a + (z_b - z_a)\frac{y_p - y_b}{y_b - x_a}
\end{aligned}
$$



WebGL グラフィックスパイプライン

WebGL is ...

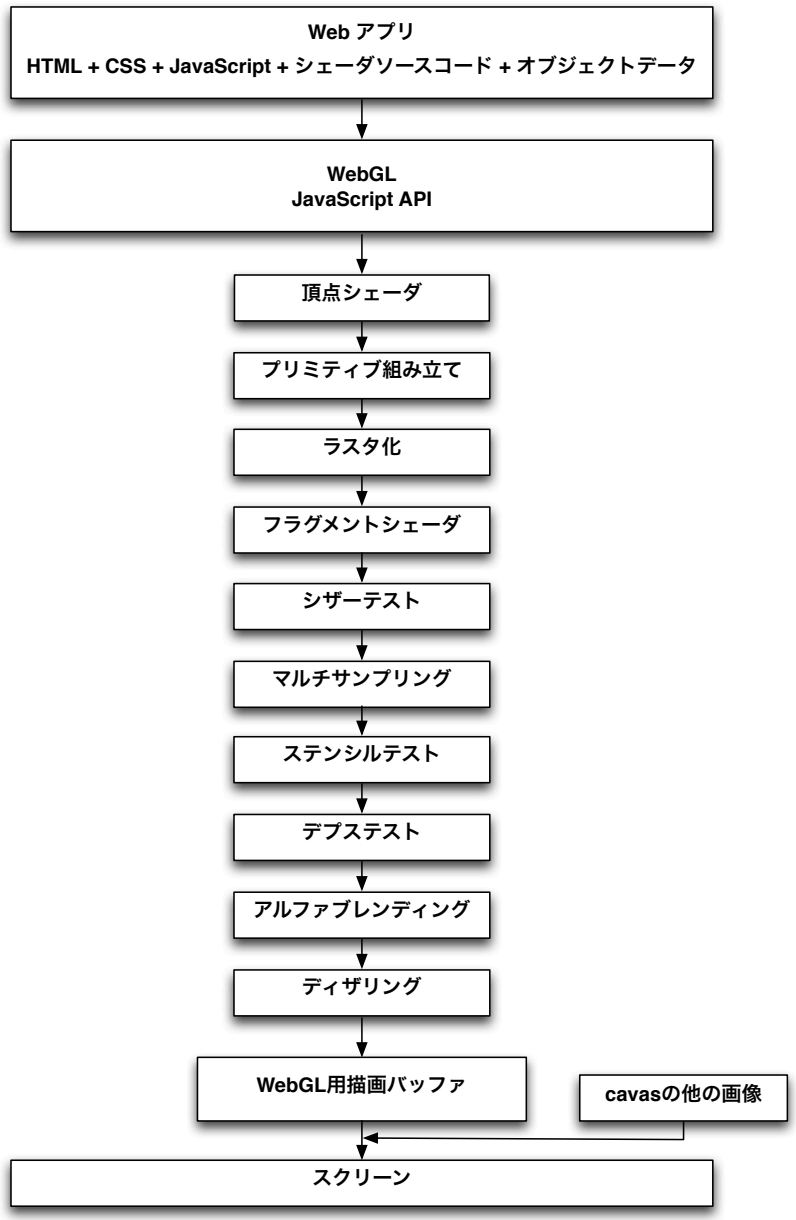
- OpenGL ES 2.0 for the Web
- JavaScript binding for OpenGL ES

`http://www.khronos.org/webgl/`

- ブラウズの環境設定が必要。以下のページで表示されることを確認せよ。

`http://tinyurl.com/kageyama2018v`

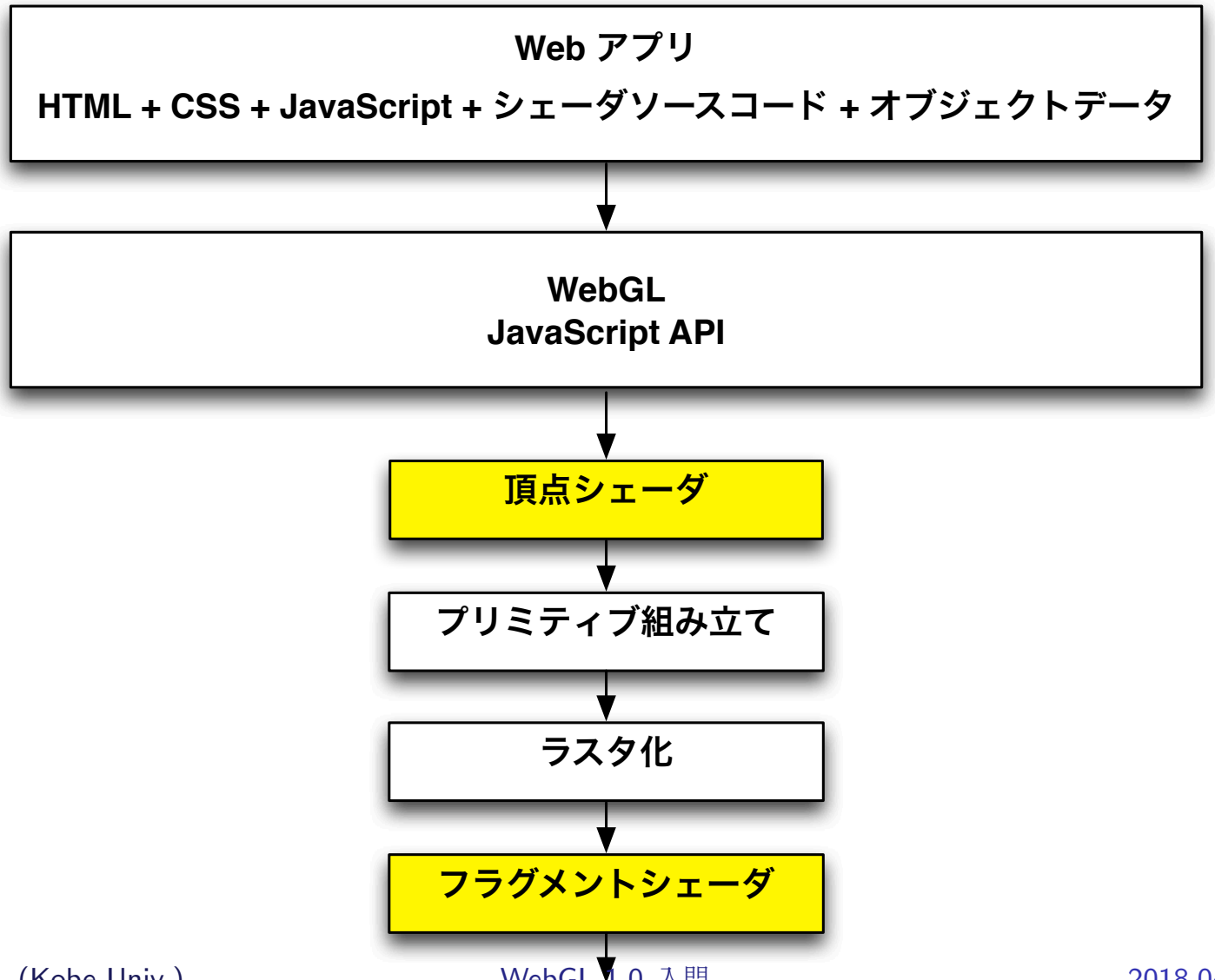
WebGL のグラフィックスパイプライン



シェーダ（拡大図は次のページ）

シェーダ

頂点シェーダ（バーテックスシェーダ）とフラグメントシェーダ



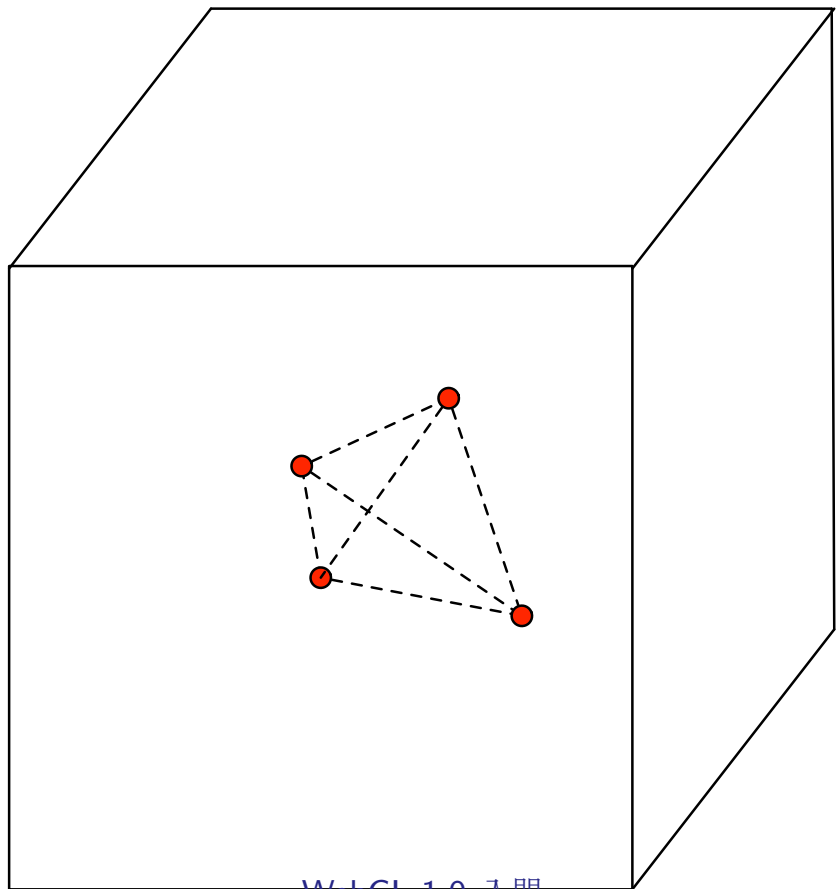
WebGL アプリケーション

Web アプリ ＝ HTML + CSS + JavaScript

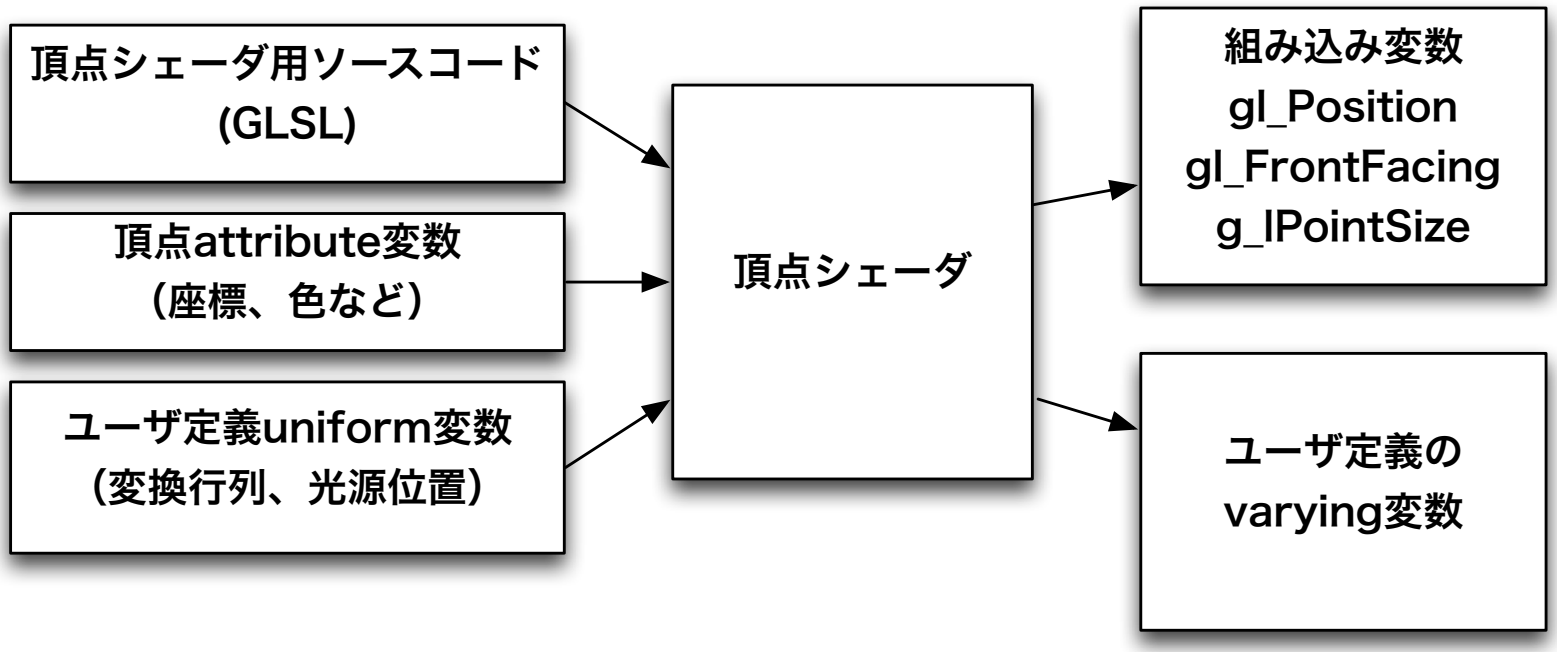
WebGL アプリ ＝ HTML + CSS + JavaScript + シェーダ言語（OpenGL SL）

頂点シェーダ

- 各頂点に対して処理を行う
- 並列処理
- n 個の頂点があれば n 個の頂点シェーダプロセッサを同時に実行させる



頂点シェーダの入出力データ



頂点シェーダプログラム

- C 言語に似ている。
- OpenGL SL (Shading Language)
- 4 行 4 列の行列ベクトル演算が組み込み関数

```
attribute vec3 aVertexPos;
atrribute vec4 aVertexColor;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

varying vec4 vColor;

void main() {
  gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPos, 1.0);
  vColor = aVertexColor;
}
```

頂点シェーダプログラム

```
attribute vec3 aVertexPos;
atrribute vec4 aVertexColor;
```

attribute（属性）変数とは

- ユーザが定義する変数
- 各頂点に固有のデータ（位置や色）

RGBA で 4 成分のベクトル

頂点シェーダプログラム

```
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
```

mat4 は、4×4 の行列の型

uniform 変数とは

- ユーザが定義する変数
- （その時刻（フレーム）に）全ての頂点で同じ値を持つデータ

頂点シェーダプログラム

```
varying vec4 vColor;
```

varing 変数 (varying variable) とは

- フラグメントシェーダに情報を渡すための変数
- ユーザが定義できる
- 組み込み varying 変数もある
  - gl_Position
  - gl_FrontFacing
  - gl_PointSize

頂点シェーダプログラム

```
void main() {
  gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPos, 1.0);
  vColor = aVertexColor;
```

エントリーポイントは main

返値はなし

1. 今処理している頂点の位置（3 次元規格化デバイス座標）を 4 次元にして
2. モデルビュー変換行列をかけて
3. 射影変換行列をかけて
4. 組み込み varying 変数である gl_Position に代入する

最後にこの頂点の色を varying 変数である vColor に書き込む

プリミティブ組み立て

primitive assembly

プリミティブ

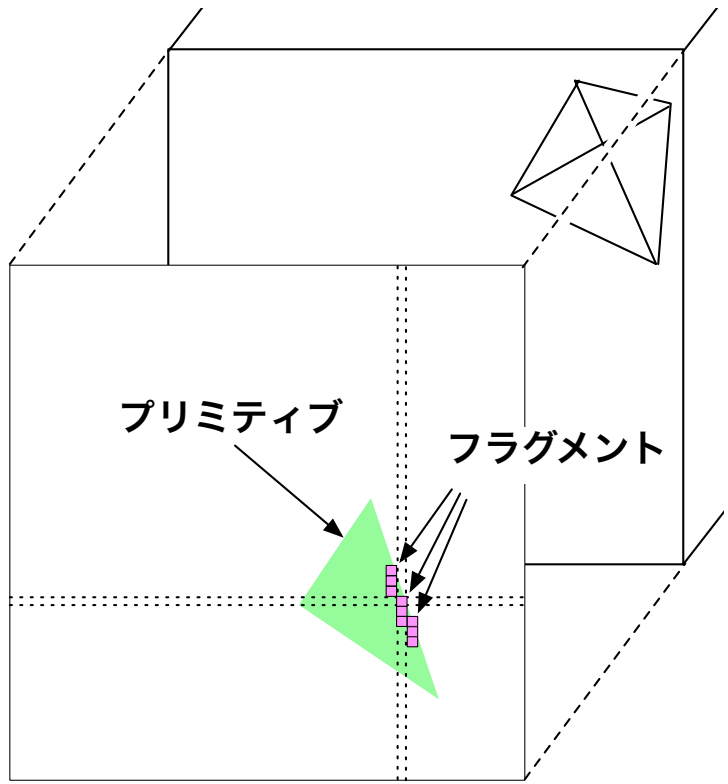
- 3 角形（OpenGL 1.x では沢山のプリミティブがあったがいまは 3 角形と線分、点のみ。）
- 線分
- ポイントスプライト

クリッピング処理はここで行われる

ラスタ化

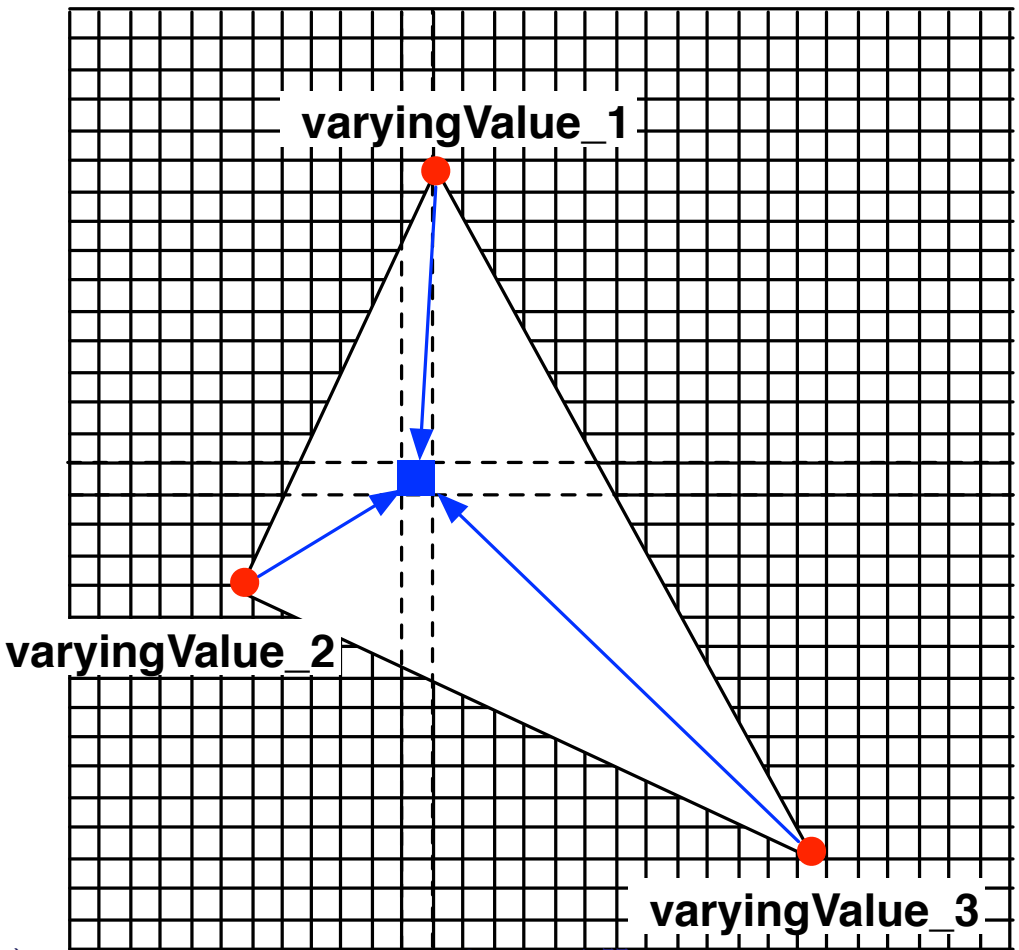
プリミティブからフラグメントを作る処理

フラグメント ≈ ピクセル（様々なテストに合格したフラグメントだけが描画ピクセルになる）



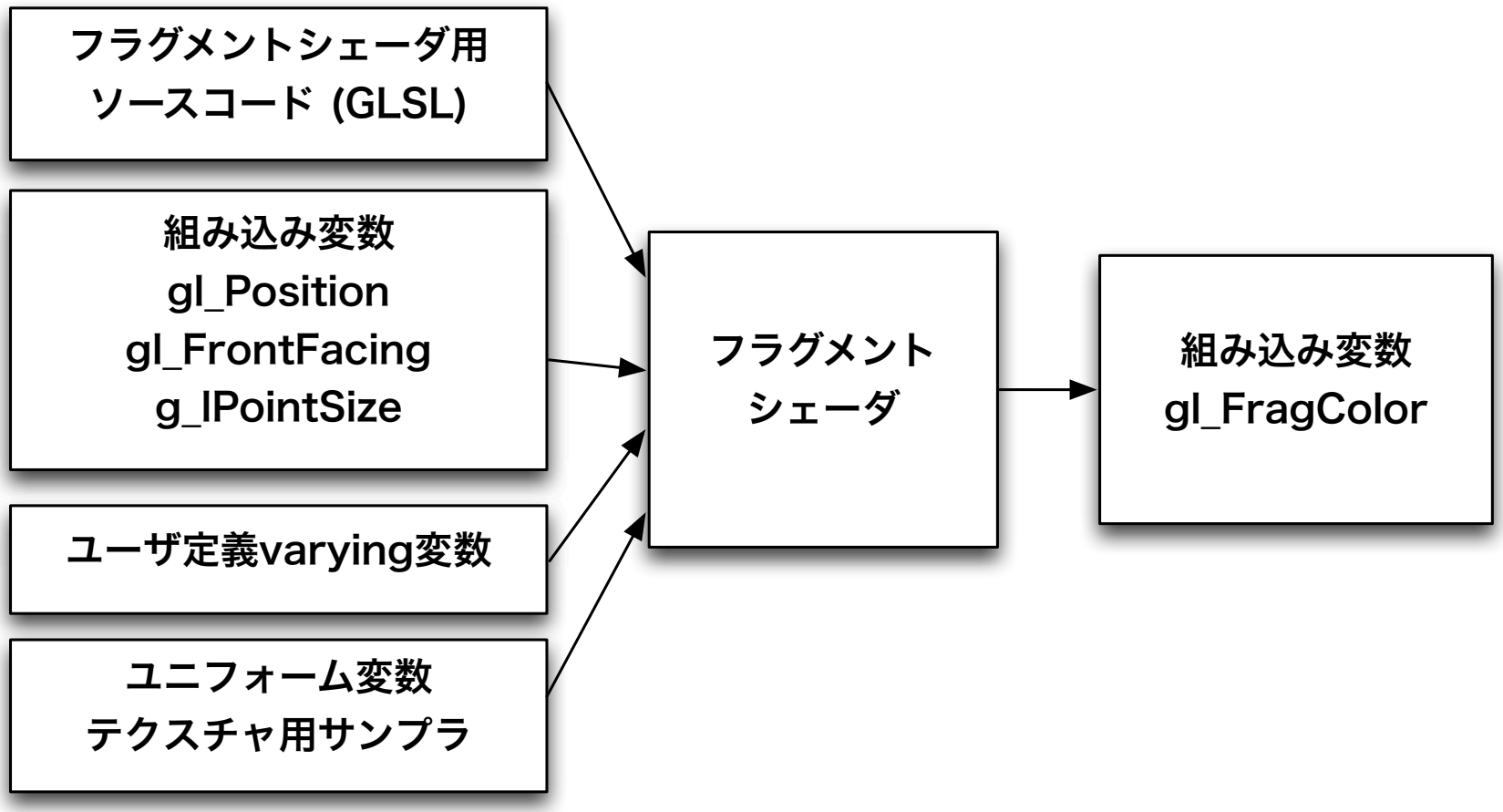
varying 変数の補間

- 頂点シェーダ からフラグメントシェーダへは varying 変数を通じて情報を送る。
- 各フラグメントの varying 変数値は自動的に線形補間される。



フラグメントシェーダの入出力

全てのフラグメントで並列処理。シェーディング言語でプログラム。

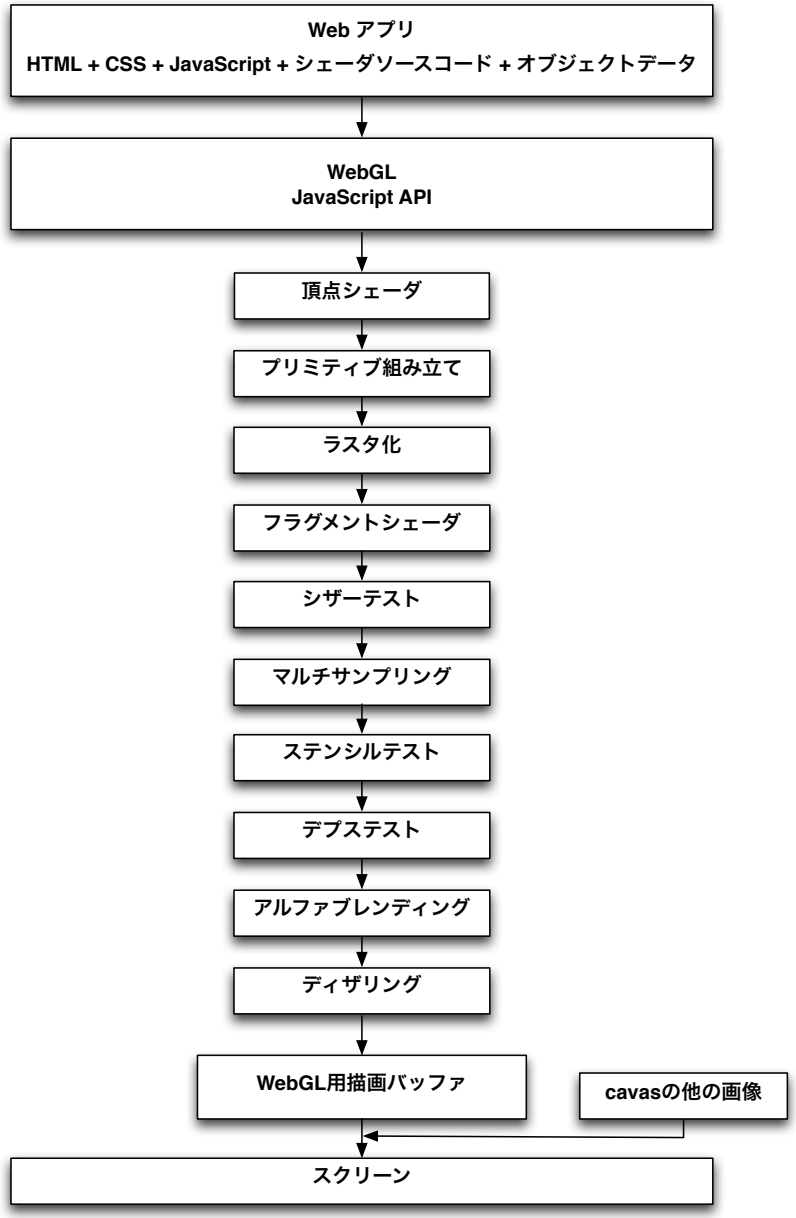


フラグメントシェーダプログラム

```
precision mediump float; // precision qualifier （精度修飾子）

varying vec4 vColor;  // 補間された値

void main() {
  gl_FragColor = vColor;
}
```

精度修飾子： 最低保証する精度。

WebGL のグラフィックスパイプライン（再掲）

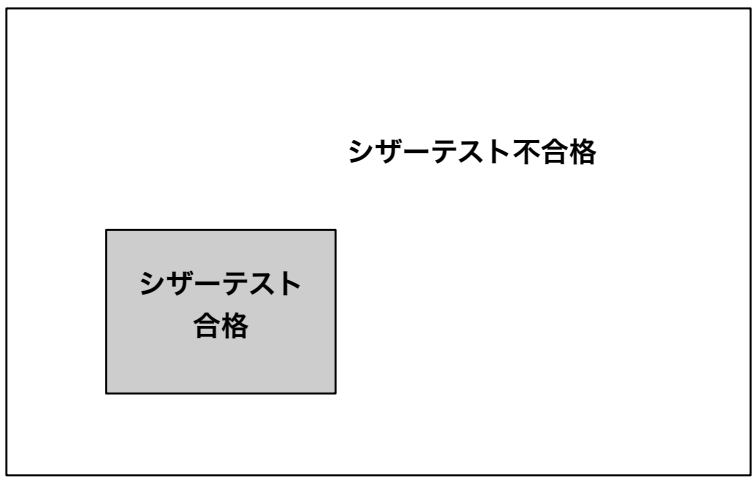


シザーテスト

描画ウィンドウの一部の領域だけを「はさみ（scissors）」で切り取る（はさみといっても任意の形ではない。長方形のみ）。

テストに合格したフラグメントだけ描画。不合格フラグメントはそれ以降のパイプラインを通らない → 処理の高速化

シザーテストの簡単な例： OpenGL Super Bible (2011, p.112)



マルチサンプリング

アンチエイリアジング＝斜めの線（特にほぼ水平な線）のギザギザをとる方法

マルチサンプリング＝周囲の複数のフラグメントをランダムに選択して色を混ぜる

OpenGL Super Bible (2011, p.382) 参照。

ステンシルテスト

ステンシルバッファの対応する位置の値と比較テストする。

不合格フラグメントは破棄

OpenGL Super Bible (2011, p.399) 参照。

デプステスト

既に述べた。

アルファブレンディング

半透明な物体の表現

後述

ディザリング

カラーバッファのビット数が少ないとき、中間色を表現する処理。

WebGLによる三角形の描画

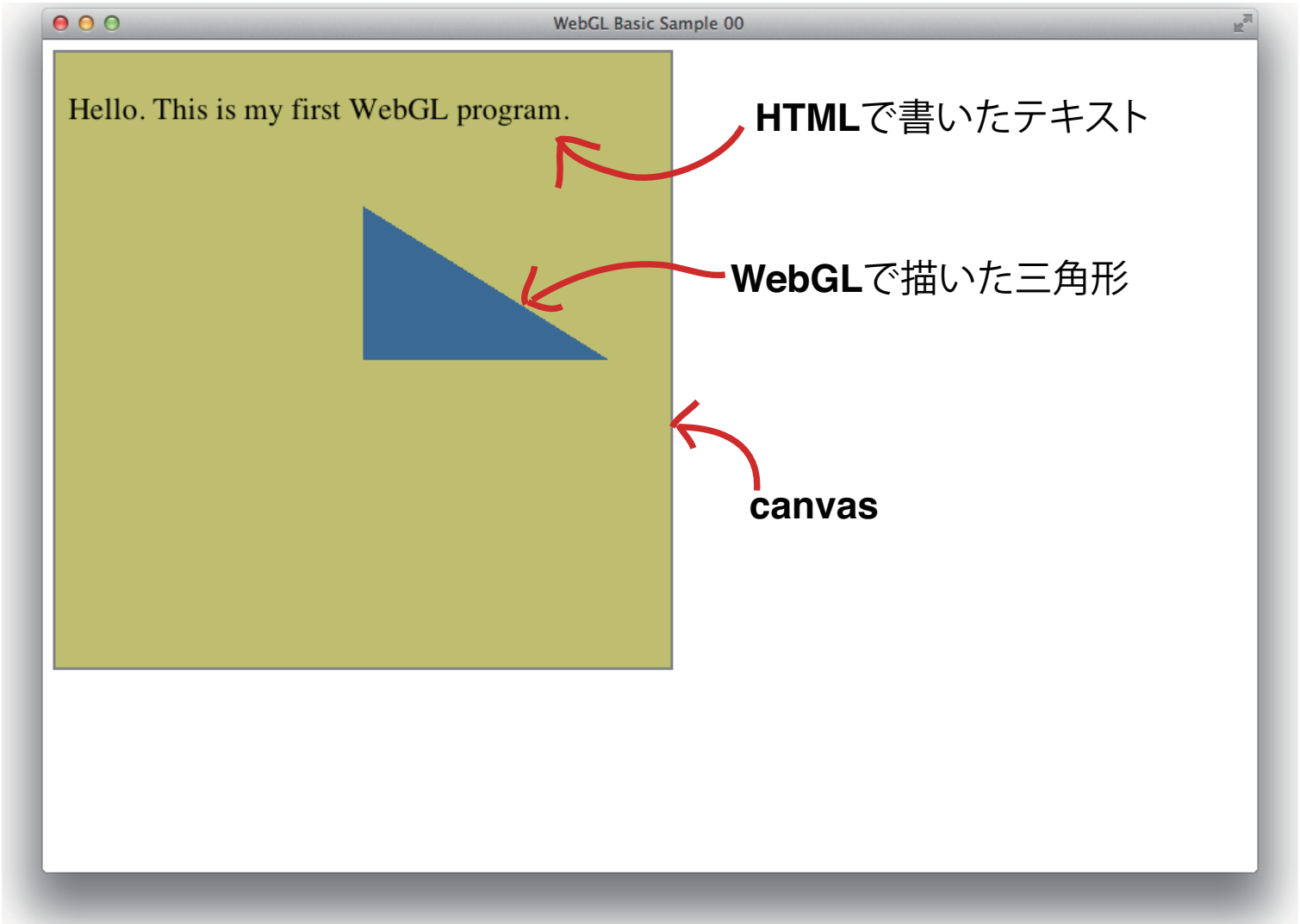
サンプルコード

WebGL での三角形の描画

webgl_sample_triangle_00.html

ソースコードの見方： ［Safari］ 開発 → ページのソースを表示

3 角形の描画


WebGL Basic Sample 00
Hello. This is my first WebGL program.
HTMLで書いたテキスト
WebGLで描いた三角形
canvas

webgl_sample_triangle_00.html

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>WebGL Sample Triangle 00</title>
<meta charset="utf-8">

<style type="text/css">
    canvas {
        border: 2px solid grey;
    }
    .text {
        position:absolute;
        top: 40px;
        left: 20px;
        font-size: 1.5em;
```

```
        color: black;
    }
</style>

<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i<names.length; i++) {
    try {
```

```
      context = canvas.getContext(names[i]);
    } catch(e) {}
    if (context) {
      break;
    }
  }
  if (context) {
    context.viewportWidth = canvas.width;
    context.viewportHeight = canvas.height;
  } else {
    alert("Failed to create context.");
  }
  return context;
}
```

```
function loadShader(type, shaderSource) {
  var shader = gl.createShader(type);
  gl.shaderSource(shader,shaderSource);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert("Error compiling shader" + gl.getShaderInfoLog(
        shader));
    gl.deleteShader(shader);
    return null;
  }
  return shader;
}
```

```
function setupShaders() {
  var vertexShaderSource =
    "attribute vec3 aVertexPosition;  \n" +
    "void main() { \n" +
    "  gl_Position = vec4(aVertexPosition, 1.0); \n" +
    "} \n";
```

```
var fragmentShaderSource =
  "precision mediump float; \n" +
  "void main() {     \n" +
  "   gl_FragColor = vec4(0.2, 0.4, 0.6, 1.0); \n" +
  "} \n";

var vertexShader = loadShader(gl.VERTEX_SHADER,
    vertexShaderSource);
var fragmentShader = loadShader(gl.FRAGMENT_SHADER,
    fragmentShaderSource);

shaderProgram = gl.createProgram();
```

```
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS))
        {
      alert("Failed to setup shader.");
    }

    gl.useProgram(shaderProgram);

    shaderProgram.vertexPositionAttribute =
      gl.getAttribLocation(shaderProgram, "aVertexPosition");
}

function setupBuffers() {
```

```
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  var triangleVertices = [
        0.0, 0.0, 0.0,
        0.8, 0.0, 0.0,
        0.0, 0.5, 0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(
      triangleVertices),
                gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numberOfItems = 3;
}

function draw() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
```

```
  gl.clear(gl.COLOR_BUFFER_BIT);

  gl.vertexAttribPointer(shaderProgram.
      vertexPositionAttribute,
                         vertexBuffer.itemSize, gl.FLOAT,
                             false, 0, 0);
  gl.enableVertexAttribArray(shaderProgram.
      vertexPositionAttribute);
  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}

function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = createGLContext(canvas);
  setupShaders();
  setupBuffers();
  gl.clearColor(0.8, 0.8, 0.4, 1.0);
  draw();
```

```
}

</script>
</head>

<body onload="startup();">
  <canvas id="myGLCanvas" width="480" height="480"></canvas>
  <div class="text">Hello. This is my first WebGL program.</
      div>
</body>
</html>
```

コンテキスト作成

- HTML5 の canvas 要素の getContext method を呼ぶ
- 引数（文字列）は “experimental-webgl” を渡す（将来これは “webgl” に変更されるので、どちらにも対応できるようにしておく）。
- getContext method が返すのは WebGLRenderingContext オブジェクト。
  - WebGL の変数や関数はこの WebGLRenderingContext のメンバー。
  - このサンプルプログラムでは gl という変数にこのオブジェクトを代入。

シェーダプログラム

- 頂点シェーダ用の（GLSL 言語の）ソースコードは、GPU に送り、GPU にコンパイルさせる。
- HTML（あるいは JavaScript）の文脈では GLSL ソースコードは単なる文字列。
- ここでは文字列変数 fragmentShaderSource にソースコード文字列を代入して送っている。
- 【注意】普通はこうはしない。もっと便利な方法がある。後述。

シェーダプログラムの作成

1. 頂点シェーダオブジェクトを作る
   1.1 頂点シェーダソースコードを頂点シェーダオブジェクトにロードする
   1.2 コンパイルする
2. フラグメントシェーダオブジェクトを作る
   2.1 フラグメントシェーダソースコードをフラグメントシェーダオブジェクトにロードする
   2.2 コンパイルする
3. プログラムオブジェクトを作る
4. プログラムオブジェクトに頂点シェーダオブジェクトをアタッチする（こちらが先）
5. プログラムオブジェクトにフラグメントシェーダオブジェクトをアタッチする
6. リンクする
7. このプログラムオブジェクトを WebGLRenderingContext に登録する

（GLSL の）頂点属性について

詳細は『OpenGL 4.0 グラフィックスシステム』の p.60 を参照

頂点シェーダ内で定義できる attribute 変数

float, vec2, vec3, vec4,

mat2, mat3x2, mat4x2,

mat2x3, mat3, mat4x3,

mat4, mat2x4, mat3x4

宣言されていても使われていない attribute 変数 → アクティブでない

それ以外 → アクティブ

アクティブな attribute 変数全体はまとめて整数インデックスで管理される（最大サイズは MAX_VERTEX_ATTRIBS）。それぞれの attribute 変数にはこのインデックスで参照する。

`int BindAttribLocation(unit program, unit index, const char *name);`

- name という attribute 変数を index にバインドする（上書き）。
- program はリンクしたプログラムオブジェクトの名前。
- 自分でバインドしない場合はGL が自動的にバインドする。
- そのときの index は以下の関数で取得できる。

`int GetAttribLocation(unit program, const char *name);`

attribute 変数名 (name) を指定して、その変数がどのインデックスにバインドされているかを問い合わせる。

シェーダプログラム

gl.useProgram(shaderProgram);

shaderProgram . vertexPositionAttribute =
gl.getAttribLocation(shaderProgram, ” aVertexPosition ”);

上で、シェーダプログラムの vertexPositionAttribute というプロパティ名は任意。

```
shaderProgram.anyNameIsOK =
  gl.getAttribLocation(shaderProgram, "aVertexPosition");
```

バッファの作成

```
function setupBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  var triangleVertices = [
        0.0, 0.0, 0.0,
        0.8, 0.0, 0.0,
        0.0, 0.5, 0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(
      triangleVertices),
                gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numberOfItems = 3;
}
```

バッファの作成

- 三角形の頂点データをシェーダに送るためにはバッファオブジェクトを使う。
- バッファオブジェクトは WebGLRenderingContext の createBuffer() メソッドで作成する。
- 作成した WebGLBuffer オブジェクトを JavaScript のグローバル変数 vertexBuffer に代入
- vertexBuffer を「現在の配列バッファオブジェクト」にバインドする。
- 後で使えるようにこの頂点配列の情報（頂点数等）も書いておく（どうやって渡してもよい）。

描画

```
function draw() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT);

  gl.vertexAttribPointer(shaderProgram.
      vertexPositionAttribute,
                         vertexBuffer.itemSize, gl.FLOAT,
                             false, 0, 0);
  gl.enableVertexAttribArray(shaderProgram.
      vertexPositionAttribute);
  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}
```

演習

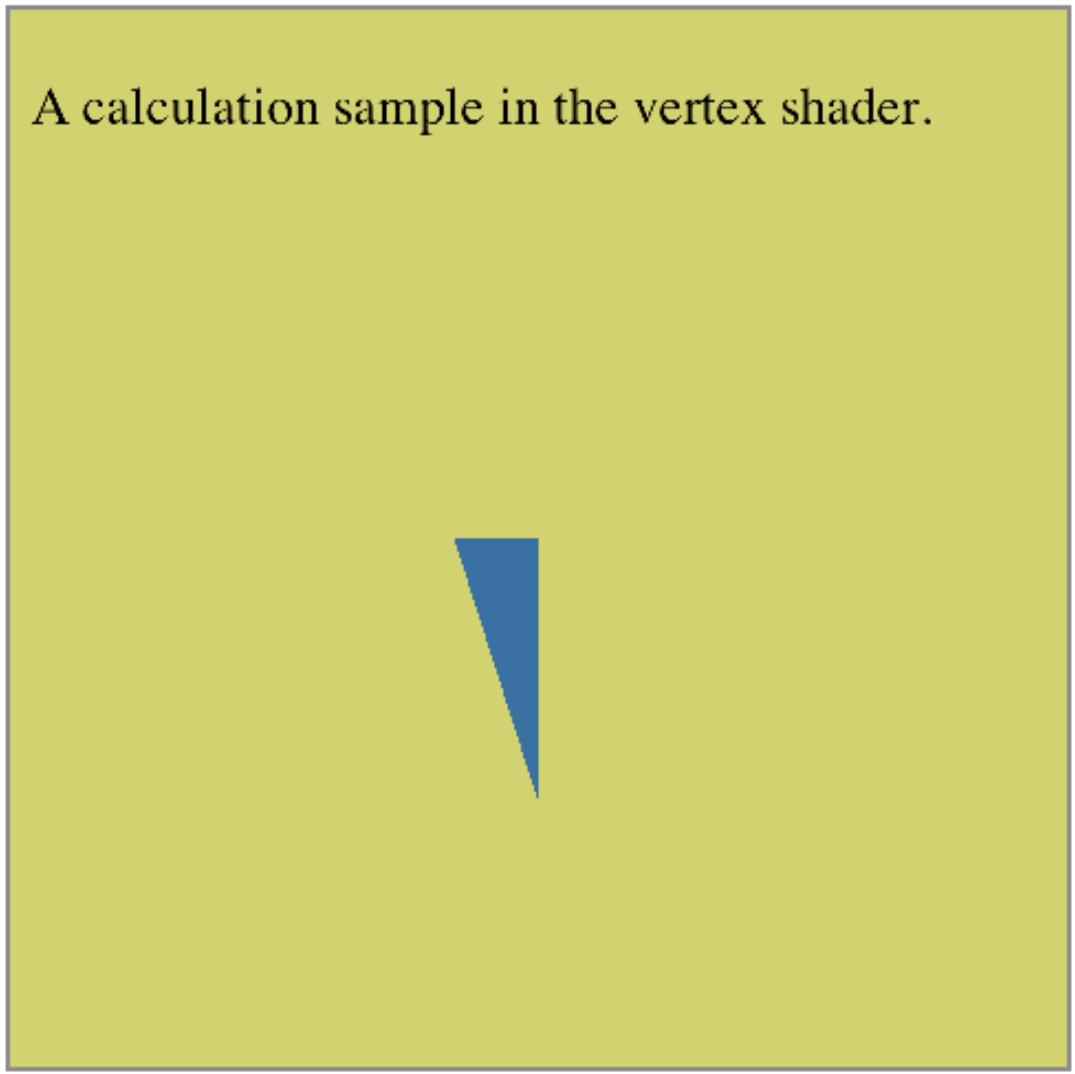
頂点シェーダでの演算

- 頂点シェーダの中で簡単な数値計算をしてみよう。
- 例題：webgl_sample_triangle_01.html

webgl_sample_triangle_01.html

