

# Delphiで超高速OpenGL 2D/3D描画

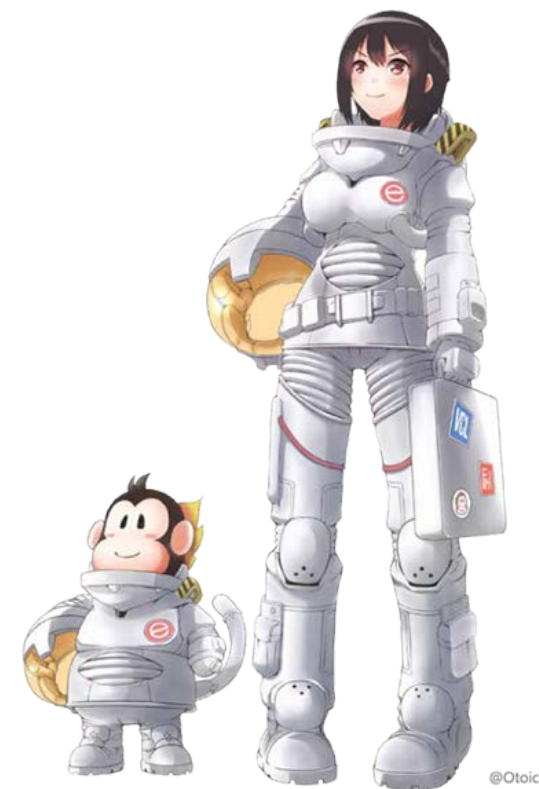
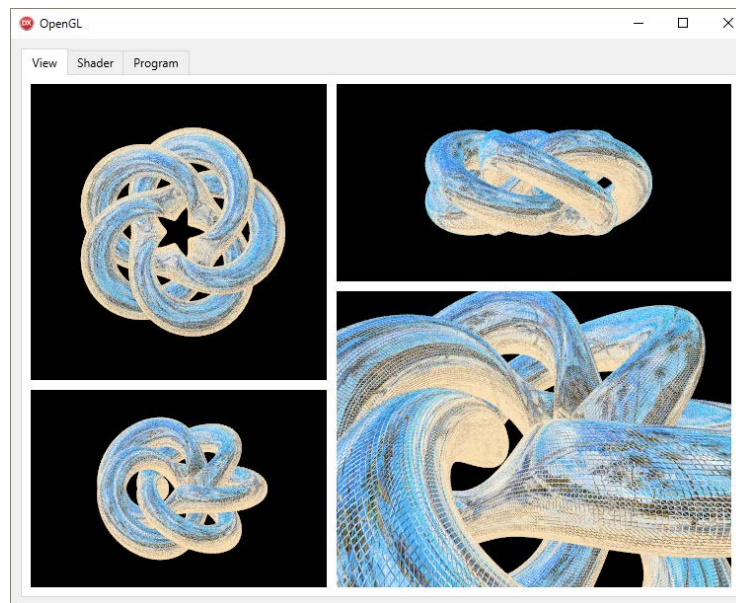
～ FMX / VCL コンポーネントで驚きの性能実現 ～

第34回 エンバカデロ・デベロッパーキャンプ

慶應義塾大学藤代研究室 特別研究員  
+ 玉泉山安国院 住職

中山 雅紀

contact@luxidea.net



**e**mbarcadero®  
DEVELOPER CAMP

# アジェンダ

- OpenGLとは？
- OpenGLの初期化
  - FMX の場合
  - VCL の場合
- コマンドの実行
- シーンの構築
- 物体の定義
- OpenGL の歴史
- ミニмумライブラリ
- マキシмумライブラリ

- FMX版：[github.com/LUX0PHIA/OpenGL](https://github.com/LUX0PHIA/OpenGL)
- VCL版：[github.com/LUX0PHIA/OpenGL\\_VCL](https://github.com/LUX0PHIA/OpenGL_VCL)

