チュートリアル:
ゲームグラフィクス・プログラミング入門
1時限目

(株)セガ AM R&D 2 山之内毅

このセッションについて

- 1時限目
  - ターゲットに2Dと3Dの絵を表示する
  - 多数のオブジェクトを動かす
  - ➡ グラフィクスAPIの使い方を学びます
- 2時限目
  - 3Dモデリングツールを使って任意形状をつくる
  - それをターゲットに即したデータにコンバート
  - ターゲットで表示
  - ➡ 描画のワークフローを学びます

サンプルの開発環境

- グラフィクスAPI
  - OpenGL ES 2.0
    - ➡固定機能を廃した、プログラマブルシェーダのみのグラフィクスAPI
    - ➡組み込み向け
- ターゲット
  - iPhone 3GS (XCODE, iPhone SDK 3.0)
  - GL ES 2.0エミュレータ (VS 2008 EE)

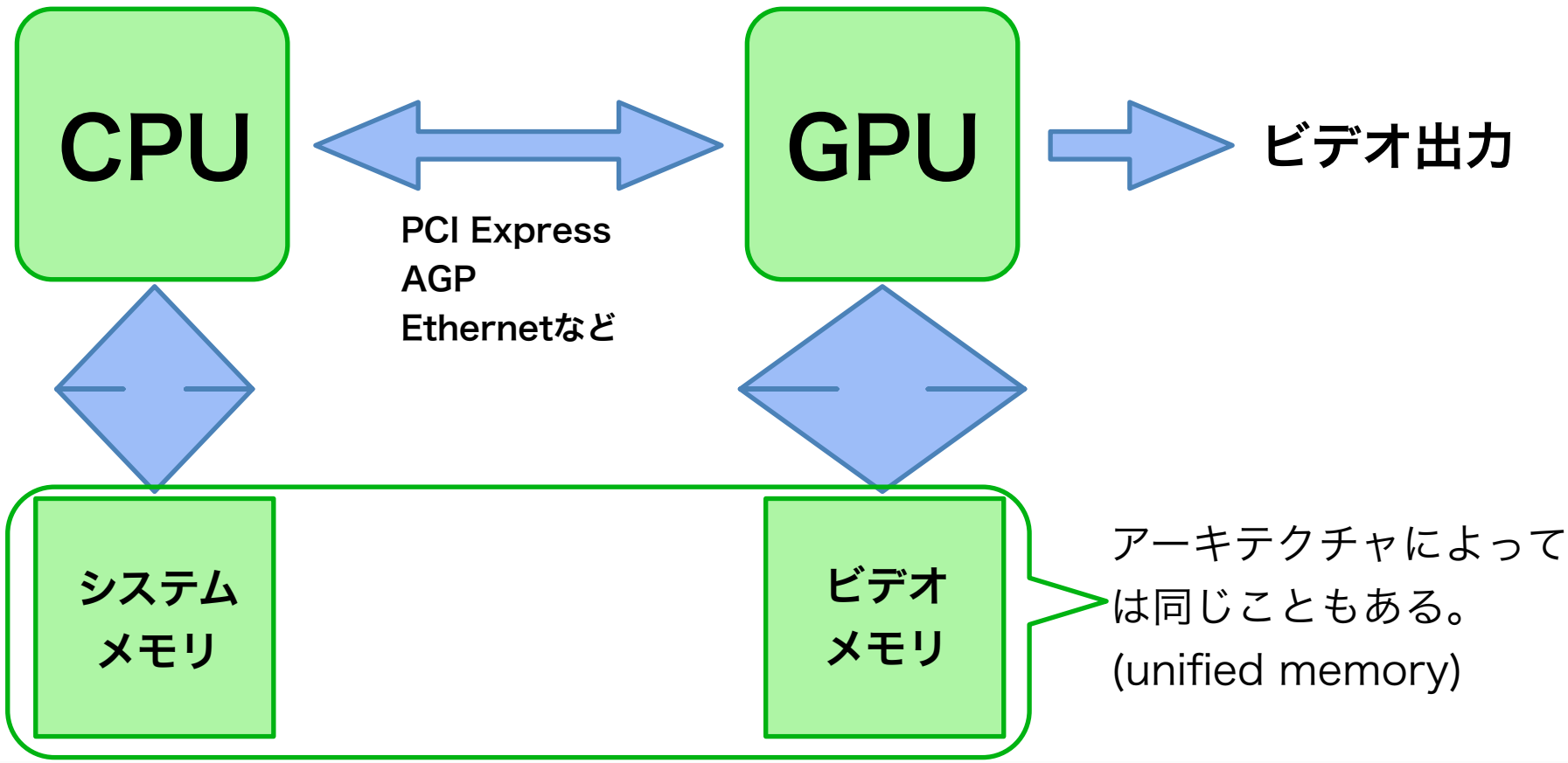
1時限目、目次

- グラフィクスハードウェア
- OpenGL(ES)概説
- 2Dの三角形を描く
- 3Dの三角錐を描く
- パフォーマンスのために

グラフィクスハードウェア

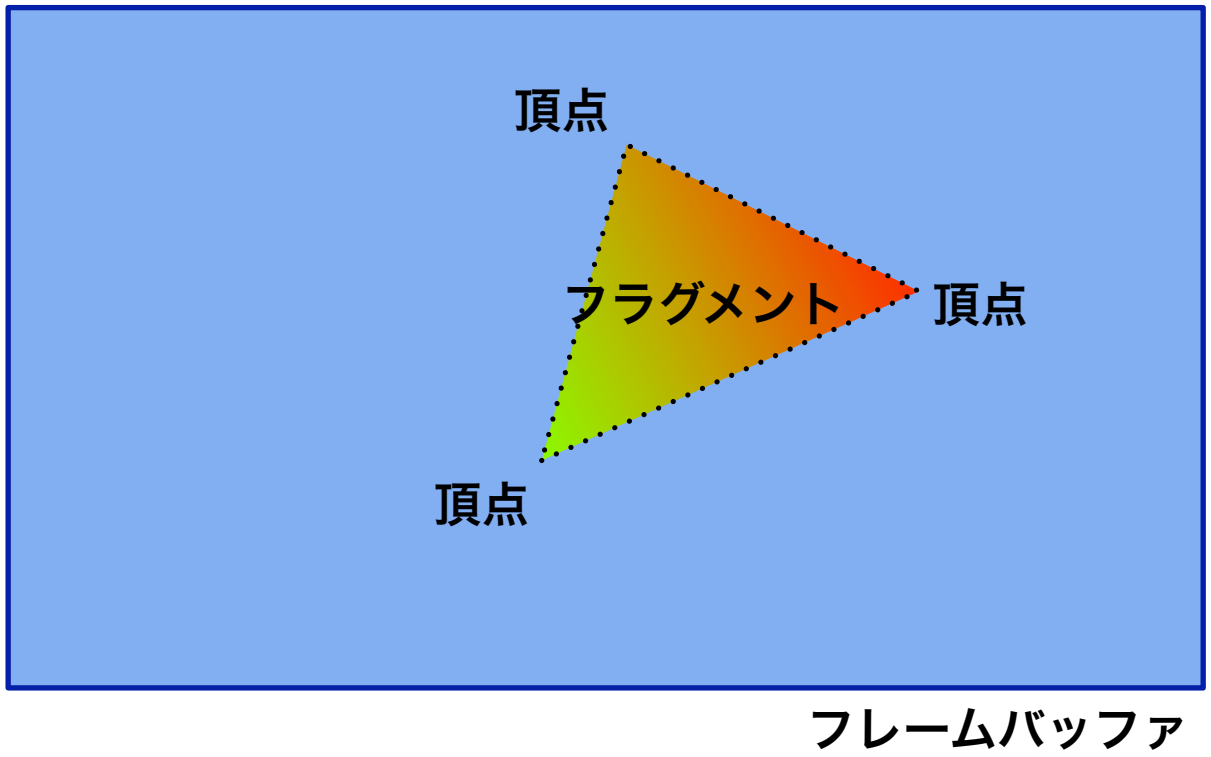
CPUとGPU

- GPUは画面を出力する機械
  - ➡ 高性能なGPUは、高帯域の専用メモリを持ち、多数の並列プロセッサでイメージを処理する。



GPUの描画機能

- 基本的に三角形を塗るだけ。
  - 頂点の座標計算
  - フラグメントの色計算



プログラマブルシェーダ

- 昔は、頂点やフラグメントの計算方法が固定されていた。パラメータの変更のみ可能。
  - ➡ 固定機能
  - ➡ OpenGL ES 1.x (iPhone 3Gまで搭載)

- 今は、自分の好きな処理をするコードを送り込むことが可能。
  - ➡ プログラマブルシェーダ
  - ➡ OpenGL ES 2.x (iPhone 3GS以降に搭載)

ベクトルとGPU

- ベクトルは数をならべたもの。
  - v = (3, 1, 4, 1, 5, 9, 2, 6, 5, ...)
- 3D CGの世界では、座標や色を表現するのに使う。
  - vec3 pos = vec3(x, y, z);
  - vec4 color = vec4(r, g, b, a);