```
function setupShaders() {
  var vertexShaderSource =
    "attribute vec3 aVertexPosition;  \n" +
    "void main() { \n" +
    "   gl_Position = vec4(aVertexPosition, 1.0); \n" +
    "   gl_Position *= vec4(-0.2,-1.0,1.0,1.0); //  Here !! \n
        " +
    "} \n";
```

頂点シェーダでの演算



演習

- webgl_sample_triangle_00.html の頂点シェーダを変更して元の三角形とは別の図形（図）を描こう。

描画：DrawArrays と DrawElements

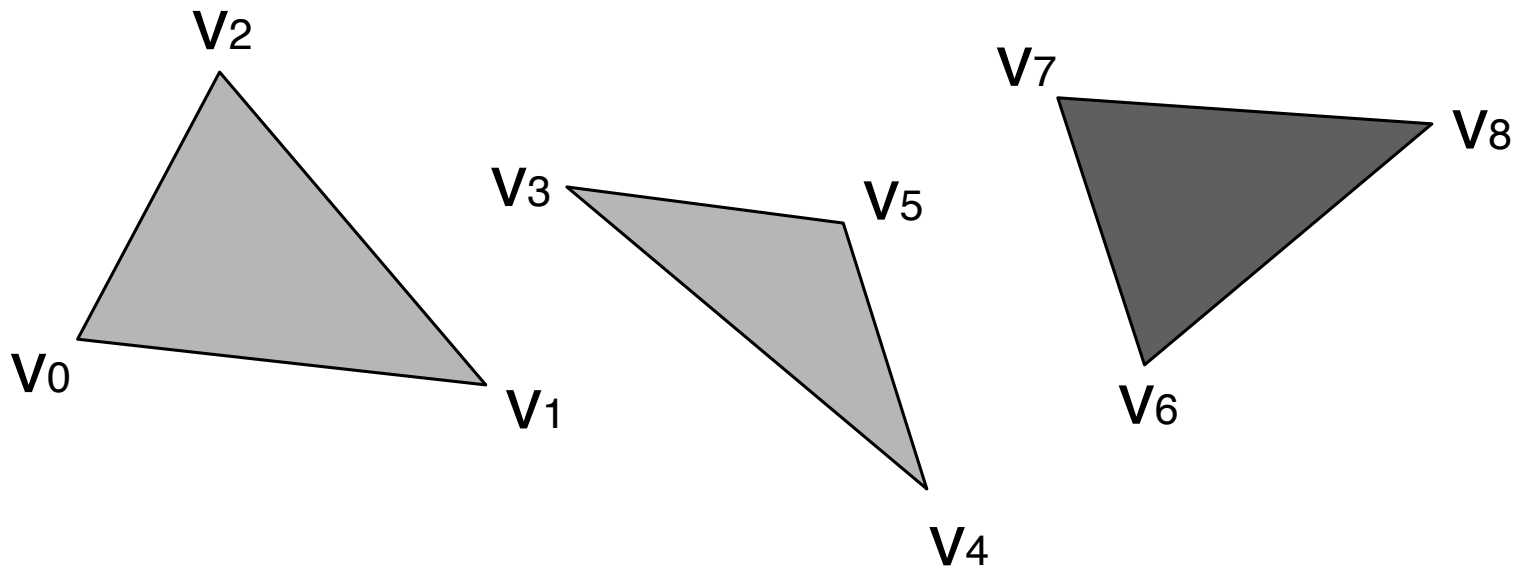
最も基本的な描画

- 指定した色で全てのピクセルを描く
- `gl.clear()`
- 色の指定 `gl.clearColor()`
- 「背景色」

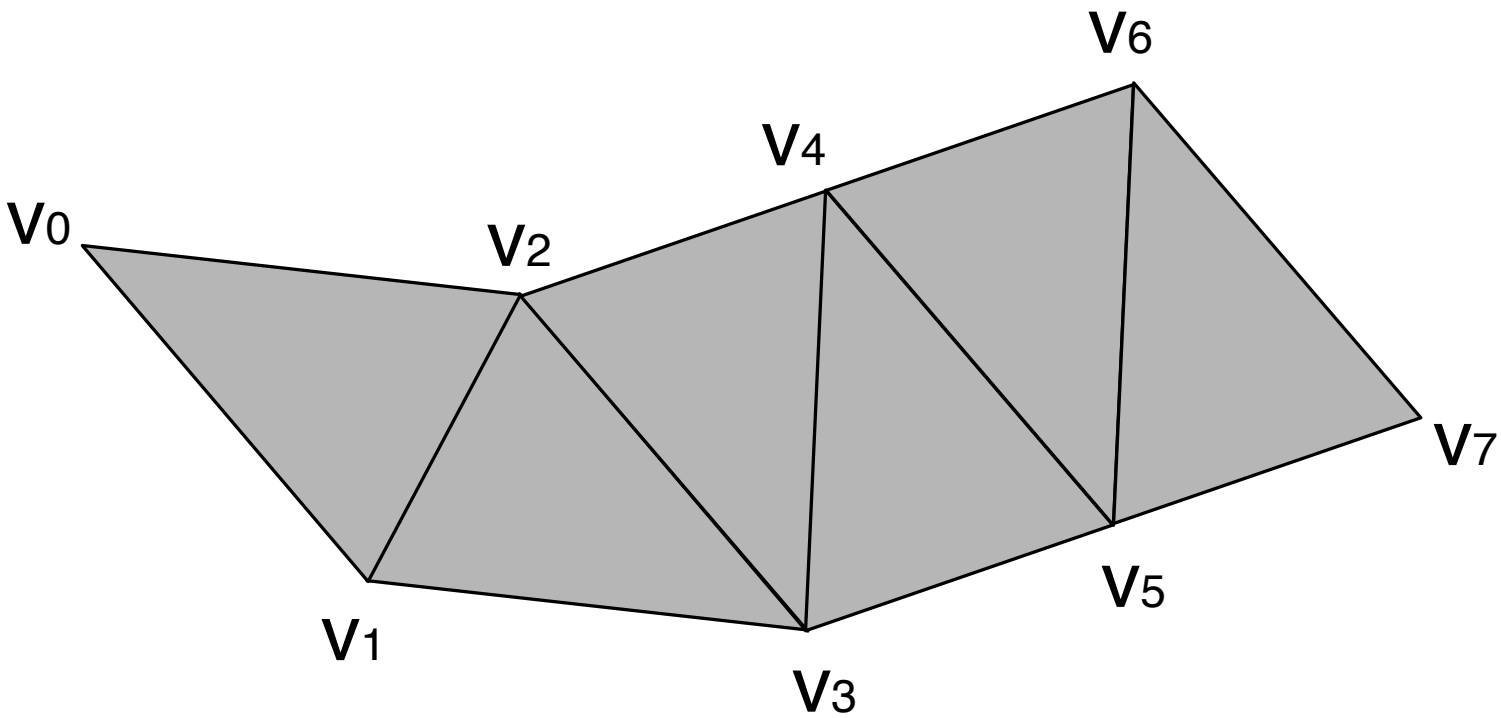
プリミティブ

- 面は 3 角形で作る
- OpenGL 1.x と基本は同じ（ただしプリミティブは三角形だけになった。）
- TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN

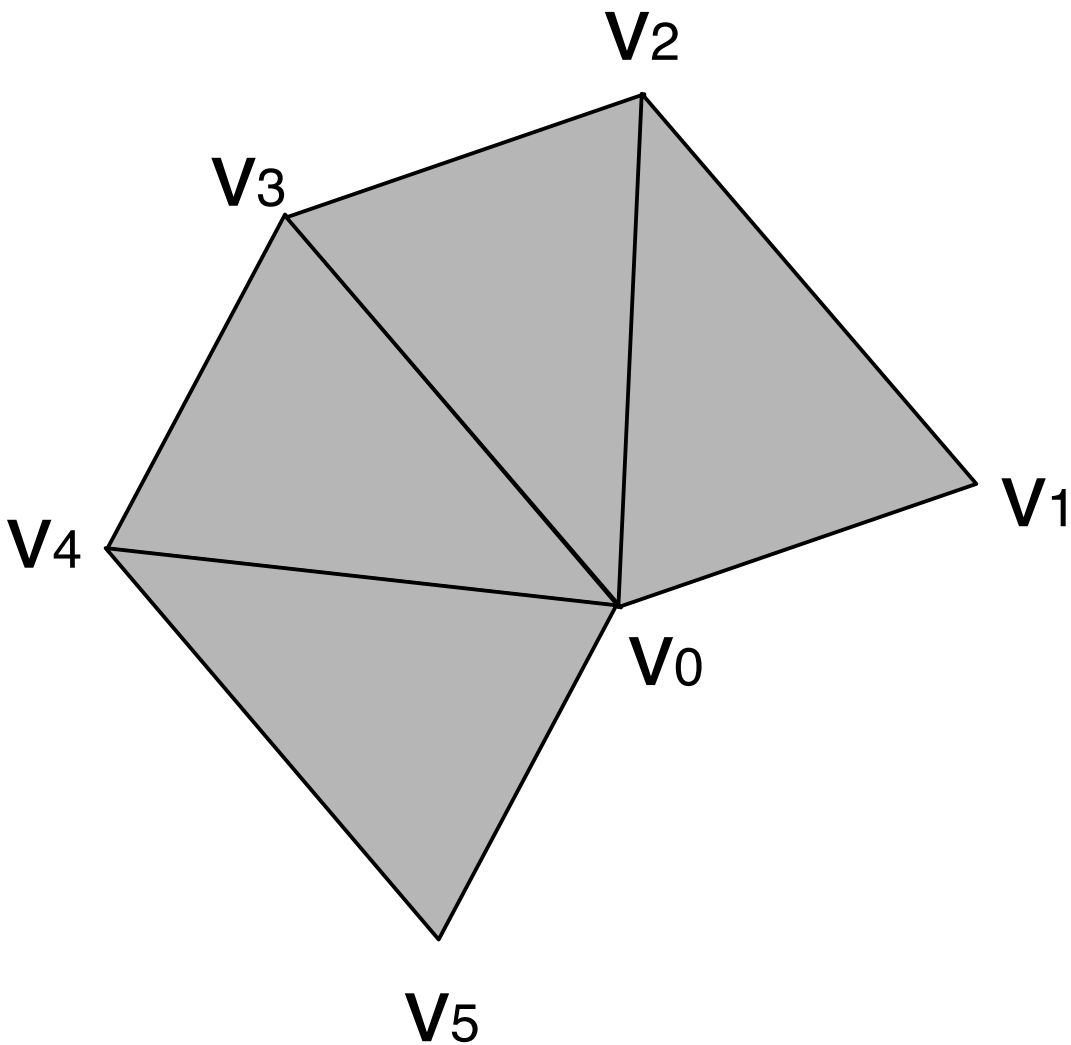
gl.TRIANGLES



gl.TRIANGLE_STRIP



gl.TRIANGLE_FAN



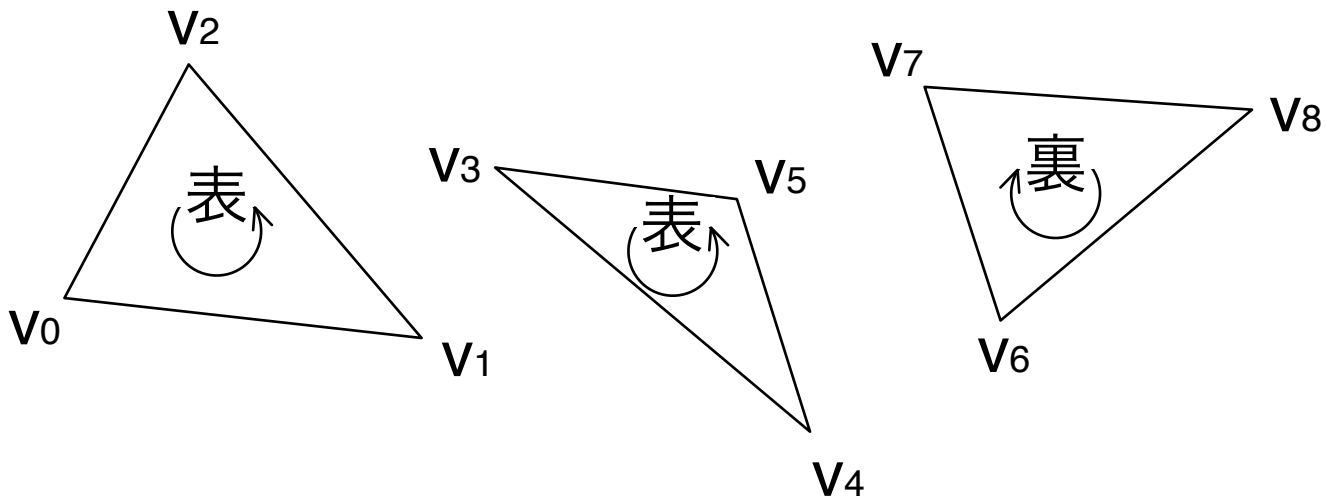
Front Face と Winding Order

三角形には表面と裏面がある

表面は頂点の番号順で決まる。デフォルトは “右手系”。

反時計方向（Counter Clock Wise, CCW）

逆は時計方向（Clock Wise, CW）



裏面のカリング

裏面を見ることがない場合、裏面のラスター処理は省略すればいい。

高速化。

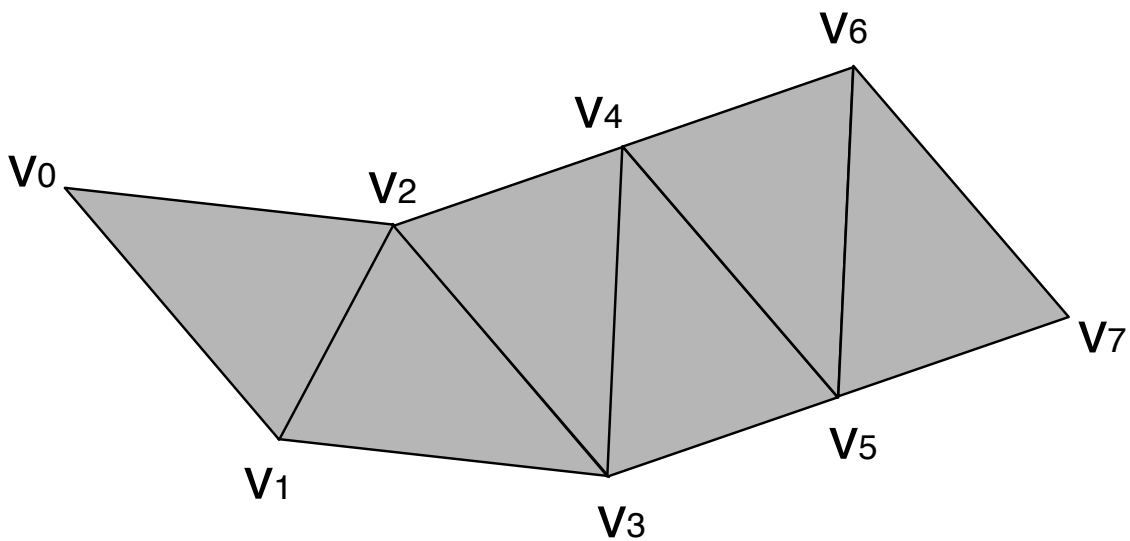
カリング （culling）

```
gl.frontFace(gl.CCW);          // デフォルト
gl.enable(gl.CULL_FACE);       // デフォルトでは disabled
gl.cullFace(gl.BACK);          // デフォルト
```

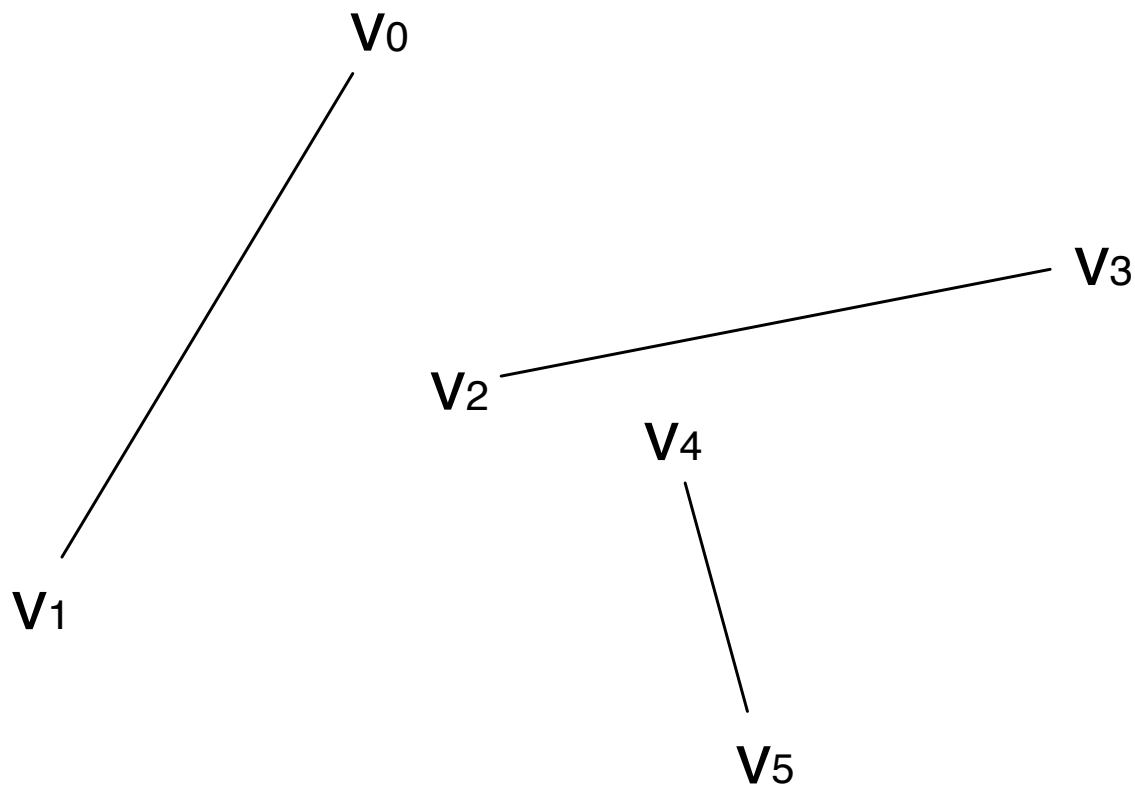
TRIANGLE_STRIP

ワインディングオーダーを保存した三角形列を自動的に構成

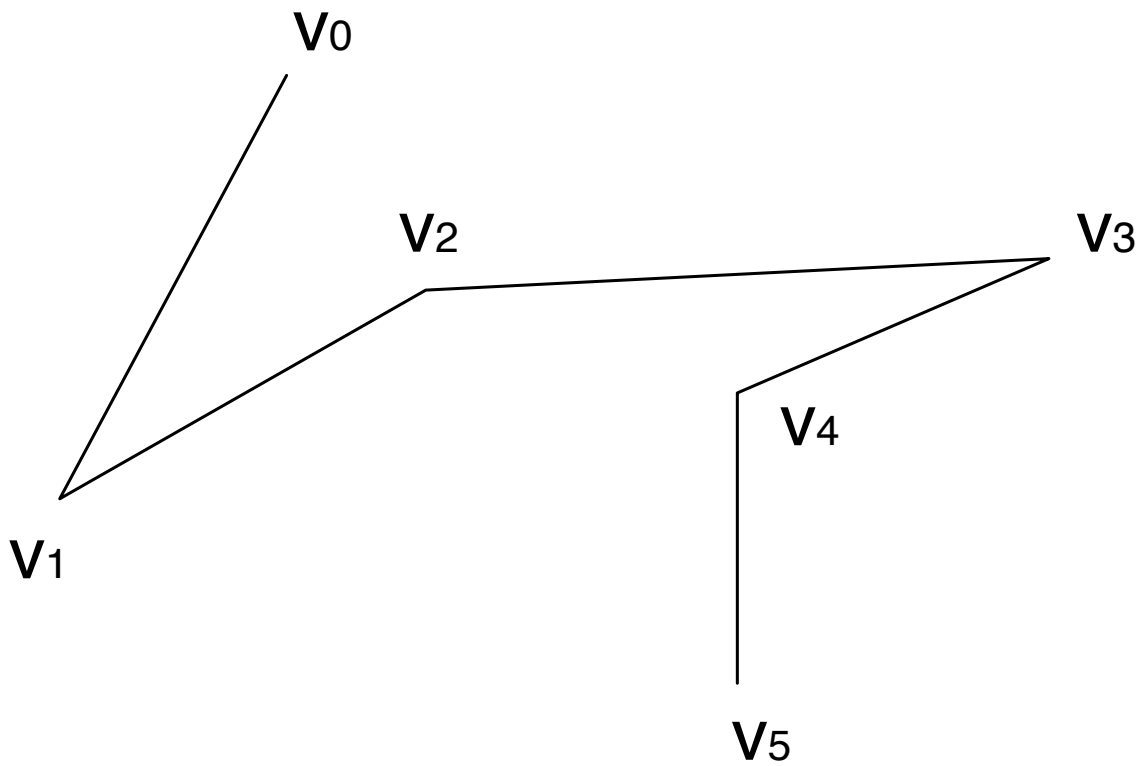
三角形 1	v_0	v_1	v_2				
三角形 2		v_2	v_1	v_3			
三角形 3			v_2	v_3	v_4		
三角形 4				v_4	v_3	v_5	
三角形 5					v_4	v_5	v_6
三角形 6						v_6	v_5	v_7



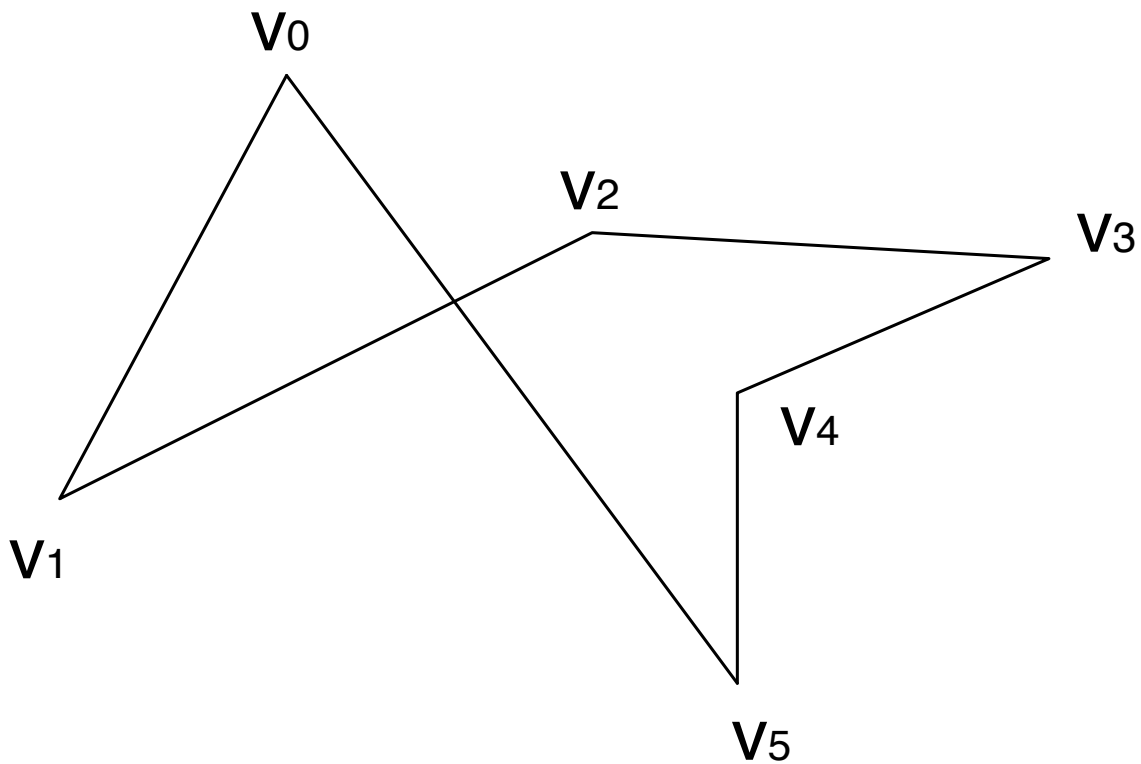
gl.LINES



gl.LINE_STRIP



gl.LINE_LOOP



点

ポイントスプライト（point sprite）

“大きさを持った点”

`gl_PointSize` で指定（シェーダで）

二つの描画メソッド

`gl.drawArrays()`： 配列に納められた順番通りに頂点からプリミティブを構成する

`gl.drawElements()`：別の配列（要素配列）を使って頂点を再利用する

gl.drawArrays()

```
void drawArrays(GLenum mode, GLint first, GLsizei count)
```

ここで mode は gl.X: X={POINTS, LINES, LINE_LOOP, LINE_STRIP, TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN}

first は頂点データ配列のうち、最初に使用する要素のインデックス

count は何個の頂点を使うか

何の配列を使うかの指定がないことに注意。

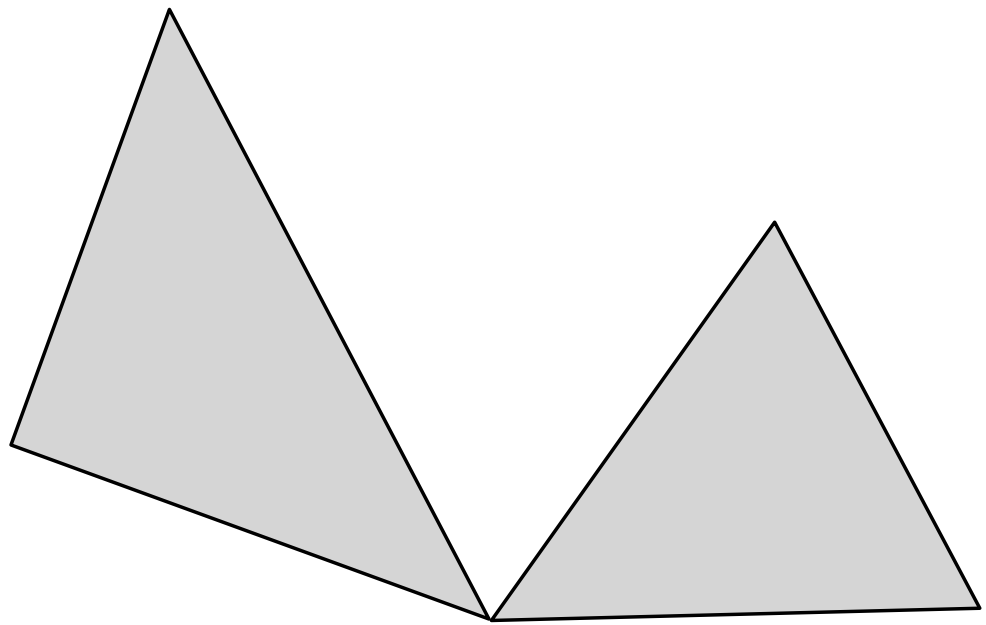
描画する頂点データ配列は、gl.ARRAY_BUFFER と決まっている。

gl.ARRAY_BUFFER にバインドされた配列バッファに頂点の座標を入れる。

ソースコード

```
// バッファオブジェクト作成
  vertexBuffer = gl.createBuffer();
// それをバインドする
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
//  （普通の）配列で座標を用意
  var triangleVertices = [0.0, ...
// 普通の配列から型付き配列を作り、それをバッファにアップロード
  gl.bufferData(gl.ARRAY_BUFFER,
                new Float32Array(triangleVertices)...
// 頂点シェーダの属性としてこのバッファを使うことを指定する
  gl.vertexAttribPointer(shaderProgram.
      vertexPositionAttribute,...
// 頂点シェーダの頂点属性配列を使うことを宣言
  gl.enableVertexAttribArray(shaderProgram.
      vertexPositionAttribute);
// 三角形描画
  gl.drawArrays(gl.TRIANGLES, ...
```

頂点の重複



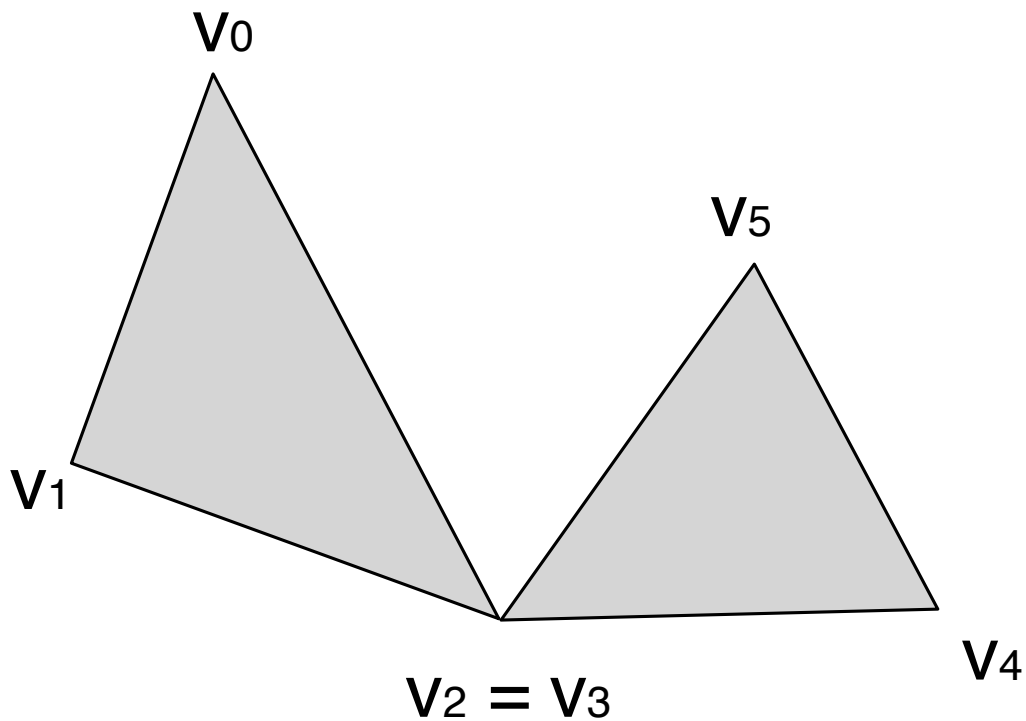
drawArrays() で描く場合

配列バッファ

v_0	v_1	v_2	v_3	v_4	v_5

を用意して gl.TRIANGLES で描く。

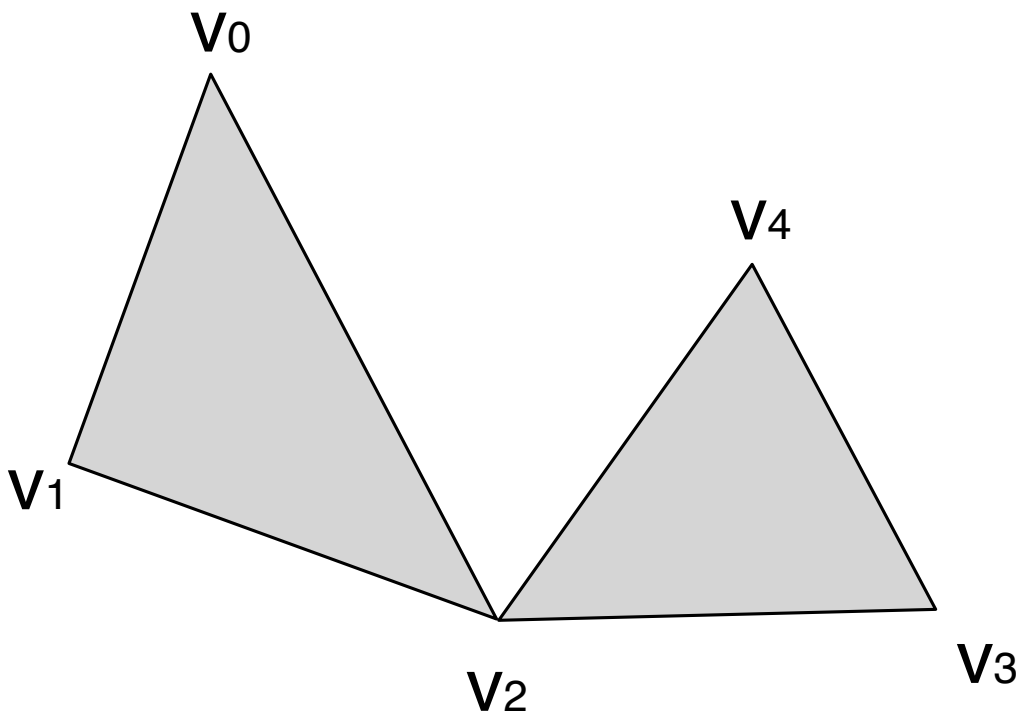
頂点 v_2 と v_3 は重複。無駄なメモリと通信。



drawElements()

配列バッファ | v_0 | v_1 | v_2 | v_3 | v_4 | と、

要素配列バッファ | 0 | 1 | 2 | 2 | 3 | 4 | を用意して gl.TRIANGLES で描く。（WebGLBuffer オブジェクトを二つ使う。）



gl.drawElements()

`void drawElements(GLenum mode, GLsizei count, GLenum type, GLintptr offset)`

mode は `gl.X: X={POINTS, LINES, LINE_LOOP, LINE_STRIP, TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN}`

以下、ab を `gl.ELEMENT_ARRAY_BUFFER` にバインドされた要素配列バッファとすると、

count は ab に何個のインデックスがあるか

type は ab の要素インデックスの型。`gl.UNSIGNED_BYTE` または `gl.UNSIGNED_SHORT` のどちらか

offset は ab の中で実際に使うインデックスの開始位置（オフセット）

サンプルコード webgl_sample_triangle_02.html

```
function setupBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  var triangleVertices = [
        0.000000,  0.866025, 0.0,
       -0.500000,  0.000000, 0.0,
       -1.000000, -0.866025, 0.0,
        0.000000, -0.866025, 0.0,
        1.000000, -0.866025, 0.0,
        0.500000,  0.000000, 0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(
      triangleVertices),
                gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
```

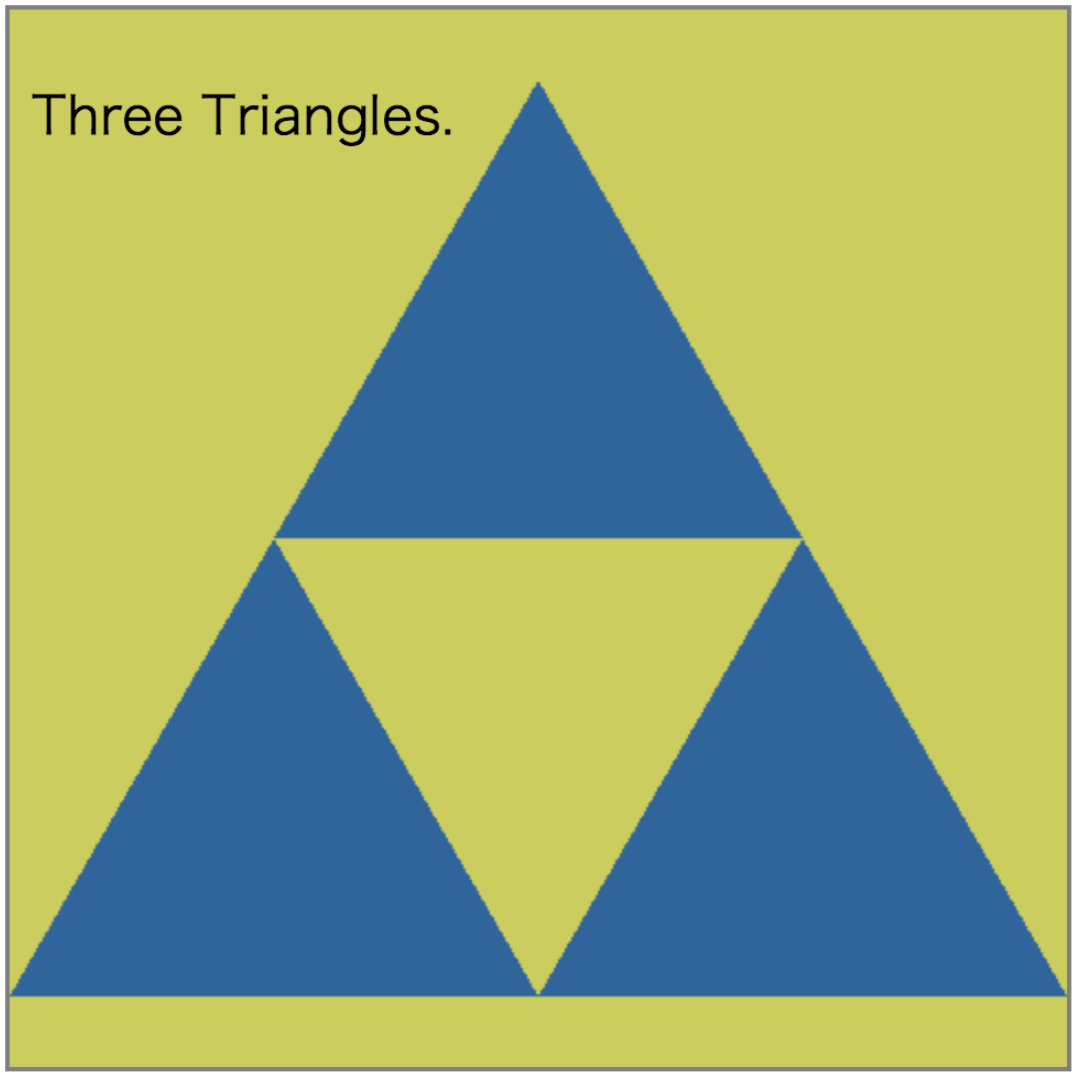
サンプルコード webgl_sample_triangle_02.html

```
    indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    var indexNumbers = [
            0, 1, 5,
            1, 2, 3,
            3, 4, 5
    ];
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(
        indexNumbers),
                  gl.STATIC_DRAW);
    indexBuffer.size = 9;
}

function draw() {
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);
```

サンプルコード webgl_sample_triangle_02.html