📖 README.md

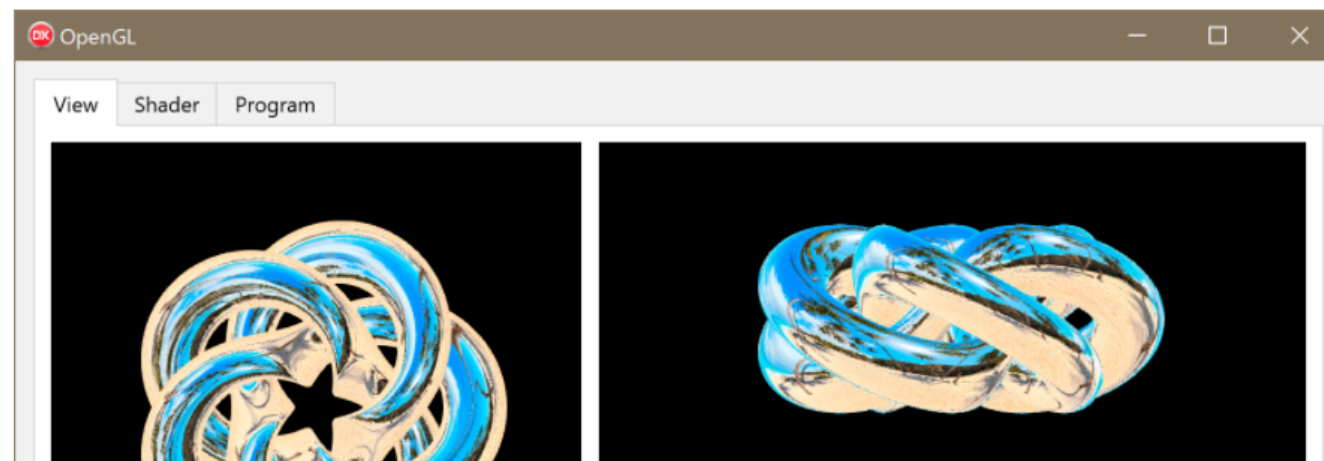
## 歴史に学ぶ OpenGL

OpenGL のバージョンを辿りながら実装していくことで、新しい API が追加された意図を理解しながら、パイプラインの構造を把握していきます。

- [OpenGL 1.0](#)
- [OpenGL 1.1](#)
- [OpenGL 1.5](#)
- [OpenGL 2.1](#)
- [OpenGL 3.0](#)

## OpenGL

FMX : [FireMonkey](#) フレームワークのコンポーネントとして [OpenGL](#) の描画領域を埋め込む方法。



# 自己紹介.身分

## ■ 慶應義塾大学 藤代研究室

[www.fj.ics.keio.ac.jp](http://www.fj.ics.keio.ac.jp)

### ● リサーチフェロー

- [nakayama@fj.ics.keio.ac.jp](mailto:nakayama@fj.ics.keio.ac.jp)
- CGの研究

## ■ 和洋女子大学 山本研究室

[www.a-cad.net](http://www.a-cad.net)

### ● 共同研究員

- 3DアパレルCADの研究

## ■ 玉泉山 安国院

[www.ankokuin.or.jp](http://www.ankokuin.or.jp)