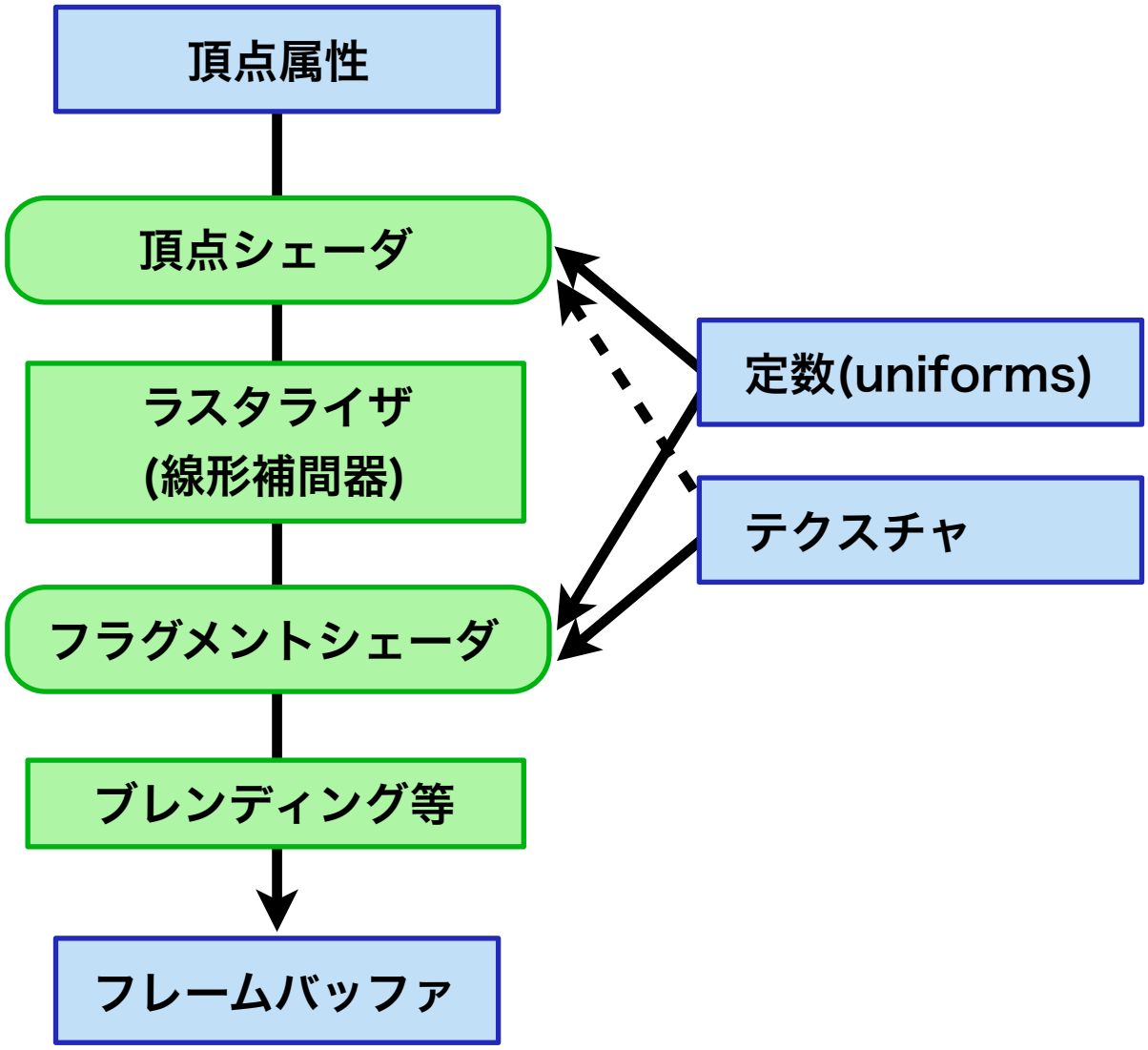
    ➡ まとめられて便利。
- 通常のGPUでは、4要素のベクトルを扱える。
  - pos = pos + vec3(1, 2, 0);

    ➡ 座標を x方向へ+1, y方向へ+2 移動する。

OpenGL(ES)概説

レンダリングパイプライン

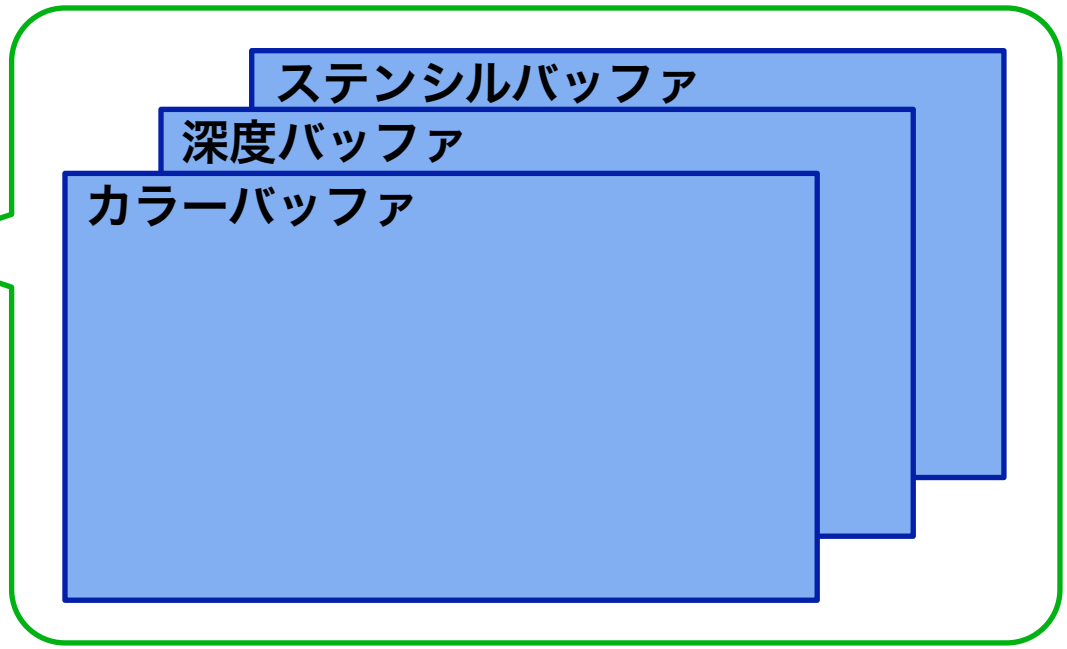
- OpenGL ES 2.0のパイプライン



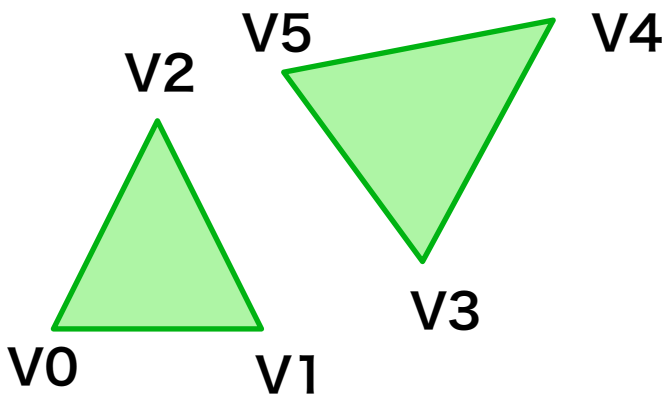
フレームバッファ

- 絵を描くカンバスのこと。一般に次の三種で構成。
  - カラーバッファ(R, G, B, A)
  - 深度バッファ(隠面処理用)
  - ステンシルバッファ(特殊効果用)

iPhone SDKでは、これらを
FramebufferObject(FBO)
と言うオブジェクトで管理。

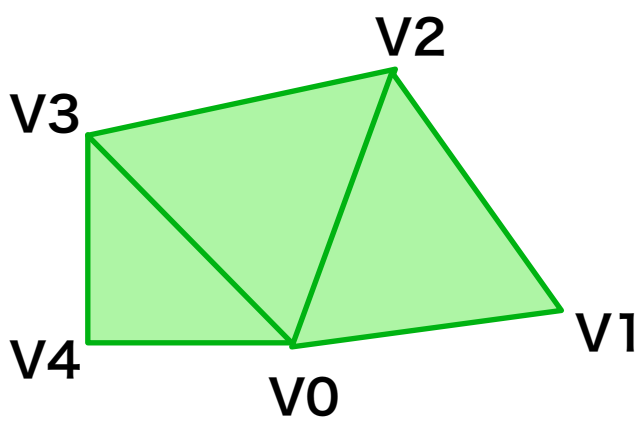


プリミティブ(三角形)