```
    gl.vertexAttribPointer(shaderProgram.
        vertexPositionAttribute,
                               vertexBuffer.itemSize, gl.FLOAT,
                                   false, 0, 0);
    gl.enableVertexAttribArray(shaderProgram.
        vertexPositionAttribute);

    gl.drawElements(gl.TRIANGLES, indexBuffer.size,
                        gl.UNSIGNED_SHORT, 0);
}

function startup() {
    canvas = document.getElementById("myGLCanvas");
    gl = createGLContext(canvas);
    setupShaders();
    setupBuffers();
    gl.clearColor(0.8, 0.8, 0.4, 1.0);
    draw();
```

webgl_sample_triangle_02.html の実行結果



WebGL のコーディングスタイル

（この講義での）命名規則

- canvas.getContext() で取り出した WebGLRenderingContext は常に gl という変数に保存。
- gl はグローバル変数とする
- JavaScript もシェーダも camelCase
- シェーダの変数では、属性（attribute）には a のプリフィックス。 e.g., aVertexPosition
- シェーダの変数では、varying 変数には v のプリフィックス。 e.g., vColor
- シェーダの変数では、uniform には u のプリフィックス。 e.g., uMVMatrix

WebGL コードのデバッグ

JavaScript コンソール

- それぞれのブラウザで「コンソール表示」
  - Safari の場合：開発 → エラーコンソールを表示
- エラーメッセージが表示される
- JavaScript の console.log(" 文字列 "); → 文字列をプリント
- “printf デバッグ” に使える

演習

Safari のエラーコンソールを開き、

- 自作 HTML ファイル中の JavaScript プログラムから console.log() 関数で文字列を書き出せ。

WebGL 用のデバッグツール

- Chrome: Chrome デベロッパーツール
- Firefox: Firebug （Firefox のエクステンション）
- 各種ブラウザ: Web Inspector

Safari Web Inspector

- Sarafi 用 Web 開発ツール
- OS X & iOS

Safari Web Inspector

使い方

- WebGL アプリを含むページを表示
- メニュー： 開発 → Web インスペクタを表示

ツールバーに置くと便利：

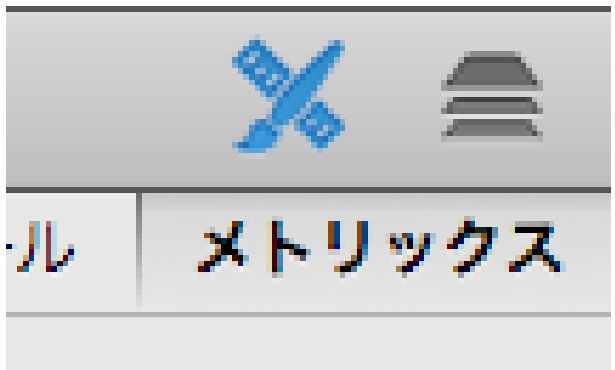
- メニュー: 表示 → ツールバーをカスタマイズ... → Web インスペクタのアイコンをツールバーにドラッグ

Web Inspector パネル の detatch

Inspecting DOM

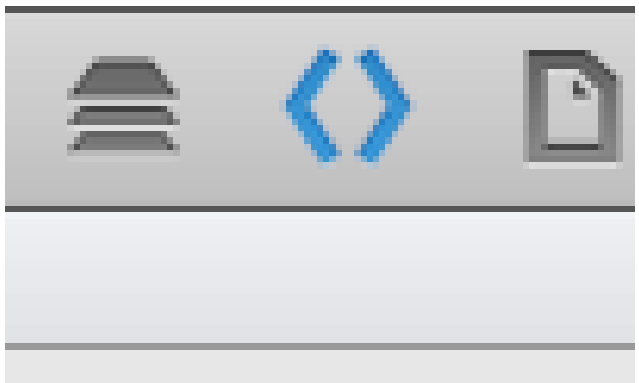
- DOM tree 表示
- DOM のノードにカーソルを置くと、対応するエレメントが色で表示
- ソースか DOM か切替可能

サイズ表示



- たとえば、HTML の canvas にマウス → 配置サイズが表示

DOM ノード情報



- 選択した HTML の要素（DOM のノード）の情報

Timelines

- ダウンロード時間の解析
- JavaScript のプロファイラ

Timelines

時計のアイコン（タイムライン）をクリック

- ダウンロード時間の解析 & JavaScript のプロファイラ
- ネットワーク要求
- レイアウトとレンダリング
- JavaScript とイベント

時間計測法：

- プロファイル（左下） → 記録ボタン

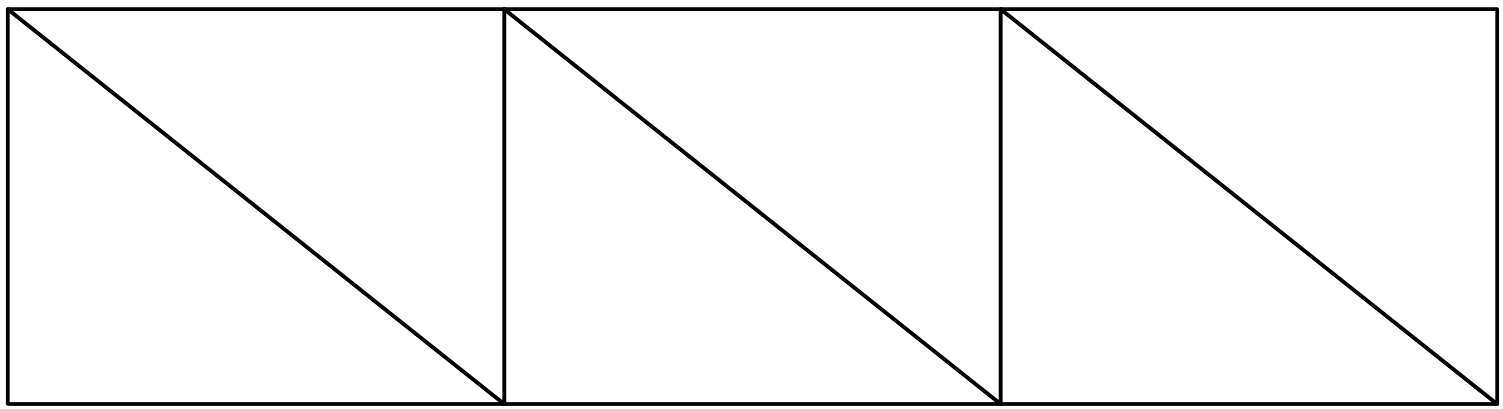
デバッガ

- ブレークポイント（左下）
- コールスタック（左上）

演習

演習

- 前回の演習課題の続き。下の図（あるいは他でも OK）を drawElements と TRIANLGE_STRIP を使って描け。
- ただし、今日は WebGL Inspector 等のツールも使ってみよう。



DOM

DOM とは

Document Object Model (DOM)

- HTML や XML 文書にアクセスするための API
  - プログラム（スクリプト言語）から（構造をもつ）文書にアクセスする
  - プログラム（スクリプト言語）から文書の構造、スタイル、内容を変更する

URL: `http://www.w3.org/DOM/DOMTR`

DOM の例

サンプル：document_object.html

Web Inspector（またはエディタ）でソースコードを見てみよう。

実行結果と見比べよう。

DOM ツリーを見てみよう。

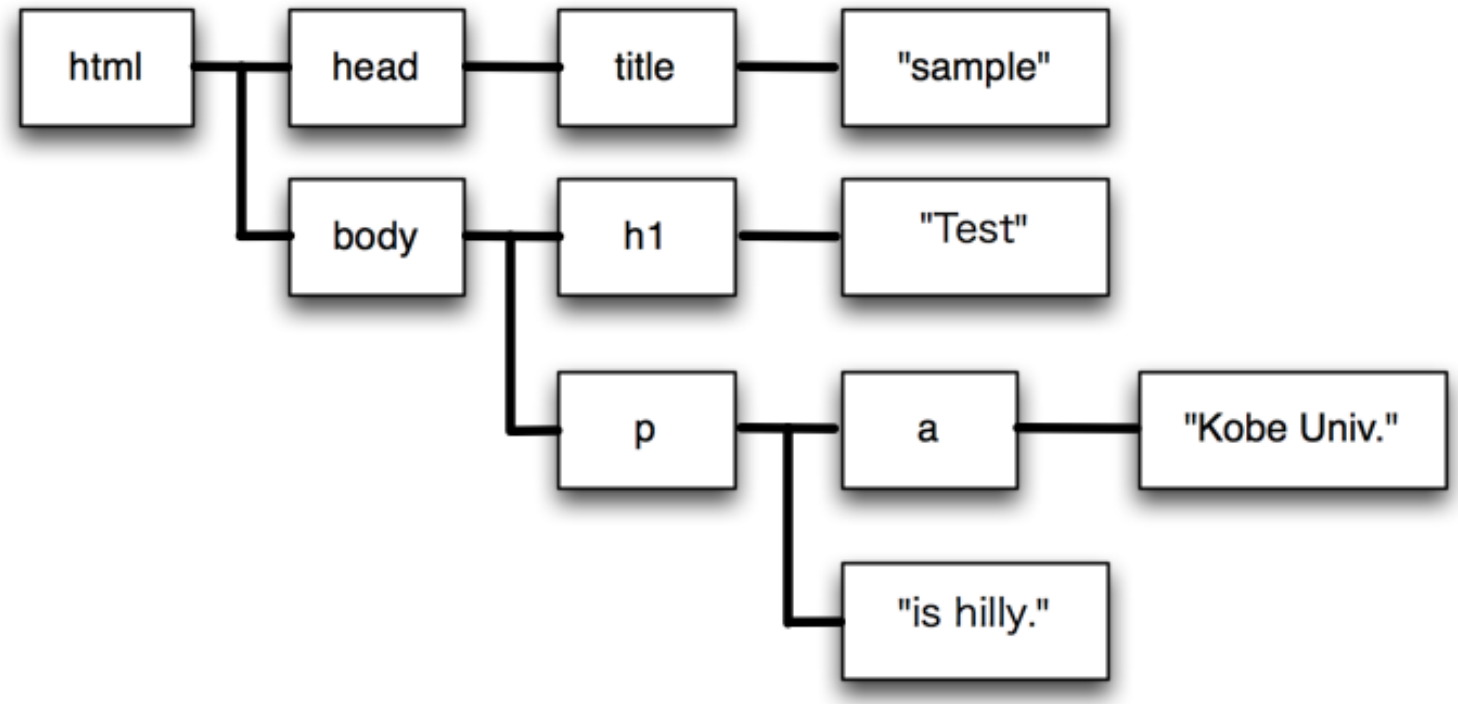
DOM node tree

階層構造を持った文書 = ノードの木構造とみる

```
<html>
  <head>
    <title>sample</title>
  </head>
  <body>
    <h1>Test</h1>
      <p><a href="http://www.kobe-u.ac.jp/">Kobe Univ.</a>
         is hilly.
      </p>
  </body>
</html>
```

DOM node tree

階層構造を持った文書 ＝ ノードの木構造とみる



DOM は各ノードへのアクセス手段を提供する。

ノードへの指定方法

- ツリーを root からたどる。例：
  document.firstChild.childNodes[1].childNodes[1].childNodes[1].nodeValue
  で “is hilly.” を取得。
- 要素名（タグ名）で指定する getElementsByTagName() メソッド
- 要素の id で指定する getElementById() メソッド

DOM を使うための準備

特になし

ブラウザに組み込まれている

例：

サンプルコード dom_sample_00.html

Web Inspector のソース表示（またはエディタ）でソースを見てみよう。

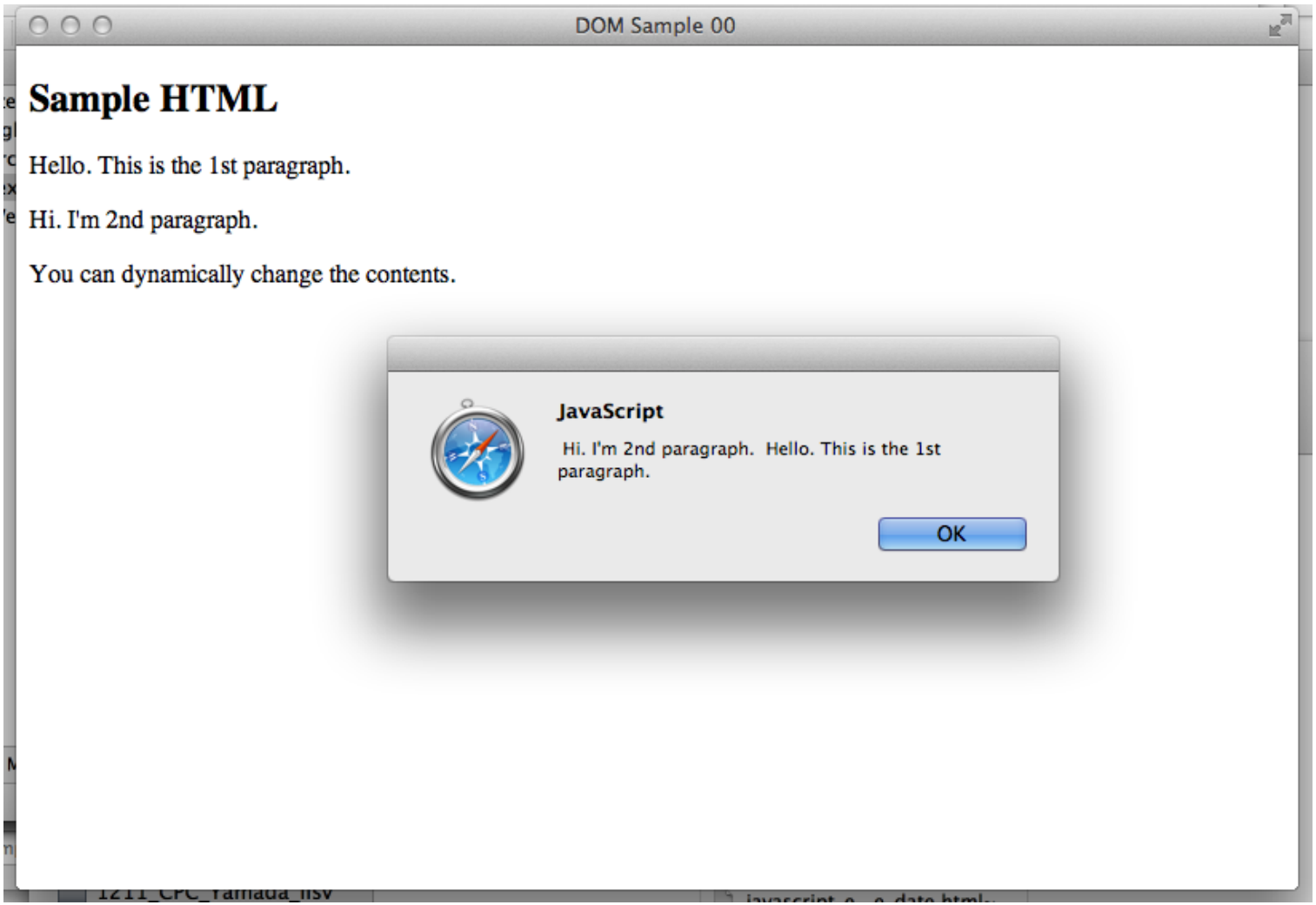
Web Inspector で DOM ツリーを表示してみよう。

DOMを使ったサンプルプログラム：dom_sample_00.html

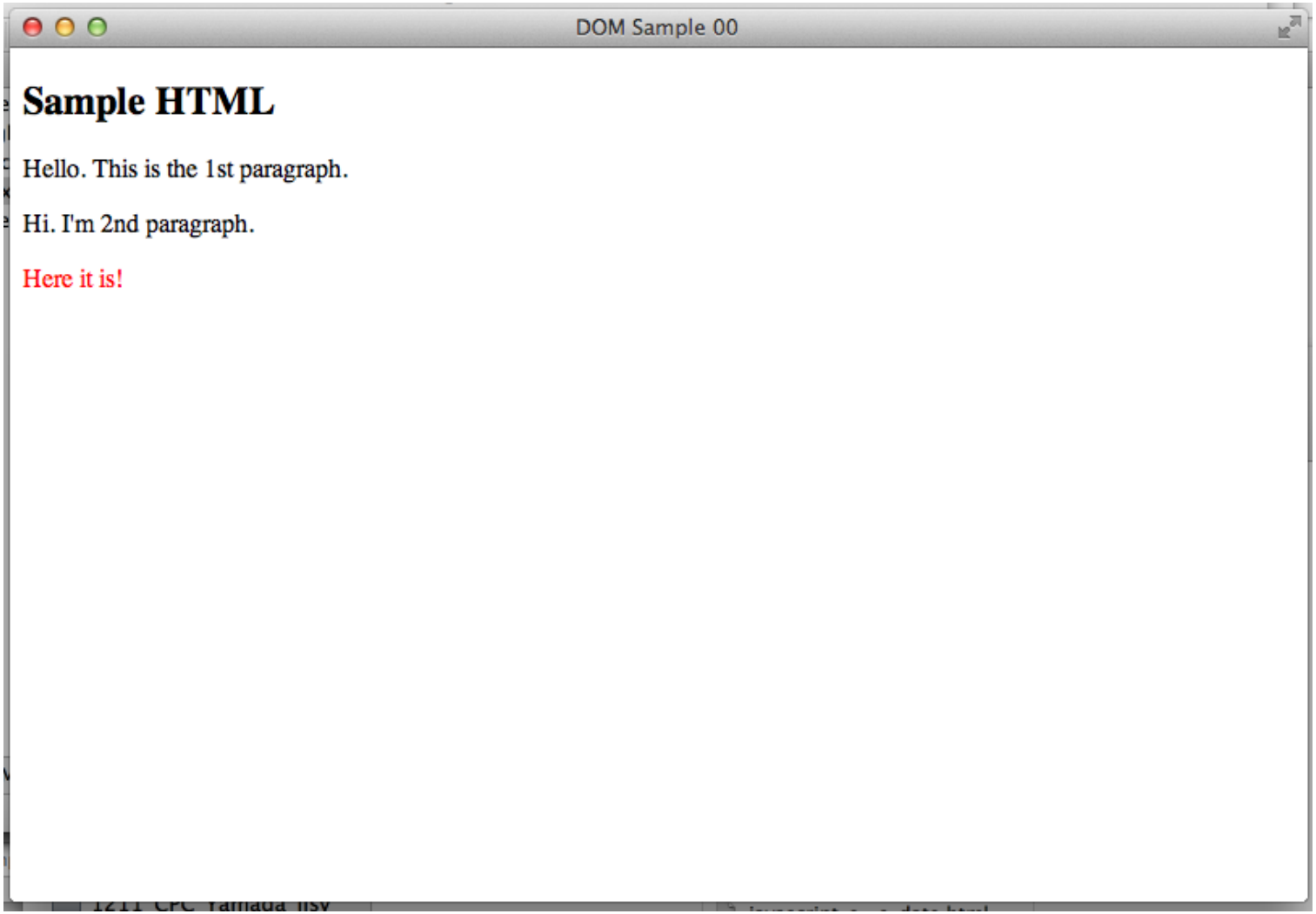
```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>DOM Sample 00</title>
<meta charset="utf-8">
<script type="text/javascript">
window.onload = function() {
   var paragraphs = document.getElementsByTagName("p");
   var p00 = paragraphs[0].textContent; // 1st paragraph
   var p01 = paragraphs[1].textContent; // 2nd paragraph
   alert(p01 + p00);
   document.getElementById("id000").style.color = "red";
   document.getElementById("id000").textContent = "Here it is!
       " ;
}
```

```
</head>

<body>

<h2> Sample HTML </h2>
<p> Hello. This is the 1st paragraph. </p>

<p> Hi. I'm 2nd paragraph. </p>

<div id="id000"> You can dynamically change the contents. </
    div>
</body>
</html>
```

読み込み結果


DOM Sample 00
Sample HTML
Hello. This is the 1st paragraph.
Hi. I'm 2nd paragraph.
You can dynamically change the contents.
JavaScript
Hi. I'm 2nd paragraph. Hello. This is the 1st paragraph.
OK

読み込み結果


DOM Sample 00
Sample HTML
Hello. This is the 1st paragraph.
Hi. I'm 2nd paragraph.
Here it is!

DOM を使ってシェーダソースコードをロードする

- これまでのサンプルプログラムでは、シェーダソースコードは JavaScript の文字列変数として直接書いていた。
- DOM API を使えばもっと読みやすくできる
- HTML の script タグとして書く：
  - 頂点シェーダ：
    ```
    <script id="shader-vs" type="x-shader/x-vertex">
    ```
  - フラグメントシェーダ：
    ```
    <script id="shader-fs" type="x-shader/x-fragment">
    ```
- この二つの script type をブラウザは知らないので無視する。
- ［注意］HTML のヘッダに書かずにファイルから読み出す方法もある（後述）。

webgl_sample_triangle_02_shader_from_DOM.html

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;

  void main() {
    gl_Position = vec4(aVertexPosition, 1.0);
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;

  void main() {
    gl_FragColor = vec4(0.2, 0.4, 0.6, 1.0);
  }
</script>
```

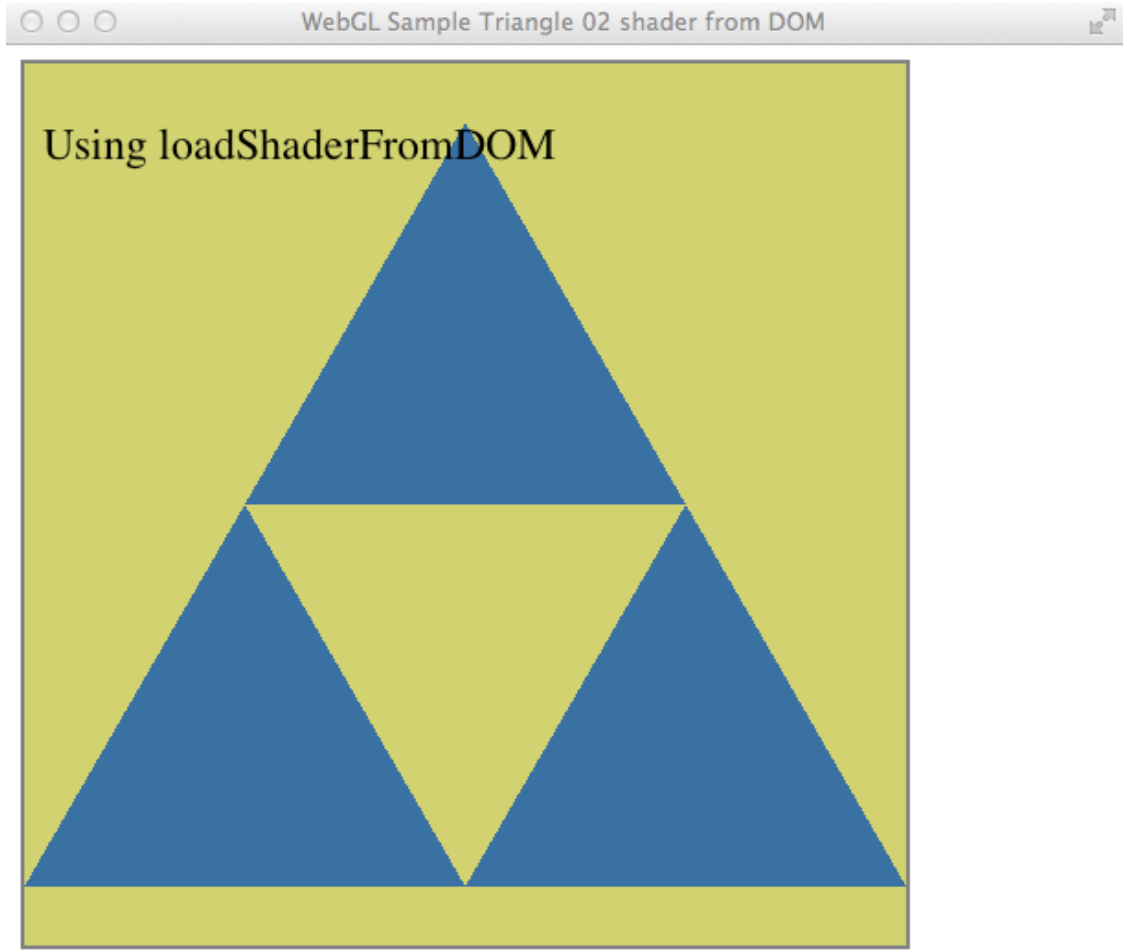
あとはDOMを使って読み込めばいい

```
function loadShaderFromDOM(id) {
  var shaderScript = document.getElementById(id);

  if (!shaderScript) {
    return null;
  }

  var shaderSource = "";
  var currentChild = shaderScript.firstChild;
  while (currentChild) {
    if (currentChild.nodeType == 3) { // 3 <= TEXT_NODE
      shaderSource += currentChild.textContent;
    }
    currentChild = currentChild.nextSibling;
  }
```

```
var shader;
if (shaderScript.type == "x-shader/x-fragment") {
  shader = gl.createShader(gl.FRAGMENT_SHADER);
} else if (shaderScript.type == "x-shader/x-vertex") {
  shader = gl.createShader(gl.VERTEX_SHADER);
} else {
  return null;
}

gl.shaderSource(shader, shaderSource);
gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
  alert(gl.getShaderInfoLog(shader));
```

```
    return null;
  }
  return shader;
}

function setupShaders() {

  var vertexShader = loadShaderFromDOM("shader-vs");
  var fragmentShader = loadShaderFromDOM("shader-fs");
```

実行結果


WebGL Sample Triangle 02 shader from DOM
Using loadShaderFromDOM

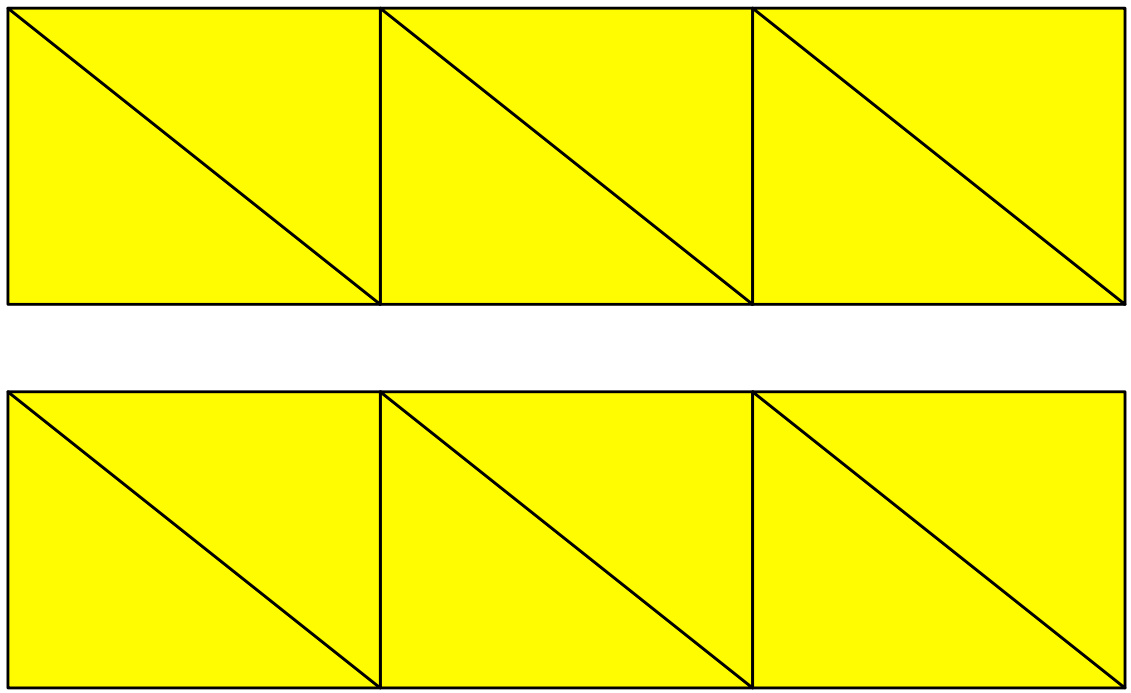
演習

演習

- WebGL で好きな図を描け。
- ただしシェーダプログラムは DOM API を使ってロードすること。

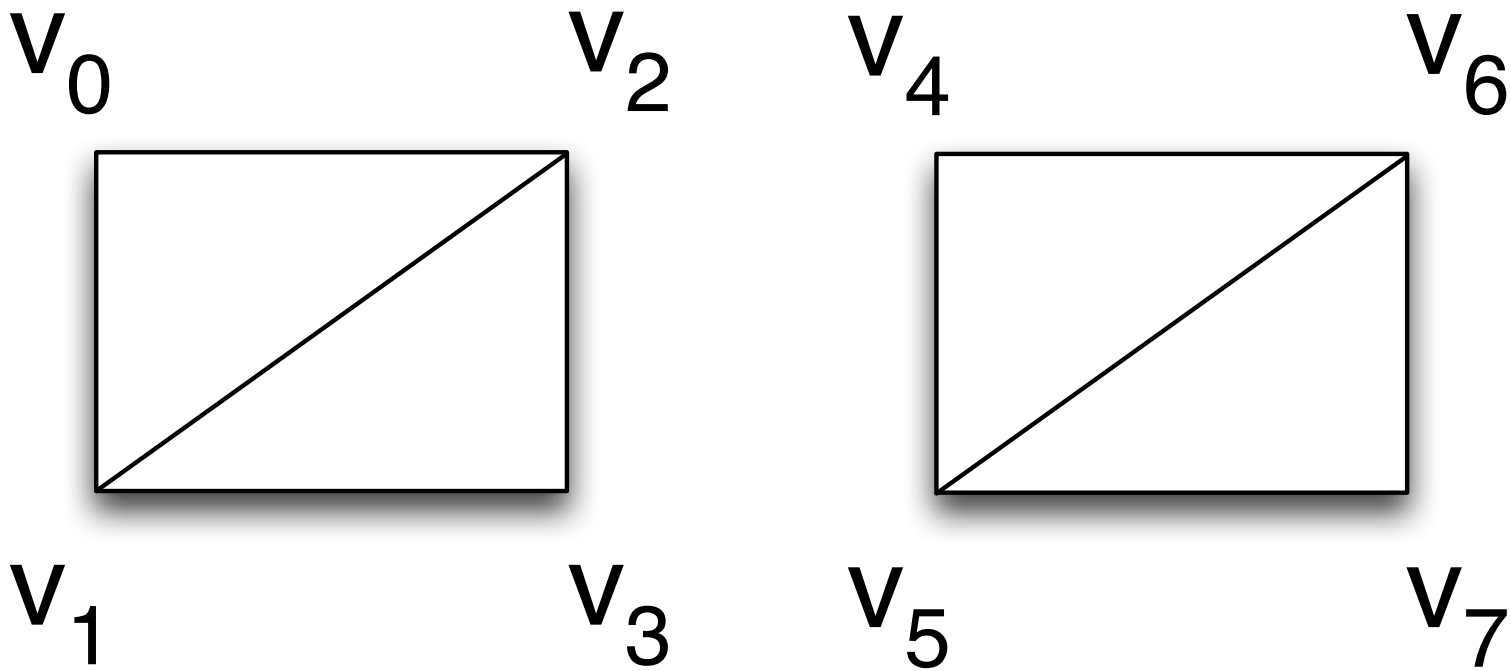
縮退三角形

二つの長方形



飛びのある TRIANGLE_STRIP

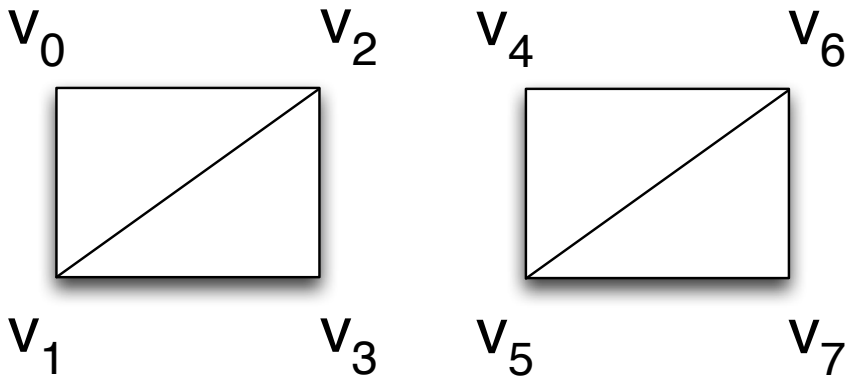
- gl.drawArrays() や gl.drawElements() を呼び出す回数は少ない方がいい
- 飛びのある TRIANGLE_STRIP



縮退三角形

- gl.TRIANGLE_STRIP で連続していないストリップを結合する
- ジャンプする部分に「縮退三角形」をおく
- ダミーの頂点をおく
- 縮退三角形＝面積ゼロの三角形 → GPU が検出、自動的に破棄

赤がダミーの頂点



0	1	2	3	3	4	4	5	6	7

(0,1,2)

(2,1,3)

(2,3,3)

(3,3,4)

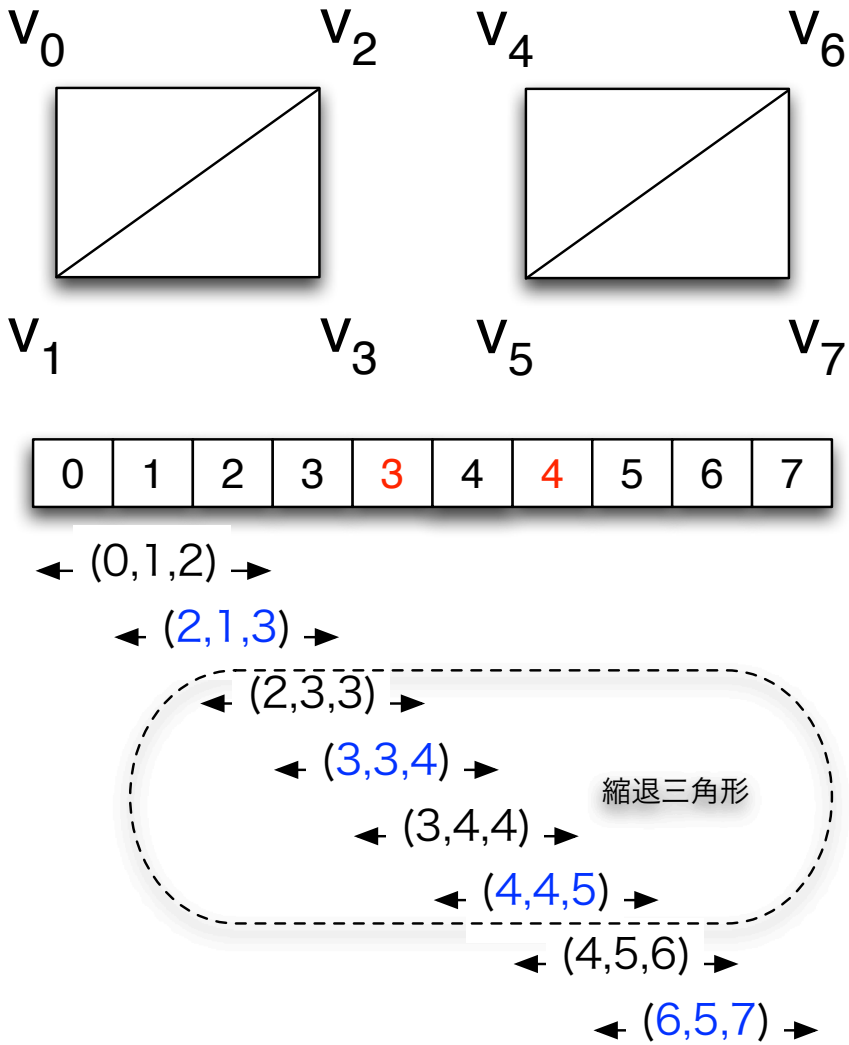
(3,4,4)

(4,4,5)

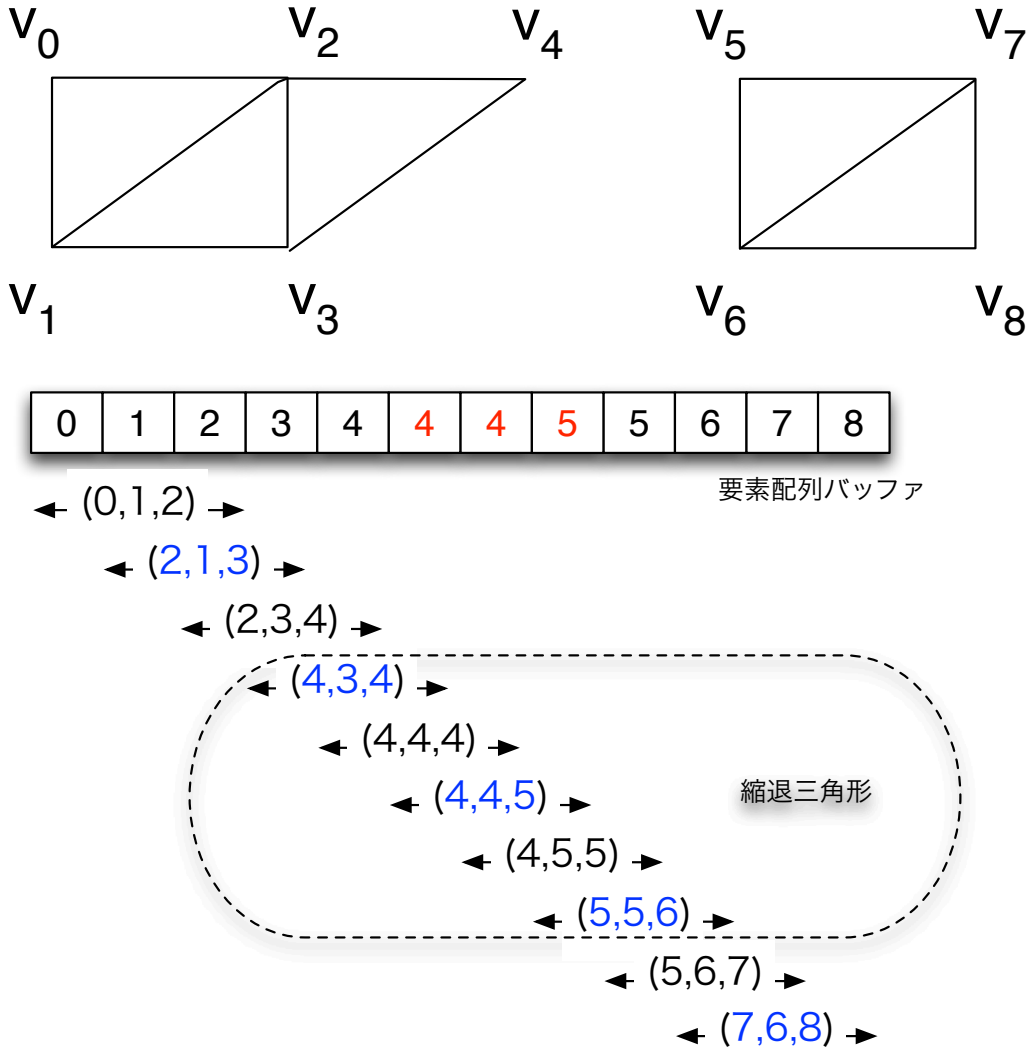
(4,5,6)

(6,5,7)

縮退三角形


v0
v2
v4
v6
v1
v3
v5
v7
0 1 2 3 3 4 4 5 6 7
(0,1,2)
(2,1,3)
(2,3,3)
(3,3,4)
(3,4,4)
(4,4,5)
(4,5,6)
(6,5,7)
縮退三角形

最初のストリップに三角形が奇数個ある場合



型付き配列

型付き配列

- JavaScript はバイナリーデータの処理が不得意であった
- 型付き配列（typed array）の導入

型付き配列

var buffer = new ArrayBuffer(8);

- 8 バイトのバッファ
- バッファ内のデータを直接操作できない
- → ビューの導入

var viewFloat32 = new Float32Array(buffer);

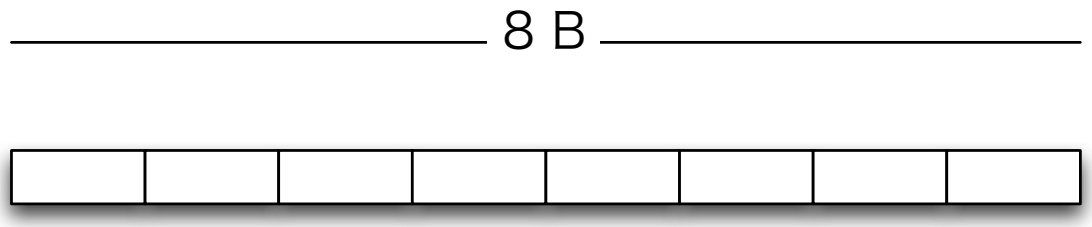
同じバッファに複数のビューを割り当てることができる。

var viewUint16 = new Uint16Array(buffer);

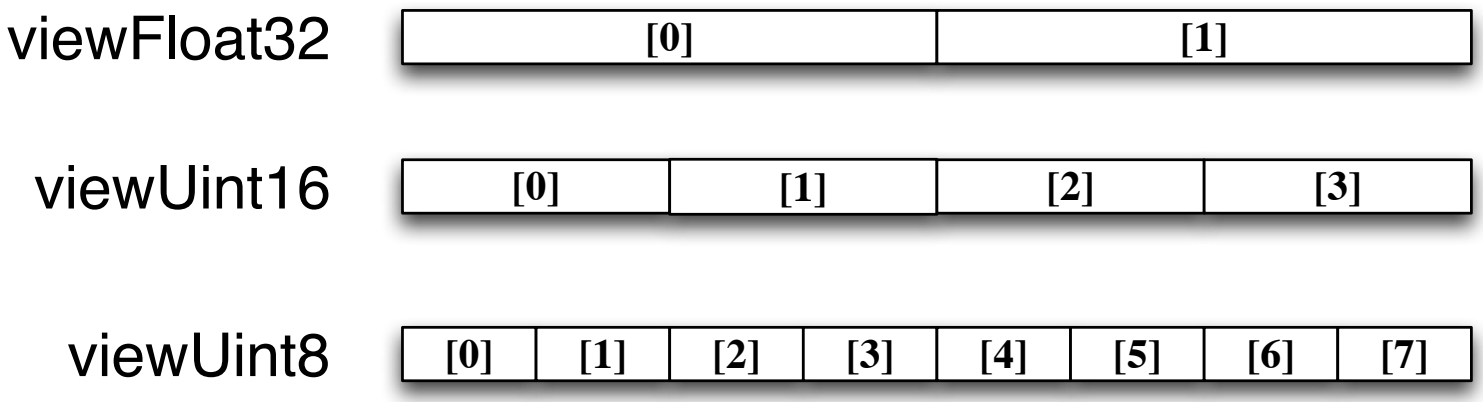
var viewUint8 = new Uint8Array(buffer);