### ● 住職

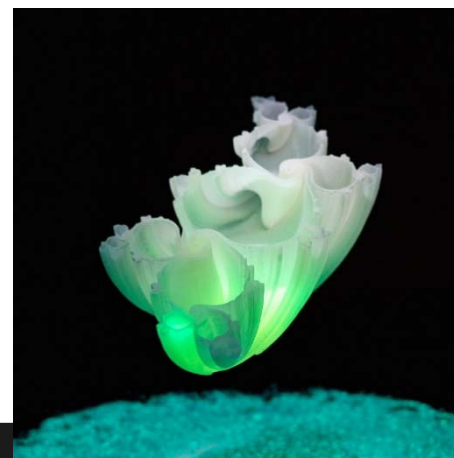
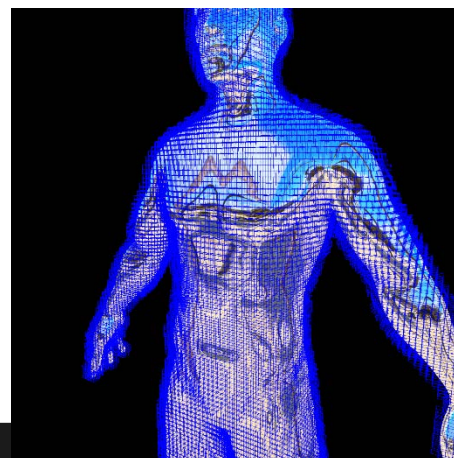
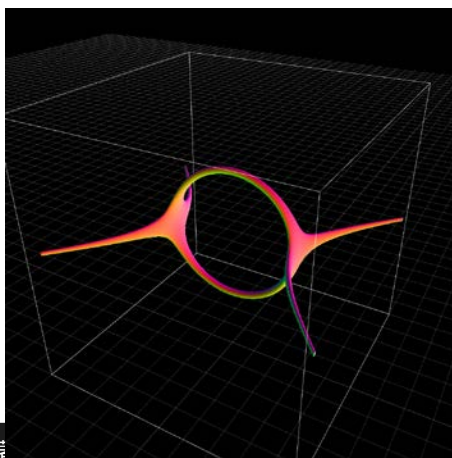
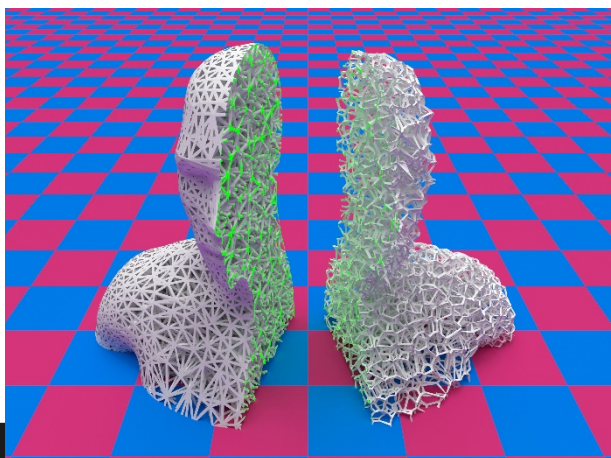
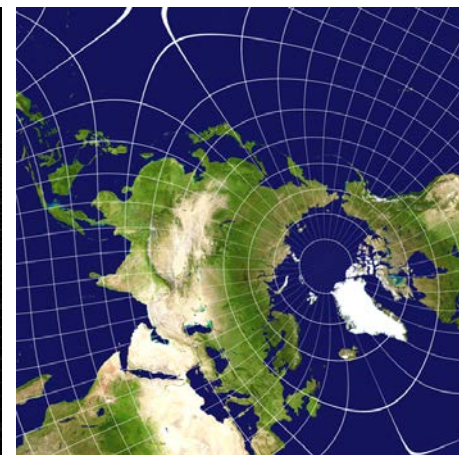
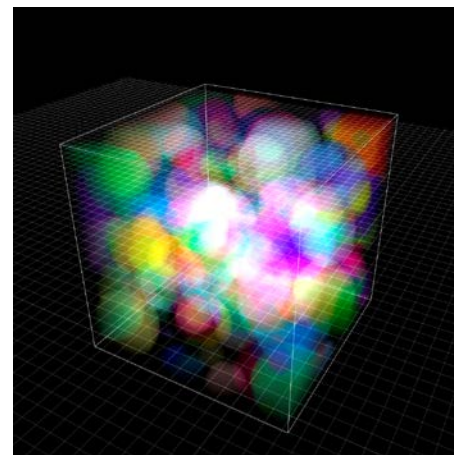
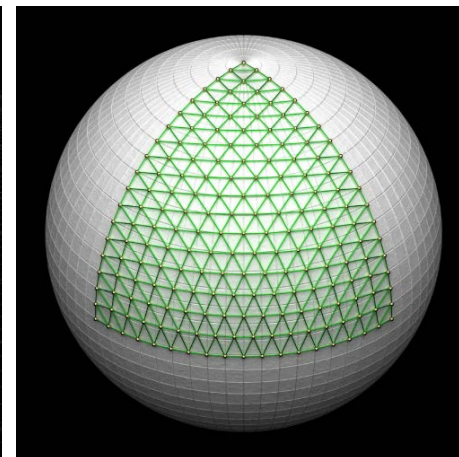
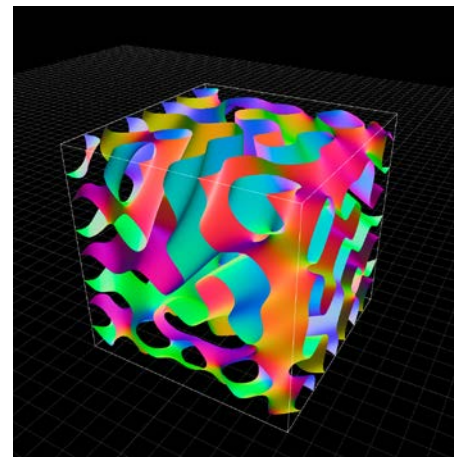
- [contact@ankokuin.or.jp](mailto:contact@ankokuin.or.jp)
- 中道思想の研究