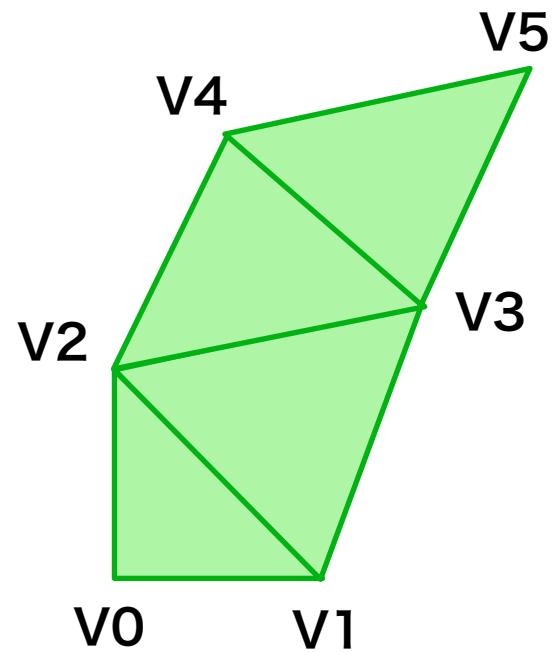
GL_TRIANGLES

基本。使いやすい。



GL_TRIANGLE_FAN

球の極部分など特殊用途。



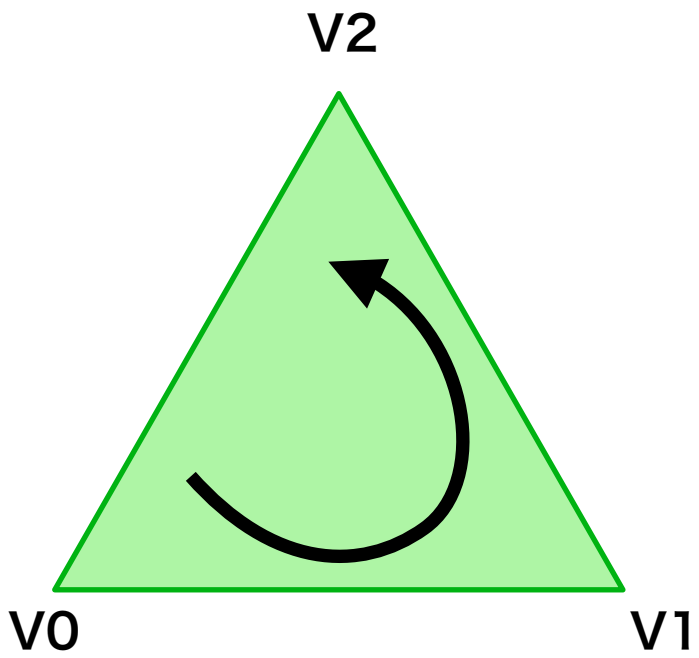
GL_TRIANGLE_STRIP

一般に効率が良い。

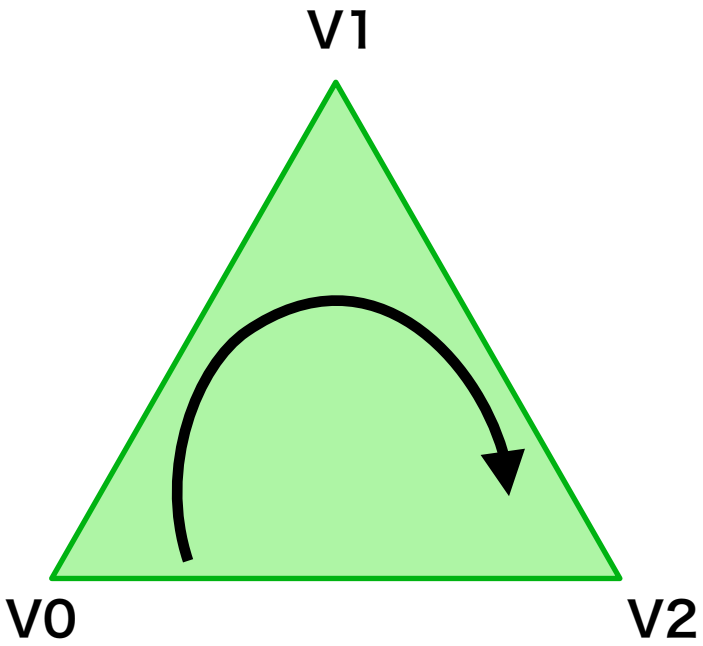
ハードウェアの特性によってストリップ長を決める。

面の表裏

- glFrontFace() で指定
  - ➡デフォルトはGL_CCW



GL_CCW
(Counter-Clockwise:反時計回り)

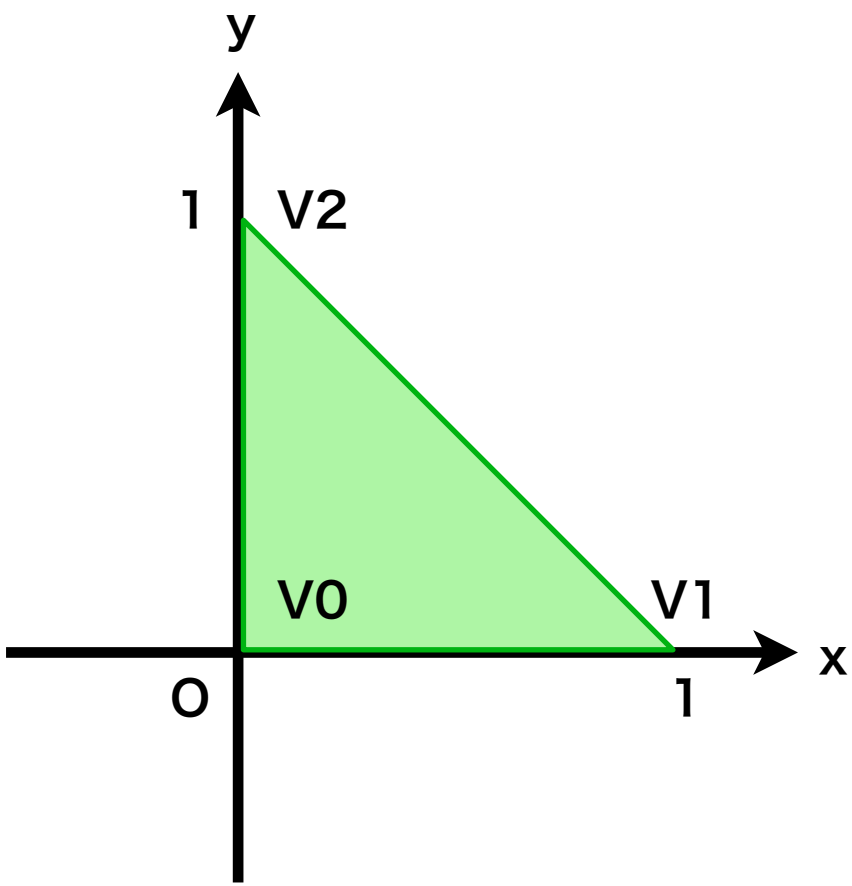


GL_CW
(Clockwise:時計回り)

２Dの三角形を描く

三角形のモデル

- 頂点とインデックスの配列で表現




```
// 頂点配列
GLfloat vertices[] ={
    0,0,  1,0,  0,1,   // x, y
};
```

```
// インデックス配列
GLushort indices[] = {
    0, 1, 2,
};
```

これらをシステムメモリから
GL側(ビデオメモリ)へ転送。

バッファオブジェクト

- GL側で保持するバイナリイメージのこと。

```
GLuint     vb, ib;

// 頂点バッファ生成
glGenBuffers(1, &vb);                          // バッファオブジェクトを生成
glBindBuffer(GL_ARRAY_BUFFER, vb);      // 頂点配列としてバインド
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
                    GL_STATIC_DRAW);            // 初期データを定義
glBindBuffer(GL_ARRAY_BUFFER, 0);      // バインド解除

// インデックスバッファ生成
glGenBuffers(1, &ib);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ib);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
                    GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

頂点バッファオブジェクトの描画

- glDrawElements()でインデックス付頂点配列を描画

```
// 頂点バッファのバインド
glBindBuffer(GL_ARRAY_BUFFER, vb);
glVertexAttribPointer(0,                    // 属性インデックス
                      2,                    // 要素数。x, yで2。
                      GL_FLOAT,             // 型
                      GL_FALSE,             // normalized指定。falseにしておく。
                      sizeof(GLfloat)*2, //1頂点のサイズ。float 2個分。
                      (void *)0);           // 頂点内要素のオフセット
glEnableVertexAttribArray(0);   // 頂点配列を有効にする

// インデックスバッファのバインド
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ib);

// 描画
glDrawElements(GL_TRIANGLE_STRIP, 3, GL_UNSIGNED_SHORT, (void *)0);
```

しかし…

- これだけでは絵が出ません。

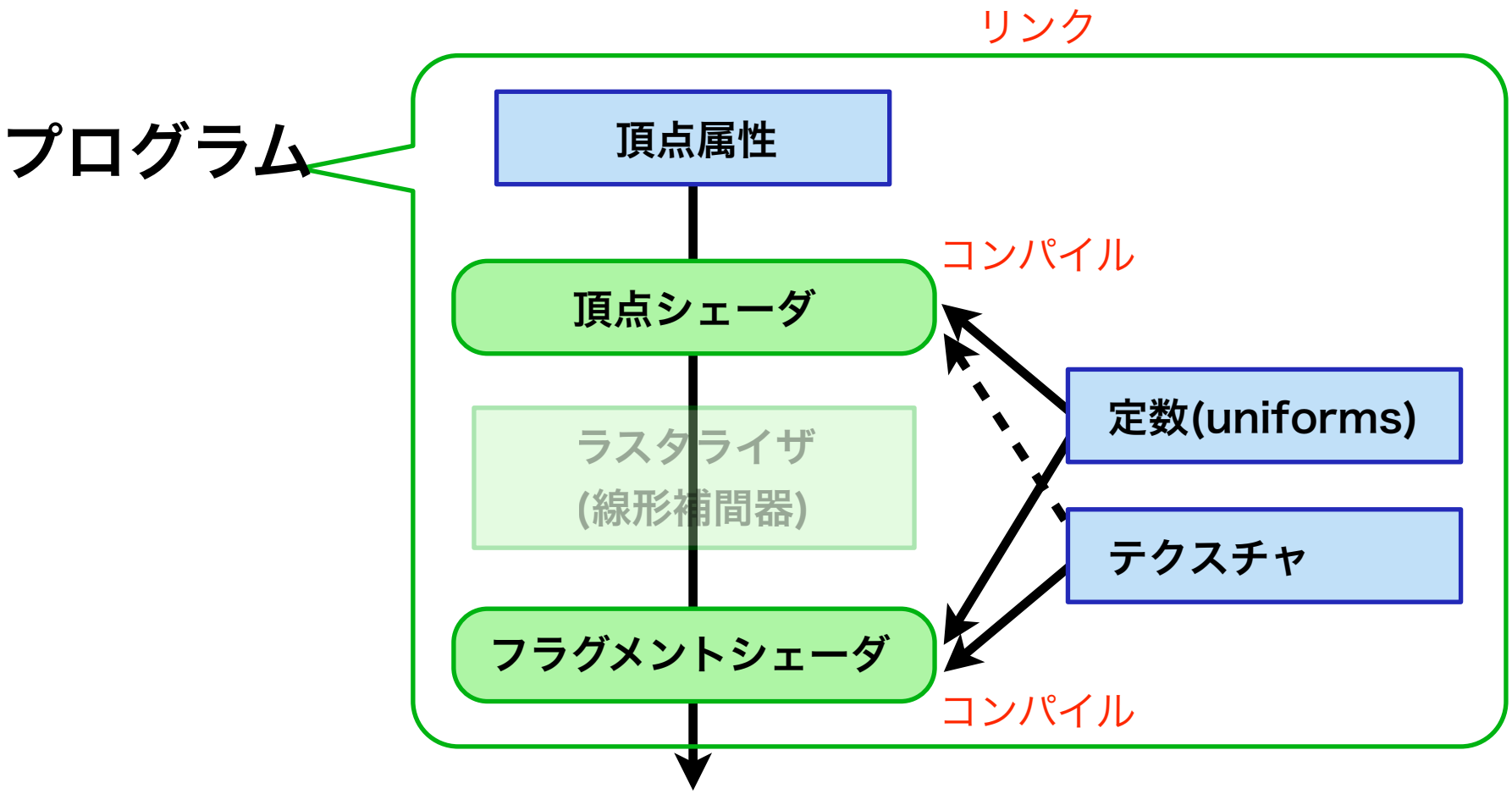
GL「どうやって描いたらいいか判らないの」

↓

シェーダプログラムの設定が必要

シェーダプログラム

- 頂点シェーダ、フラグメントシェーダは個々にコンパイルされ、リンクされて一つのプログラムになる。



頂点シェーダ

頂点シェーダのコンパイル