型付き配列

バッファ



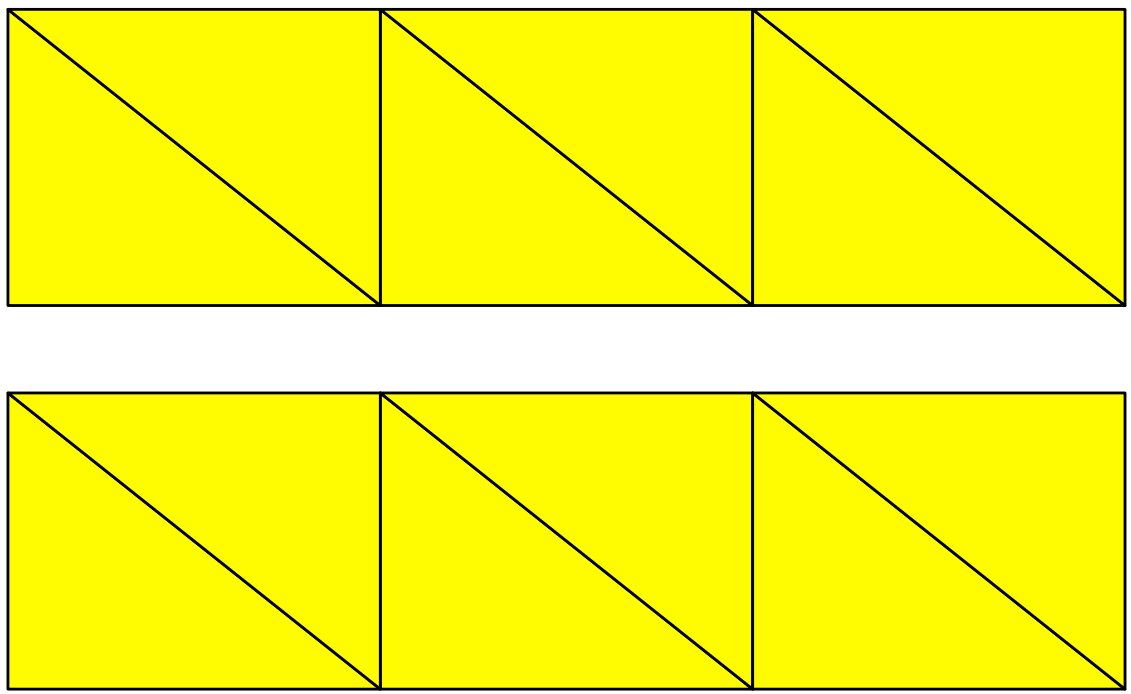
ビュー



演習 01

演習 01

- 二つの長方形を TRIANGLE_STRIP 一回の呼び出しで描け。
- 面の色は任意。三角形の枠線は描かなくてもよい。



頂点属性のインターリーブ

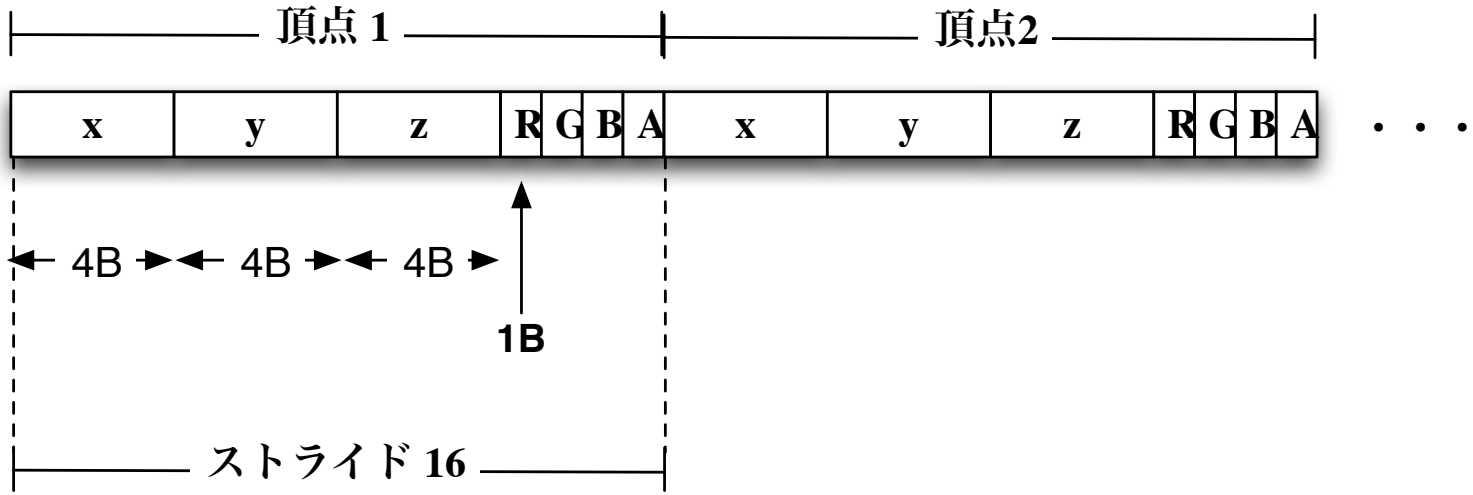
頂点属性（attribute）

- 座標
- 色
- 法線ベクトル
- テクスチャ座標

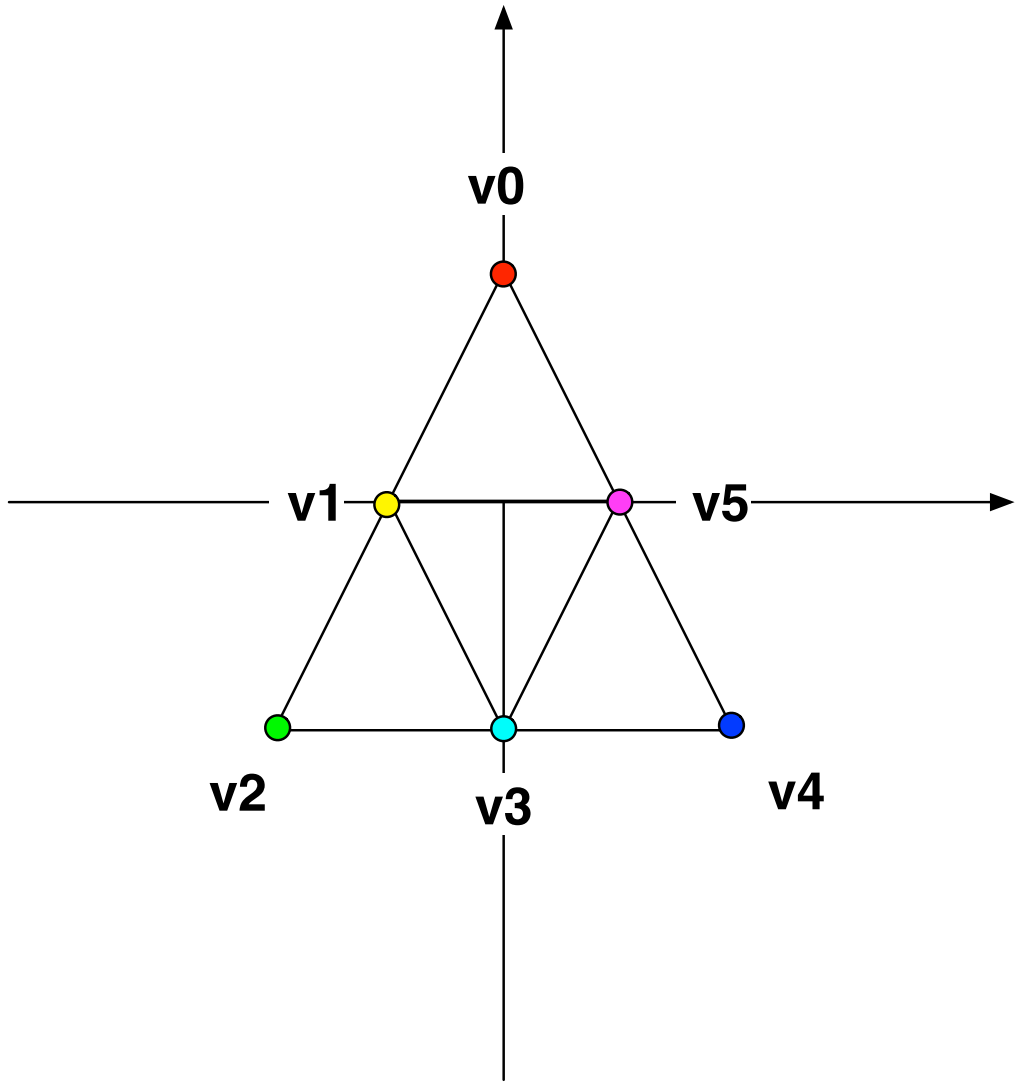
必要な属性を頂点毎にまとめて一つの大きなバッファに納める。

それぞれの属性を別々のバッファに納める方法もあるが、その方法は遅い。

データ (位置と色属性) のインターリーブ



例題



webgl_sample_triangle_03.html

頂点シェーダ

```
<script id="shader-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexPosition;
   attribute vec4 aVertexColor;
   varying vec4 vColor;

   void main() {
     vColor = aVertexColor;
     gl_Position = vec4(aVertexPosition, 1.0);
   }
</script>
```

webgl_sample_triangle_03.html

フラグメントシェーダ

```
<script id="shader-fs" type="x-shader/x-fragment">
   precision mediump float;

   varying vec4 vColor;
   void main() {
      gl_FragColor = vColor;
   }
</script>
```

webgl_sample_triangle_03.html

シェーダのセットアップ

```
function setupShaders() {

  var vertexShader = loadShaderFromDOM("shader-vs");
  var fragmentShader = loadShaderFromDOM("shader-fs");


  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS))
      {
    alert("Failed to setup shader.");
  }
```

```
  gl.useProgram(shaderProgram);

  shaderProgram.vertexPositionAttribute =
    gl.getAttribLocation(shaderProgram, "aVertexPosition");

  shaderProgram.vertexColorAttribute =
    gl.getAttribLocation(shaderProgram, "aVertexColor");

  gl.enableVertexAttribArray(shaderProgram.
      vertexPositionAttribute);
  gl.enableVertexAttribArray(shaderProgram.
      vertexColorAttribute);
}
```

webgl_sample_triangle_03.html

JavaScript 配列でデータを用意

```
var triangleVertices = [
 // (    x          y          z   )(   r     g     b     a   )
     0.000000,  0.866025,  0.0,   255,    0,    0,  255,  // red
    -0.500000,  0.000000,  0.0,   255,  255,    0,  255,  // yellow
    -1.000000, -0.866025,  0.0,     0,  255,    0,  255,  // green
     0.000000, -0.866025,  0.0,     0,  255,  255,  255,  // cyan
     1.000000, -0.866025,  0.0,     0,    0,  255,  255,  // blue
     0.500000,  0.000000,  0.0,   255,    0,  255,  255   // magenda
];
```

webgl_sample_triangle_03.html

あとは言葉で説明

バッファのセットアップ

1. バッファのアロケート （new ArrayBuffer）
2. バッファに Float32Array ビューをマップ（位置座標用）
3. 同じバッファに Uinit8Array ビューをマップ（色用）
4. JavaScript 配列の値をバッファにロード

webgl_sample_triangle_03.html

(続き)

要素バッファのセットアップ

1. 要素バッファのセットアップ
2. 要素バッファを ELEMENT_ARRAY_BUFFER にバインド
3. 要素バッファに要素のデータをロード

webgl_sample_triangle_03.html

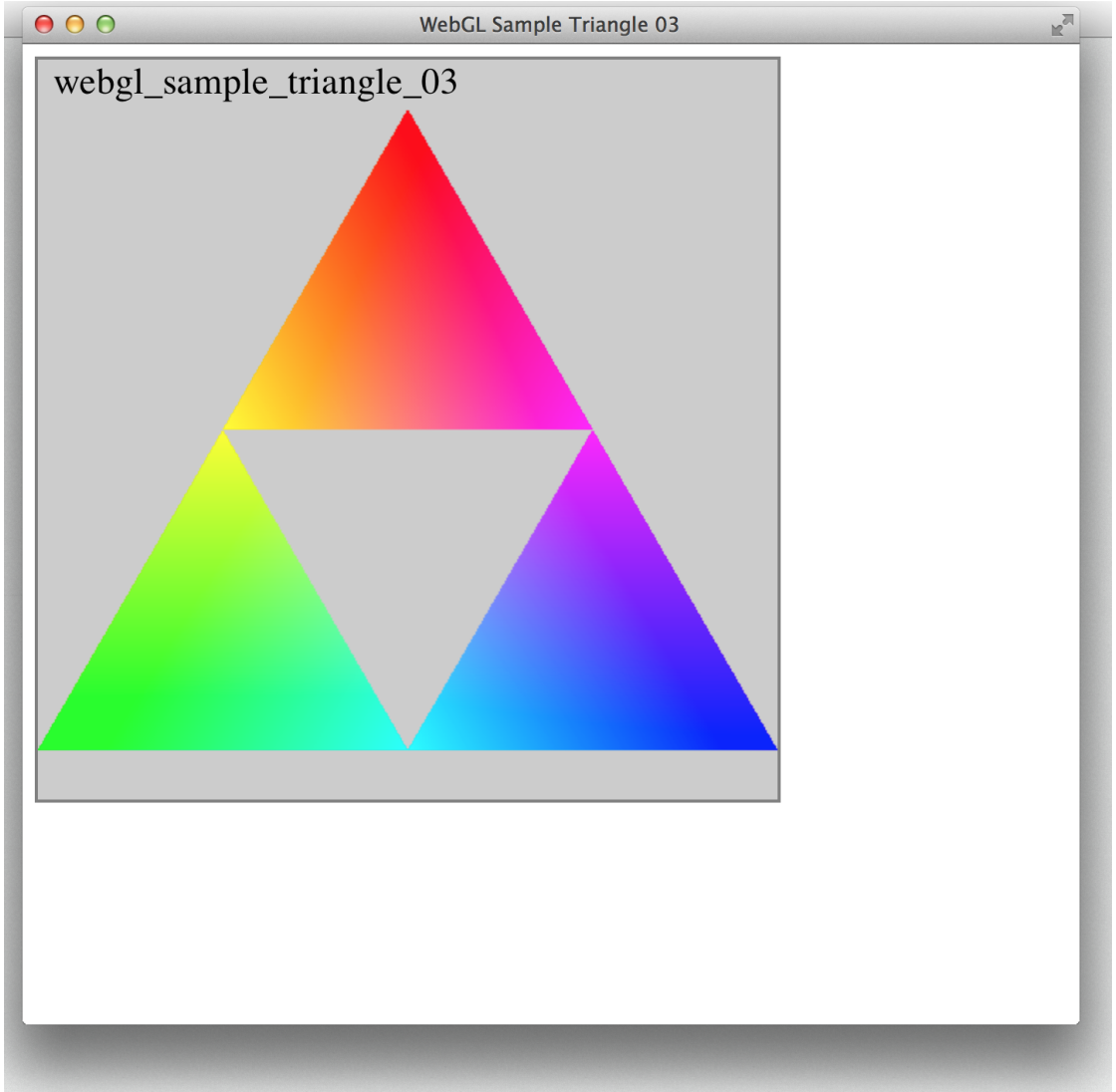
（続き）

描画

1. バッファを ARRAY_BUFFER にバインド （bindBuffer）
2. バッファ中の位置データの配置を記述 （vertexAtrribPointer）
3. バッファ中の色データの配置を記述 （vertexAtrribPointer）
4. 描画 （drawElements）

以上（詳しくはソースコード参照）

webgl_sample_triangle_03.html



演習 02

演習 02

- 頂点データのインターリーブの練習をしよう。
- 対象は任意。

定数頂点データ

定数頂点データ

複数の頂点で属性が共通な場合（例えば複数の頂点で色が同じなど）

定数頂点データ（constant vertex data）を指定する

方法は簡単

1. その属性の汎用属性インデックスを無効化する
   （disableVertexAttribArray）
2. 定数属性データを指定する（vertexAttrib4f 等）
   - gl.vertexAttrib4f(index, v0, v1, v2, v3)
   - gl.vertexAttrib3f(index, v0, v1, v2)
   - gl.vertexAttrib2f(index, v0, v1)
   - gl.vertexAttrib1f(index, v0)

頂点配列と定数頂点データが混在するとき

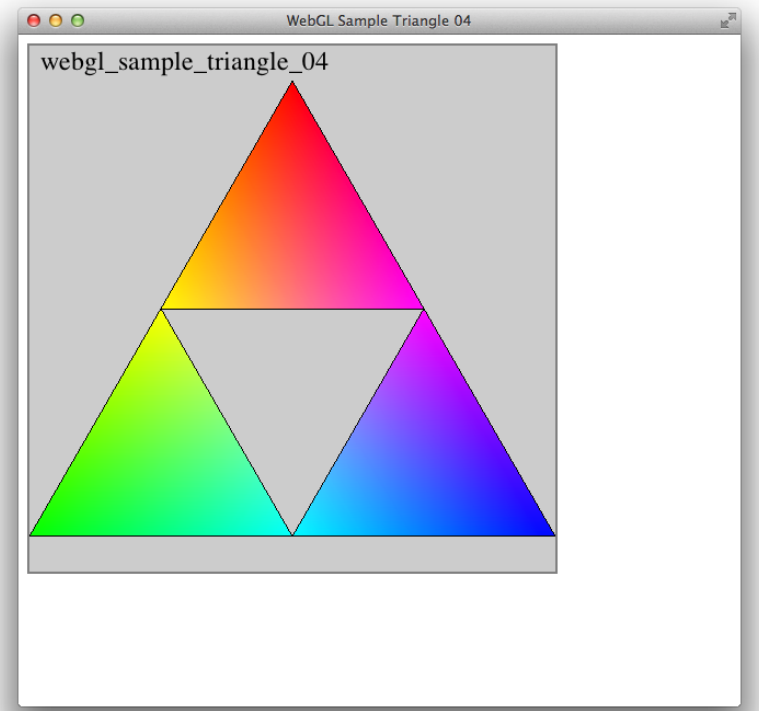
gl.enableVertexAttribArray がコールされている → 頂点配列から読む

gl.disableVertexAttribArray がコールされている → gl.vertexAttrib() で設定された値を使う

例題：webgl_sample_triangle_04.html

面の色：頂点配列

線の色：定数頂点データ（黒）



演習兼復習

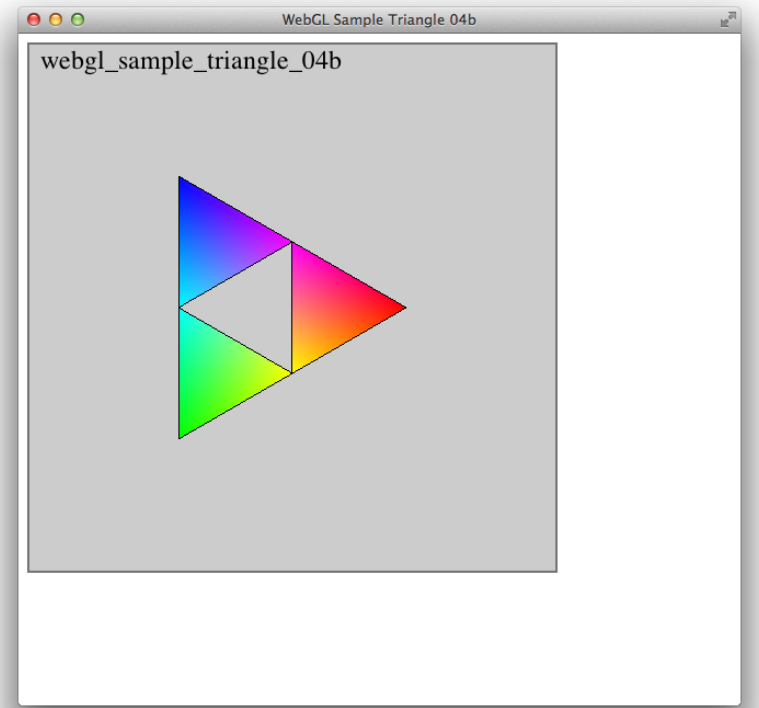
ちょっと復習：シェーダ内部での計算

頂点シェーダ内部での計算

webgl_sample_triangle_04b.html

$(x, y, z, 1) \Longrightarrow (y/2, x/2, z, 1)$ （縮小して裏返し）

裏面を表示（表の面をカリング）



演習 03

演習 03

- 定数頂点データの練習をしよう
- 対象は任意

glMatrix の紹介

行列ライブラリ

- WebGL では JavaScript で行列を扱う必要がある
- モデル変換、ビュー変換、射影変換
- 4x4 または 3x3 の行列演算
- JavaScript で行列を扱うライブラリは複数ある
- その中で、ここでは glMatrix を使う

glMatrix.js

http://glmatrix.net

最新バージョンは 2.0

この講義ではバージョン 0.9.6 を使用

以下の URL からソースコードを入手

http://www.research.kobe-u.ac.jp/csi-viz/members/kageyama/lec

自分の WebGL ソースコードと同じディレクトリに glMatrix.js という名前で保存

glMatrix.js Copyright

```
/*
 * glMatrix.js - High performance matrix and vector
     operations for WebGL
 * version 0.9.6
 */

/*
 * Copyright (c) 2011 Brandon Jones
 *
 * This software is provided 'as-is', without any express or
     implied
 * warranty. In no event will the authors be held liable for
     any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for
     any purpose,
 * including commercial applications, and to alter it and
     redistribute it
 * freely, subject to the following restrictions:
```

glMatrix のベクトルと行列

vec3, vec4, mat3, mat4, quat4

全て 型付き配列 Float32Array

glMatrix の関数

vec3.{create/set/add/subtract/negate/scale/normalize/cross/length/
dot/direction/lerp/str}

mat3.{create/set/identity/transpose/toMat4/str}

mat4.{create/set/identity/transpose/determinant/inverse/
toRotationMat/toMat3/toInverseMat3/multiply/multiplyVec3/
multiplyVec4/translate/scale/rotate/rotateX/rotateY/rotateZ/
frustum/perspective/ortho/lookAt/str}

quat4.{create/set/calculateW/inverse/length/normalize/multiply/
multiplyVec3/toMat3/toMat4/slerp/str}

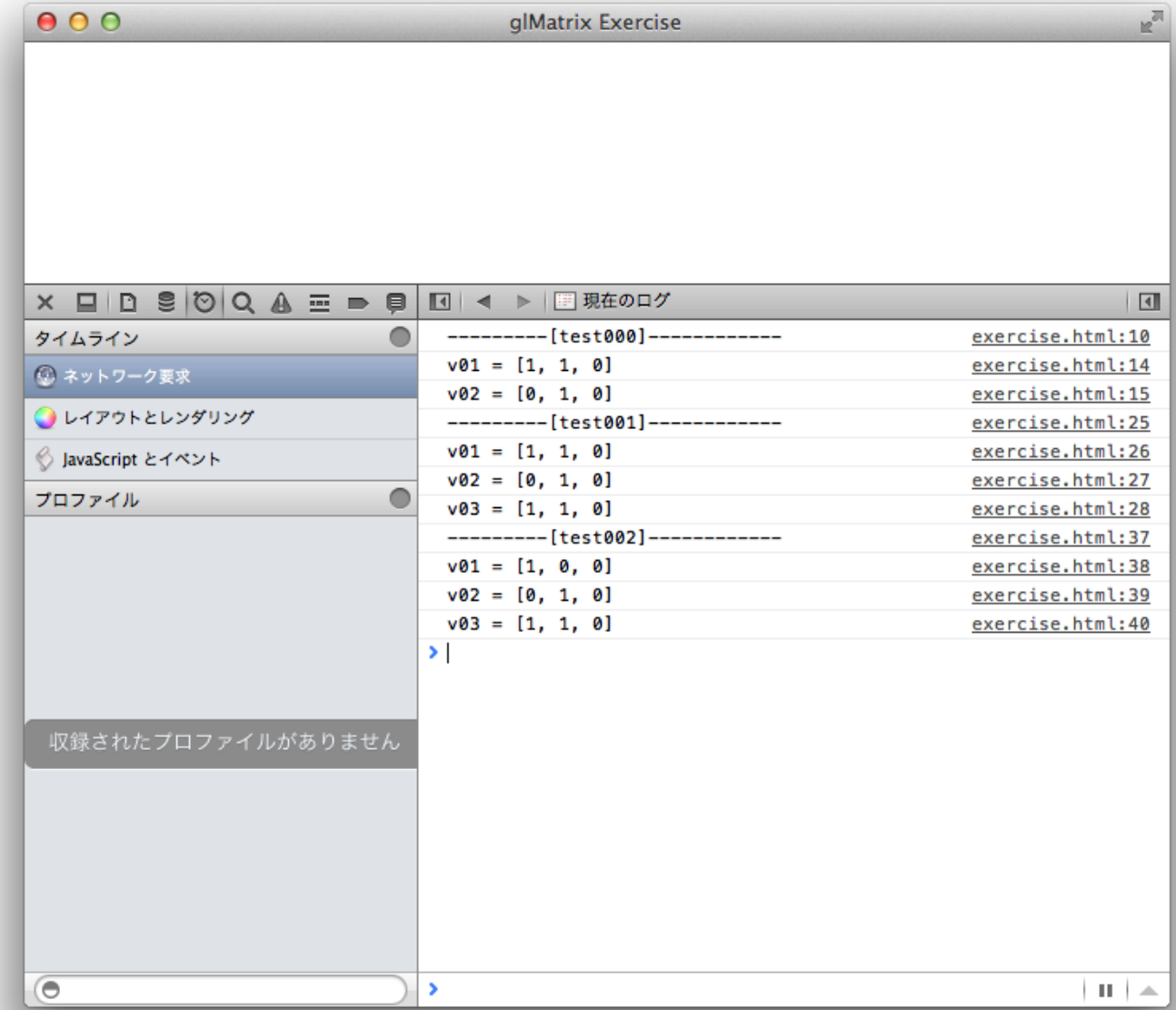
例： glMatrix_exercise_00.html

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>glMatrix Exercise</title>
<meta charset="utf-8">
<script type="text/javascript" src="glMatrix.js"></script>
<script type="text/javascript">

function test000() {
  console.log("----------[test000]----------");
  var v01 = vec3.create([1.0, 0.0, 0.0]);
  var v02 = vec3.create([0.0, 1.0, 0.0]);
  vec3.add(v01,v02);
  console.log("v01 = " + vec3.str(v01));
  console.log("v02 = " + vec3.str(v02));
```

```
  test001();
}

function test001() {
  var v01 = vec3.create([1.0, 0.0, 0.0]);
  var v02 = vec3.create([0.0, 1.0, 0.0]);
  var v03 = vec3.create();

  v03 = vec3.add(v01,v02);
  console.log("----------[test001]----------");
  console.log("v01 = " + vec3.str(v01));
  console.log("v02 = " + vec3.str(v02));
  console.log("v03 = " + vec3.str(v03));
  test002();
}
```

```
function test002() {
  var v01 = vec3.create([1.0, 0.0, 0.0]);
  var v02 = vec3.create([0.0, 1.0, 0.0]);
  var v03 = vec3.create();
  vec3.add(v01,v02,v03);
  console.log("----------[test002]----------");
  console.log("v01 = " + vec3.str(v01));
  console.log("v02 = " + vec3.str(v02));
  console.log("v03 = " + vec3.str(v03));
}

</script>
</head>
```

glMatrix_exercise_00.html：結果



glMatrix における引数のルール

ベクトルの足し算

第一引数が更新される

```
vec3.add(v01, v02);          // v01 + v02 ⇒ v01
```

左辺と第一引数が更新される

```
v03 = vec3.add(v01, v02);          // v01 + v02 ⇒ v01 & v03
```

第一引数を更新したくない場合はこうする：

vec3.add(v01, v02, v03);　　// v01 + v02 ⇒ v03

ベクトルの内積と外積
glMatrix_exercise_01.html

ベクトルの内積
glMatrix_exercise_01.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var v02 = vec3.create([1.0, 1.0, -1.0]);
var ans = vec3.dot(v01, v02);          // ans = 0
```

ベクトルの外積 (a)
glMatrix_exercise_01.html

```
var v01 = vec3.create([1.0, 0.0, 0.0]);
var v02 = vec3.create([0.0, 1.0, 0.0]);
var v03 = vec3.create();
v03 = vec3.cross(v01, v02);        // v01 は更新される（=v03）
```

ベクトルの外積 (b)
glMatrix_exercise_01.html

```
var v01 = vec3.create([1.0, 0.0, 0.0]);
var v02 = vec3.create([0.0, 1.0, 0.0]);
var v03 = vec3.create();
vec3.cross(v01, v02, v03);        // v01 は更新されない。
```

演習

演習

- glMatrix_exercise_00.html と
- glMatrix_exercise_01.html を動かしてみよう。

ブラウザでコンソールを開くこと。

ベクトルの規格化 (a)
glMatrix_exercise_02.html

長さを 1 にする

```
var v01 = vec3.create([3.0, 4.0, 0.0]);
vec3.normalize(v01);
```

ベクトルの規格化 (b)
glMatrix_exercise_02.html

```
var v01 = vec3.create([3.0, 4.0, 0.0]);
var v02 = vec3.create();
vec3.normalize(v01, v02);
        // v01 を規格化したものが v02 に入る。
        // v01 は更新されない。
```

ベクトルの大きさ
glMatrix_exercise_03.html

```
var v01 = vec3.create([3.0, 4.0, 12.0]);
var amp = vec3.length(v01);
```

ベクトルのコピー

glMatrix_exercise_03.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var v02 = vec3.create();
vec3.set(v01,v02)
        // v01 の 3 成分全てが v02 にコピーされる
```

ベクトルの符号反転 (a)
glMatrix_exercise_03.html

```
vec3.negate(v01);
        // v01 の符号を変える
```

ベクトルの符号反転 (b)
glMatrix_exercise_03.html

```
var v01 = vec3.create([3.0, 4.0, .0]);
var v02 = vec3.create();
vec3.negate(v01, v02);
        // v01 の符号を変えたものが v02 に入る。
        // v01 は変わらない。
```

ベクトルの引き算
glMatrix_exercise_03.html

add と同じ

vec3.add ⇒ vec3.subtract

とすればいい。

add と同様に 3 通りの書き方がある。

ベクトルの拡大縮小 (a)
glMatrix_exercise_03.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var s = 0.1;
vec3.scale(v01, s);
        // v01 が s 倍される。
```

ベクトルの拡大縮小 (b)
glMatrix_exercise_03.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var v02 = vec3.create();
var s = 0.1;
vec3.scale(v01, s, v02);
        // v01 を s 倍したものが v02 に入る。
        // v01 は変わらず。
```

二つの点を結ぶ単位方向ベクトル (a)
glMatrix_exercise_03.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var v02 = vec3.create([1.0, 2.0, -2.0]);
vec3.direction(v01, v02);
        // 点 v01 から点 v02 に向かう方向の単位ベクトルが v01 に入る。
```

二つの点を結ぶ単位方向ベクトル (b)
glMatrix_exercise_03.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var v02 = vec3.create([1.0, 2.0, -2.0]);
var v03 = vec3.create();
vec3.direction(v01, v02, v03);
        // 点 v01 から点 v02 に向かう方向の単位ベクトルが v03 に入る。
        // v01 は変わらず。
```

二点間の線形補間 (a)
glMatrix_exercise_03.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var v02 = vec3.create([1.0, 2.0, -2.0]);
var s = 0.6;
vec3.lerp(v01, v02, s);
      // 点 v01 と点 v02 を結ぶ線分を比率 s で線形補間した位置ベクトル
      // 結果は v01 に入る。
```

二点間の線形補間 (b)
glMatrix_exercise_03.html

```
var v01 = vec3.create([1.0, 2.0, 3.0]);
var v02 = vec3.create([1.0, 2.0, -2.0]);
var v03 = vec3.create();
var s = 0.6;
vec3.lerp(v01, v02, s, v03);
      // 点 v01 と点 v02 を結ぶ線分を比率 s で線形補間した位置ベクトル
      // 結果は v03 に入る。v01 は変わらず。
```

4x4 単位行列
glMatrix_exercise_04.html

単位行列（もっと便利な作り方は次のページ）var m01 = mat4.create([
1.0, 0.0, 0.0, 0.0, // 1st column
0.0, 1.0, 0.0, 0.0, // 2nd column
0.0, 0.0, 1.0, 0.0, // 3rd column
0.0, 0.0, 0.0, 1.0 // 4th column
]);

4x4 単位単位行列
glMatrix_exercise_04.html

単位行列を作るにはこうするのが簡単

```
var m01 = mat4.create();
mat4.identity(m01);
```

転置 (a)
glMatrix_exercise_04.html

```
var m01 = mat4.create([
      1.0, 1.0, 1.0, 1.0, // 1st column
      0.0, 1.0, 1.0, 1.0, // 2nd column
      0.0, 0.0, 1.0, 1.0, // 3rd column
      0.0, 0.0, 0.0, 1.0 // 4th column
      ]);
mat4.transpose(m01);
      // m01 の転置
```

転置 (b)
glMatrix_exercise_04.html

```
var m01 = mat4.create([
        1.0, 1.0, 1.0, 1.0, // 1st column
        0.0, 1.0, 1.0, 1.0, // 2nd column
        0.0, 0.0, 1.0, 1.0, // 3rd column
        0.0, 0.0, 0.0, 1.0 // 4th column
        ]);
var m02 = mat4.create();
mat4.transpose(m01,m02);
        // m01 の転置が m02 に入る。m01 は変わらず。
```

行列

行列成分の配置は列優先であることに注意

```
mat4.create([
        a00, a10, a20, a30, // 第 1 列
        a01, a11, a21, a31, // 第 2 列
        a02, a12, a22, a32, // 第 3 列
        a03, a13, a23, a33 // 第 4 列
        ]);
```

行列のかけ算

var m03 = mat4.multiply(m01, m02)

行列 m01 と行列 m02 をかけた結果が m03 に入る。

行列 m01 も書き換わる（=m03）

行列 m01 を変更したくない場合は

mat4.multiply(m01, m02, m03)

とする。

行列式
glMatrix_exercise_04.html

```
var m01 = mat4.create([
        1.0, 1.0, 2.0, 1.0, // 1st column
        2.0, 1.0, 2.0, 2.0, // 2nd column
        0.0, 2.0, 3.0, 3.0, // 3rd column
        0.0, 3.0, 4.0, 4.0 // 4th column
        ]);
console.log(" det(m01) = " + mat4.determinant(m01));      // = -2
```

逆行列 (a)
glMatrix_exercise_04.html

```
var m01 = mat4.create([
      1.0, 1.0, 2.0, 1.0, // 1st column
      2.0, 1.0, 2.0, 2.0, // 2nd column
      0.0, 2.0, 3.0, 3.0, // 3rd column
      0.0, 3.0, 4.0, 4.0 // 4th column
      ]);
mat4.inverse(m01);
```

m01 が自分の逆行列に置き換わる

逆行列 (b)
glMatrix_exercise_04.html

```
var m01 = mat4.create([
      1.0, 1.0, 2.0, 1.0, // 1st column
      2.0, 1.0, 2.0, 2.0, // 2nd column
      0.0, 2.0, 3.0, 3.0, // 3rd column
      0.0, 3.0, 4.0, 4.0 // 4th column
      ]);
var m02 = mat4.inverse(m01);
```

m01 が自分の逆行列に置き換わる。m02 に同じものが入る。

逆行列 (c)
glMatrix_exercise_04.html

```
var m01 = mat4.create([
      1.0, 1.0, 2.0, 1.0, // 1st column
      2.0, 1.0, 2.0, 2.0, // 2nd column
      0.0, 2.0, 3.0, 3.0, // 3rd column
      0.0, 3.0, 4.0, 4.0 // 4th column
      ]);
var m02 = mat4.create();
mat4.inverse(m01,m02);     //m01 が自分の逆になる。m02 も同じもの
var m03 = mat4.create(); mat4.multiply(m01,m02,m03)   // m01 は不変
```

3x3 から 4x4 行列へ (a)
glMatrix_exercise_05.html

```
var m01 = mat3.create([
      1.0, 2.0, 3.0, // 1st column
      0.0, 1.0, 2.0, // 2nd column
      0.0, 0.0, 1.0 // 3rd column
      ]);
var m02 = mat3.toMat4(m01);
```

3x3 から 4x4 行列へ (b)
glMatrix_exercise_05.html

左上の 3 行 3 列に要素をコピーする。それ以外は、4 行 4 列目は 1、それ以外は 0 でうめる。

```
var m01 = mat3.create([
        1.0, 2.0, 3.0, // 1st column
        0.0, 1.0, 2.0, // 2nd column
        0.0, 0.0, 1.0 // 3rd column
        ]);
var m02 = mat4.create();
mat3.toMat4(m01,m02)
            // m01 = [1, 2, 3, 0, 1, 2, 0, 0, 1]
            // m02 = [1, 2, 3, 0, 0, 1, 2, 0, 0, 0, 1, 0, 0, 0, 0, 1]
```

4x4 から 3x3 行列へ (a)
glMatrix_exercise_05.html

左上の 3x3 成分をとりだす。

```
var m01 = mat4.create([
      1.0, 1.0, 2.0, 1.0, // 1st column
      2.0, 1.0, 2.0, 2.0, // 2nd column
      0.0, 2.0, 3.0, 3.0, // 3rd column
      0.0, 3.0, 4.0, 4.0 // 4th column
      ]);
var m02 = mat4.toMat3(m01);

    // m01 = [1, 1, 2, 1, 2, 1, 2, 2, 0, 2, 3, 3, 0, 3, 4, 4]
    // m02 = [1, 1, 2, 2, 1, 2, 0, 2, 3]
```

4x4 から 3x3 行列へ (b)
glMatrix_exercise_05.html

左上の 3x3 成分をとりだす。

```
var m01 = mat4.create([
        1.0, 1.0, 2.0, 1.0, // 1st column
        2.0, 1.0, 2.0, 2.0, // 2nd column
        0.0, 2.0, 3.0, 3.0, // 3rd column
        0.0, 3.0, 4.0, 4.0 // 4th column
        ]);
var m02 = mat4.create();
mat4.toMat3(m01, m02);

    // m01 = [1, 1, 2, 1, 2, 1, 2, 2, 0, 2, 3, 3, 0, 3, 4, 4]
    // m02 = [1, 1, 2, 2, 1, 2, 0, 2, 3]
```

3x3 行列のコピー

mat3.set()

使い方は vec3.set() と同じ

これまでの説明で以下の引数入出力パターンには十分なじんだと思うので、以後、並記はしない。

mat4.operator(src_and_dest, param)

dest = mat4.operator(src_and_dest, param)

mat4.operator(src, param, dest)

ただし、次の multiplyVec3 は

mat4.multiplyVec3(param, src_and_dest);

mat4.multiplyVec3(param, src, dest);

というパターンである。

4x4 行列と vec3 の積 (a)
glMatrix_exercise_05.html

```
var M = mat4.create([
1.0, 0.0, 0.0, 0.0,
2.0, 0.0, 1.0, 0.0,
3.0, 1.0, 0.0, 0.0,
4.0, 0.0, 0.0, 1.0
]);
var v01 = vec3.create([1, 0, 0]);
      // [1, 0, 0, 1] という vec4 に変換されてから、積をとる
mat4.multiplyVec3(M,v01);    //積の結果の vec4 の第 4 成分は捨てられる
      // v01 = [5, 0, 0]
```

4x4 行列と vec3 の積 (b)
glMatrix_exercise_05.html

```
var m01 = mat4.create();
mat4.identity(m01);
var v01 = vec3.create([1.0, 0.0, 0.0]);
var v02 = vec3.create();
mat4.multiplyVec3(m01, v01, v02);
```

z 軸の周りの回転行列を右からかける
glMatrix_exercise_05.html

```
var M = mat4.create([
      1.0, 0.0, 0.0, 0.0, // rotate
      0.0, 0.0, 1.0, 0.0, // around
      0.0,-1.0, 0.0, 0.0, // x-axis
      0.0, 0.0, 0.0, 1.0 // for 90 deg.
      ]);
var angle = Math.PI/2;
var N = mat4.create();
mat4.rotateZ(M, angle, N);    // N = MR (not RM)
var v01 = vec3.create([1,0,0]);
mat4.multiplyVec3(N, v01);     // v01 = [0, 0, 1]
```

同様な関数：

mat4.rotateX()：x 軸周りの回転行列をかける

mat4.rotateY()：y 軸周りの回転行列をかける

mat4.rotate()： 指定した軸の周りの回転行列をかける

平行移動行列を右からかける
glMatrix_exercise_05.html

```
var M = mat4.create([
      1.0, 0.0, 0.0, 0.0, // rotate
      0.0, 0.0, 1.0, 0.0, // around
      0.0,-1.0, 0.0, 0.0, // x-axis
      0.0, 0.0, 0.0, 1.0 // for 90 deg.
      ]);
var transv = vec3.create([0,1,0]);   // translate in y
var N = mat4.create();
mat4.translate(M, transv, N);       // N = MT
var v01 = vec3.create([1,0,0]);
mat4.multiplyVec3(N, v01);       // v01 = [1, 0, 1]
```

射影行列の作成

- モデルビュー変換と射影変換については後で説明（復習）する。
- それぞれの関数の使い方はこの後のサンプルコードで。
- 詳細はソースコード（glMatrix.js）を参照。

```
mat4.frustum(left, right, bottom, top, near, far, dest);
mat4.perspective(fovy, aspect, near, far, dest);
mat4.ortho(left, right, bottom, top, near, far, dest);
mat4.lookAt(eye, center, up, dest);
```

その他便利な関数

mat4.toInverseMat3： 左上 3x3 行列の逆行列を取り出す。法線ベクトルの変換に便利

mat4.scale： 引数の vec3 で指定した成分でスケールする

quat4.*：4 元数関係

{vec3,vec4,mat3,mat4,quat4}.str：成分の書きだし

演習

演習

- glMatrix_exercise_02.html から
- glMatrix_exercise_05.html までをベースに各自自由に。

射影変換と同次座標

クリップ座標

これまではクリップ座標の立方体の中だけで描画していた。つまりビューボリュームは正規化ビューボリューム

$$[-1,+1]\times[-1,+1]\times[-1,+1]$$

で固定であった。これは不便。

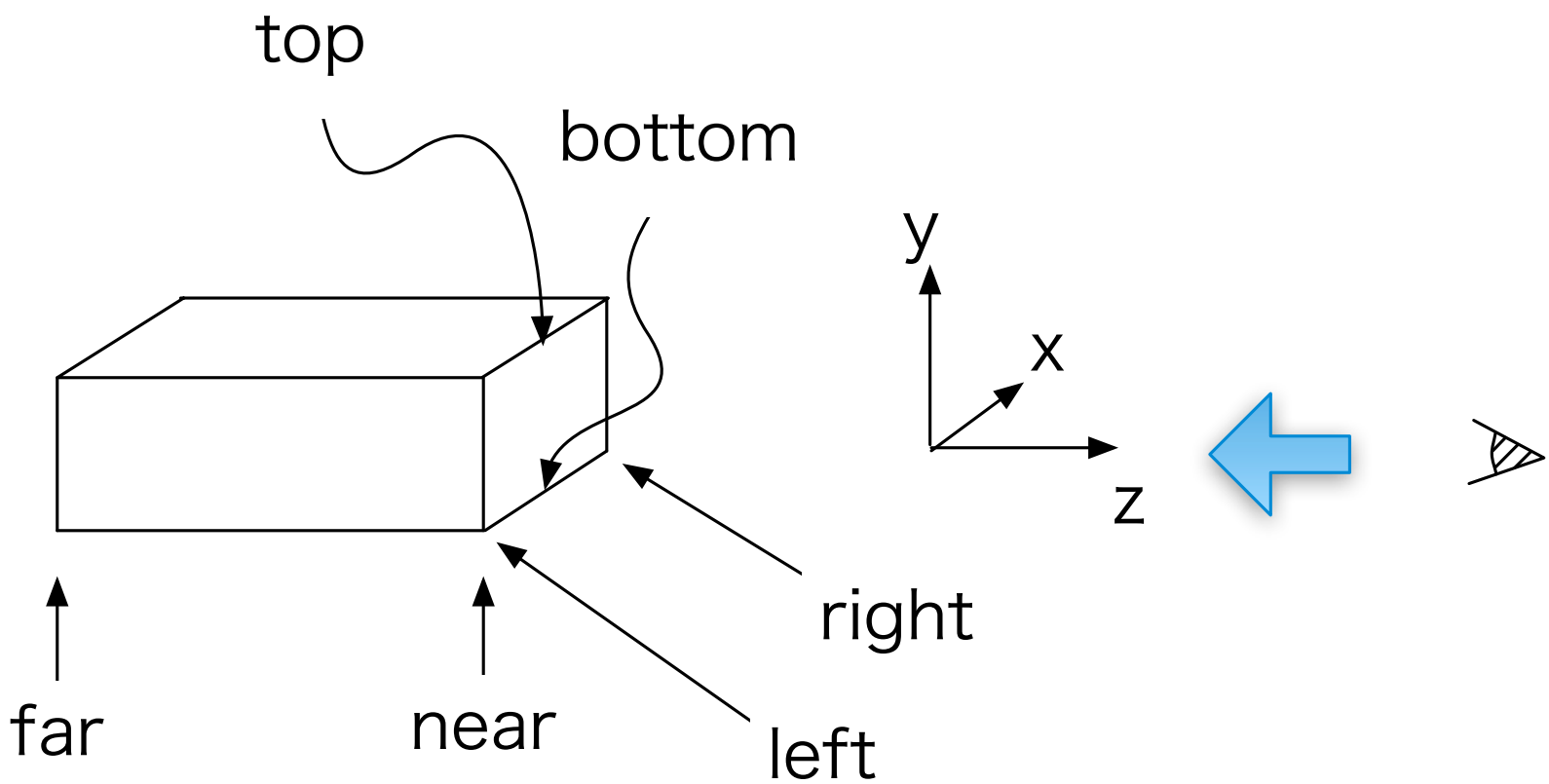
長さの単位（大きさ）を気にしないで、つまりワールド座標で物体を自由に定義したい。