# 自己紹介.専門

- 球面幾何学
- 写實的レンダリング
- 立体視
- アパレル設計
- 数学可視化
- 体積ベースモデリング
- 3Dプリンタ

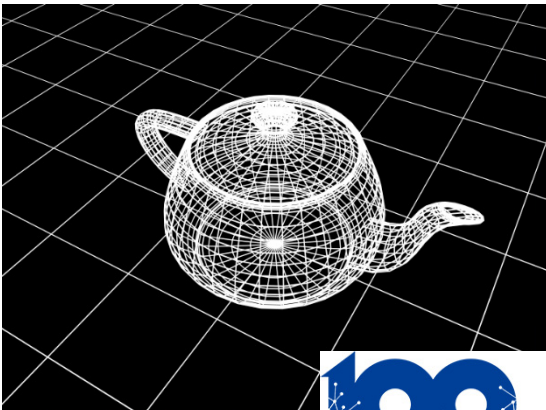
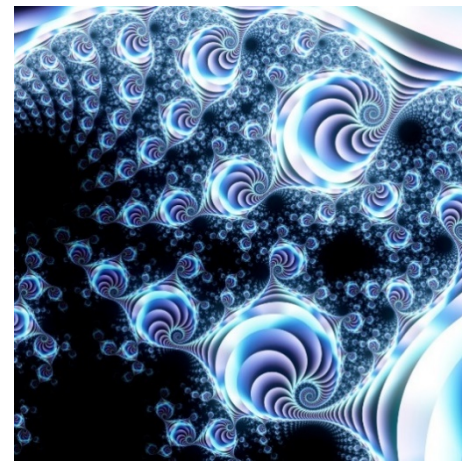
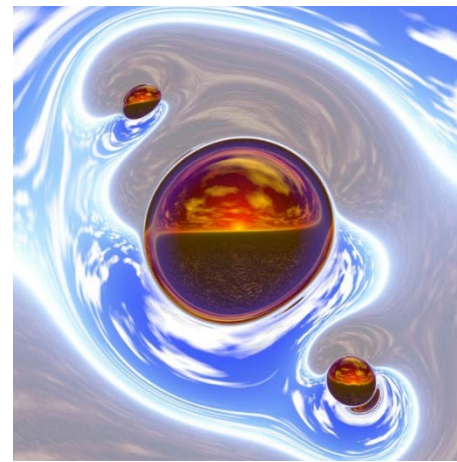
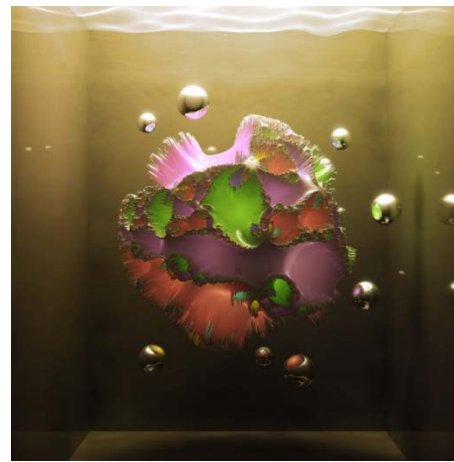