```
const char vert_src[] =
        "attribute  vec2  a_position;          \n"
        "void main() {                         \n"
        "    gl_Position.xy = a_position;      \n"    // 座標コピーするだけ
        "    gl_Position.z = 0.0;              \n"
        "    gl_Position.w = 1.0;              \n"
        "}\n";
GLuint vert_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vert_shader, 1, &vert_src, NULL);
glCompileShader(vert_shader);
check_shader_status(vert_shader);  // 正常にコンパイルできたかチェック
```

attribute: 頂点属性を示すqualifier
vec2: float*2の型
gl_Position: 座標出力用の組み込み変数

フラグメントシェーダのコンパイル

```
const char frag_src[] =
        "precision mediump float;          \n"
        "uniform vec4 base_color;  \n"
        "void main() {                     \n"
        "    gl_FragColor = base_color;    \n"
        "}\n";
GLuint frag_shader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(frag_shader, 1, &frag_src, NULL);
glCompileShader(frag_shader);
check_shader_status(frag_shader); // 正常にコンパイルできたかチェック
```

precison mediump float: 浮動小数点の精度を中程度(16〜24bit)にする
uniform: アプリから指定する不変量(定数)を示すqualifier
glFragColor: 色出力用の組み込み変数

シェーダプログラム

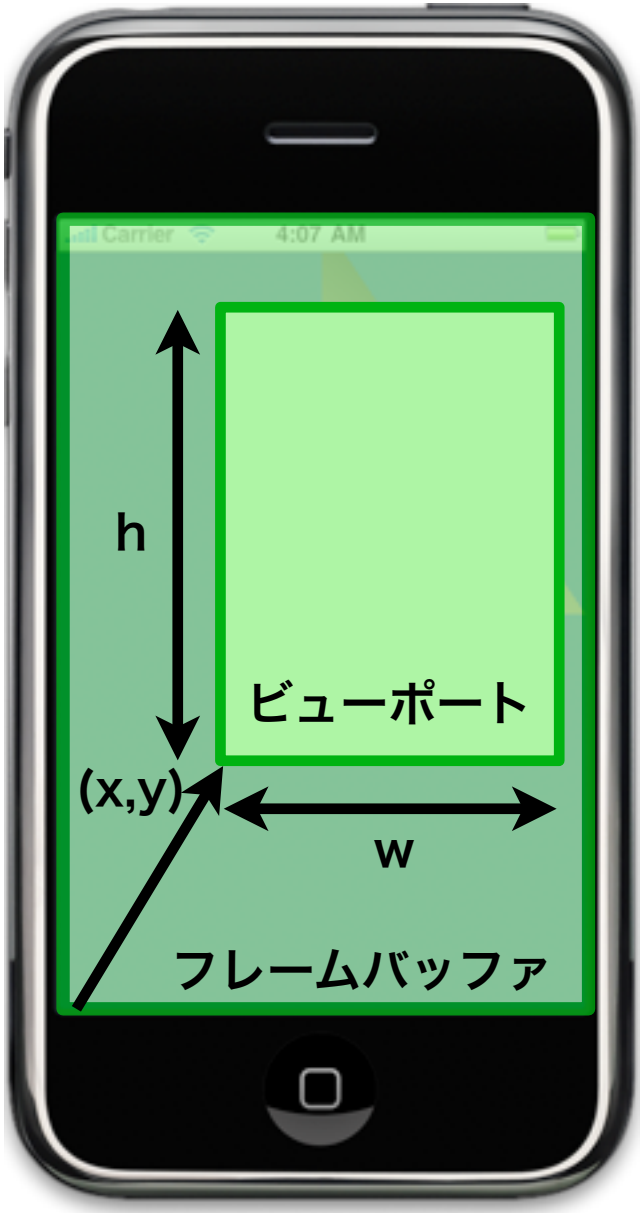
プログラムの生成とリンク

```
program = glCreateProgram();
// シェーダのアタッチ。アタッチ後はシェーダを破棄して構わない。
glAttachShader(program, vert_shader);
glAttachShader(program, frag_shader);
glDeleteShader(vert_shader);
glDeleteShader(frag_shader);
// 頂点属性のバインド
glBindAttribLocation(program, 0, “a_position”);
// リンク
glLinkProgram(program);
check_program_status(program);  // 正常にリンクできたかチェック

// uniform location の取得
GLint base_color_loc = glGetUniformLocation(program, "base_color");
```

描画範囲

- glViewport(x, y, w, h);
  で、フレームバッファ内の
  描画範囲を指定



描画再挑戦

プログラムを使って描画

```
// ビューポート設定と画面クリア
glViewport(0, 0, backingWidth, backingHeight);
glClear(GL_COLOR_BUFFER_BIT);

// シェーダプログラムを使用
glUseProgram(program);
// uniformを設定
float base_color[] = { 1, 0, 0, 1 };    // 赤
glUniform4fv(base_color_loc, 4, base_color);