射影変換

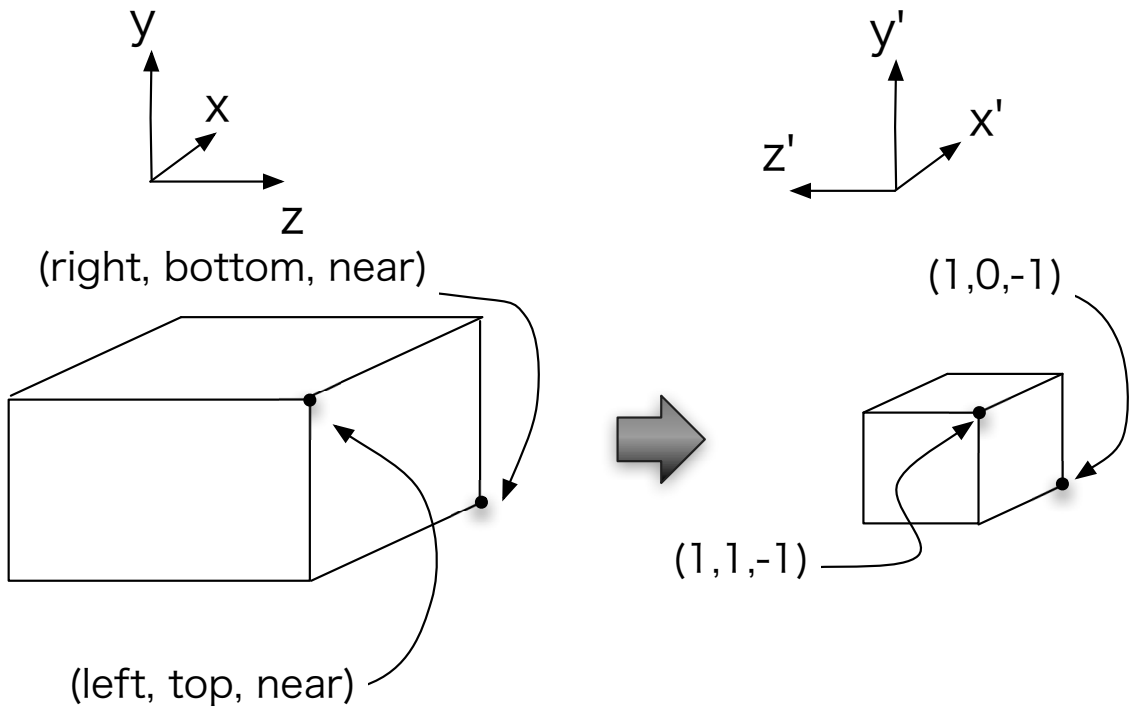
- 正射影
- 透視射影

正射影

- カメラは $z = \infty$ に位置し、$-z$ 方向を向いている。
- 正規化ビューボリュームの座標系は左手系。



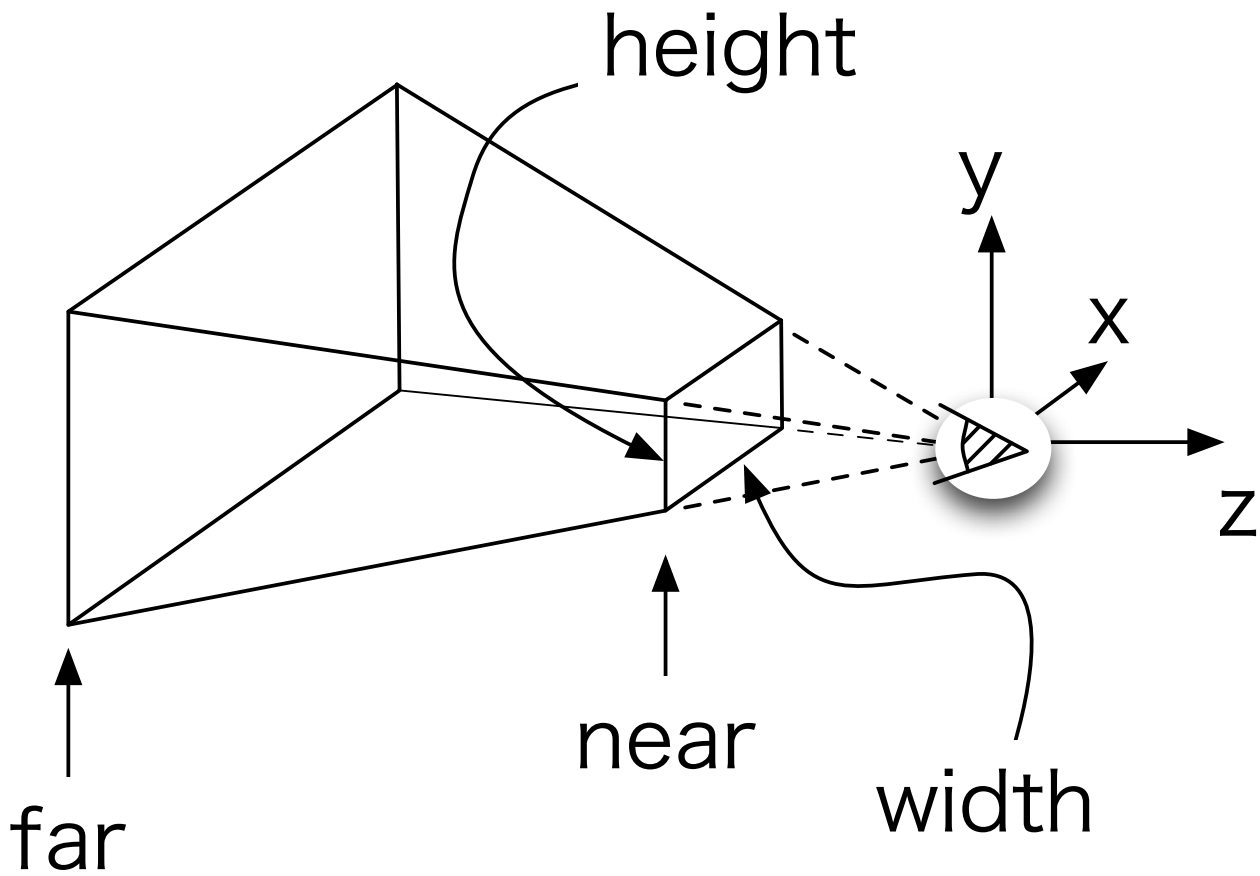
正射影



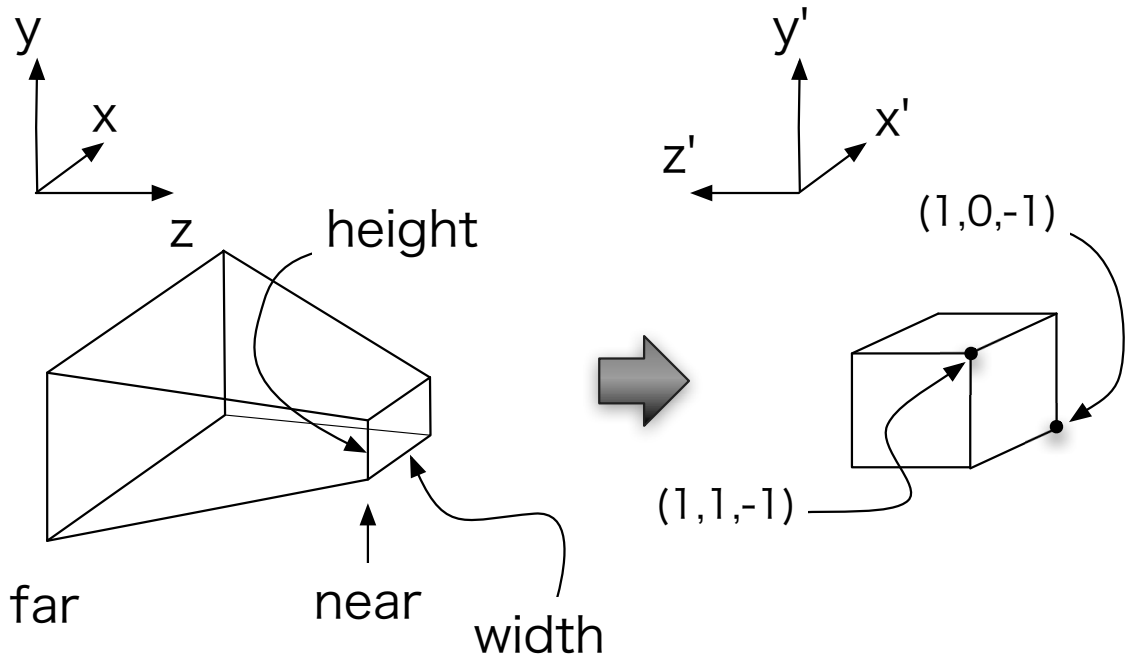
- 直方体から立方体（正規化ビューボリューム）への単純なスケール変換行列。
- 自分で行列を設定するのも簡単だが、次の関数が用意されている
- mat4.ortho(left, right, bottom, top, near, far, projectionMatrix);

透視射影

- カメラは原点に位置し、$-z$ 方向を向いている。
- 物体を回転・移動させればいつでも $-z$ 方向に見えるようにできる。

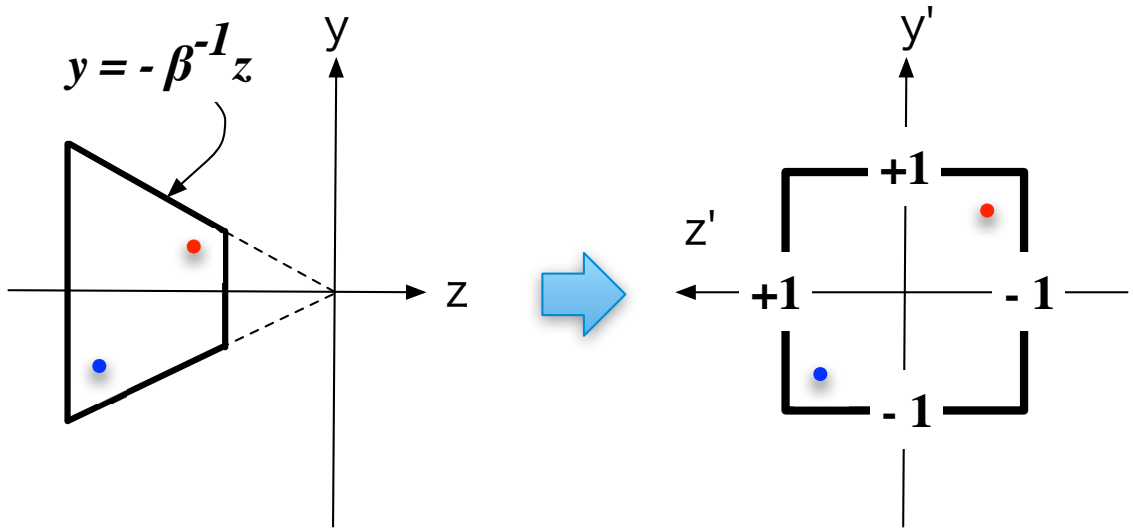


透視射影



- 視錐台形から立方体への変換。

透視射影



$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta\, x/z \\ -\beta\, y/z \\ c_1\, z + c_2 \end{pmatrix}$$

$\beta > 0, c_1, c_2$ は定数。x と y に関しては $-\beta/z$ 倍のスケール変換。

（このスケール係数は、たとえば、視錐台の上面 $y = -\beta^{-1}\, z$ が $y' = +1$ に変換されることで確認できる。）

(復習)

平行移動変換

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \end{pmatrix}$$

は同次座標を使うと行列演算で書けた。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

では、透視射影

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta\, x/z \\ -\beta\, y/z \\ c_1\, z + c_2 \end{pmatrix}$$

も工夫すれば行列で書けるか？

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

••• できない。

では、透視射影

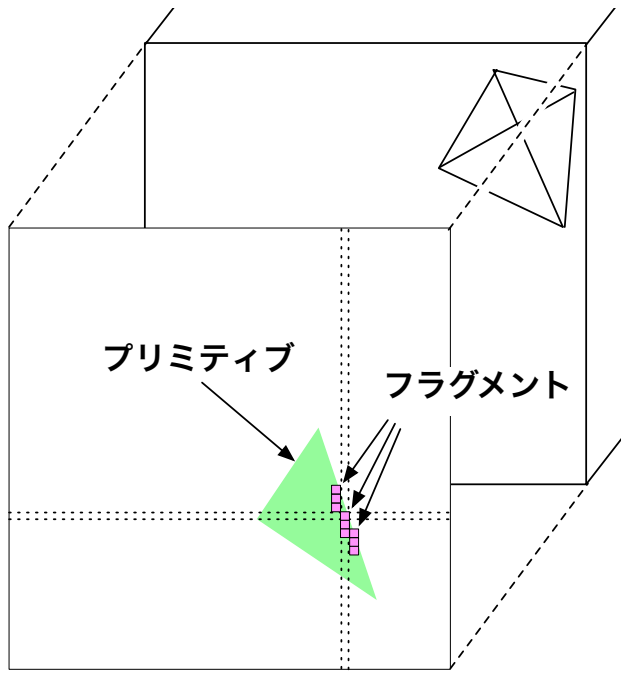
$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta\, x/z \\ -\beta\, y/z \\ c_1\, z + c_2 \end{pmatrix}$$

も工夫すれば行列で書けるか？

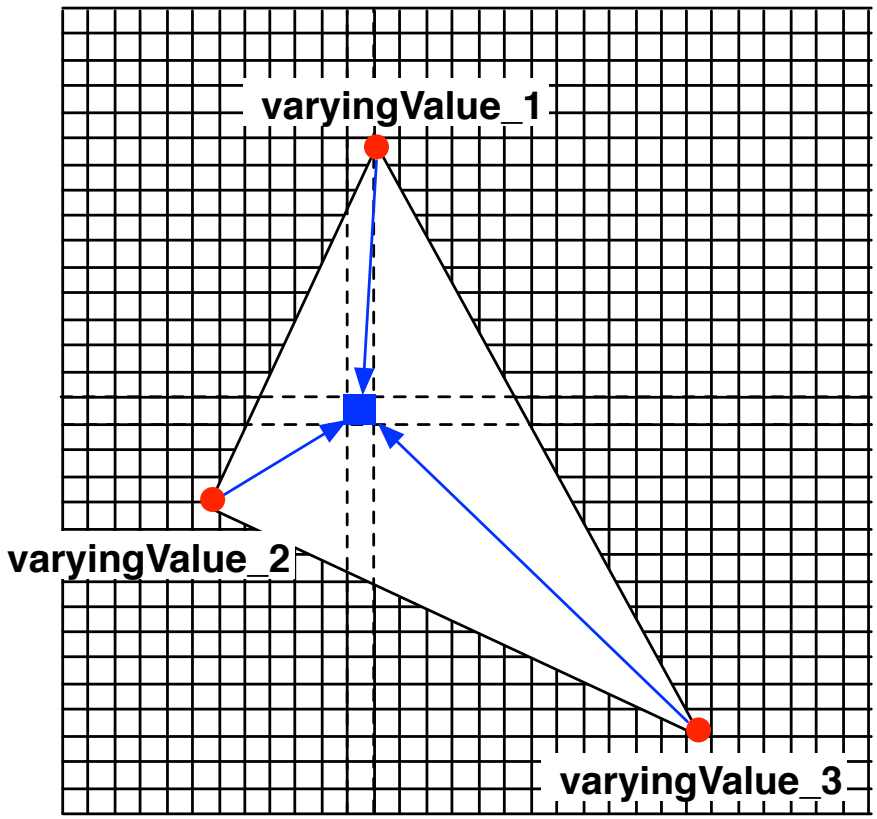
$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

z 補間の問題

- 透視射影の場合、もう一つ注意しなければいけない問題がある。
- 復習：フラグメントシェーダに渡される時の varying 値の自動線形補間

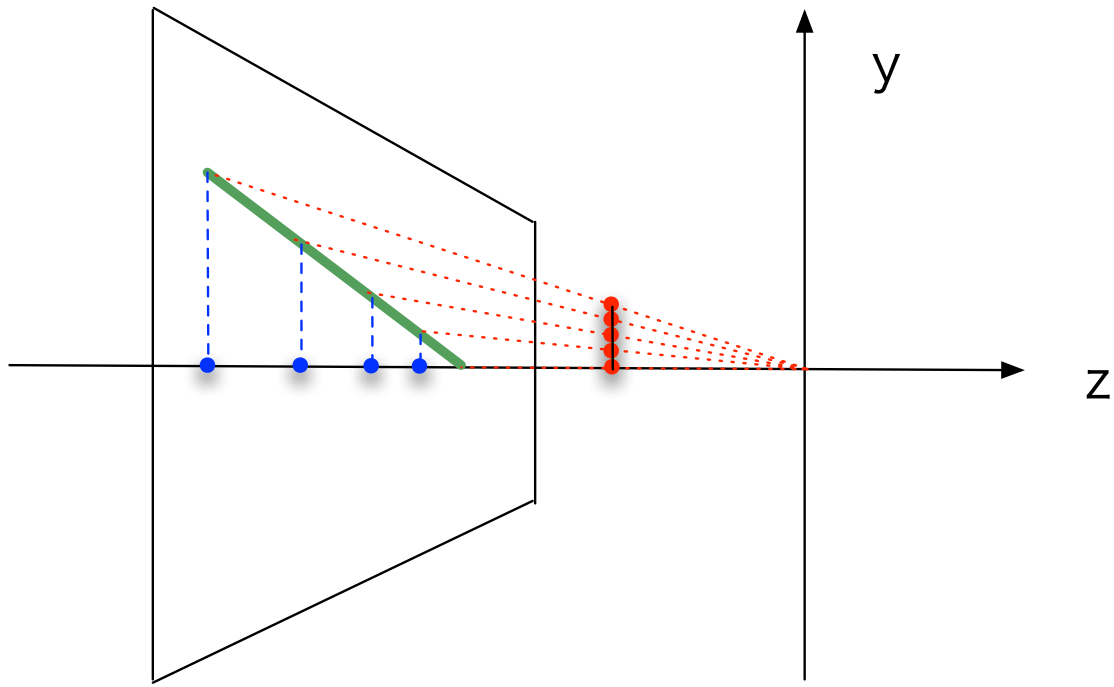


z 補間の問題

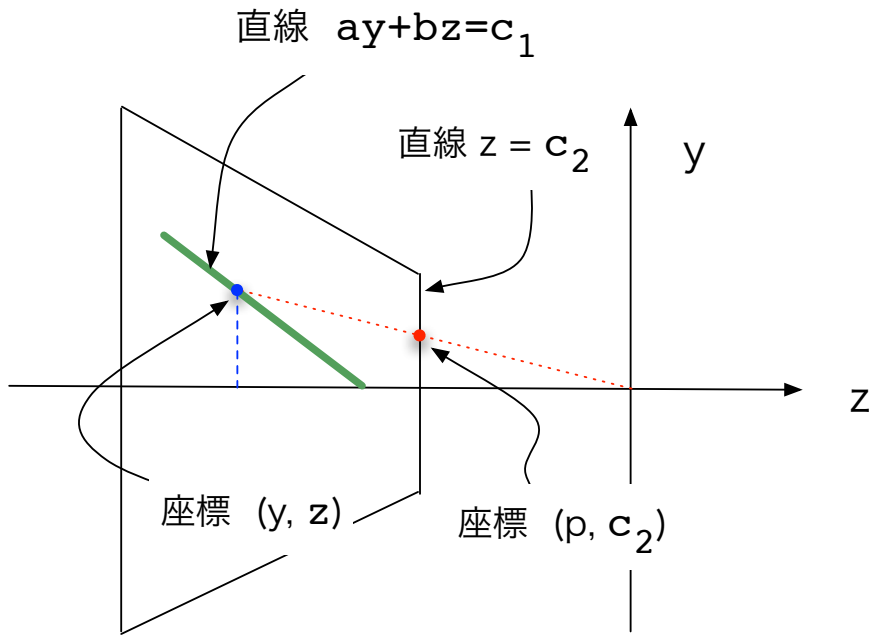


z 補間の問題

- ピクセル（赤）が等間隔に並んでいても
- プリミティブ（緑）上の補間点（青）の z 値は非等間隔。
- これは都合が悪い。



等間隔ピクセル p は等間隔の $1/z$ に対応



$$(ap/c_2 + b)z = c_1$$

$$\frac{1}{z} = \frac{a}{c_2\, c_1} p + \frac{b}{c_1}$$

そこで z の代わりに

$$z \Rightarrow z' = -\alpha - \gamma/z$$

という座標変換して z' をフラグメントシェーダ（プリミティブ合成ステージ）に渡せば、z 方向にも自然に等間隔の線形補間になる。

（負の z に対して、$\gamma > 0$ なら z' は z の単調増加関数である。）

$-1 \leq z' \leq +1$ となるように α と γ を調整する。

—

そういえば ••• 透視射影の x, y 成分も z の除算が入っていた：

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -\beta\, x/z \\ -\beta\, y/z \end{pmatrix}$$

そこで

ある点 $(x, y, z)^t$ に対して、

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta\, x/z \\ -\beta\, y/z \\ -\alpha - \gamma/z \end{pmatrix}$$

をプリミティブ組み立てステージに渡せばよい。(3 成分全てに z の除算が入っていることに注意。) これを OpenGL では、次の 2 段階に分けて実行する。

透視射影変換の二つのステップ

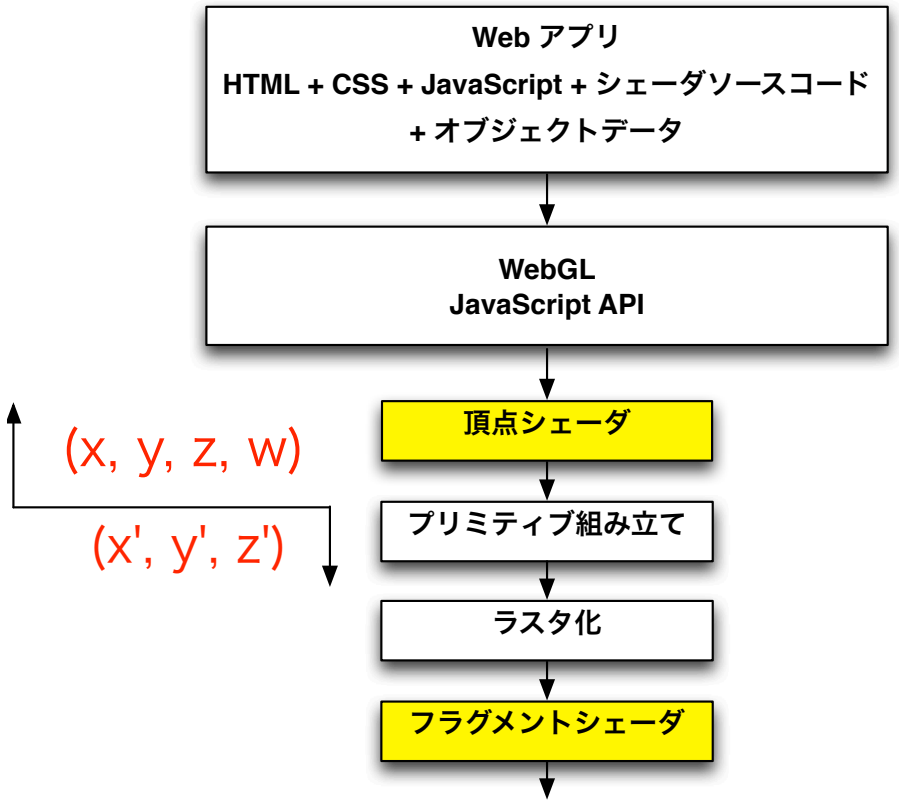
(1) 同次座標の行列演算。（この行列は一意には決まらない。ここで挙げるのは一つの例である。）

$$\begin{pmatrix} x^\dagger \\ y^\dagger \\ z^\dagger \\ w^\dagger \end{pmatrix} = \begin{pmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \alpha & \gamma \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(2) w 座標での割り算。これを透視除算という。プリミティブ組み立て時に実行される。

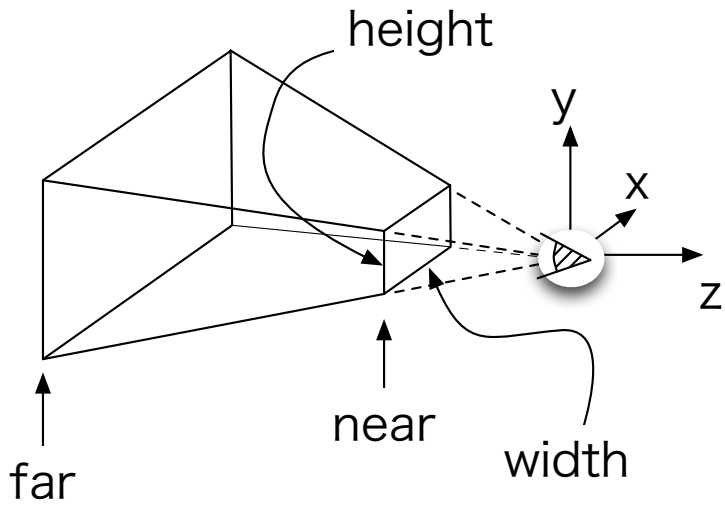
$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x^\dagger/w^\dagger \\ y^\dagger/w^\dagger \\ z^\dagger/w^\dagger \end{pmatrix}$$

パイプラインでの座標の処理



- $(x', y', z') = (x/w, y/w, z/w)$
- GPU が自動的に実行

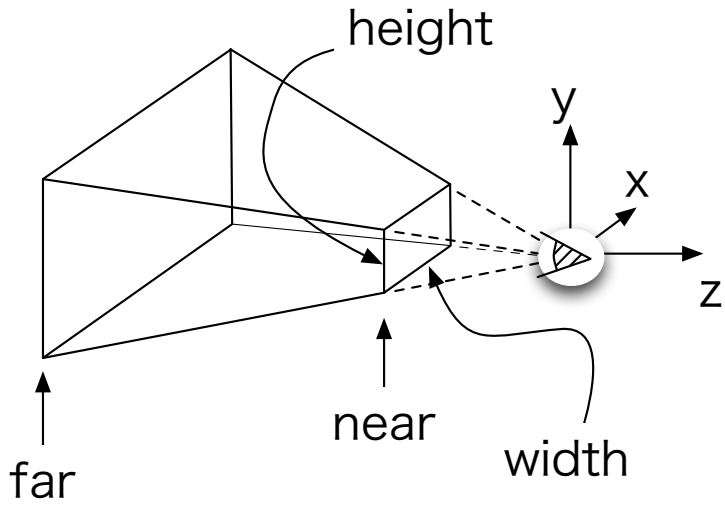
透視射影行列の設定



$$\begin{pmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \alpha & \gamma \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

パラメータ α, β, γ の決定　$\Rightarrow$　計算するのは簡単。

透視射影行列の設定



mat4.perspective(fovy, aspect, near, far, projectionMatrix);

fovy （field of view）は視野の高さ（y）方向の角度。aspect は縦横比。

上下と左右に非対象な frustum（視錐台）を作る場合は、
mat4.frustum(left, right, bottom, top, near, far, projectionMatrix);

（ステレオ画像を作る時には、右目用の画像と左目用の画像が必要である。このとき、視錐台は左右非対称。）

モデルビュー変換と射影変換

座標変換の実際

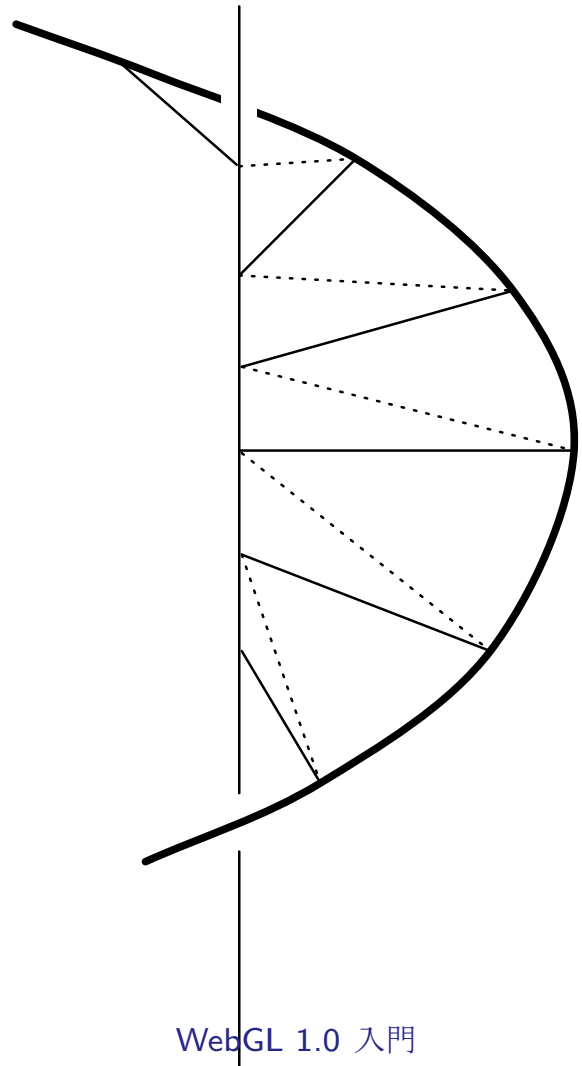
- モデルビュー変換
- 射影変換

について、あるオブジェクトに少しずつ座標変換をかけながらその効果を見る。

サンプル 3D 物体

らせん状の物体を考える

中心軸とそれにまきつく螺旋の間を面でつなぐ



webgl_sample_spiral_00.html

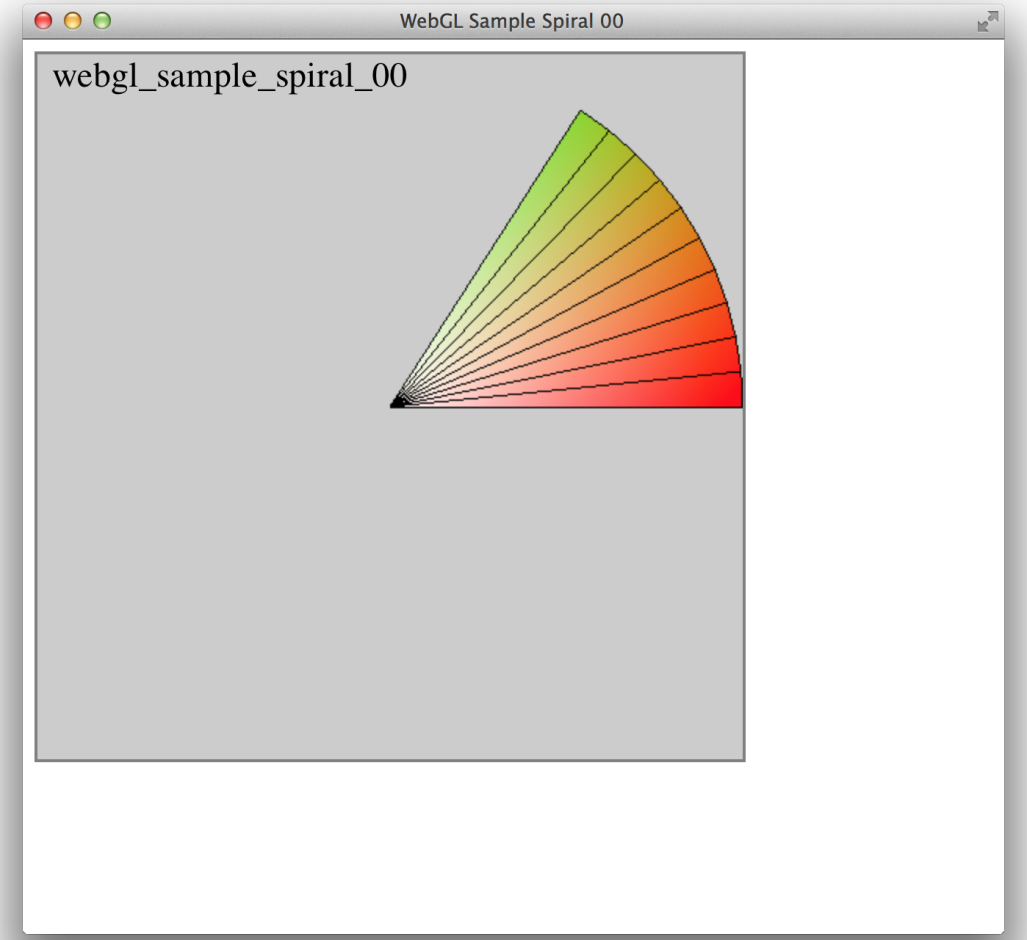
オブジェクトの構成部分

```
var dz = 0.1;
var pitch = 1.0;
var pof = 0;        // positionOffsetInFloats
var cof = 12;       // colorOffsetInBytes

for (var k=0; k<Nz; k++) {
  var z = k*dz;
  positionView[  pof] = 0.0;   // x
  positionView[1+pof] = 0.0;   // y
  positionView[2+pof] = z;     // z
  colorView[  cof] = 255;      // R
  colorView[1+cof] = 255;      // G
  colorView[2+cof] = 255;      // B
  colorView[3+cof] = 255;      // A
  pof +=vertexSizeInFloats;
```

```
    cof +=vertexSizeInBytes;

    phase = pitch*z;
    positionView[  pof] = Math.cos(phase);        // x
    positionView[1+pof] = Math.sin(phase);        // y
    positionView[2+pof] = z;                      // z
    colorView[  cof] = 255*Math.cos(phase);       // R
    colorView[1+cof] = 255*Math.sin(phase);       // G
    colorView[2+cof] = 255*Math.sin(phase*0.2);   // B
    colorView[3+cof] = 255;                       // A
    pof +=vertexSizeInFloats;
    cof +=vertexSizeInBytes;
}
```

結果

何も変換していないので、見にくい。正規化ビューボリュームから外れている部分が見えない。



演習

- webgl_sample_spiral_00.html の dz（156 行目）を変えてその効果をみよう。

モデル変換
webgl_sample_spiral_01.html

モデルを移動・縮小・回転する行列 modelViewMatrix をつくり、頂点シェーダに送る。これも頂点属性（attribute）。

```
mat4.identity(modelViewMatrix);
mat4.translate(modelViewMatrix, [0.8,-0.5, -0.8]);
mat4.scale(modelViewMatrix,[0.3,0.3,0.3]);
mat4.rotateX(modelViewMatrix,-Math.PI/3);
mat4.rotateY(modelViewMatrix,-Math.PI/3);
gl.uniformMatrix4fv(uniLocation,
                    false,
                    modelViewMatrix);
//--</new>--


gl.clear(gl.COLOR_BUFFER_BIT);
```

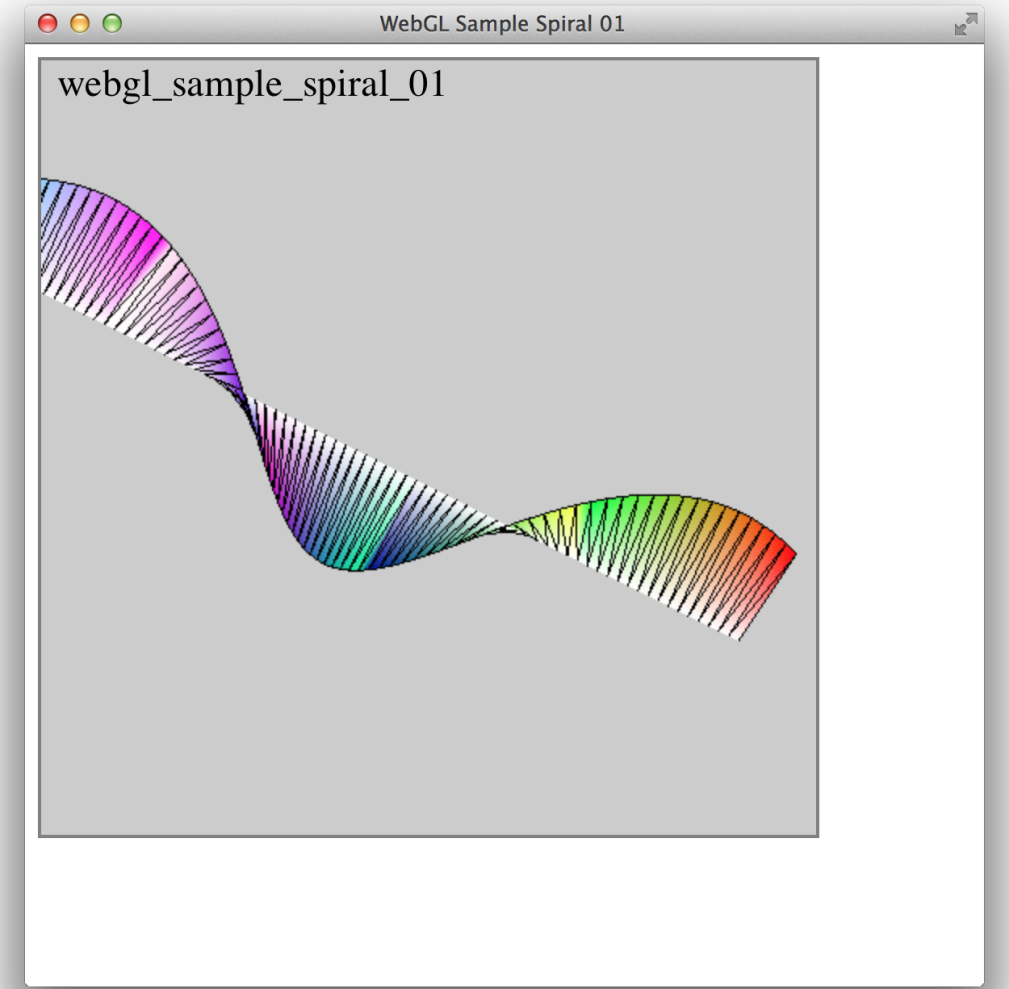
頂点シェーダ

modelViewMatrix も頂点の attribute

復習：全ての頂点に共通した attribute 値は uniform

```
<!-- new -->

<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;
  uniform   mat4 uMVMatrix;       //<--new
  varying   vec4 vColor;

  void main() {
    vColor = aVertexColor;
    gl_Position = uMVMatrix * vec4(aVertexPosition, 1.0); //<
        --new
  }
```

スナップショット
webgl_sample_spiral_01.html



演習

- webgl_sample_spiral_01.html の回転軸と回転角を変更してその効果をみよう。

射影変換
webgl_sample_spiral_02.html

常に正規化ビューボリュームのサイズを意識してモデルを移動・縮小・回転を設定するよりも、ワールド座標（CG 世界）の中にカメラを設定する、と考えた方が自然。

⇒ 射影変換。

mat4.perspective で projectionMatrix を作る

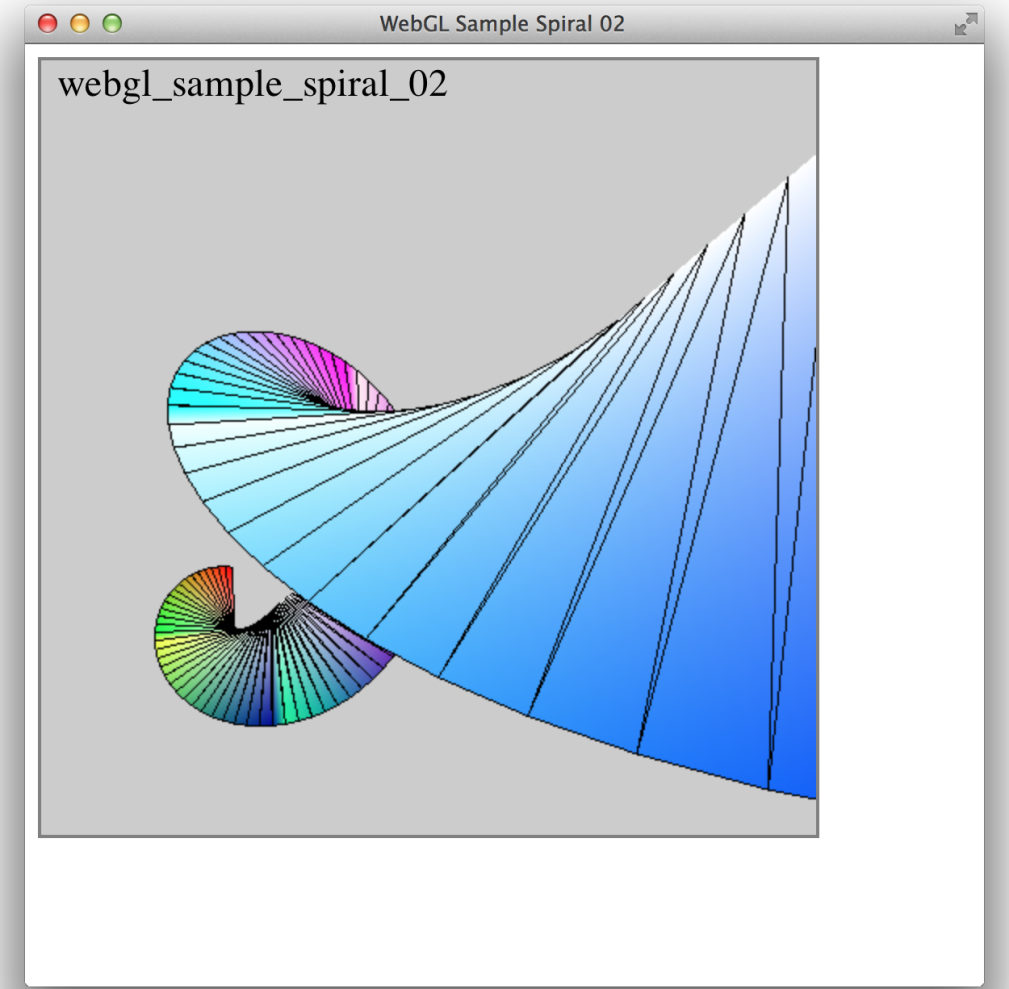
projectionMatrix を uniform として頂点シェーダに送る

シェーダではこの二つの行列をかける

gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);