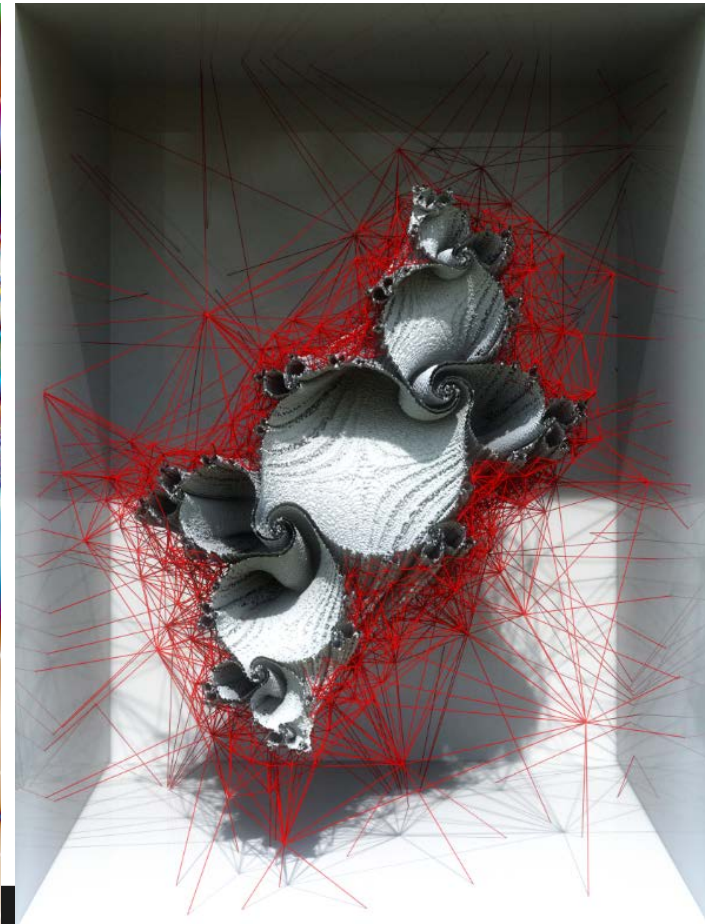
# 自己紹介.趣味

なんちゃって

- C G アート制作
  - フラクタル大好き！
- Delphi 大好き！

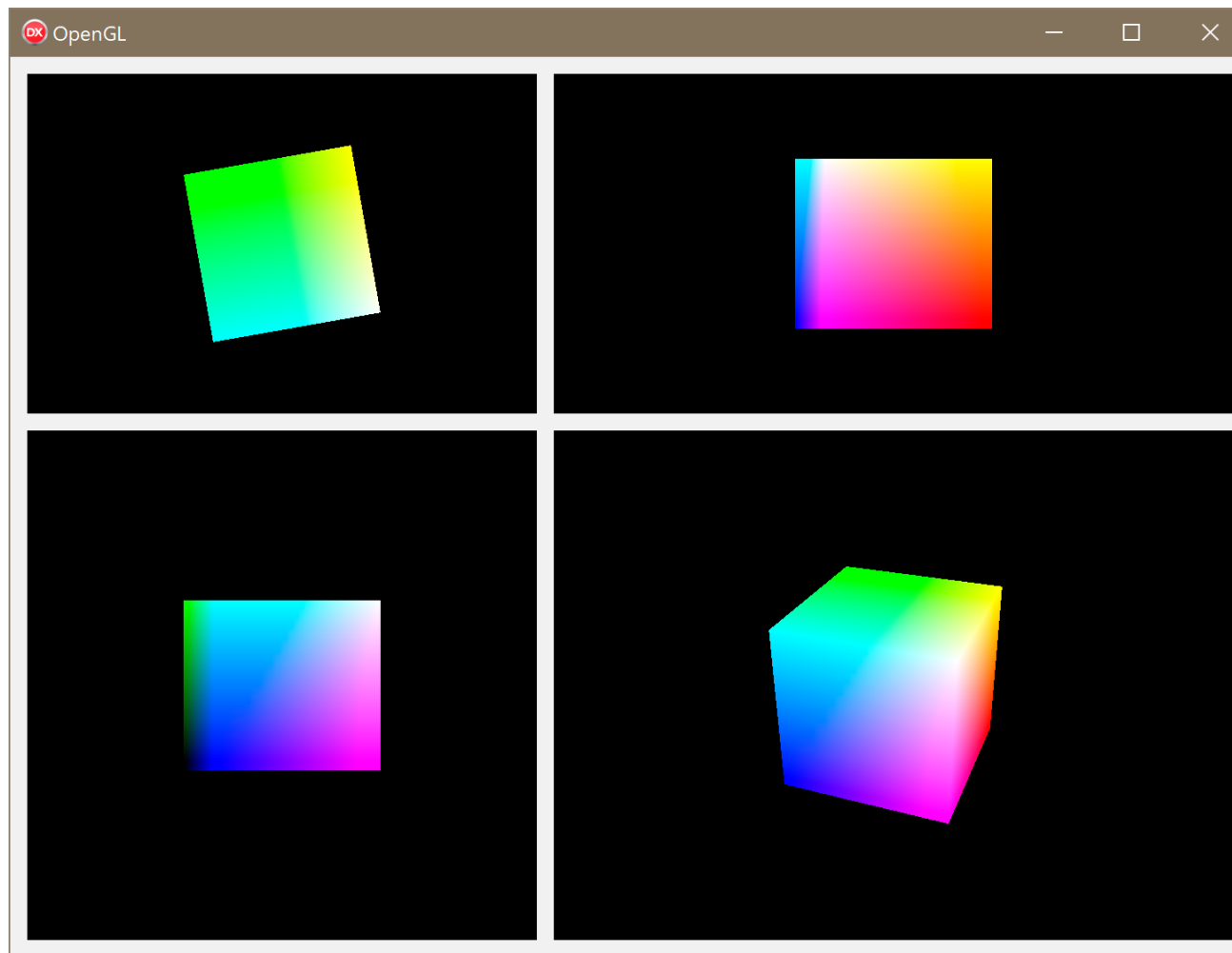