// 先の頂点バッファオブジェクトをバインドして、
glDrawElements();
```

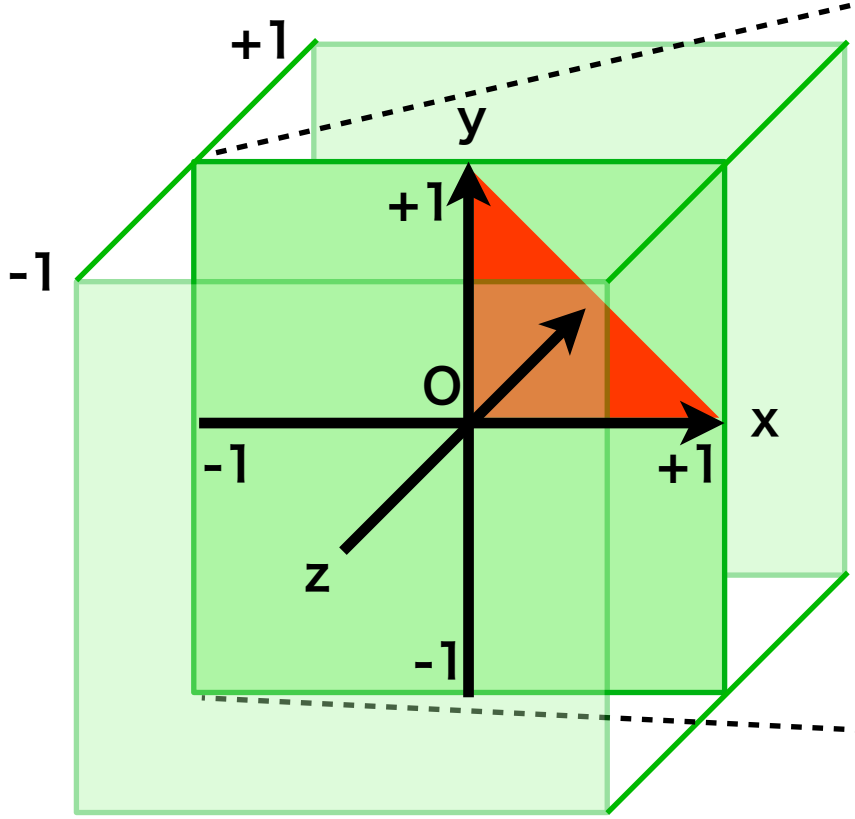
backingWidth, backingHeight:
ウィンドウシステムからもらった画面のサイズ

2D三角形描画の実行結果(予想図)

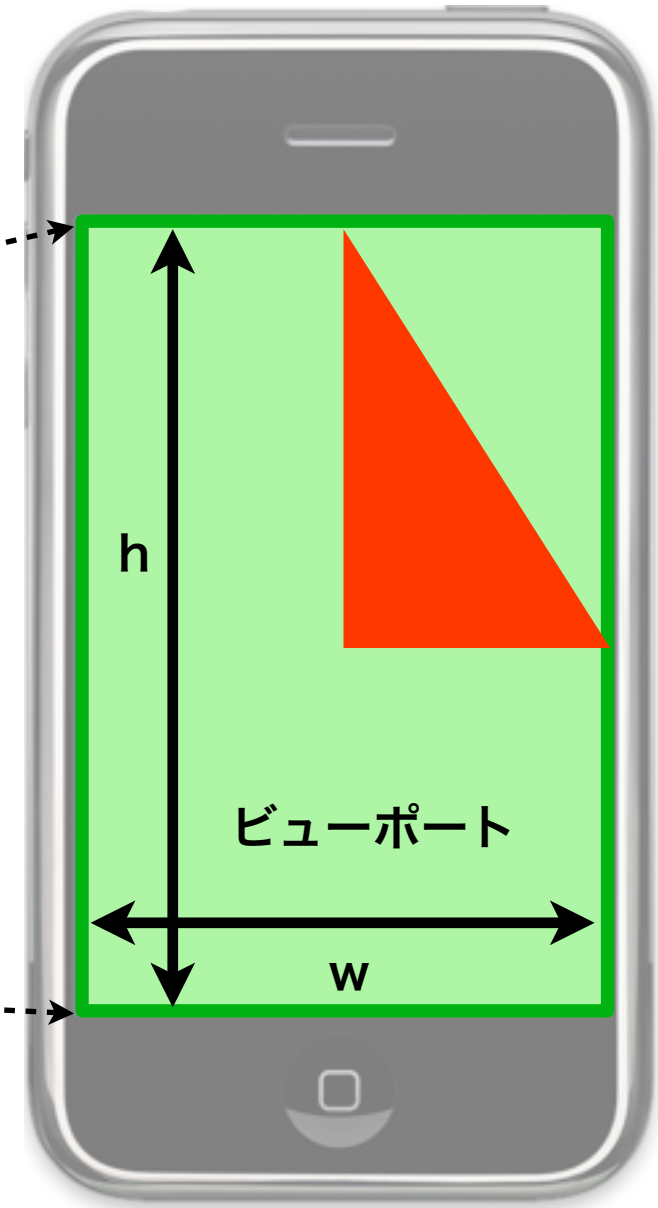
描けた！でも縦長…

正規化デバイス座標系

プリミティブは、-1<=(x,y,z)<=+1 の
立方体内に描かれている。



正規化デバイス座標系



２D三角形の描画(実機デモ)

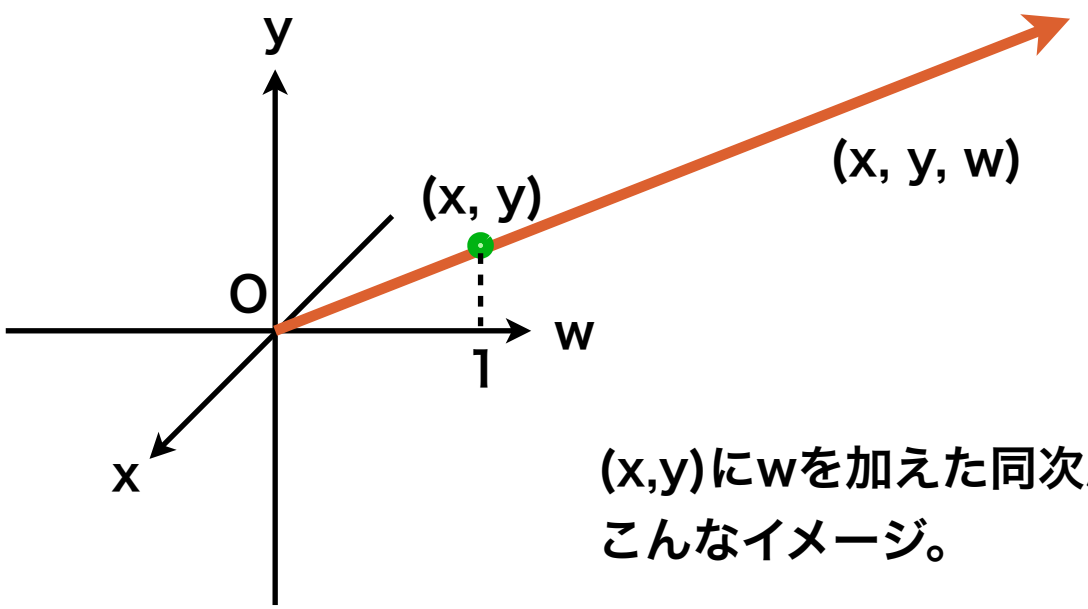
- ソースコードの説明と実機デモ、です。

ちょっといじって、表示位置と色のアニメーションも。

3Dの三角錐を描く

同次座標

- (x, y, z) にwを加えて、4次元ベクトルにしたもの。
- (x, y, z, w) を (x/w, y/w, z/w) の座標として扱う。
- w=0 の場合は (x, y, z) 方向の無限遠を示す。
  - ➡ ライティングの平行光源などで使う。



(x,y)にwを加えた同次座標の例。
こんなイメージ。

行列と変換

- 同次座標を使うと、平行移動を行列の乗算で行える。

$$\begin{pmatrix} x+a \\ y+b \\ z+c \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- 事前に変換行列を乗算しておくことで、複数の変換を一回で行える。
  - $p' = [P][S][R][T]\ p = [M]\ p$

要するに

- 行列はベクトルを変換してくれる便利なもの。