このサンプルプログラムのこれ以外のポイントとして、デプステストとポリゴンオフセットがあるが説明は省略。

スナップショット
webgl_sample_spiral_02.html



演習

- webgl_sample_spiral_02.html の射影行列の設定 mat4.perspective(...) を変更してその効果を見よう。

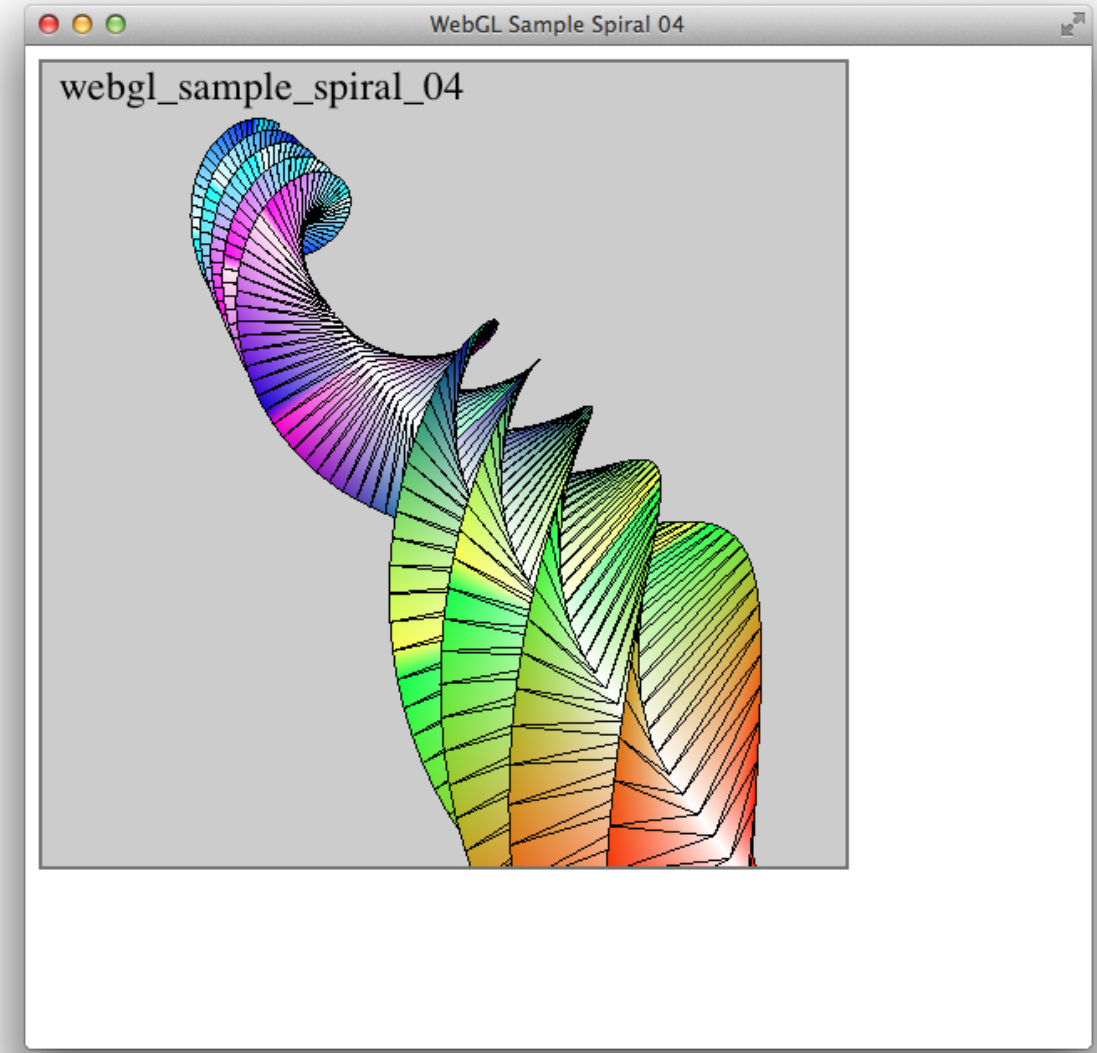
複数の変換行列の管理

描画する物体が複数の要素から構成されているときなど、カメラ（射影行列）は固定していて、モデルビュー行列は頻繁に更新してロードする場合には、モデルビュー行列のスタックを使うのが便利。（以前の OpenGL （OpenGL 1.x）には glPushMatrix(), glPopMatrix() があったが、今の OpenGL にはない。）

JavaScript には配列に push と pop のメソッドが備わっているのでこれを使えばいい。

練習

螺旋型の面を位相をずらして 5 枚描く



変換行列の push と pop
webgl_sample_spiral_04.html

```
    }
    modelViewMatrix = modelViewMatrixStack.pop();
}

//--new function
function uploadModelViewMatrixToShader() {
    gl.uniformMatrix4fv(uniLocation[1],
                        false,
                        modelViewMatrix);
}
```

変換行列のシェーダへのロード
webgl_sample_spiral_04.html

```
function uploadProjectionMatrixToShader() {
  gl.uniformMatrix4fv(uniLocation[0],
                      false,
                      projectionMatrix);
}

//--new function--
function draw_a_spiral() {
  // Draw triangles
```

描画
webgl_sample_spiral_04.html

```
}

function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = createGLContext(canvas);

  shader_program = create_shader_program();
```

演習

- webgl_sample_spiral_04.html を自由に変更してみよう。

演習

- glMatrix を使い、任意の計算をせよ。画像不要。

準備

サンプルテクスチャのダウンロード

この講義のウェブページから、今日の演習で使用する次のテクスチャデータをローカルディスクに保存してください。

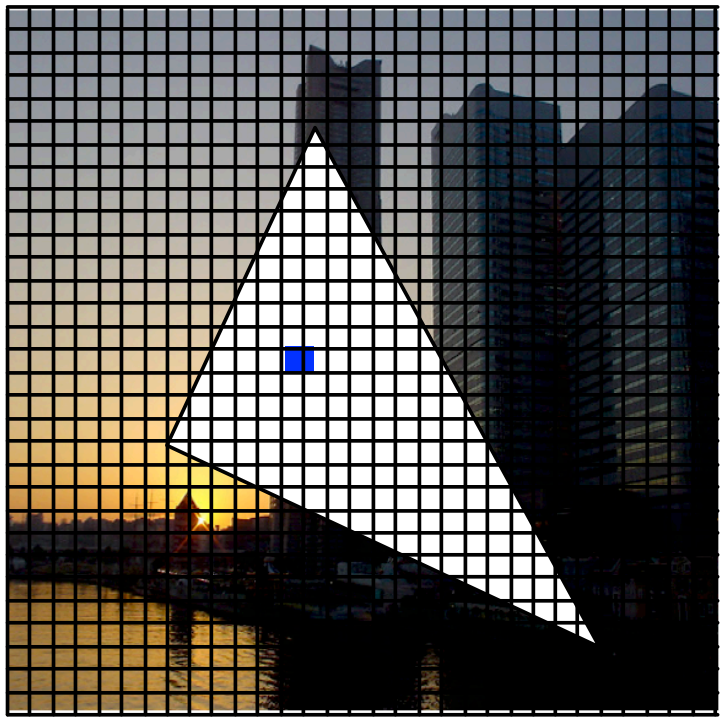
- kobe_u_logo_512x256.jpg
- yokohama_512x512.jpg
- uribos_256x256.jpg
- kobe_1024x512.jpg

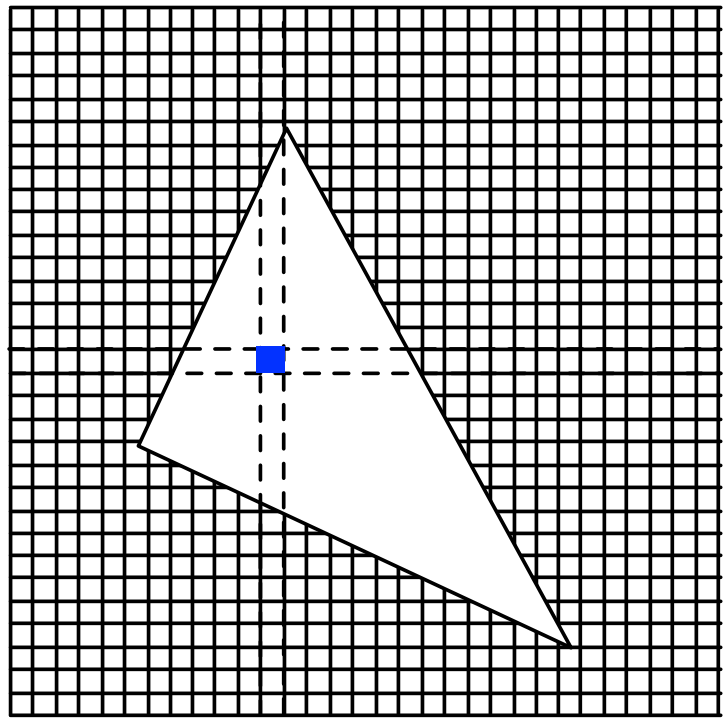
保存手順（例）：ブラウザで表示 → Ctrl + クリック → イメージを保存

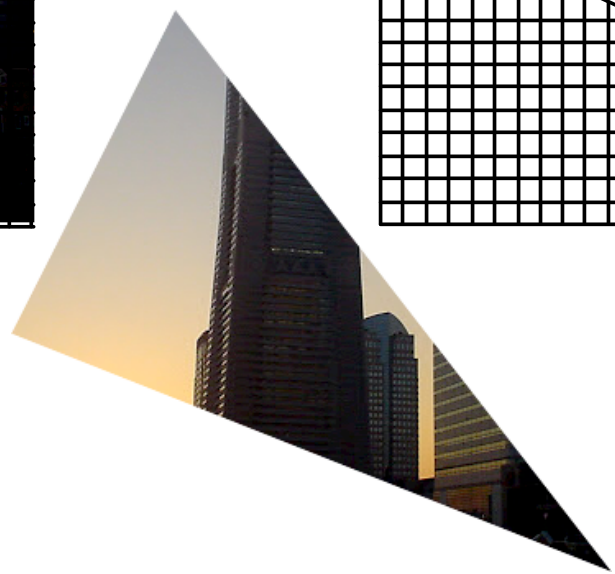
テクスチャマッピング

テクスチャマッピング（テクスチャリング）とは

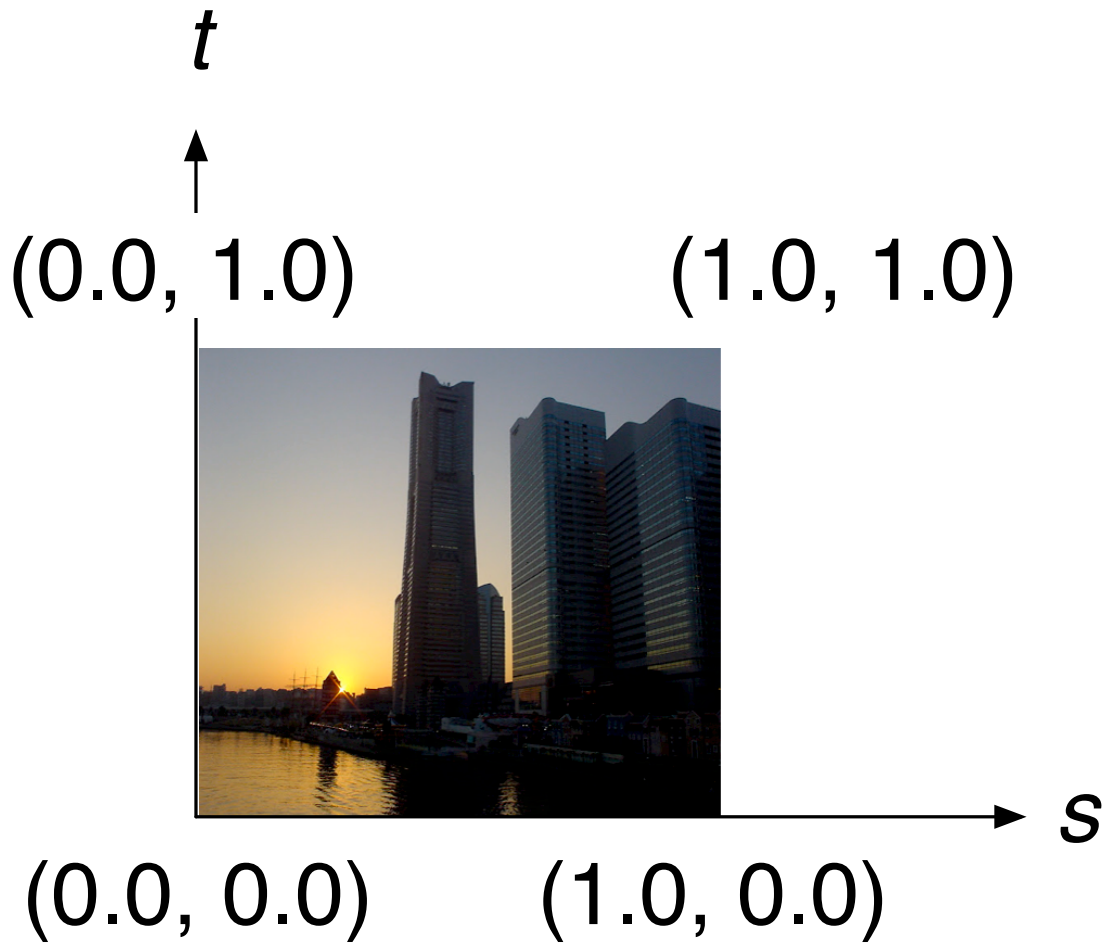
テクセル



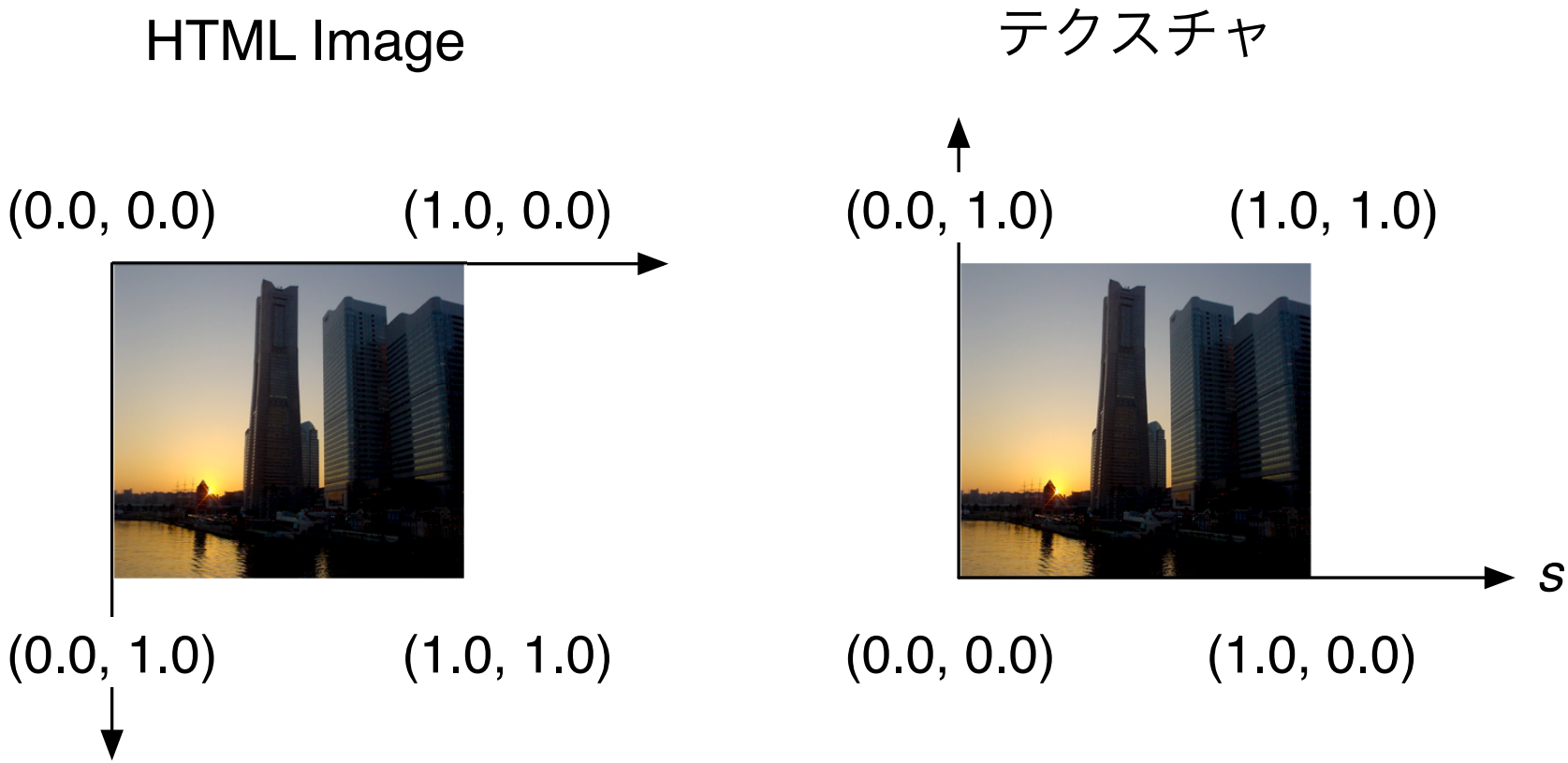




テクスチャ座標



HTML image と WebGL image



読み込んだ画像を上下反転させるために以下を呼ぶ

gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL,true);

テクスチャのロード手順の概観

1. gl.createTexture() で WebGLTexture オブジェクトを作成する
2. new Image() で HTML DOM の Image オブジェクトを作成する
3. Image オブジェクトの onload ハンドラを設定する
4. Image オブジェクトの src プロパティにロードする画像の URL をセットする
5. （必要なら）gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL,true); で上下反転する
6. gl.texParameteri() でテクスチャパラメータの設定をする

1. WebGLTexture オブジェクトを作成する

```
var texture = gl.createTexture();
```

2. テクスチャをバインドする

```
gl.bindTexture(gl.TEXTURE_2D, texture);
```

3. 画像データをロードする

PNG, GIF, JPEG ファイルなどが可能

手順 :

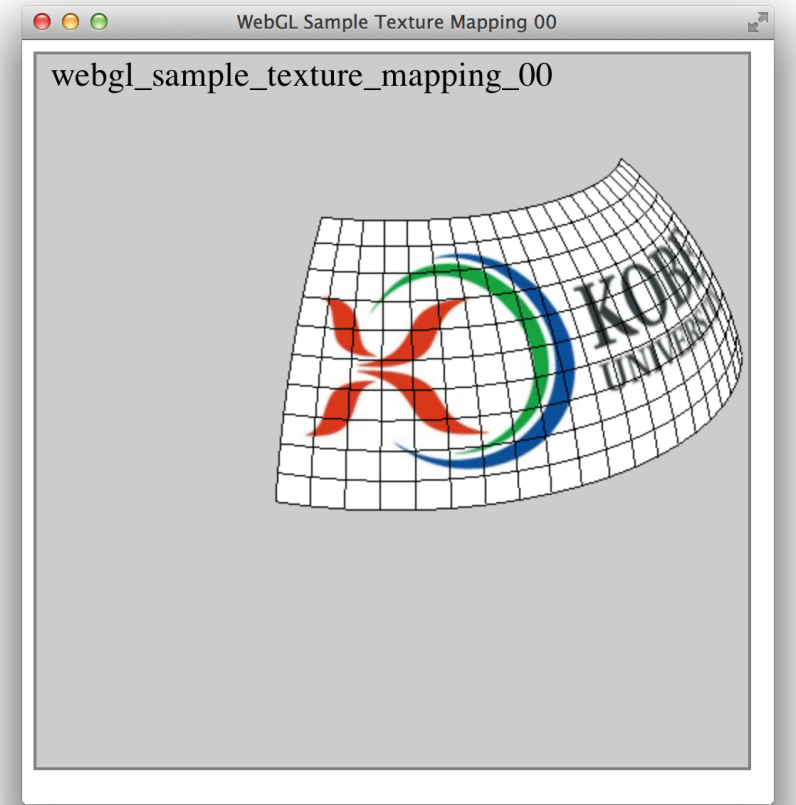
1. HTML DOM の Image オブジェクトを作成する
   image = new Image();
2. image の src プロパティにロードする画像の URL を設定する。
3. データのロードが終わったら GPU にデータをアップロードする

サンプルプログラム

基本的なテクスチャマッピング

webgl_sample_texture_mapping_00.html

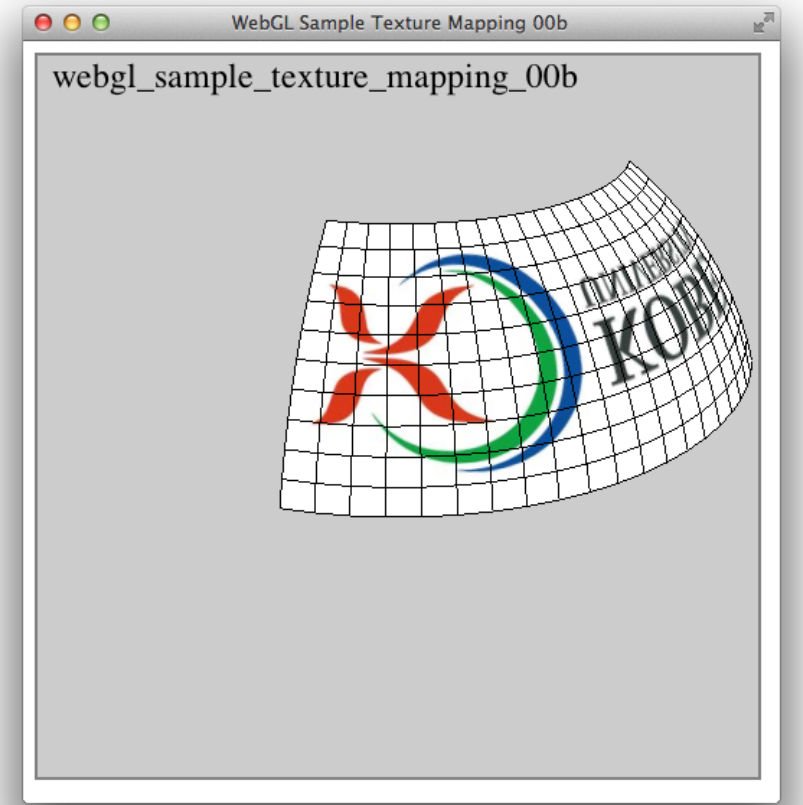
FLIP_Y なし



サンプルプログラム 00b

webgl_sample_texture_mapping_00b.html

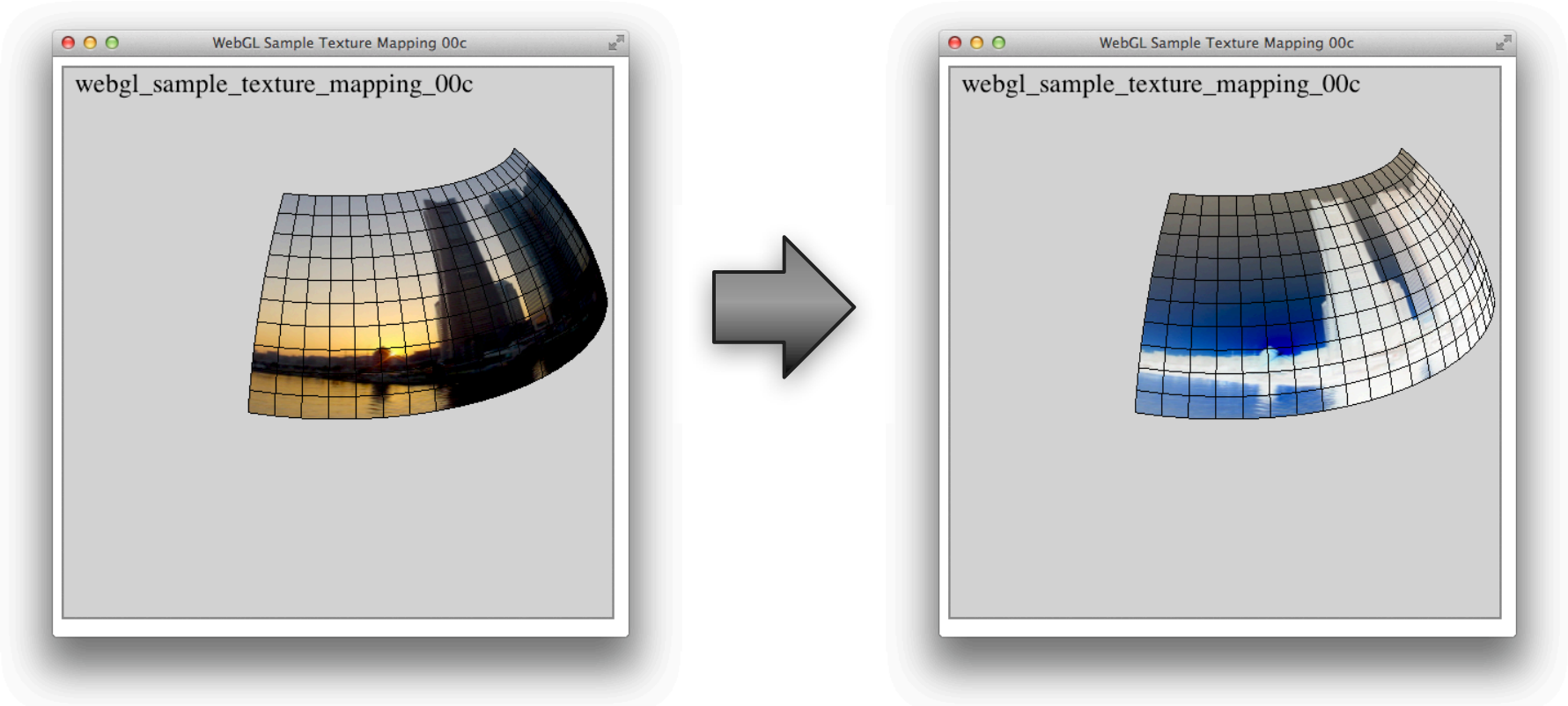
FLIP_Y あり



サンプルプログラム 00c

webgl_sample_texture_mapping_00c.html

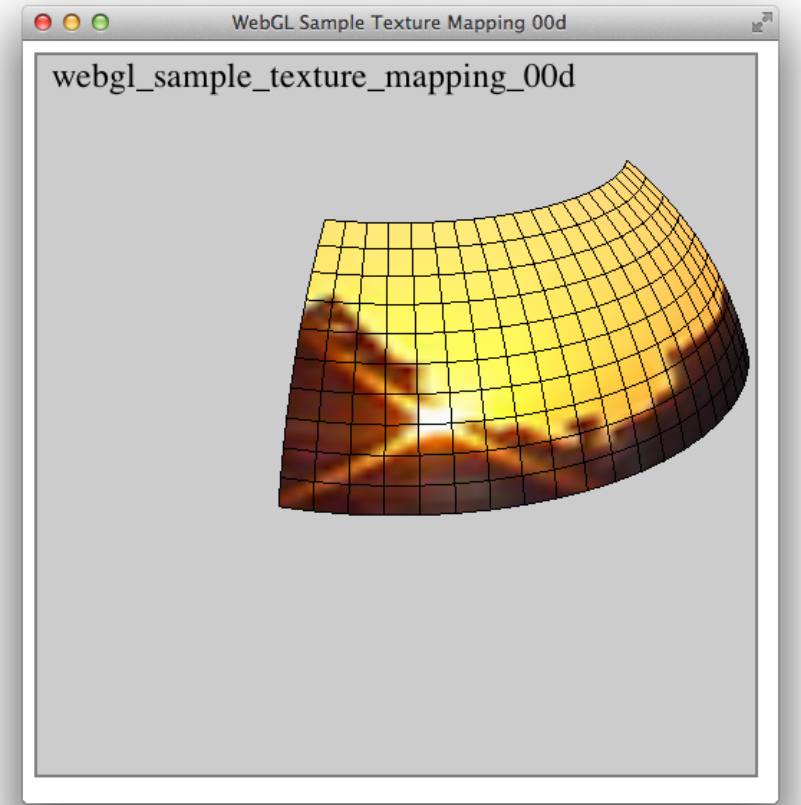
フラグメントシェーダでのテクスチャデータの演算



サンプルプログラム 00d

webgl_sample_texture_mapping_00d.html

画像データの一部だけをマッピング

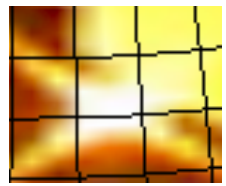


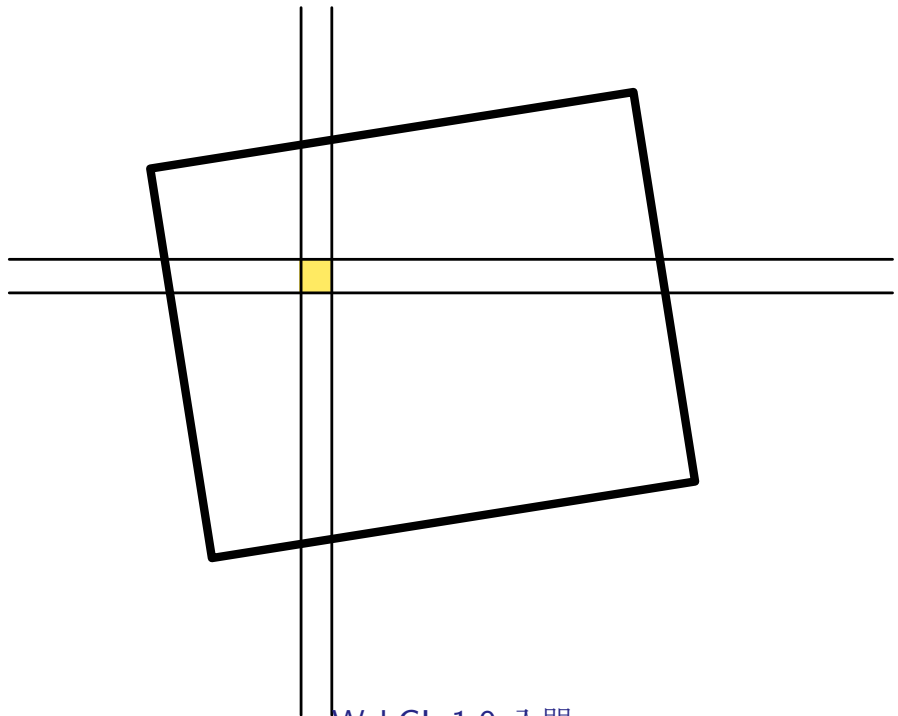
テクスチャの拡大・縮小

テクスチャの拡大

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);

線形補間（gl.LINEAR）





テクスチャの補間（拡大）

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);

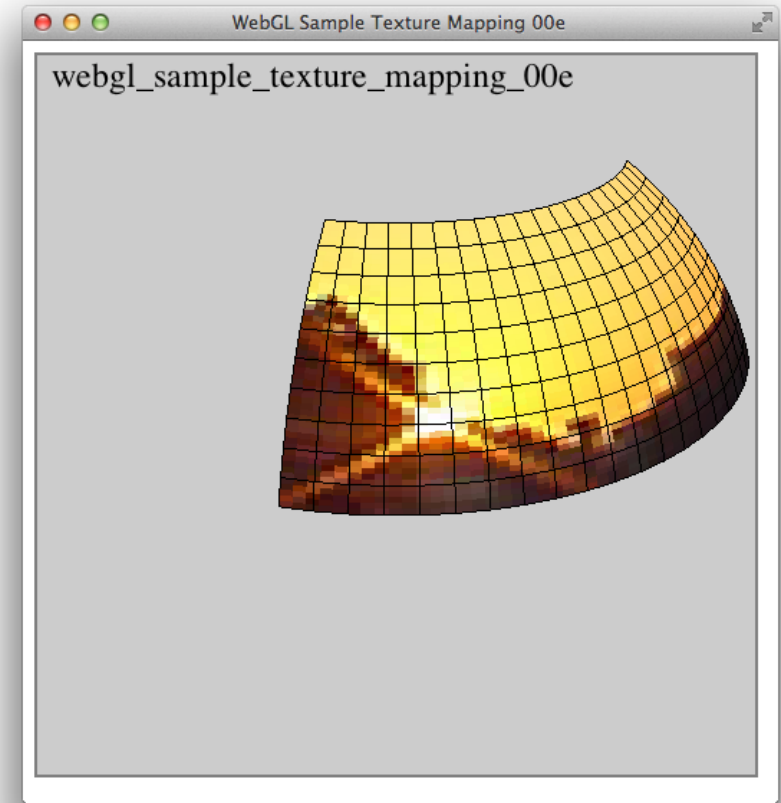
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

- gl.LINEAR そのテクスチャ座標を囲む 4 つのテクセル値の双線形補間
- gl.NEAREST テクスチャ座標に最も近いテクセルの色を使う

ピクセレーションの問題

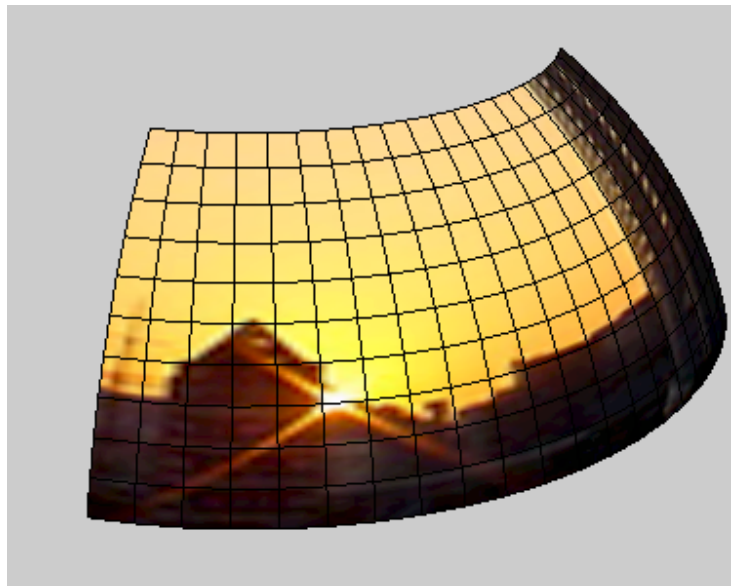
gl.NEAREST

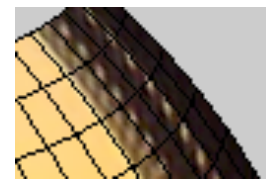
webgl_sample_texture_mapping_00e.html

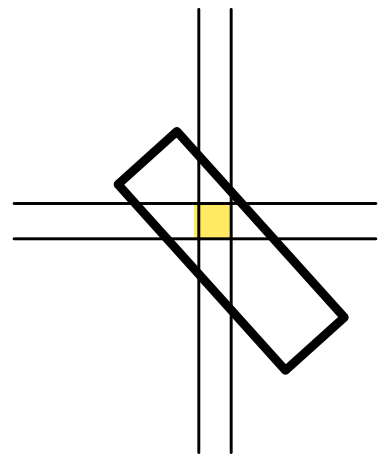


テクスチャの縮小

一つのピクセル位置に複数のテクセルが対応







テクスチャの補間（縮小）

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

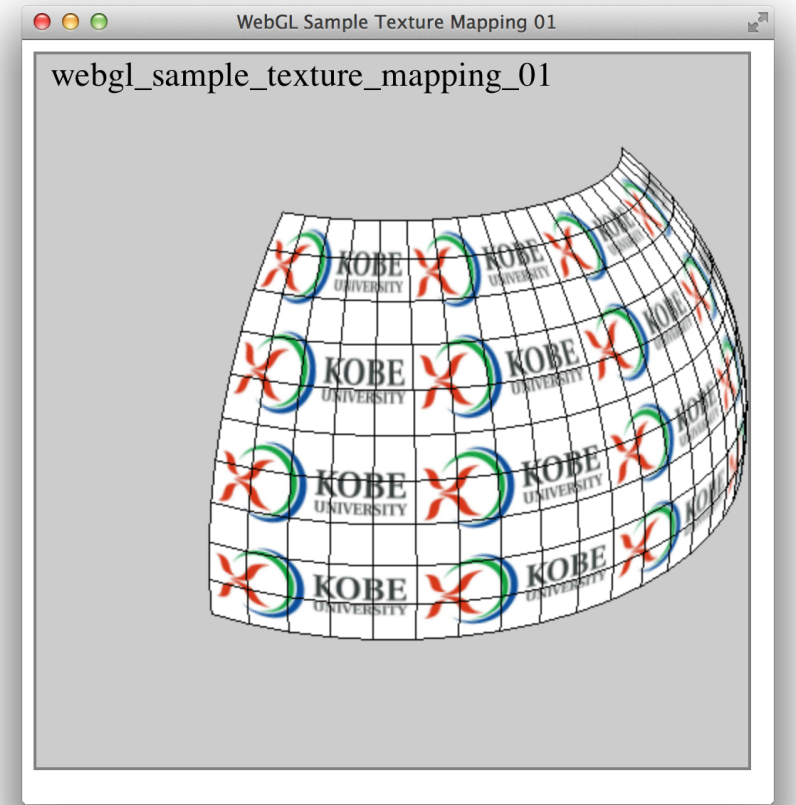
- gl.LINEAR そのテクスチャ座標に対応する 4 つ（固定）のテクセル値の加重平均
- gl.NEAREST テクスチャ座標に最も近いテクセルの色を使う

ラップモード

サンプルプログラム 01

webgl_sample_texture_mapping_01.html

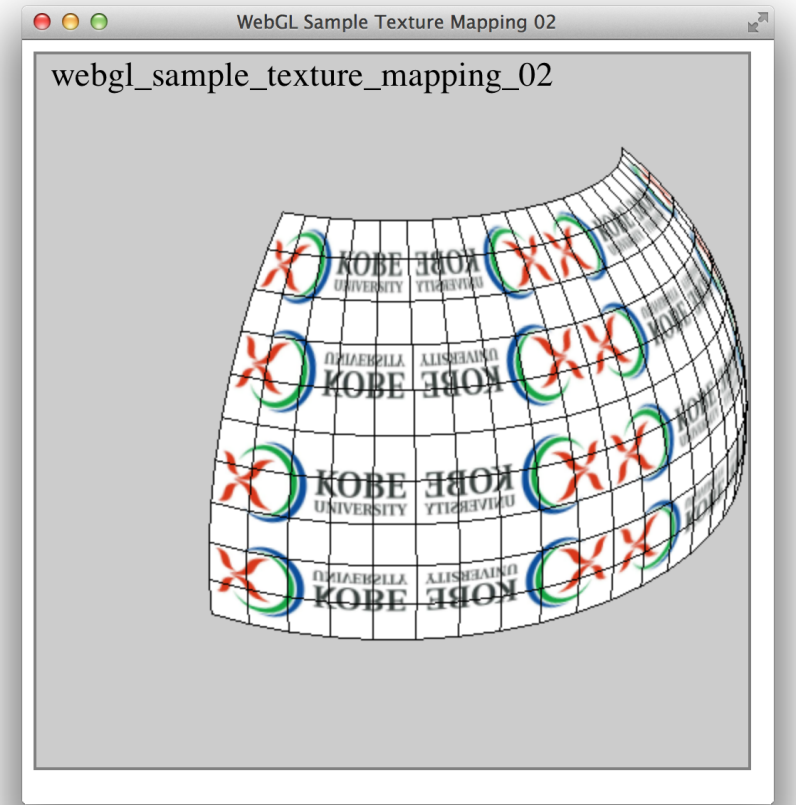
ラップモード： REPEAT



サンプルプログラム 02

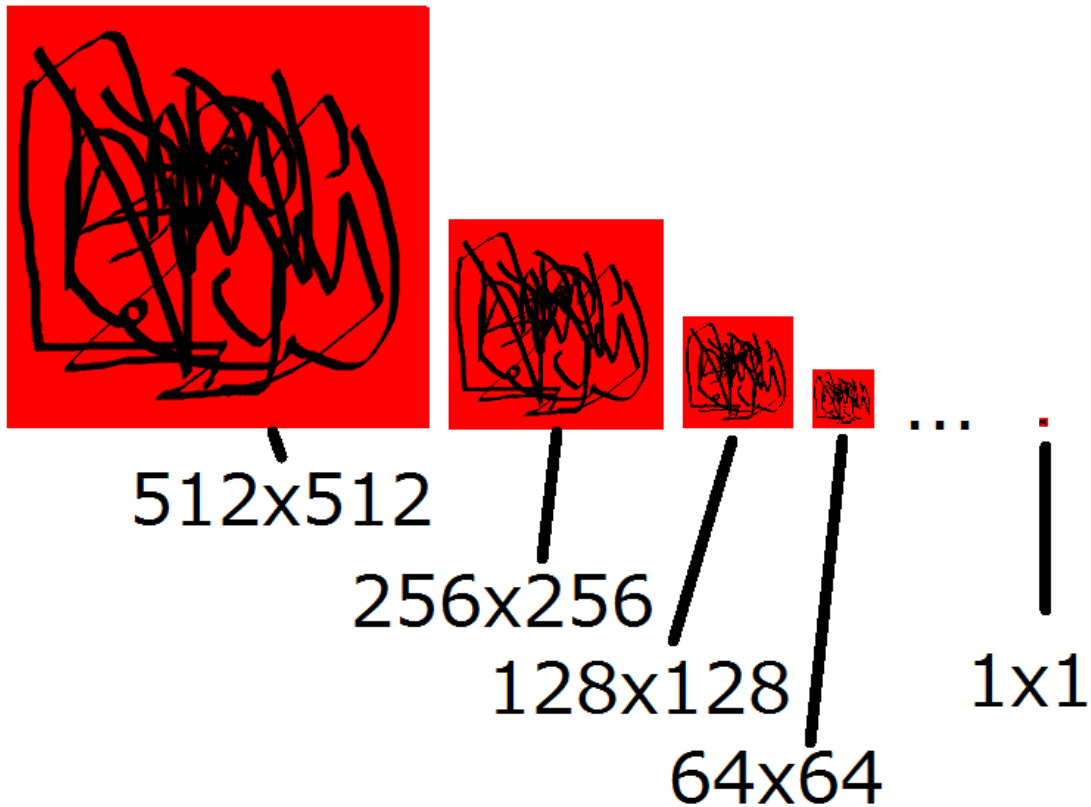
webgl_sample_texture_mapping_02.html

ラップモード： MIRRORED_REPEAT



ミップマップ

ミップマップ



ミップマップ

```
gl.texeImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.UNSIGNED_BYTE, image);
gl.generateMipmap(gl.TEXTURE_2D);
```

画像サイズ（幅と高さ）は２のベキでなければいけない。

演習

- WebGL でテクスチャマッピングを使った画像を作れ。

アニメーション

二つの方法

1. JavaScript の setInterval() を使う
2. requestAnimationFrame() を使う

requestAnimationFrame() は HTML DOM の window オブジェクトのメソッド。

setInterval

setInterval(codeToCall, timeoutInMilliSeconds)