輝かせたい  
コミュニケーションの  
夢・未来  
IEICE100周年





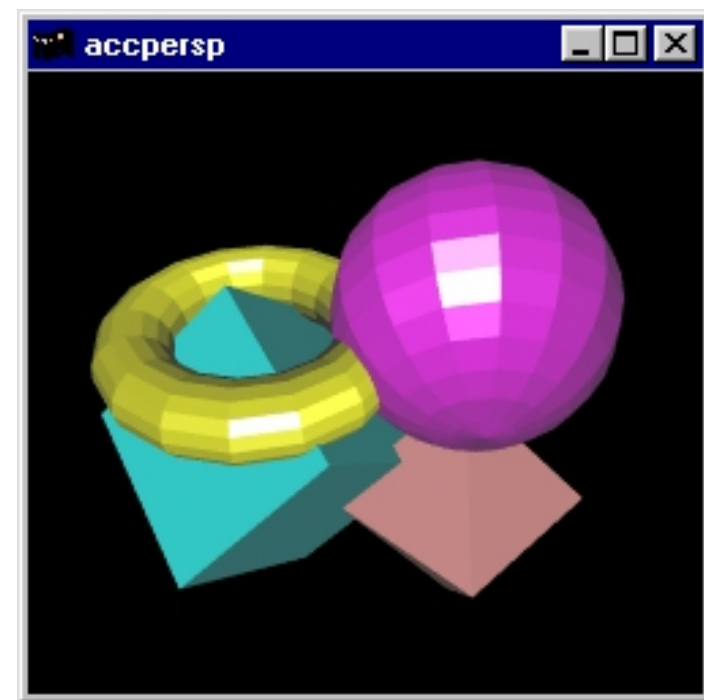
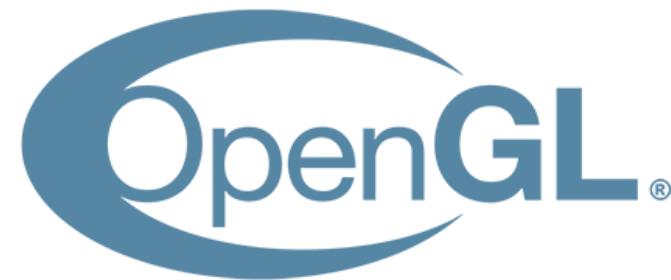
## ■ OpenGLとは？