  - 移動 (Translation)
  - 回転 (Rotation)
  - 拡大縮小 (Scaling)
  - 射影 (Projection)

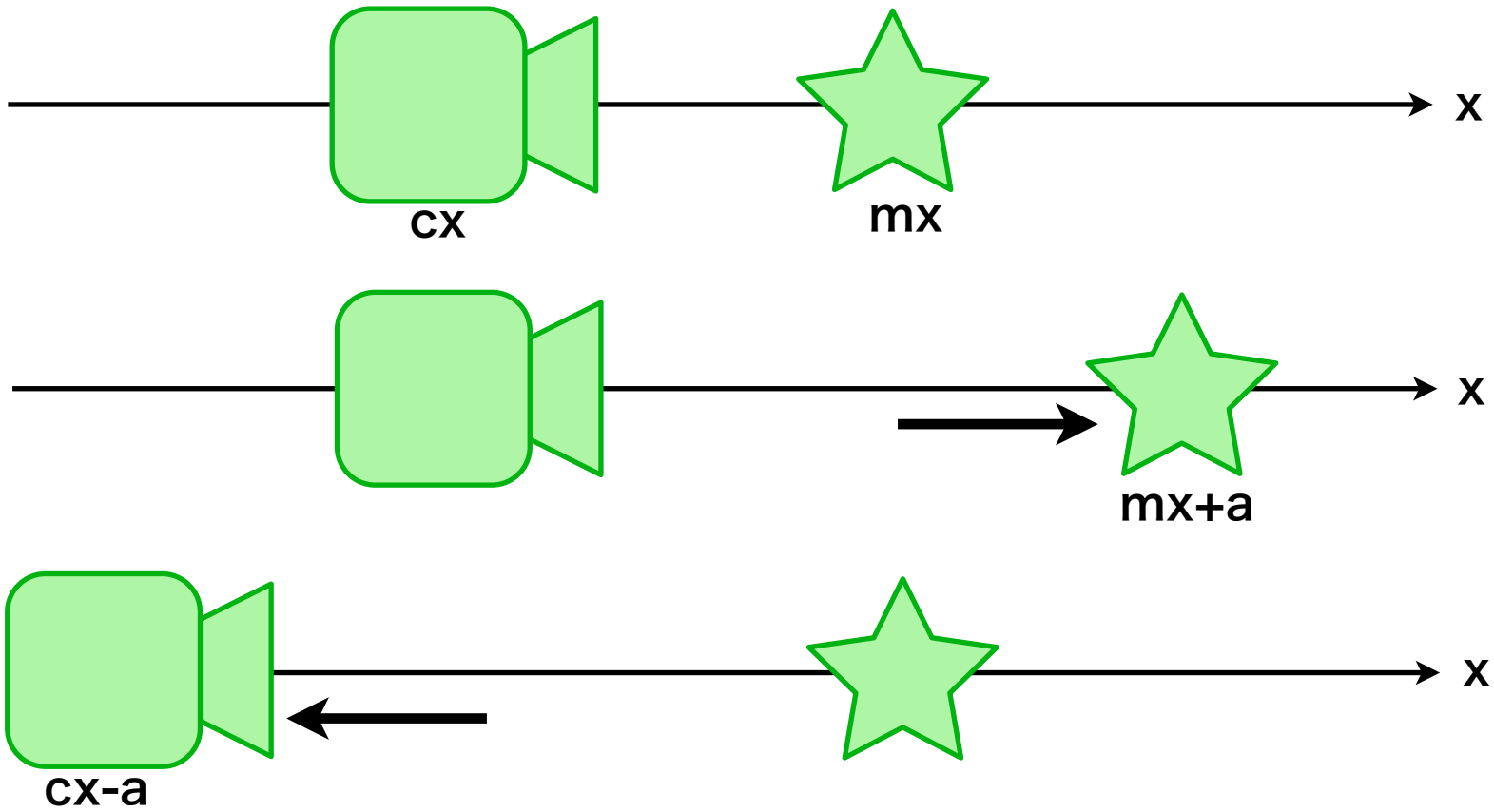
要するに

- 行列はベクトルを変換してくれる便利なもの。
  - 移動 (Translation)
  - 回転 (Rotation)
  - 拡大縮小 (Scaling)
  - 射影 (Projection)
- 事前に行列を乗算しておけば、頂点毎に1回行列に乗算するだけで、全部の変換を行える。

  ➡大事なことなので2回言いました。

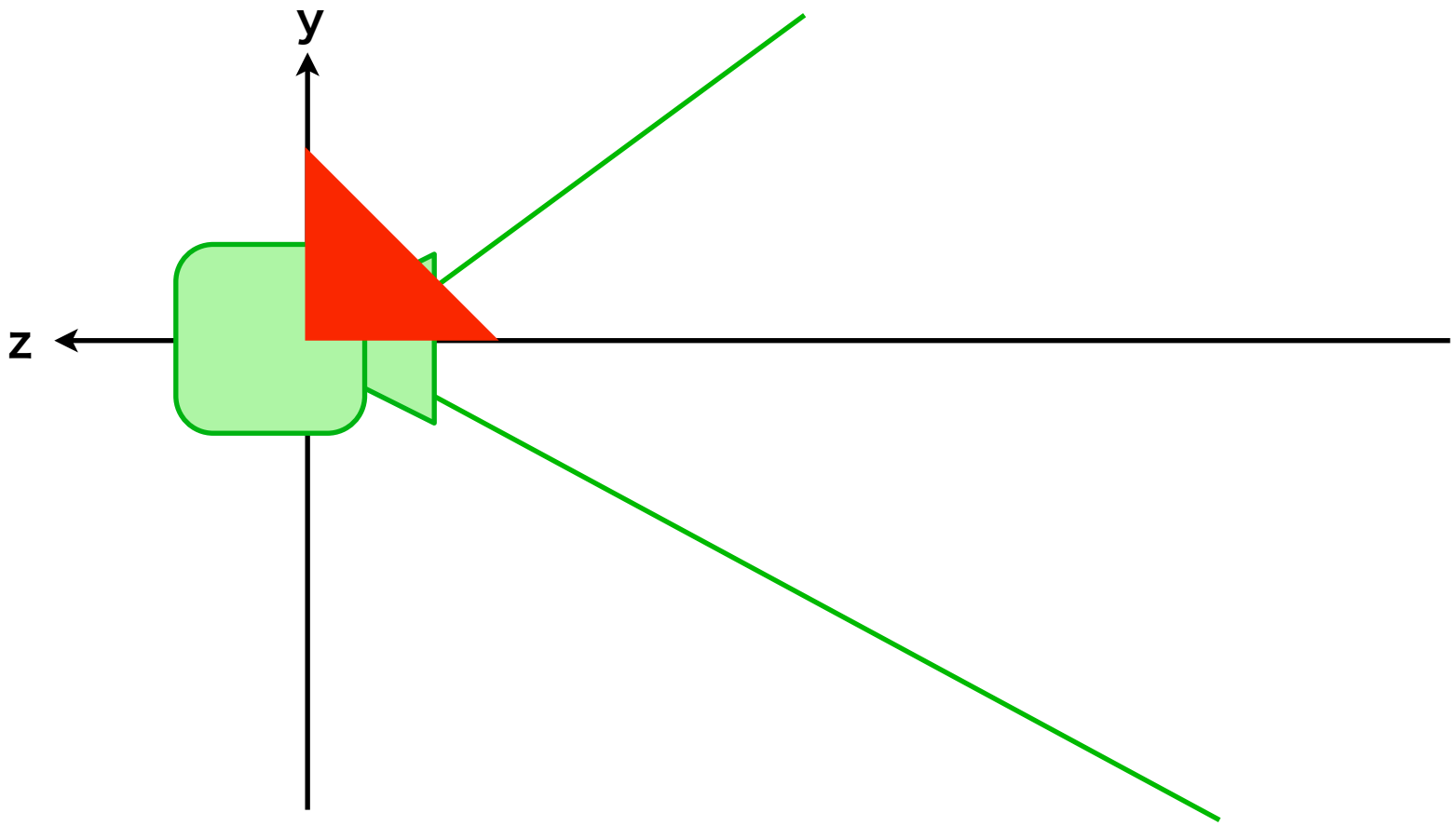
モデルビュー変換

- モデルが動くのと、カメラが動くのは等価。
  - ➡一緒にしてモデルビュー変換と呼ばれる。



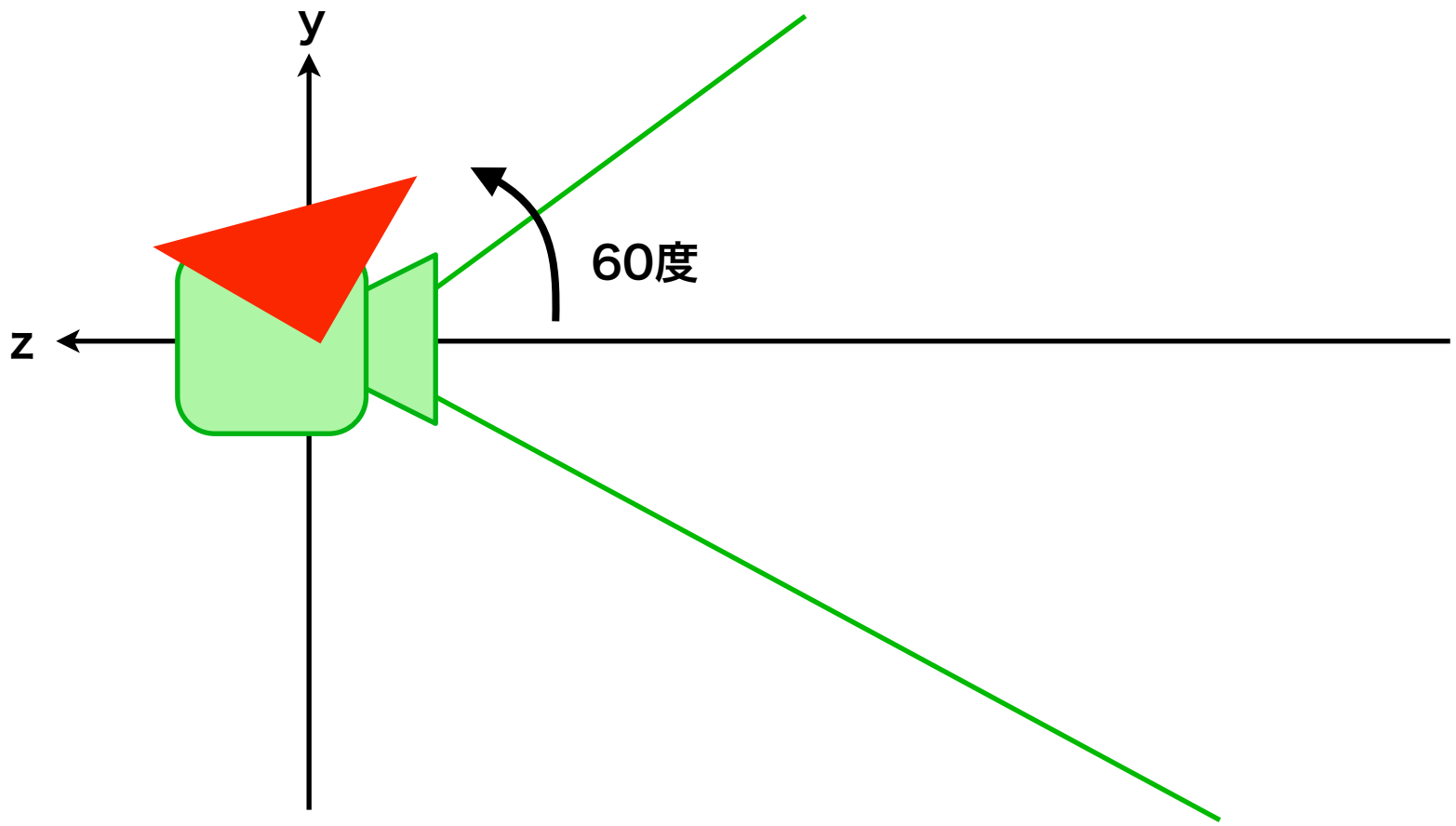
カメラ変換行列の設定

- カメラ位置は原点、視線方向は -Z 方向。
- 見やすい位置にオブジェクトを回転、移動。
  - ➡ カメラを逆に回転、移動させたことになる。



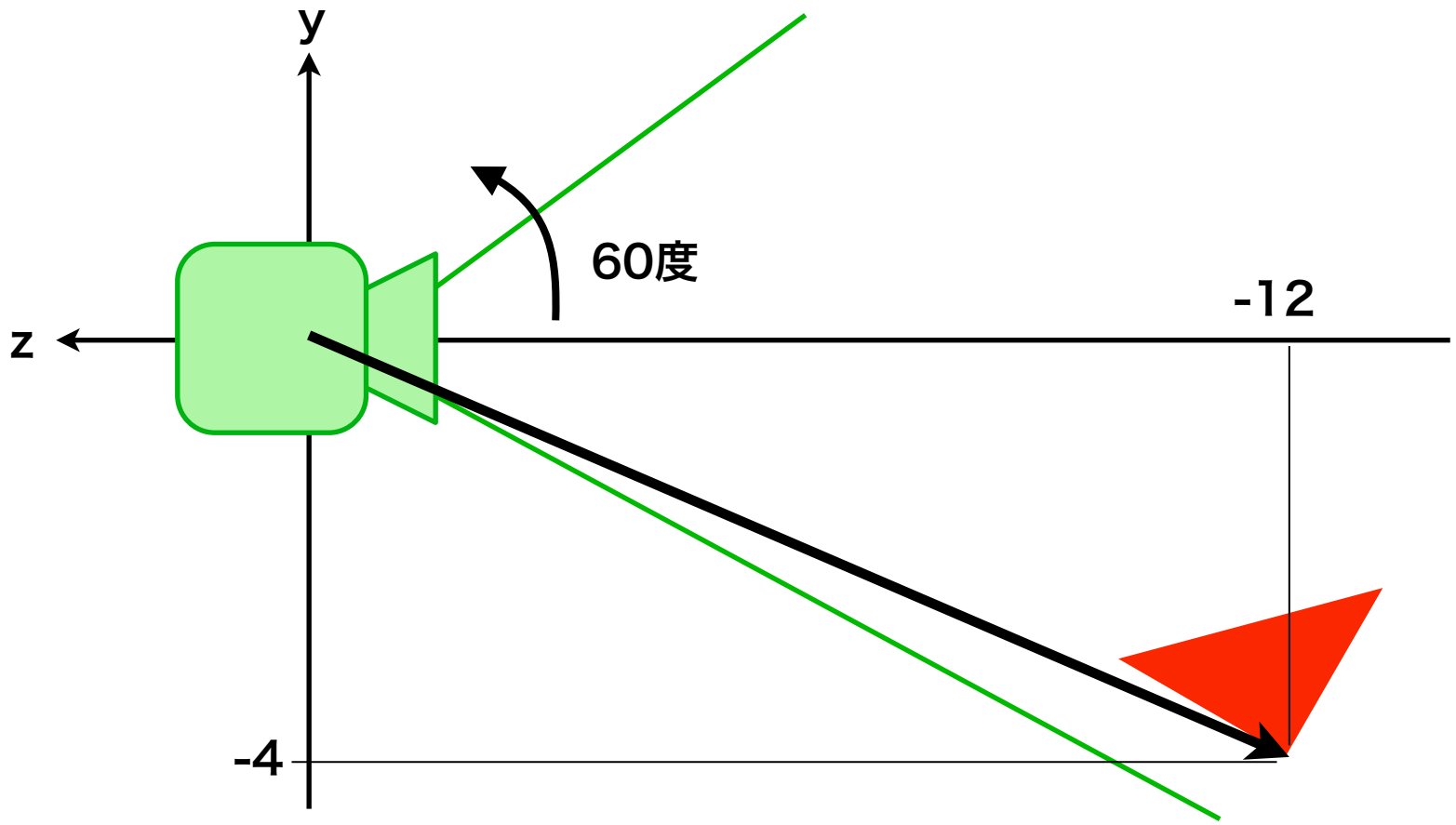
カメラ変換行列の設定

- カメラ位置は原点、視線方向は -Z 方向。
- 見やすい位置にオブジェクトを回転、移動。
  - ➡ カメラを逆に回転、移動させたことになる。



カメラ変換行列の設定

- カメラ位置は原点、視線方向は -Z 方向。
- 見やすい位置にオブジェクトを回転、移動。
  - ➡ カメラを逆に回転、移動させたことになる。



カメラ変換行列の設定

- サンプルコードでのビュー行列の設定部分。