使い方：

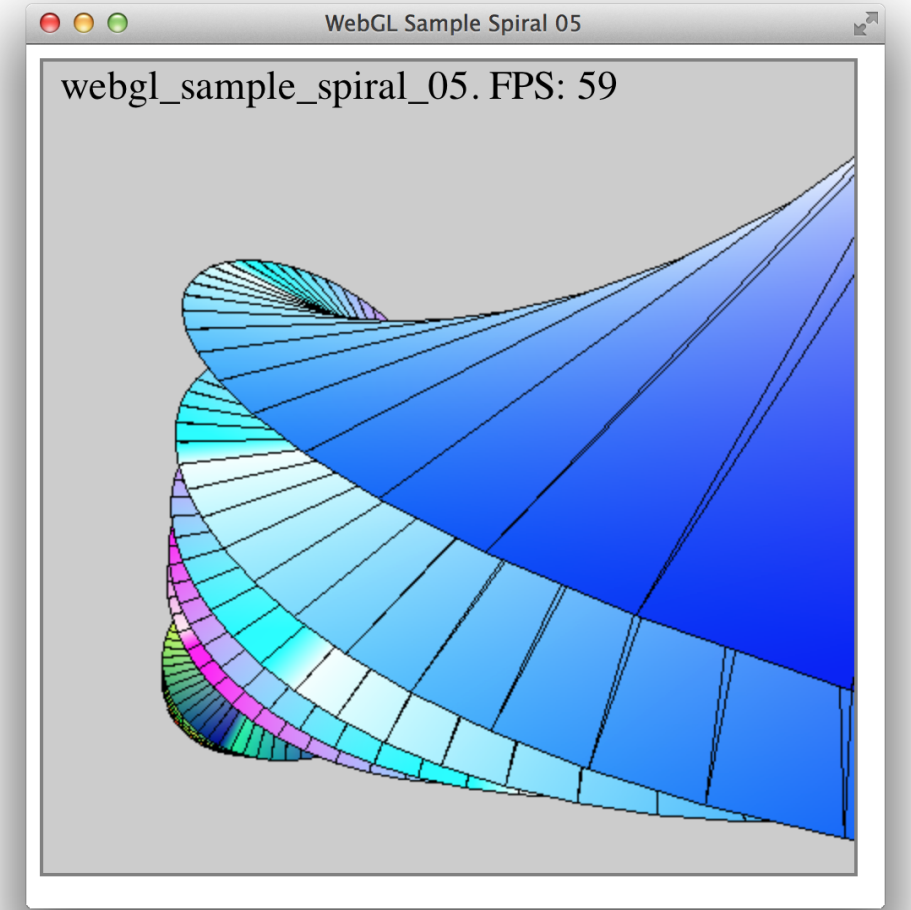
```
function draw() {
    . . .
}

function start() {
    . . .
    setInterval(draw, 16.7);
}
```

16.7 ミリ秒毎に draw() を呼ぶ。

$1/60 \sim 0.0167$

setInterval を使ったサンプルコード

webgl_sample_spiral_05.html

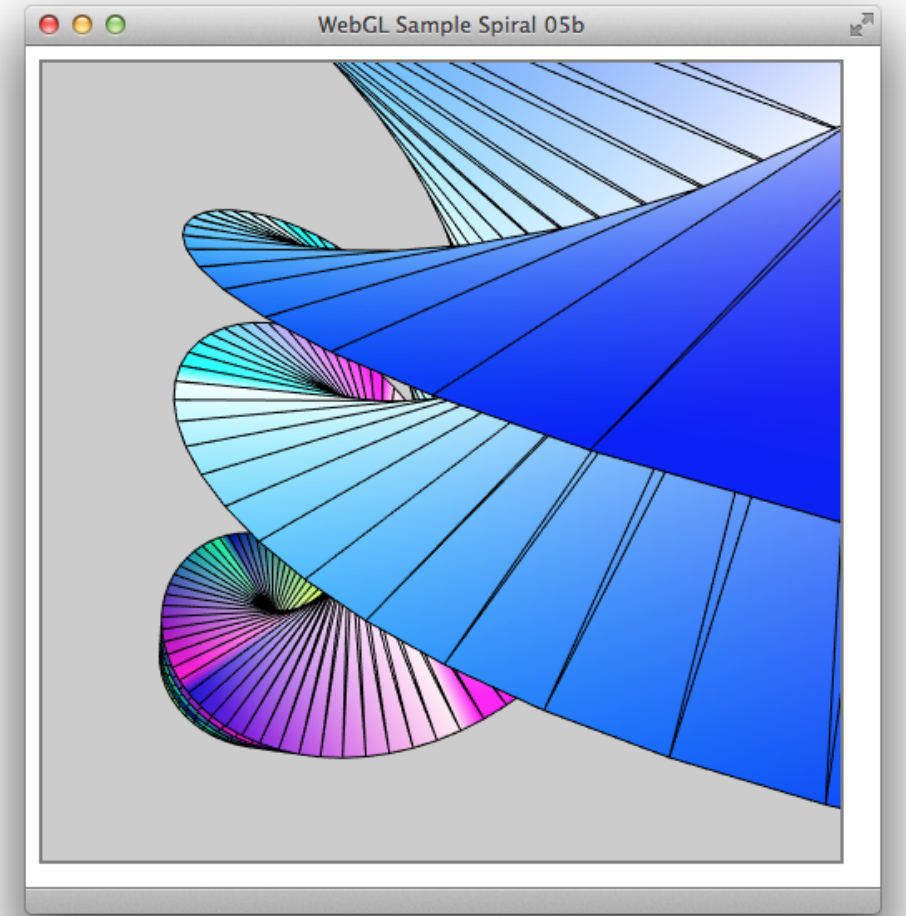


requestAnimationFrame

- シーンの更新タイミングをブラウザに任せる。
- こちらの方がよい。

requestAnimationFrame を使ったサンプルコード

webgl_sample_spiral_05b2.html



requestAnimationFrame の使い方

引数にブラウザから呼び出すコールバック関数を渡す。

requestAnimationFrame(draw);

指定したコールバック draw 関数が呼び出されるときには、引数として現在時刻（currentTime）が自動的に渡される。

つまり、draw() 関数は currentTime を引数として受け取るように定義する。

ただし、startup 関数から最初に draw() を呼び出すときには引数なしでコールする。（JavaScript では文法エラーにならない。）

requestAnimationFrame の使い方

```
function draw(currentTime) {
    requestAnimationFrame(draw);
    . . .
}

function startup() {
    . . .
    draw();
}
```

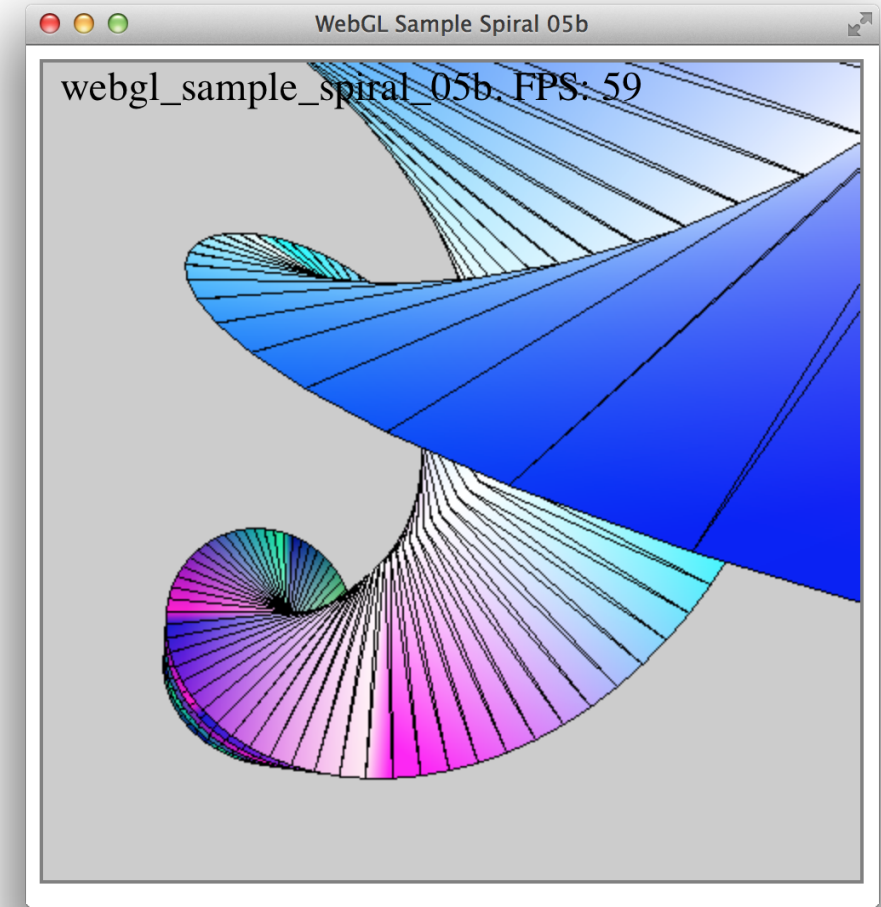
```
function draw(currentTime) {
  // 1. 現在のフレーム描画を開始する前に次のフレームを
  //    描画する新たな呼び出しを要求

  requestAnimationFrame(draw);

  // 2. シーンの中で動くオブジェクトの位置を更新
  ...
  // 3. シーンを描画
```

演習

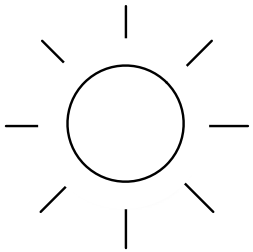
- webgl_sample_spiral_05b2.html を自由に変更し、アニメーションで遊んでみよう。

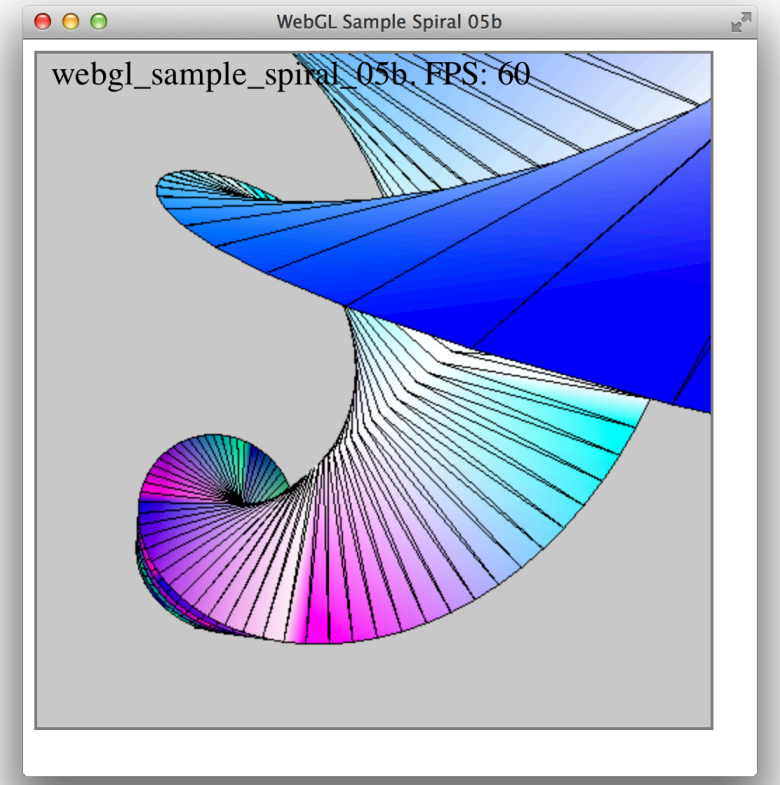
照明

- これまでは照明を全く考えていなかった
- 各頂点がそれぞれ指定された色で「光る」

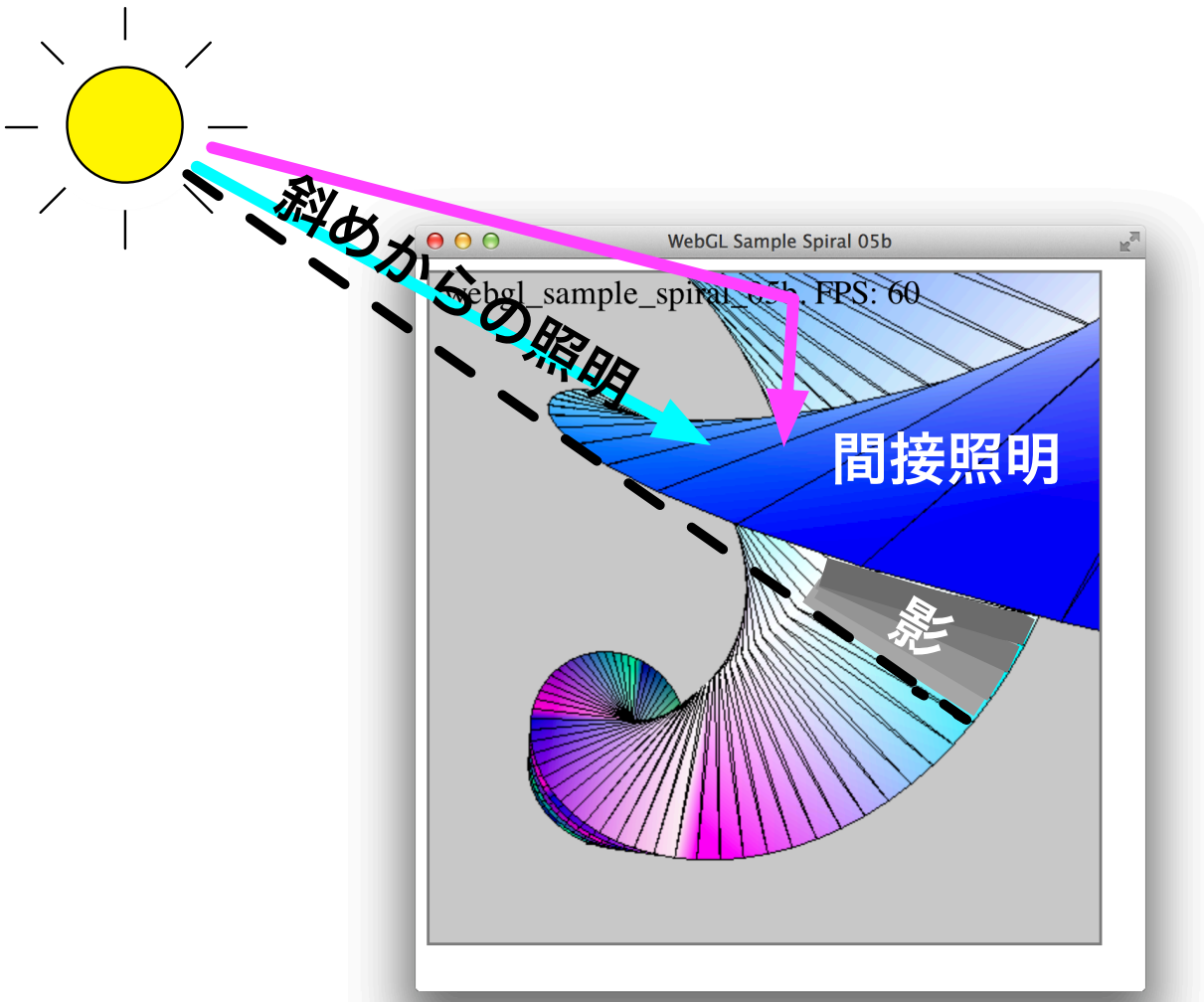


光源を置く





結構大変



二つの照明モデル

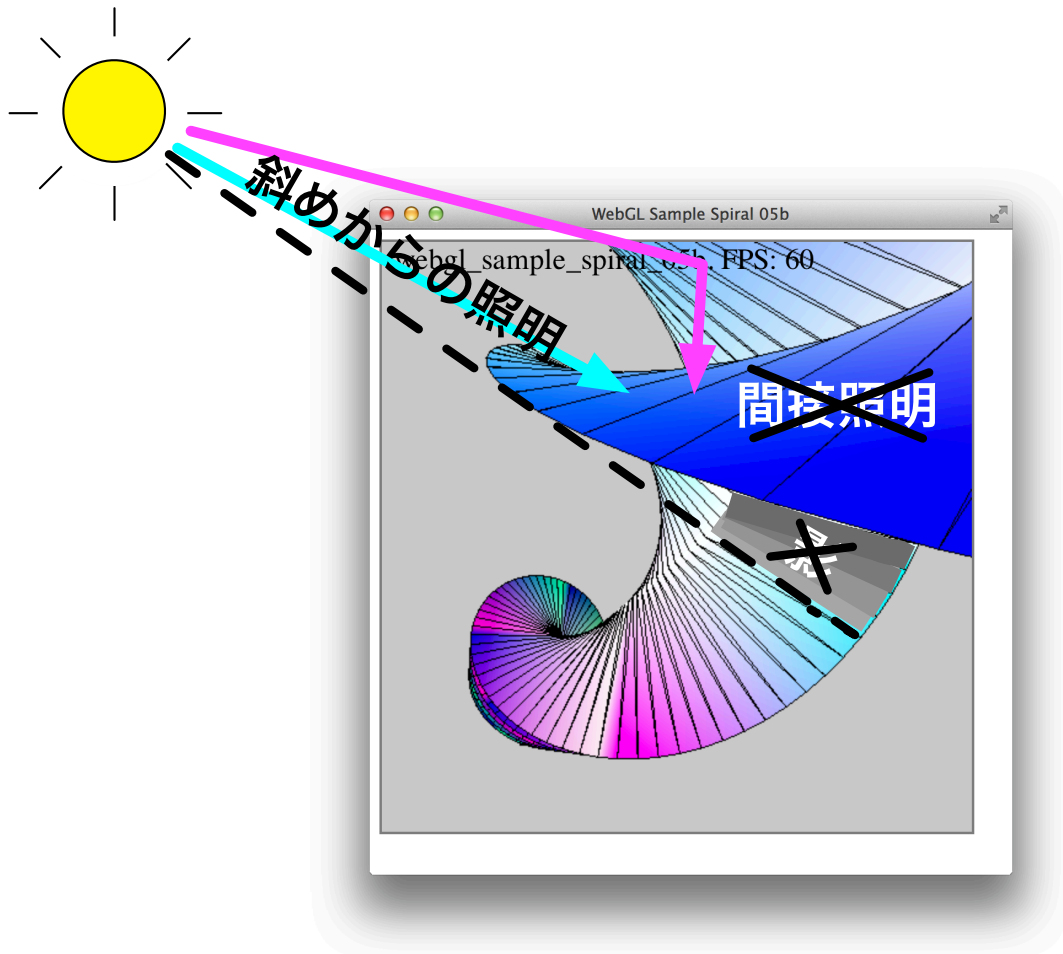
ある頂点の照明状態を計算するとき、

1. 大域照明モデル： 他の物体の存在を計算に入れる。
2. 局所照明モデル： 他の物体の存在を無視する 。

局所照明モデル

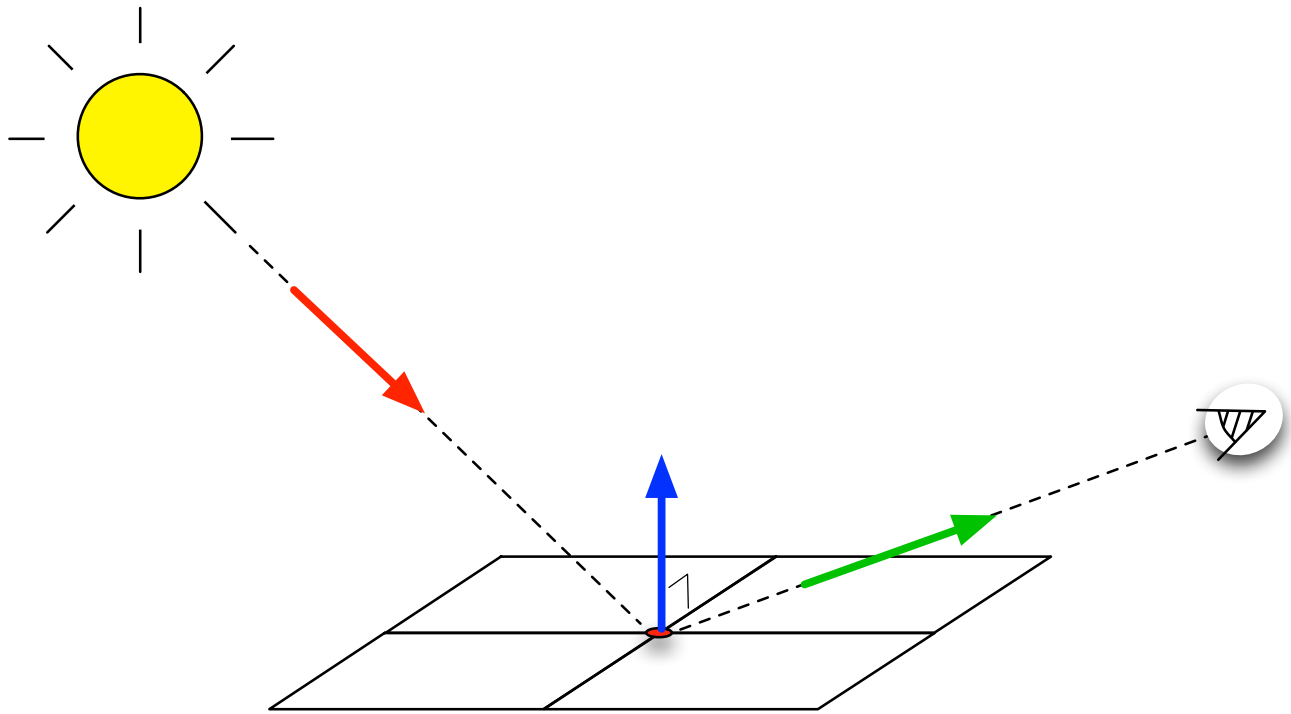
間接照明や影はできない。

間接照明は後述する環境光によって擬似的にその効果を入れる。



局所照明モデル

ある頂点での色（反射）は、3 つの単位ベクトルの関数。



フォン（Phong）の反射モデル

Bui Tuong Phong (1942–1975)

- 局所モデル
- 3 種類の光を考える。
  - 環境光（Ambient Light）
  - 拡散光（Diffuse Light）
  - 鏡面光（Specular Light）
- それぞれが独立に物体を照らす。
- それぞれの光は独立の色（RGB）を持つ。

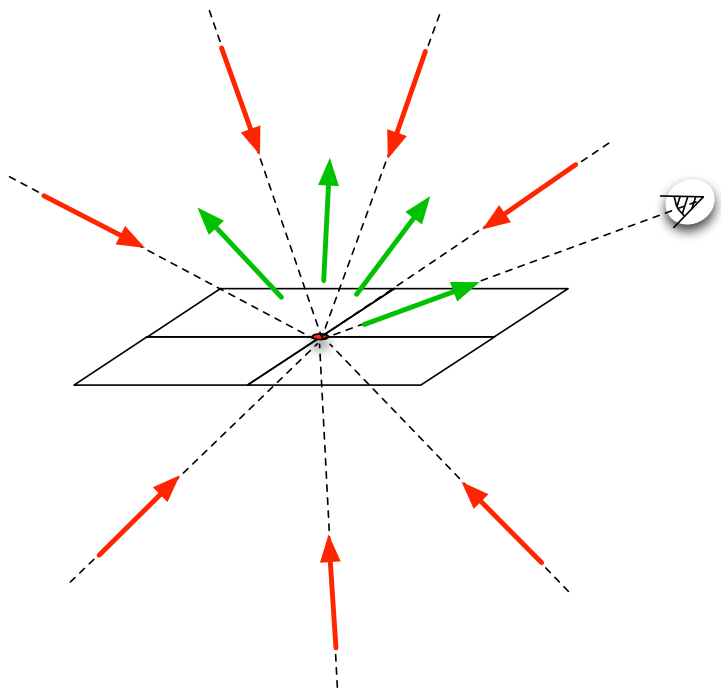
フォン（Phong）の反射モデル

物体の見え方（色や質感）の表現

- 3 つの色のそれぞれに対して頂点が独立に反射する。
  - 環境光反射（Ambient Reflection）
  - 拡散反射（Diffuse Reflection）
  - 鏡面反射（Specular Reflection）

環境光

- あらゆる方向からくる光による照明
- 面の裏側からも光が来る。
- 間接照明の模擬
- 弱めに入れる。例： $(0.2, 0.2, 0.2, 1.0)$



環境光のシェーダコード

バーテックスシェーダ

```
uniform vec3 uAmbientLightColor;
varying vec3 vLightWeighting;

vLightWeighting = uAmbientLightColor;
```

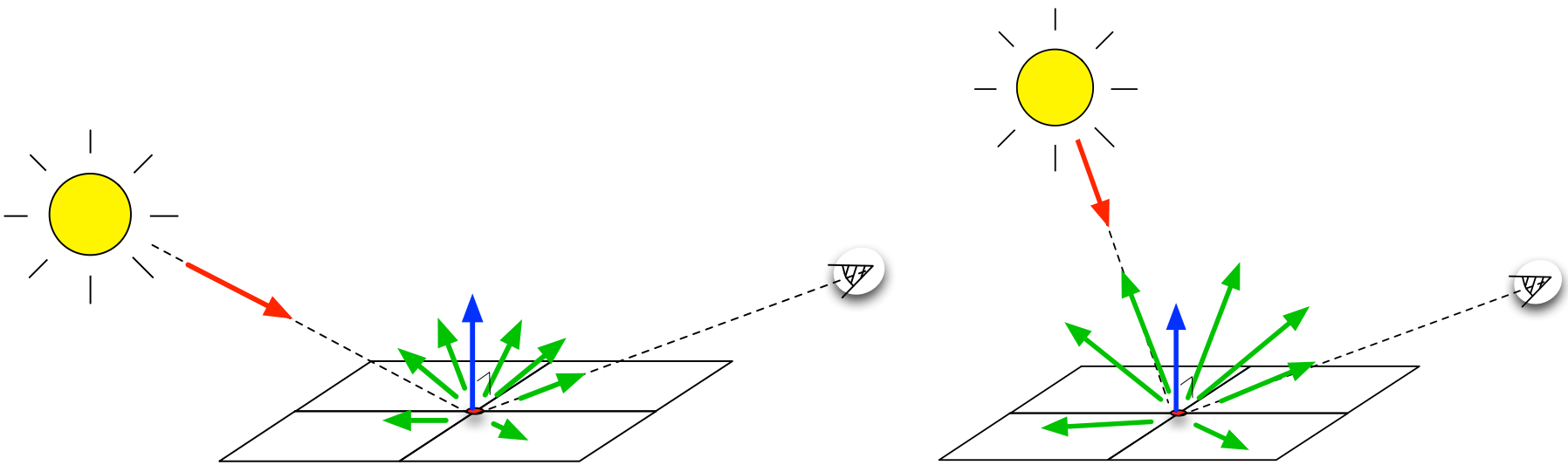
環境光のシェーダコード

フラグメントシェーダ

```
varying vec3 vLightWeighting;
vec4 texelColor = texture2D(uSampler, vTextureCoordinates);
gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb,
texelColor.a);
```

拡散光

- 光の入射角に依存。
- 頂点の法線ベクトル $\boldsymbol{n}$ と、光の方向ベクトル $\boldsymbol{\ell}$ のなす角度で決まる。
- $\boldsymbol{n} \cdot \boldsymbol{\ell}$ の関数。
- カメラの方向 $\boldsymbol{e}$ には依存しない。



拡散光のシェーダコード

バーテックスシェーダ

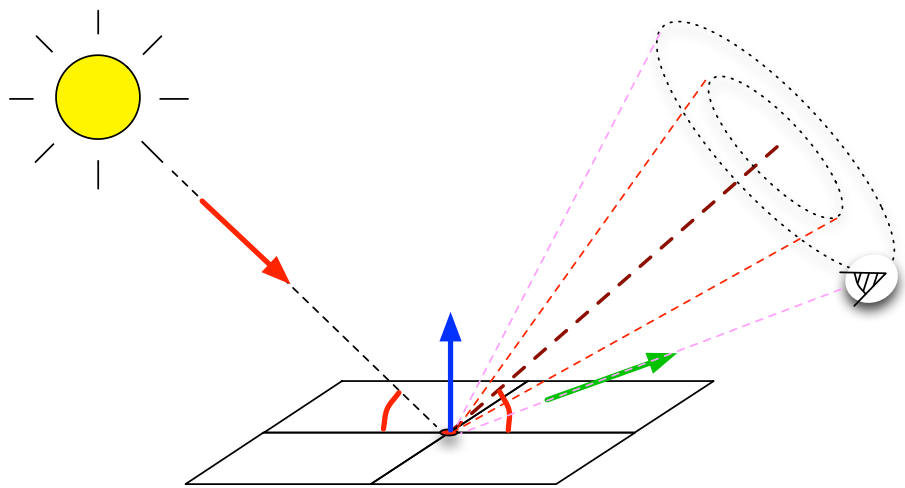
```
float diffuseLightWeightning = max(dot(normalEye,
     vectorToLightSource), 0.0);
vLightWeighting = uAmbientLightColor +
     uDiffuseLightColor * diffuseLightWeightning;
```

拡散光のシェーダコード

フラグメントシェーダ【変更なし】

鏡面光

- 金属のような光沢
- 頂点位置に鏡があるとして、入射する光が反射する方向にカメラがあるときに最も明るくなる。
- 頂点の法線ベクトル $\boldsymbol{n}$ と、光の方向ベクトル $\boldsymbol{\ell}$ と、カメラの方向 $\boldsymbol{e}$ の関数。
- 光沢度（鏡への近さ）を決めるパラメータを使う。反射する光がどれほど広がるか。



鏡面光のシェーダコード

バーテックスシェーダ

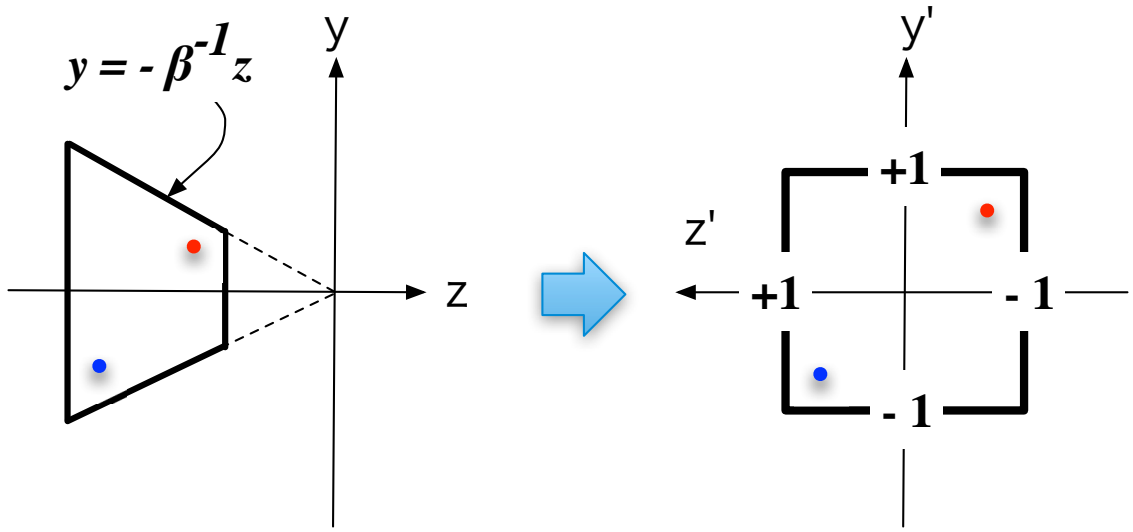
```
const float shininess = 32.0;
float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);
float specularLightWeightning = pow(rdotv, shininess);
vLightWeighting = uAmbientLightColor +
       uDiffuseLightColor * diffuseLightWeightning +
       uSpecularLightColor * specularLightWeightning;
```

鏡面光のシェーダコード

フラグメントシェーダ【変更なし】

復習

透視射影（以前の講義資料より）



$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta\, x/z \\ -\beta\, y/z \\ c_1\, z + c_2 \end{pmatrix}$$

$\beta > 0, c_1, c_2$ は定数。x と y に関しては $-\beta/z$ 倍のスケール変換。

このスケール係数は、たとえば、視錐台の上面 $y = -\beta^{-1}\, z$ が $y' = +1$ に変換されることで容易に確認できる。

透視射影変換の二つのステップ（以前の講義資料より）

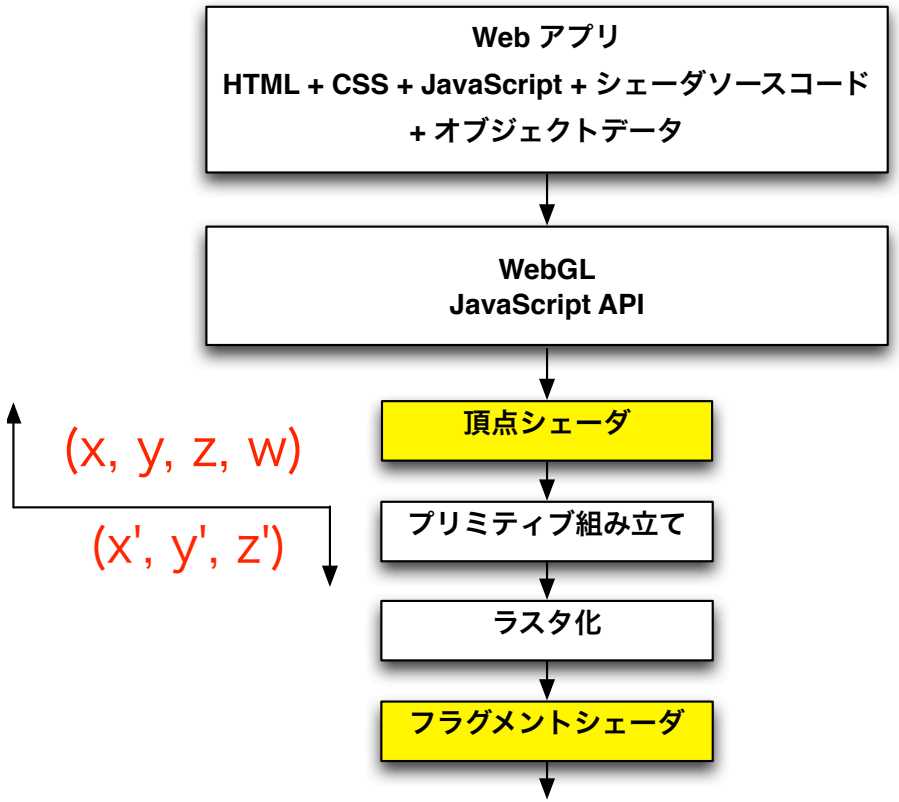
(1) 同次座標の行列演算（この行列は一意には決まらない。ここで挙げるのは一つの例。）

$$\begin{pmatrix} x^\dagger \\ y^\dagger \\ z^\dagger \\ w^\dagger \end{pmatrix} = \begin{pmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \alpha & \gamma \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(2) w 座標での割り算。これを透視除算という。プリミティブ組み立て時に実行される。

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x^\dagger/w^\dagger \\ y^\dagger/w^\dagger \\ z^\dagger/w^\dagger \end{pmatrix}$$

パイプラインでの座標の処理（以前の講義資料より）



- $(x', y', z') = (x/w, y/w, z/w)$
- GPU が自動的に実行

サンプルコード

サンプルコードを見る前に

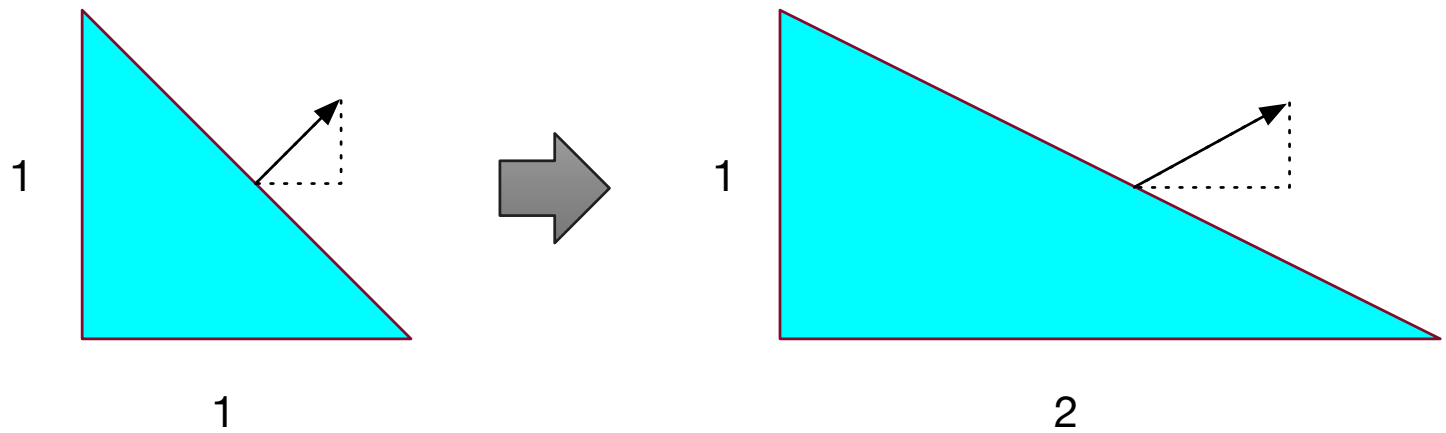
様々な方向ベクトルのシェーダでの計算方法

頂点から光源へ向かう単位ベクトル

バーテックスシェーダ

```
// Get the vertex position in eye coordinates
vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
vec3 vertexPositionEye3 = vertexPositionEye4.xyz /
vertexPositionEye4.w;

// Calculate the vector (l) to the light source
vec3 vectorToLightSource = normalize(uLightPosition -
vertexPositionEye3);
```

法線ベクトルの変換



法線ベクトルの変換

法線ベクトルの変換行列は、モデルビュー行列（の 3 行 3 列部分）の転置の逆である（証明は次のページ)。

```
// Transform the normal (n) to eye coordinates
vec3 normalEye = normalize(uNMatrix * aVertexNormal);
```

法線ベクトルの変換行列

ある点での面の法線ベクトルを $\boldsymbol{n}$, その点で面に接する任意のベクトルを $\boldsymbol{a}$ とする。

$$\boldsymbol{n} \cdot \boldsymbol{a} = \boldsymbol{n}^t \ \boldsymbol{a} = 0 \tag{1}$$

オブジェクトの変換行列を M、法線ベクトルの変換行列を N とすると、

$$\boldsymbol{a} \to \boldsymbol{a}' = M\boldsymbol{a}, \quad \boldsymbol{n} \to \boldsymbol{n}' = N\boldsymbol{n}$$

$\boldsymbol{n}'$ と $\boldsymbol{a}'$ が直交するから

$$0 = \boldsymbol{n}' \cdot \boldsymbol{a}' = (\boldsymbol{n}')^t \boldsymbol{a}' = (N\boldsymbol{n})^t \, (M\boldsymbol{a}) = \boldsymbol{n}^t N^t M \boldsymbol{a}$$

式 (1) より

$$N^t M = I$$

$$\therefore \quad N = (M^t)^{-1}$$

M が直交行列なら $N = M$.