**e**mbarcadero®  
DEVELOPER CAMP

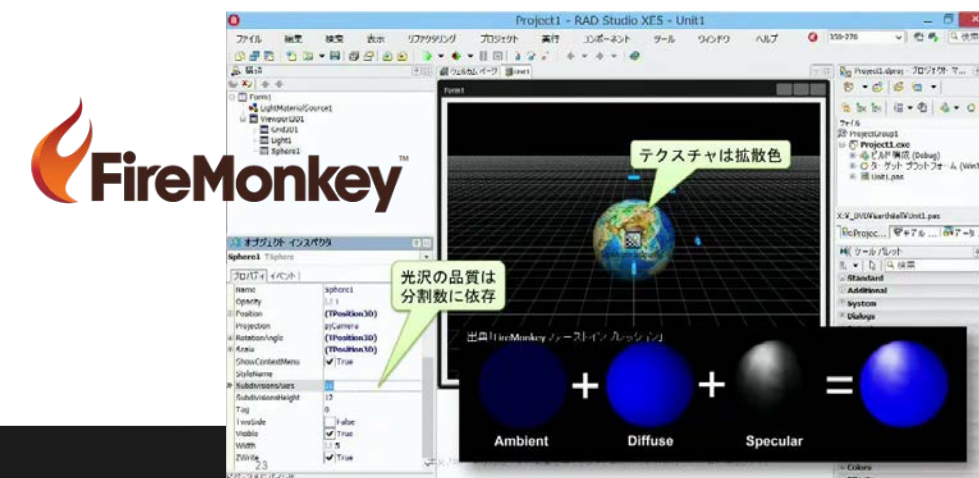
# OpenGLとは？.規格

- リアルタイム2D/3DCG用API
  - GPU : Graphics Processing Unit を活用
    - コンシューマ向けグラフィックボード
    - クリエイター向けグラフィックボード
- Khronos Group が策定
  - [www.khronos.org](http://www.khronos.org)
- クロスプラットフォーム
  - Windows, macOS, Android, iOS, Linux...
- C-API ベース
  - ラッパー作りが容易
  - Windows
    - `gl.h` → `Winapi.OpenGL.pas`
    - `glxext.h` → `Winapi.OpenGLext.pas`



# OpenGLとは？. ツール

- 高度なゲームを作りたい  
ゲームエンジンを使わざるを得ない
  - Unity
    - C#言語
  - UE : Unreal Engine
    - C++言語 / Blueprint (ノードプログラミング)
- とりあえず勉強したい  
GUIを想定していない
  - GLUT : OpenGL Utility Toolkit (開発終了)
  - GLFW : OpenGL Framework
- コンポーネントで作りたい
  - FMX : FireMonkey
    - パフォーマンス < 低負荷
  - GLScene
    - Windows, Linux, macOS?