```
void set_camera_matrix(float *mvp_mtx, float width, float height)
{
    // 射影行列
    FnMatrix4::PerspectiveMatrixVFov(mvp_mtx, 45.0f, width/height, 1.0f,
1000.0f);

    // ビュー行列
    float  view_mtx[16];
    FnMatrix4::setIdentity(view_mtx);
    FnMatrix4::setColumn(view_mtx, 3, 0.0f, -4.0f, -12.0f, 1.0f);
    FnMatrix4::mul(mvp_mtx, mvp_mtx, view_mtx);
    FnMatrix4::setRotX(view_mtx, 60.0f *M_PI/180.0f);
    FnMatrix4::mul(mvp_mtx, mvp_mtx, view_mtx);
}
```

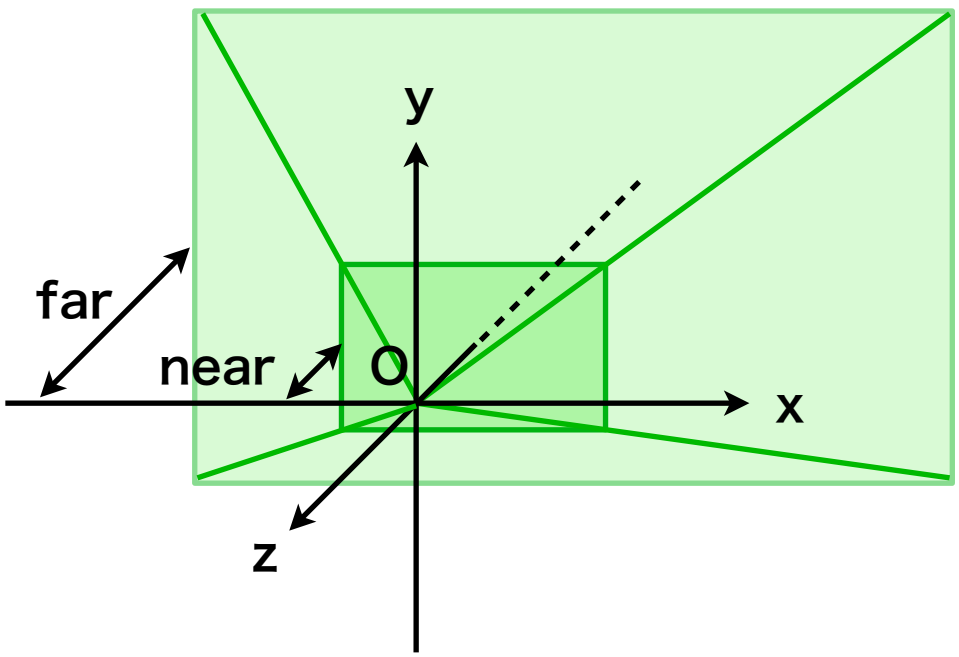
ただし…

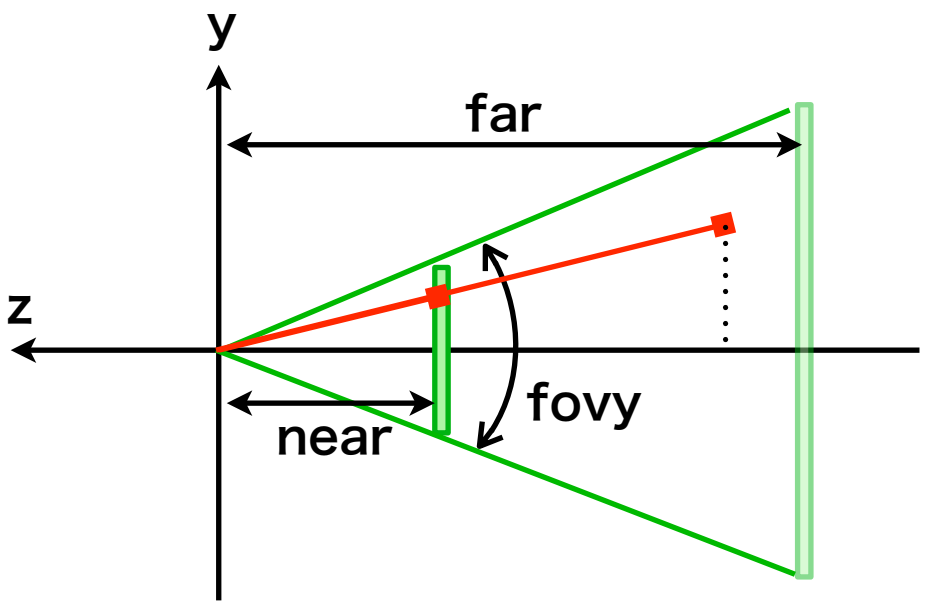
- 実際問題、分けて考えた方が便利。
- カメラも普通のオブジェクトと同じように、移動・回転させ、位置・方向が定まった時点で逆行列を求める。

視錐台(view volume)

- カメラから見える範囲のこと
  - farを底面、nearを上面とした錐台の形。
  - この範囲内のプリミティブをnear面に投影する。近いものは大きく、遠いものは小さく描画。

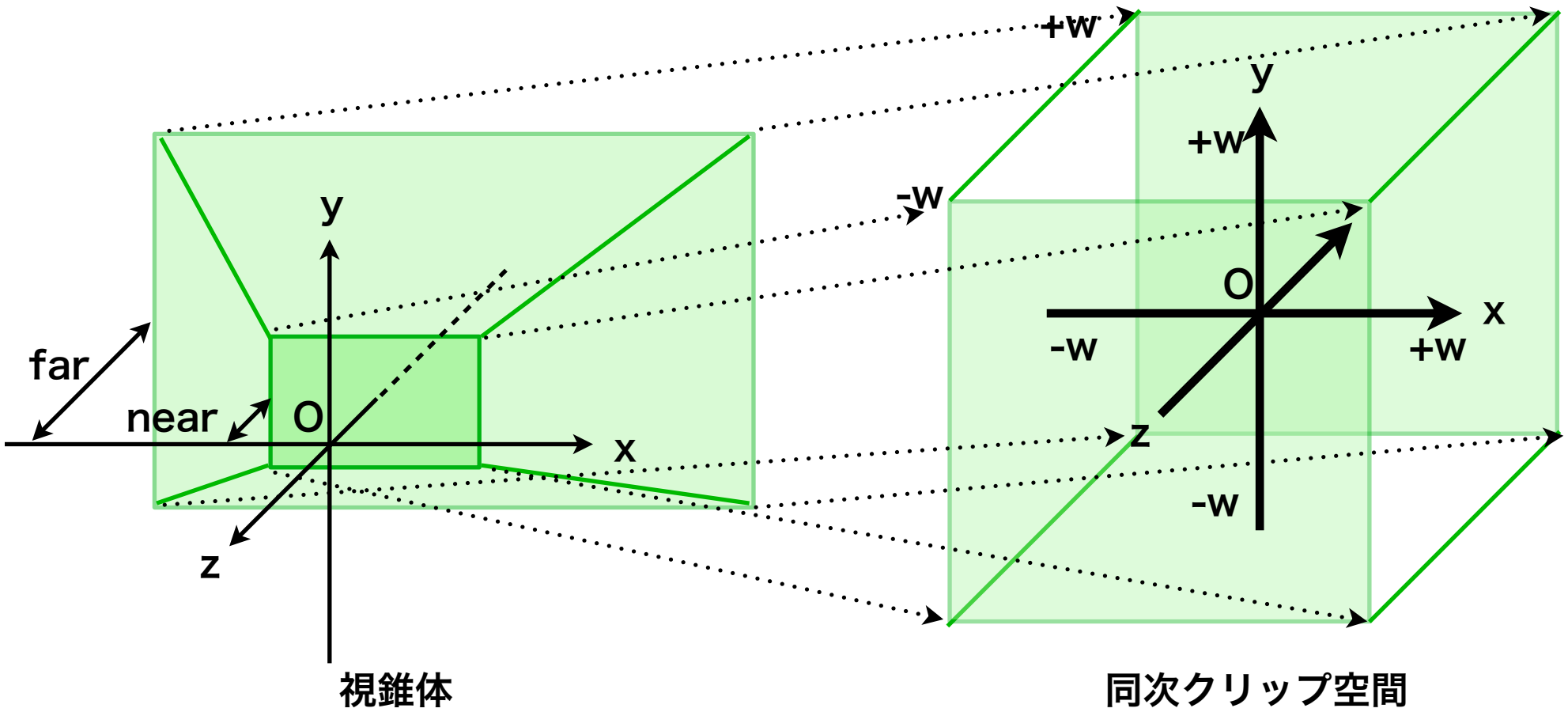
    ➡ (x, y) を -z で割る。∴ wに-zを入れればよい。





射影変換

- カメラ座標の頂点座標をクリップ座標へ変換。
  - 範囲を外れたプリミティブはクリッピングされる。
  - この後wで除算して正規化デバイス座標へ。



射影行列(projection matrix)

- 利便上、GLUライブラリのgluPerspective() で算出することが多い。
- サンプルにはこれと同じ行列を計算する関数を用意。

```
FnMatrix4::PerspectiveMatrixVFov(
    float  proj_mtx[16],
    float  fovy       /*degree*/,
    float  aspect   /* width/height */,
    float  near,
    float  far);
```

頂点シェーダ

- 頂点シェーダが出力する座標 gl_Position はこのクリップ座標。
- 2D 三角形では (x, y, 0, 1) と w=1.0 で出力していたので、クリップ座標＝正規化デバイス座標。
  - w=0.5 と出力すると、描画サイズは2倍、
    w=2.0 と出力すると、描画サイズは1/2倍になる。
    ➡ お試しあれ。

カメラ変換行列の設定

- サンプルコードでの射影行列の設定部分。

```
void set_camera_matrix(float *mvp_mtx, float width, float height)
{
    // 射影行列
    FnMatrix4::PerspectiveMatrixVFov(mvp_mtx, 45.0f, width/height, 1.0f,
1000.0f);