反射方向ベクトル

バーテックスシェーダに reflect という関数が組み込まれている。

```
// Calculate the reflection vector (r) that is needed for
specular light
vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
normalEye));
```

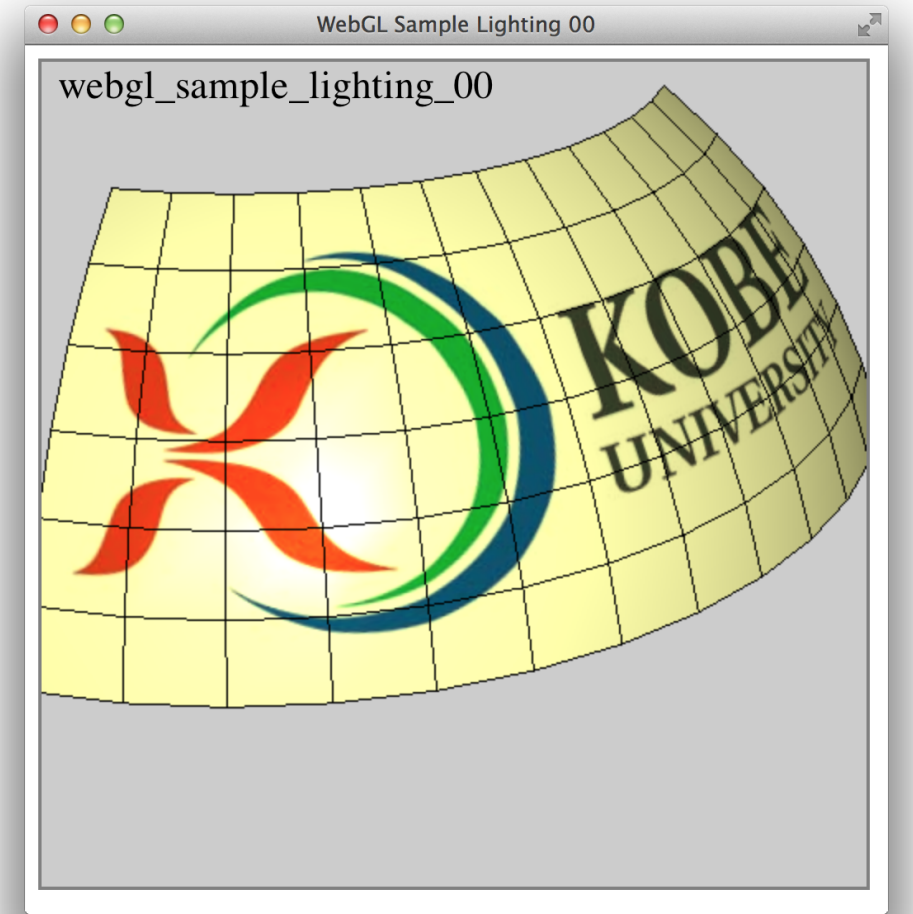
頂点からカメラ方向へのベクトル

原点（カメラ）からみた、頂点の位置ベクトルの逆。

```
vec3 viewVectorEye = -normalize(vertexPositionEye3);
```

サンプルコード01

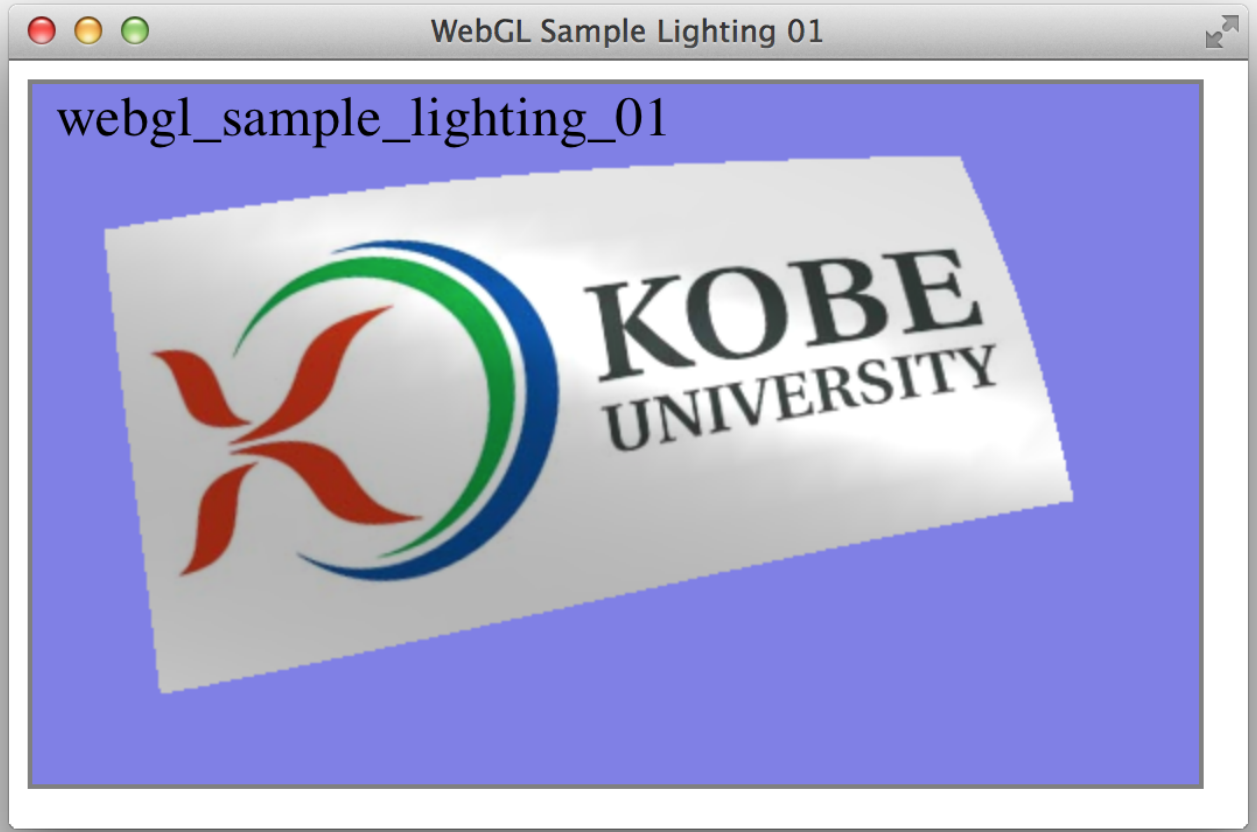
webgl_sample_lighting_00.html



サンプルコード02

webgl_sample_lighting_01.html

頂点と法線の変形をバーテックスシェーダで計算している。



サンプルコード03

webgl_spiral_kobe.html

演習

- webgl_sample_lighting_00.html をベースとして照明のパラメータを変更してその効果を見よう。

全体のまとめ

シェーダとシェーディング言語： GLSL

OpenGL シェーディング言語（OpenGL SL, GLSL）4.0

GPU を使うための言語

CG ソースコード	=	OpenGL ソースコード
	+	GLSL(フラグメントシェーダ）ソースコード
	+	GLSL(頂点シェーダ）ソースコード

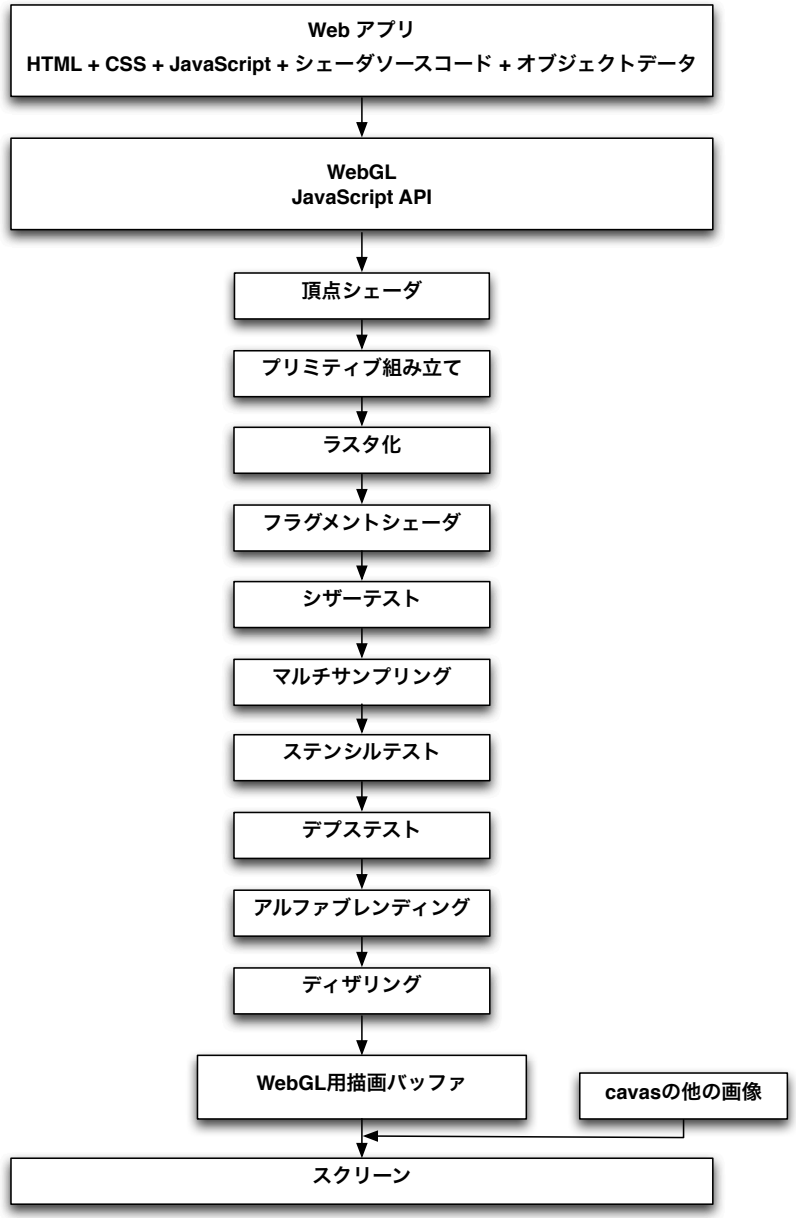
WebGL とは

WebBL = シェーダを使い、HTML5 の canvas に、JavaScript で 3D CG を書くための API

WebGL の特徴

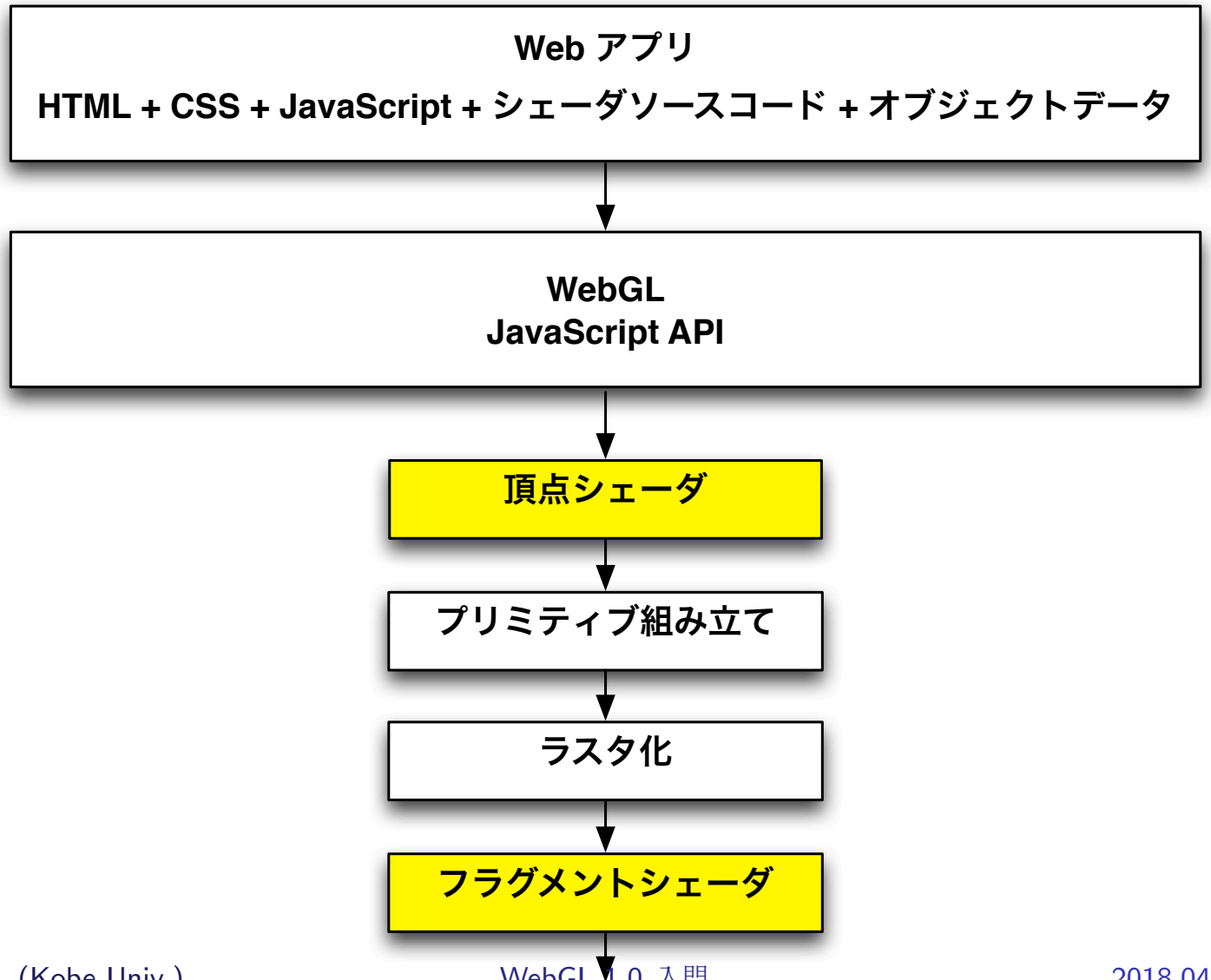
- クロスプラットフォーム
- オープンスタンダード
- Web で GPU を使ったレンダリングが可能
- 開発・利用が容易： プラグイン不要
- ソースコードが見える
- グラフィックス（OpenGL）と UI（ウィンドウ管理やイベント処理）の分離が明白

WebGL のグラフィックスパイプライン



シェーダ

頂点シェーダ（バーテックスシェーダ）とフラグメントシェーダ



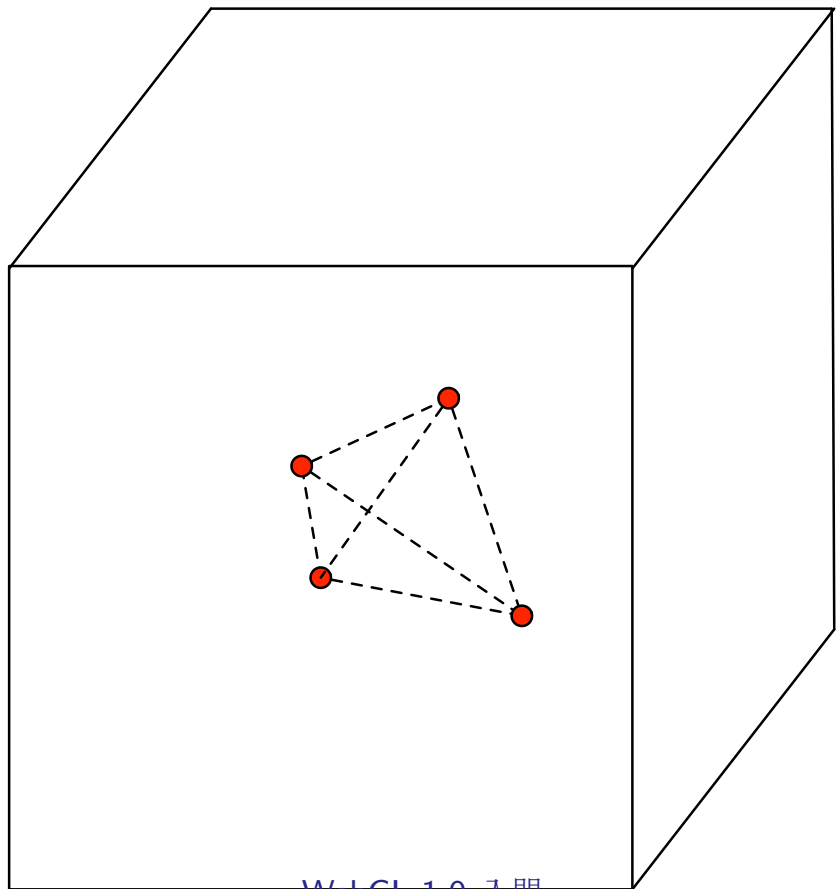
WebGL アプリケーション

Web アプリ ＝ HTML + CSS + JavaScript

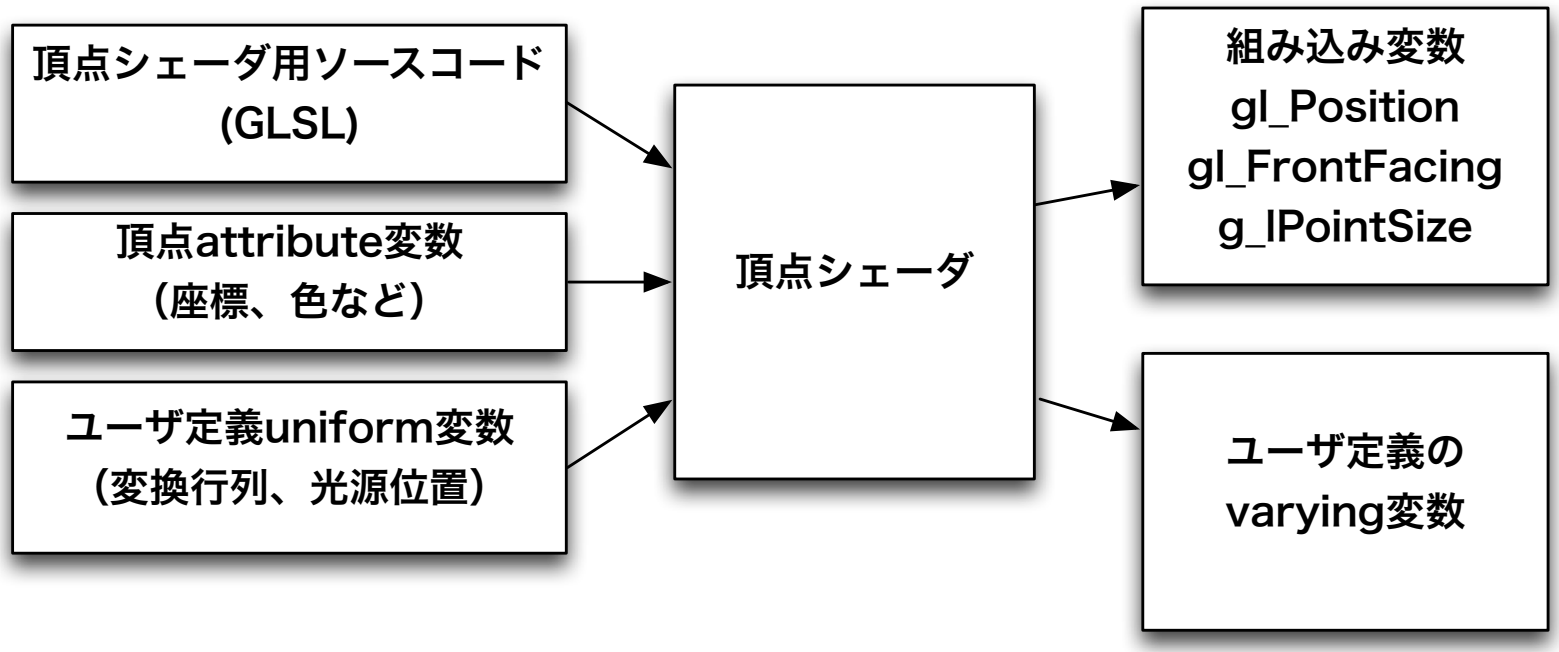
WebGL アプリ ＝ HTML + CSS + JavaScript + シェーダ言語（OpenGL SL）

頂点シェーダ

- 各頂点に対して処理を行う
- 並列処理
- n 個の頂点があれば n 個の頂点シェーダプロセッサを同時に実行させる

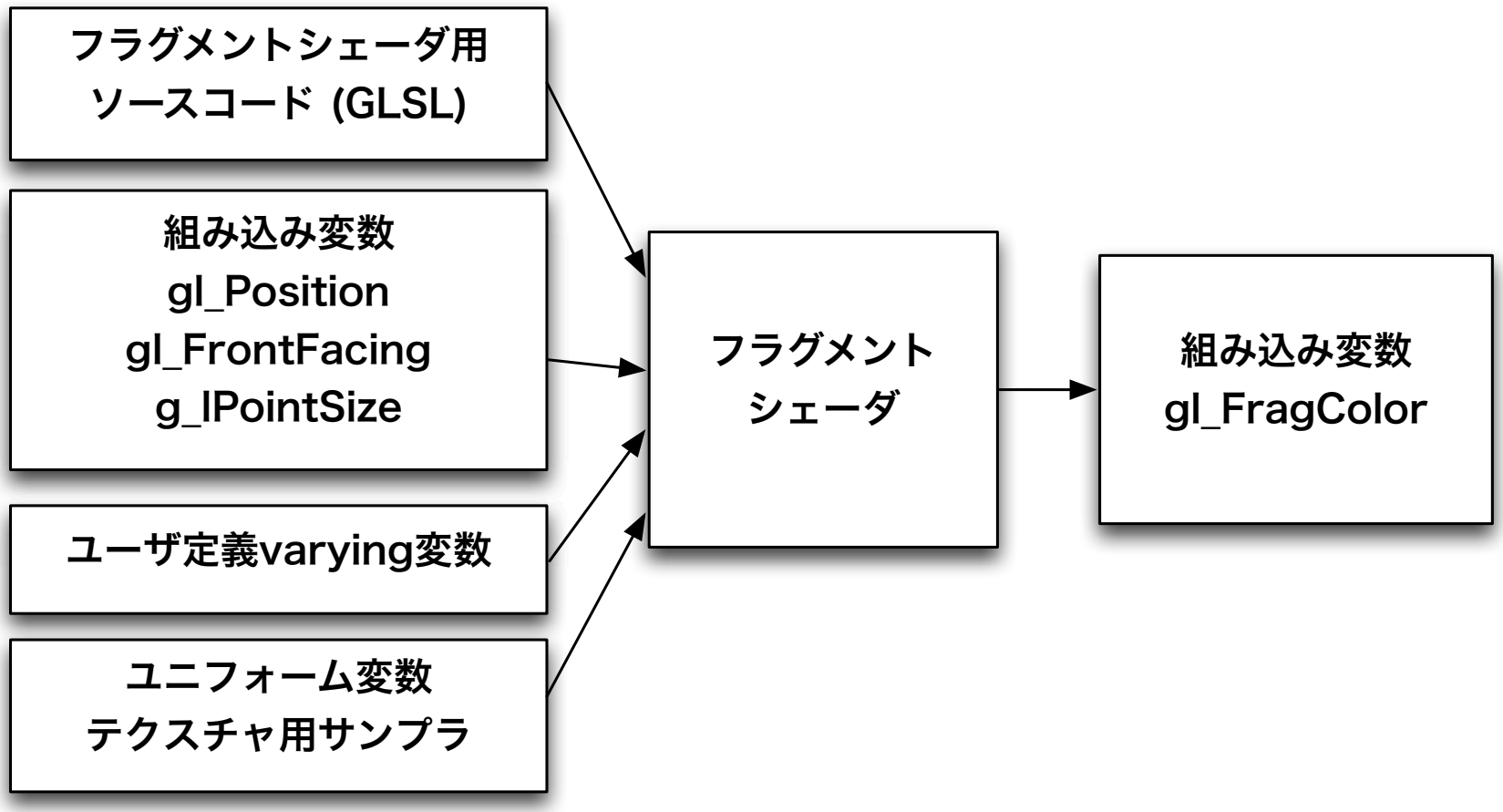


頂点シェーダの入出力データ



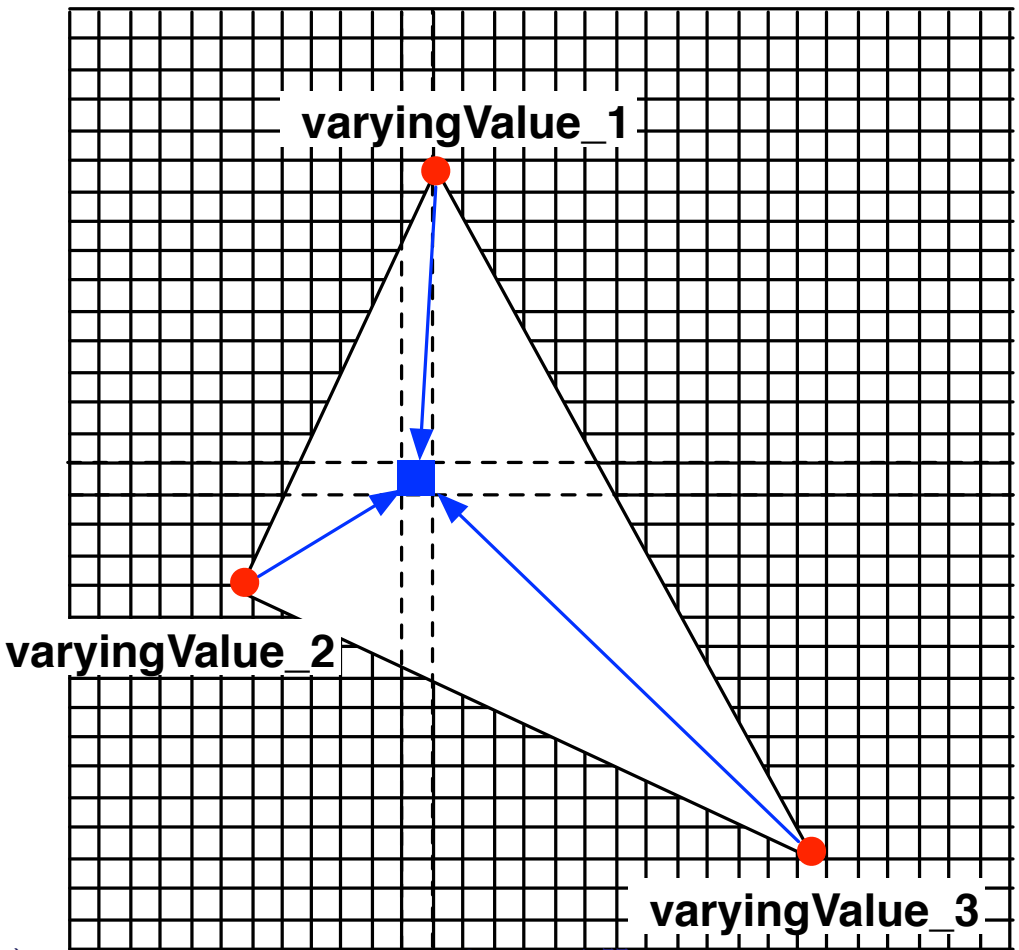
フラグメントシェーダの入出力

全てのフラグメントで並列処理。シェーディング言語でプログラム。



varying 変数の補間

- 頂点シェーダ からフラグメントシェーダへは varying 変数を通じて情報を送る。
- 各フラグメントの varying 変数値は自動的に線形補間される。



WebGL での 3D 描画プログラム

- 頂点データの生成と転送
- 法線データの生成と転送
- テクスチャデータの生成と転送
- 物体の座標変換（4 行 4 列）
- 材質（反射）特性設定
- 照明設定
- 射影変換（4 行 4 列）

  ••• 面倒

3D CGライブラリ

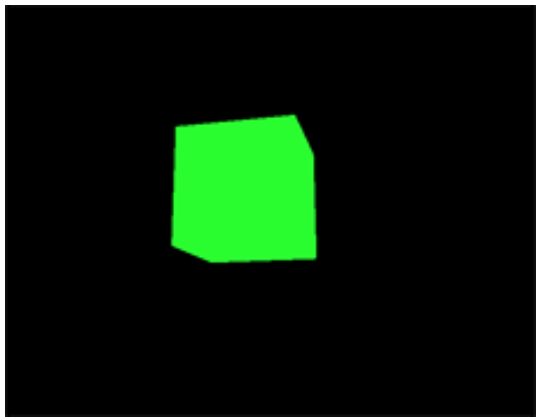
WebGL 用 JavaScript ライブラリ

WebGL のラッパ

- Three.js （`http://threejs.org`）
- Away3D TypeScript
- Babylon.js

Three.js sample

回転する直方体



サンプルコード

three_js_sample_cube.js

必要なライブラリ： three.min.js

```
<html>
<head>
<title>My first Three.js app</title>
<style>canvas { width: 100%; height: 100% }</style>
</head>

<body>
<script src="js/three.min.js"></script>
<script>
  var scene = new THREE.Scene();
  var camera = new THREE.PerspectiveCamera(75, window.innerWidth/wi
  var renderer = new THREE.WebGLRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);
  var geometry = new THREE.CubeGeometry(1,1,1);
  var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
  var cube = new THREE.Mesh(geometry, material);
```

```
  scene.add(cube);
  camera.position.z = 5;

  var render = function () {
    requestAnimationFrame(render);
    cube.rotation.x += 0.01;
    cube.rotation.y += 0.01;
    renderer.render(scene, camera);
  };

render();
</script>

</body>
</html>
```

演習

- （Three.js などのライブラリは使わずに）照明とテクスチャマッピング、アニメーションを使った WebGL プログラムを作ろう。

Appendix

数学関係のまとめ

同次座標

同次座標（homogeneous coordinates）とは 3 次元空間中の位置座標 $\boldsymbol{x}$ と、任意のベクトル $\boldsymbol{v}$ をあえて 4 成分で表現したもの。

3 次元空間中の位置座標 $\boldsymbol{x}$ を

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{2}$$

ベクトル $\boldsymbol{v}$ は

$$\begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} \tag{3}$$

アフィン変換

3 次元空間の位置座標 $\boldsymbol{x}$ や、ベクトル $\boldsymbol{v}$ の変換を考える。

$$\boldsymbol{x} \longrightarrow \boldsymbol{y} \equiv F(\boldsymbol{x}). \tag{4}$$

線形変換＝スケール変換＋回転＋剪断。

アフィン変換＝線形変換＋平行移動。

平行移動は 3 行 3 列の行列では書けない。

同次座標と 4 行 4 列の行列を使えば書ける。

線形代数の復習：内積

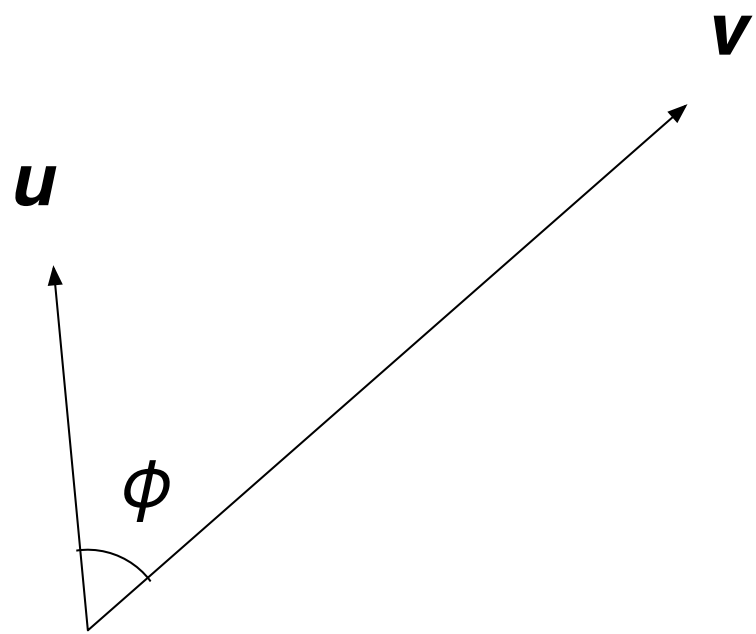
n 次元空間中のベクトルと正方行列

ベクトル $\boldsymbol{u}$ の大きさ

$$u = |\boldsymbol{u}|$$

内積

$$\boldsymbol{u} \cdot \boldsymbol{v} = u_i v_j = u\, v\, \cos\phi$$



線形代数の復習：正規直交系

$$\boldsymbol{e}_i \cdot \boldsymbol{e}_j = \delta_{ij} \quad （クロネッカーのデルタ）$$

一般のベクトル $\boldsymbol{v}$ と正規直交系 $\{\boldsymbol{e}_0, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_{n-1}\}$

$\boldsymbol{v}$ の i 成分

$$v_i = \boldsymbol{v} \cdot \boldsymbol{e}_i$$

線形代数の復習：外積

3 次元空間のベクトル

$$\boldsymbol{w} = \boldsymbol{u} \times \boldsymbol{v} \qquad w_i = \epsilon_{ijk} u_j \, v_k \quad \text{（エディントンのイプシロン）}$$

$\boldsymbol{w}$ は $\boldsymbol{u}$ と $\boldsymbol{v}$ の両方に垂直

$$w = u \, v \, \sin\phi$$

$$\boldsymbol{u} \times \boldsymbol{v} = -\boldsymbol{v} \times \boldsymbol{u}$$

$$\boldsymbol{u} \cdot (\boldsymbol{v} \times \boldsymbol{w}) = (\boldsymbol{u} \cdot \boldsymbol{w}) \, \boldsymbol{v} - (\boldsymbol{u} \cdot \boldsymbol{v}) \, \boldsymbol{w}$$

線形代数の復習：行列のかけ算

$\boldsymbol{M}$ と $\boldsymbol{N}$ は行列

行列 $\boldsymbol{M}$ の成分を $M_{ij} \quad (i, j = 0, 1, \ldots, n-1)$

行列 $\boldsymbol{N}$ の成分を $N_{ij} \quad (i, j = 0, 1, \ldots, n-1)$

とすると

$$\boldsymbol{L} = \boldsymbol{M}\boldsymbol{N}$$

の成分は

$$L_{ij} = \sum_{k=0}^{n-1} M_{ik} N_{kj} = M_{ik} N_{kj}$$

線形代数の復習：行列のかけ算

$$(\boldsymbol{LM})\,\boldsymbol{N} = \boldsymbol{L}\,(\boldsymbol{MN})$$

$$(\boldsymbol{L}+\boldsymbol{M})\,\boldsymbol{N} = \boldsymbol{LN}+\boldsymbol{MN}$$

$$\boldsymbol{MI} = \boldsymbol{IM} = \boldsymbol{M} \quad (\boldsymbol{I}：単位行列)$$

一般には非可換：

$$\boldsymbol{MN} \neq \boldsymbol{NM}$$

線形代数の復習：逆行列

正方行列 $\boldsymbol{M}$ に対して

$$\boldsymbol{M}\boldsymbol{N} = \boldsymbol{N}\boldsymbol{M} = \boldsymbol{I}$$

という行列 $\boldsymbol{N}$ を逆行列という。

逆行列を

$$\boldsymbol{M}^{-1}$$

と書く。

一般には逆行列を求めるのは大変（計算量が多い）

glMatrix.js（後述）では 4 行 4 列の逆行列を求める関数が組み込まれている。

線形代数の復習：行列式

正方行列に対して

$$\det(\boldsymbol{M})$$

$$\det(\boldsymbol{I}) = 1$$

$$\det(\boldsymbol{M}\boldsymbol{N}) = \det(\boldsymbol{M})\,\det(\boldsymbol{N})$$

$$\det\left(\boldsymbol{M}^t\right) = \det(\boldsymbol{M})$$

線形代数の復習：行列の転置

行列 $\boldsymbol{M}$ の成分を $M_{ij} \quad (i, j = 0, 1, \ldots, n-1)$

転置行列を $\boldsymbol{M}^t$ と書く

a を数、$\boldsymbol{M}$ と $\boldsymbol{N}$ を行列として

$$(a\boldsymbol{M})^t = a\boldsymbol{M}^t$$

$$(\boldsymbol{M} + \boldsymbol{N})^t = \boldsymbol{M}^t + \boldsymbol{N}^t$$

$$\left(\boldsymbol{M}^t\right)^t = \boldsymbol{M}$$

$$(\boldsymbol{M}\boldsymbol{N})^t = \boldsymbol{N}^t\boldsymbol{M}^t$$

線形代数の復習：行列のトレース

$$tr(M) = \sum_{i=0}^{n-1} M_{ii}$$

線形代数の復習：直交行列

$$\boldsymbol{M}\boldsymbol{M}^t = \boldsymbol{M}^t\boldsymbol{M} = \boldsymbol{I}$$

を満たす正方行列 $\boldsymbol{M}$ を直交行列という。

$$\boldsymbol{M}^t = \boldsymbol{M}^{-1}$$

$$\det(\boldsymbol{M}) = \pm 1$$

$\boldsymbol{M}^t$ も直交行列。

直交行列はベクトルの長さを変えない：

$$|\boldsymbol{M}\boldsymbol{u}| = |\boldsymbol{u}|$$

直交する二つのベクトルを直交行列で変換しても直交したまま。

$$\boldsymbol{u}\cdot\boldsymbol{v} = 0 \Longleftrightarrow (\boldsymbol{M}\boldsymbol{u})\cdot(\boldsymbol{M}\boldsymbol{v}) = 0$$

3 次元空間中の平面

点 $\boldsymbol{p}$ を通り、ベクトル $\boldsymbol{u}$ とベクトル $\boldsymbol{v}$ で張られる平面の式：

$$\boldsymbol{x} = \boldsymbol{p} + s\,\boldsymbol{u} + t\,\boldsymbol{v}$$

単位ベクトル $\boldsymbol{n} \equiv \boldsymbol{u} \times \boldsymbol{v}/|\boldsymbol{u} \times \boldsymbol{v}|$ を使えば、

$$\boldsymbol{n} \cdot \boldsymbol{x} + d = 0$$

$\boldsymbol{n}$ を法線ベクトルという。$f(\boldsymbol{x}) = \boldsymbol{n} \cdot \boldsymbol{x} + d$ とすると

$f(\boldsymbol{x}_0) = 0 \Longleftrightarrow$ 点 $\boldsymbol{x}_0$ はこの平面の上

$f(\boldsymbol{x}_0) > 0 \Longleftrightarrow$ 点 $\boldsymbol{x}_0$ は $\boldsymbol{p} + \boldsymbol{n}$ 側にある

$f(\boldsymbol{x}_0) < 0 \Longleftrightarrow$ 点 $\boldsymbol{x}_0$ は $\boldsymbol{p} - \boldsymbol{n}$ 側にある

面積

3 点 $\boldsymbol{p}$, $\boldsymbol{q}$, $\boldsymbol{r}$ を頂点とする 3 角形の面積

$$S = \frac{1}{2} \left| (\boldsymbol{p} - \boldsymbol{r}) \times (\boldsymbol{q} - \boldsymbol{r}) \right|$$

x-y 平面上におかれた n 角形の面積

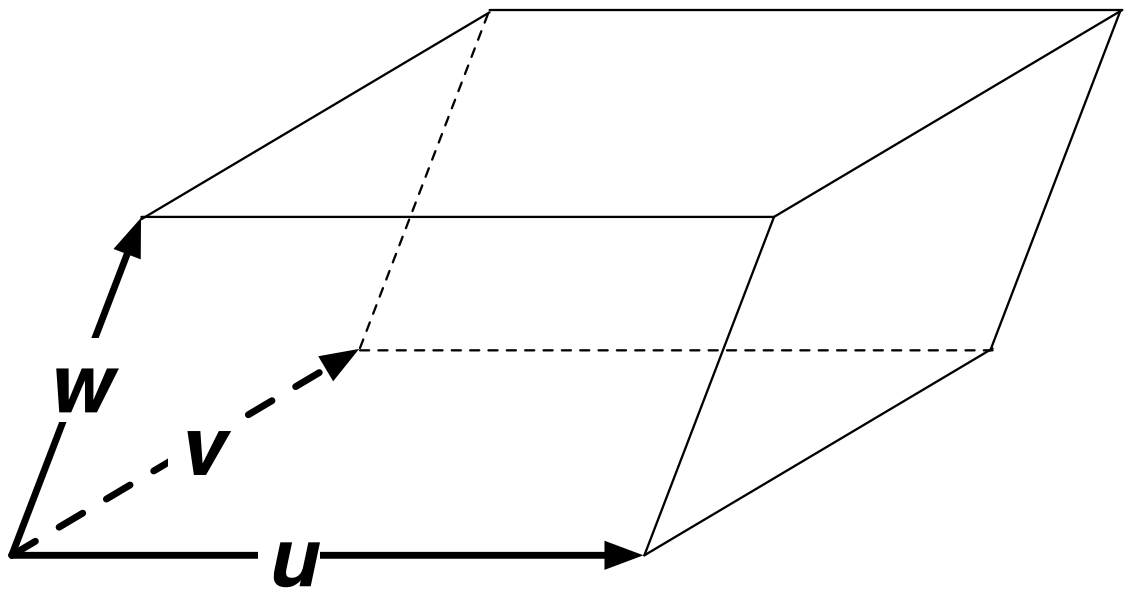
$$S = \frac{1}{2} \sum_{i=0}^{n-1} (x_i \, y_{i+1} - y_i \, x_{i+1}) = \frac{1}{2} \sum_{i=0}^{n-1} \{x_i \, (y_{i+1} - y_{i-1})\}$$

添字は　$\mathrm{mod}\ (n)$ をとる。

体積

原点を基点とする 3 つのベクトル $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$ が張る平行 6 面体の体積

$$V = \boldsymbol{u} \cdot (\boldsymbol{v} \times \boldsymbol{w}) = \boldsymbol{v} \cdot (\boldsymbol{w} \times \boldsymbol{u}) = \boldsymbol{w} \cdot (\boldsymbol{u} \times \boldsymbol{v})$$



同次座標

3 次元 $\Longrightarrow$ 4 次元

3 次元空間の位置座標

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \Longrightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

3 次元空間のベクトル

$$\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \Longrightarrow \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

同次座標

行列

$$\boldsymbol{M} = \begin{pmatrix} M_{00} & M_{01} & M_{02} & 0 \\ M_{10} & M_{11} & M_{12} & 0 \\ M_{20} & M_{21} & M_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

平行移動

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \end{pmatrix} \tag{5}$$

これを

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = T(t_x, t_y, t_z) \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{6}$$

と書けるような 3 行 3 列の行列 T は存在しない。$\Rightarrow$ 4 行 4 列にすれば OK.

平行移動

平行移動行列

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{7}$$

回転

z 軸の周りの回転

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{8}$$

スケール変換

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9}$$

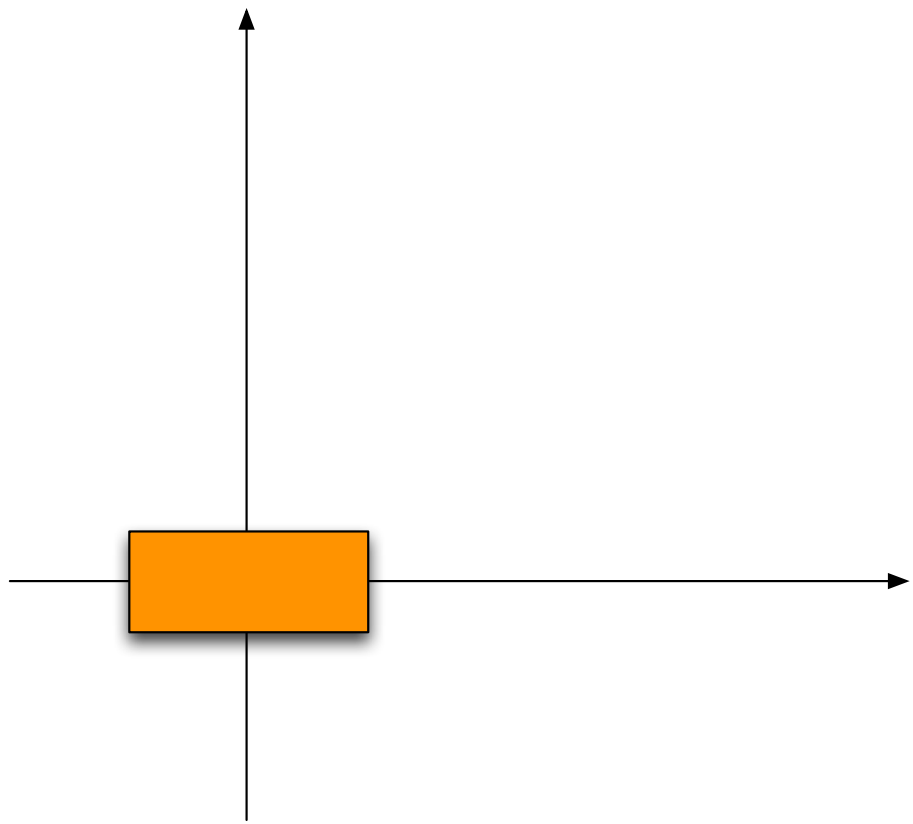
剪断

$$H_{xy}(\beta) = \begin{pmatrix} 1 & \beta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{10}$$

座標変換の合成

アフィン変換は非可換。一般に

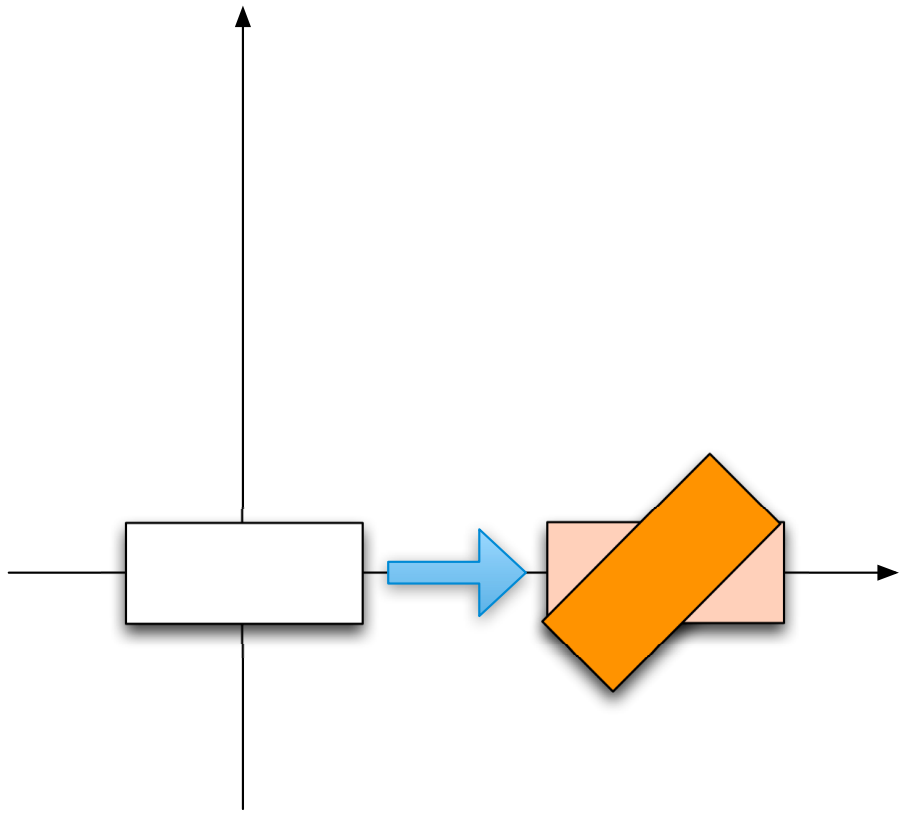
$$M_1 M_2 \neq M_2 M_1$$



座標変換の合成

アフィン変換は非可換。一般に

$$RT \neq TR$$



座標変換の合成

アフィン変換は非可換

$$RT \neq TR$$