    // ビュー行列
    float  view_mtx[16];
    FnMatrix4::setIdentity(view_mtx);
    FnMatrix4::setColumn(view_mtx, 3, 0.0f, -4.0f, -12.0f, 1.0f);
    FnMatrix4::mul(mvp_mtx, mvp_mtx, view_mtx);
    FnMatrix4::setRotX(view_mtx, 60.0f *M_PI/180.0f);
    FnMatrix4::mul(mvp_mtx, mvp_mtx, view_mtx);
}
```

挙動計算と描画の分離

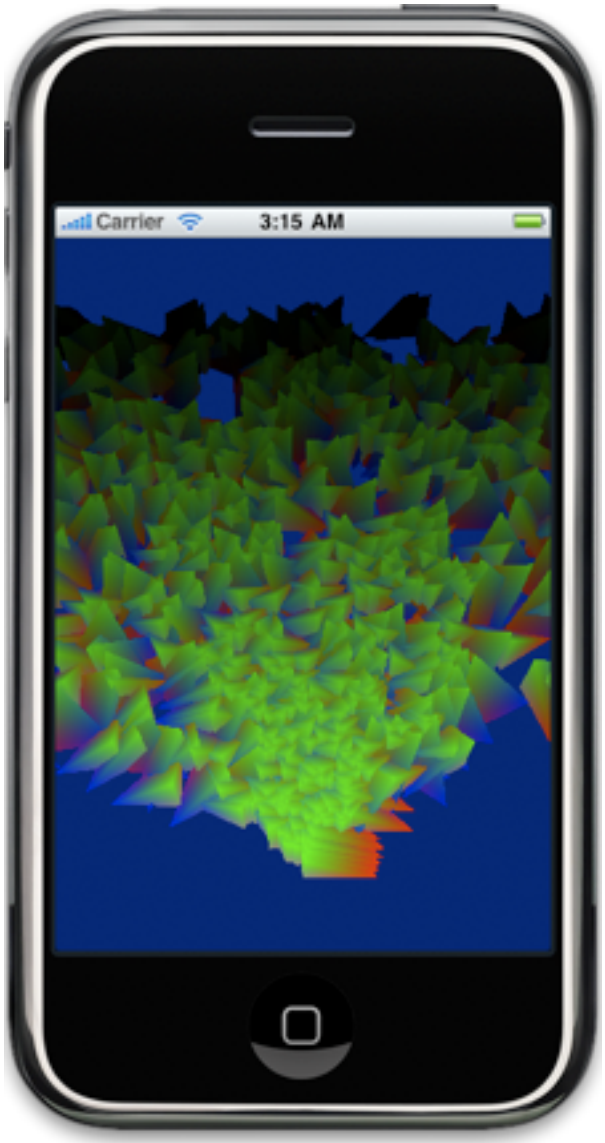
- ゲームプログラムでは、オブジェクトの挙動計算(状態更新)と描画処理は、分けておく。
  - ➡ デバグの際に挙動計算を止めて、描画だけ行うことがあるため。
- サンプルの2D三角形のアニメーションは、描画内でカウンタを更新してしまっている、典型的なまずい例。

```
void draw_object(MyObject *m) {
    m->count++;
    rotate_y(m->count);
    m->draw_mesh();
}
```

3D三角錐の描画

- ソースコードの説明と実機デモ、です。

20個

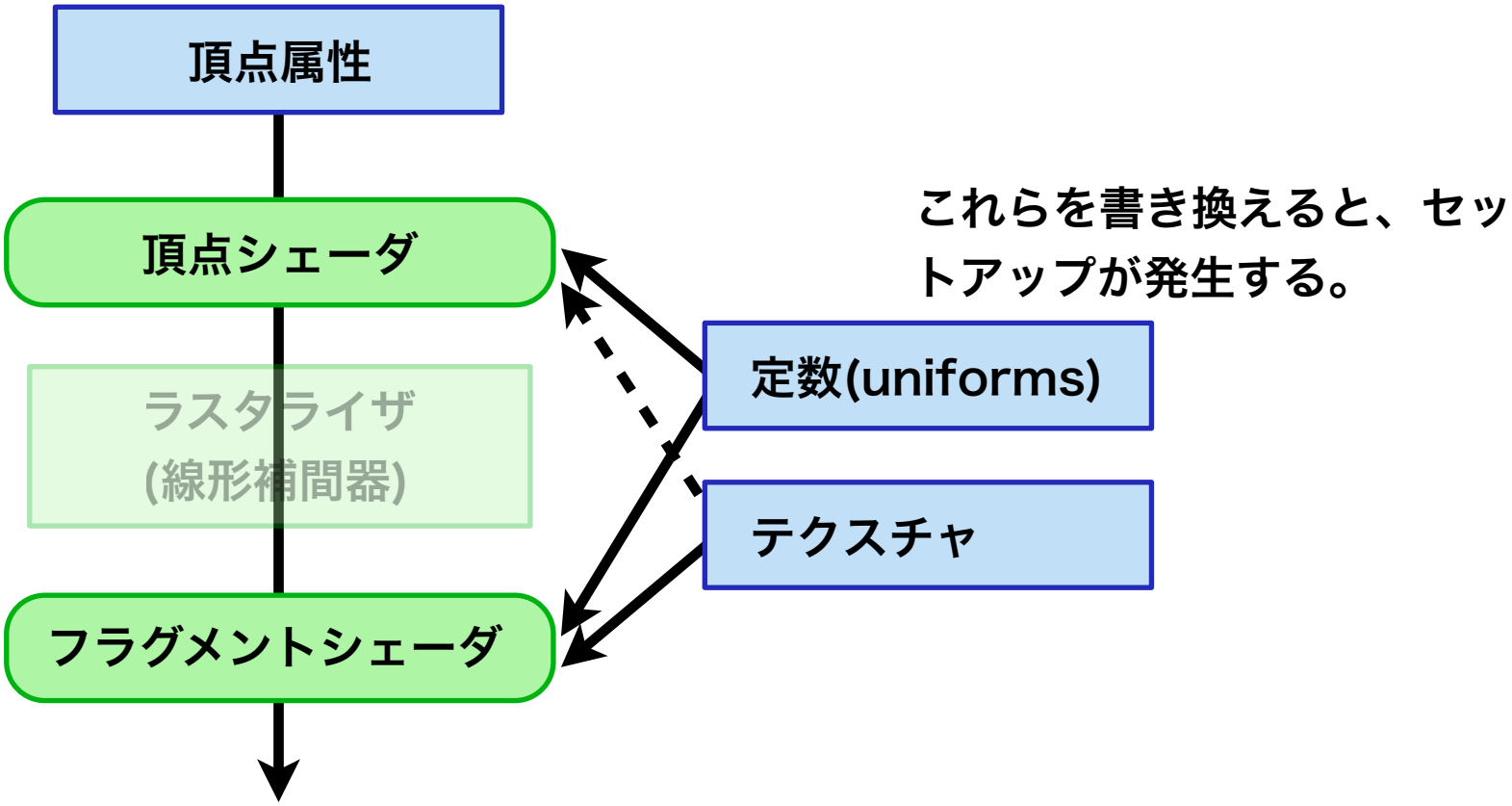


1000個

パフォーマンスのために

描画コマンドは重い

- 軽くするためには
  - 描画回数を減らす。
  - uniform, テクスチャの変更を減らす。



描画回数を減らす

- モデルデータ作成で心がけること。
  - テクスチャをまとめる
  - マテリアルをまとめる

- 結構大変です。

  ➡デザイナさんと普段から仲良くしておこう。

簡易インスタンシング

- パラメータを uniform から頂点属性へ移す。
  - ➡ シェーダセットアップがなくなり、ドローコールが速くなる。

```
#if (USE_UNIFORM)
    // set uniforms
    FnMatrix4::transpose(load_mtx, load_mtx);
    glUniformMatrix4fv(mesh->mvp_matrix_loc, 16, GL_FALSE, load_mtx);
    glUniform1f(mesh->luminance_loc, luminance);
#else
    // set attribs
    glVertexAttrib4fv(2, &load_mtx[0]);
    glVertexAttrib4fv(3, &load_mtx[4]);
    glVertexAttrib4fv(4, &load_mtx[8]);
    glVertexAttrib4fv(5, &load_mtx[12]);
    glVertexAttrib1f (6, luminance);
#endif
    glDrawElements(GL_TRIANGLE_STRIP, count, GL_UNSIGNED_SHORT, (void *)0);
```

簡易インスタンシング

- 難点。
  - 頂点属性の数が限られている。

- 通常の頂点属性数。
  - OpenGL ES 2.0 で8個
  - OpenGL で 16個。

    ➡ 行列は4x3で渡す。

- パーティクルや、弾幕で使おう！

パフォーマンスのために

- ターゲットのベンダの資料を熟読する。
  - NVIDIA
  - AMD (ATI)
  - Imagination Technologies

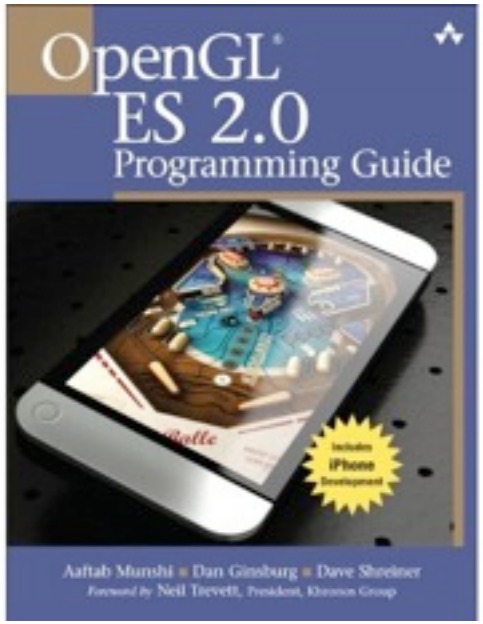
    ➡ 大抵英語です。頑張って読む。

- 仮説を立て、検証。
  - 地味な作業です。

    ➡ 工数を確保するのが勝負。

参考文献

- OpenGL ES 2.0 Programming Guide
  - Aaftab Munshi、Dan Ginsburg、Dave Shreiner
  - ISBN-13: 978-0-321-50279-7



- ゲームプログラミングのための3Dグラフィクス数学
  - Eric Lengtel、訳：狩野智英
  - ISBN4-939007-37-5

1時限目終了

- ご質問は？
- 実機デモを手元で見たい方は、休み時間にステージまでどうぞ。