# OpenGLとは？.資料

- C/C++ 向けのブログ/書籍が多い
  - A P I 名は同じなので翻訳は容易
- 昔の資料は役に立たない
  - A P I 仕様にバージョンが存在
  - レガシー A P I
    - 性能低下
    - いずれ使えなくなる
- 床井研究室 @ 和歌山大学
  - 床井浩平先生を知らないのはモグリ



床井研究室

2017年05月29日 [OpenGL][メモ] GLFW 3 で Oculus Rift を使う (1)

2017年05月30日 00:20更新

忙しい (こればかり)

でも本当に忙しいんです。意味わからなくらいに忙しいんです。たぶん、自分は仕事が遅いというか、一つの仕事に時間をかけすぎるのが問題なんだと思います。でももんはでんと言わんと結局事に迷惑をかけることになるので、ここそこ安請け合いをしないよう気を付けていたつもりなんですけど、結局いろんな仕事を引き受けてしまいました。そのせいで学生さんの面倒を十分見れてないんですけど、学生さんの方から「自分たちでやります」と言ってくれるので助かります。なんもかんも自分がやらんといかんと思ひこむのは、自分の驕りなんでしょうな。そのうちラーメンおごつたるからな (と安請け合い).

Read more...

2016年12月31日 [OpenGL][GLSL] SSAO ベースの SSRO 付きライブ放射照度マッピング

2017年02月09日 12:30更新

大晦日だ

今日は大晦日です。色々あった 2016 年も暮れていきます。今年は忙しかったです。本当に忙しかったです。iPhone の「ヘルスケア」アプリの「睡眠分析」には、今日は「1日平均: 4時間 8分」とか出ています。てなことをバンドの忘年会で自慢したら他の人も同じくらいの睡眠時間で、色々おかしかったと思います。そういえば私は先日ついに干して、任された仕事を一つ投げつけてしまいました。書きかけの原稿の催促もいただいています。返事していません。ごめんなさい。

Read more...

コメント(4) [コメントを投稿する]

Before...

- とこ [irm] さま、ご教示ありがとうございます。参考にさせていただきます。]
- irm [すみません、勘違いさせてしまいました。先生の記事がとてよくできていらっしゃるという意味です。m(\_ \_)m ま..]
- とこ [irm] さま、いえ、ポイントを頂いてありがたかったです。ありがとうございます。今後ともよろしくお願ひいたします。]

2016年12月03日 [OpenGL][GLSL] 魚眼レンズ画像の平面展開のサンプルプログラム

2016年12月10日 20:44更新

みんな頑張っている (のか?)

うちの研究室の学生さんたちがとあるコンテストに応募して審査員特別賞をもらったらしいのだけど、入賞 6 チームのうち 3 チームがうちの大学 (というか主にうちの学科) だったこともあり、これがすごいのかすごくないのかよくわからない状況のようです。彼らは優勝チームにはさすがに敵わないと思ったことと、同じ大学の他のチームに負けたことで、嬉しいのか嬉しいのかよくわからないみたいなのを言っていました。その割には記念写真でいつものよ

記事

最近の記事

- GLFW 3 で Oculus Ri..
- SSAO ベースの SSRO 付きラ..
- 魚眼レンズ画像の平面展開のサンプル..
- 矩形の描き方
- 魚眼レンズ画像の平面展開
- NuGet による freeglut..
- メッシュを使った図形描画
- 放射照度マッピング (2)
- 放射照度マッピング (1)
- 投影テクスチャマッピングとシャドウマ..

最近の更新

- FB0 を使ってデブスバッファを表示..
- GLFW 3 で Oculus Ri..
- 放射照度マッピング (2)
- SSAO ベースの SSRO 付きラ..
- 魚眼レンズ画像の平面展開
- 魚眼レンズ画像の平面展開のサンプル..
- 屈折マッピング
- 第 1 1 回 拡散反射光による陰影
- 矩形の描き方
- 第 2 回 Gouraud シェーディング..

記事カテゴリ

講義

- ゲームグラフィックス特論
- コンピュータグラフィックス
- C G 制作演習
- C G 入門
- デザイン情報入門セミナー
- デザイン情報概論
- ビジュアルデザイン
- マルチメディア技術
- メディアサイエンス基礎
- メディアデザイン演習
- 情報メディア総合演習
- メディアデザインセミナーI
- メディアデザインセミナーII
- 情報処理I
- 情報処理II
- 情報基礎演習II
- 基礎プログラミングII

その他

DEVELOPER CAMP

9

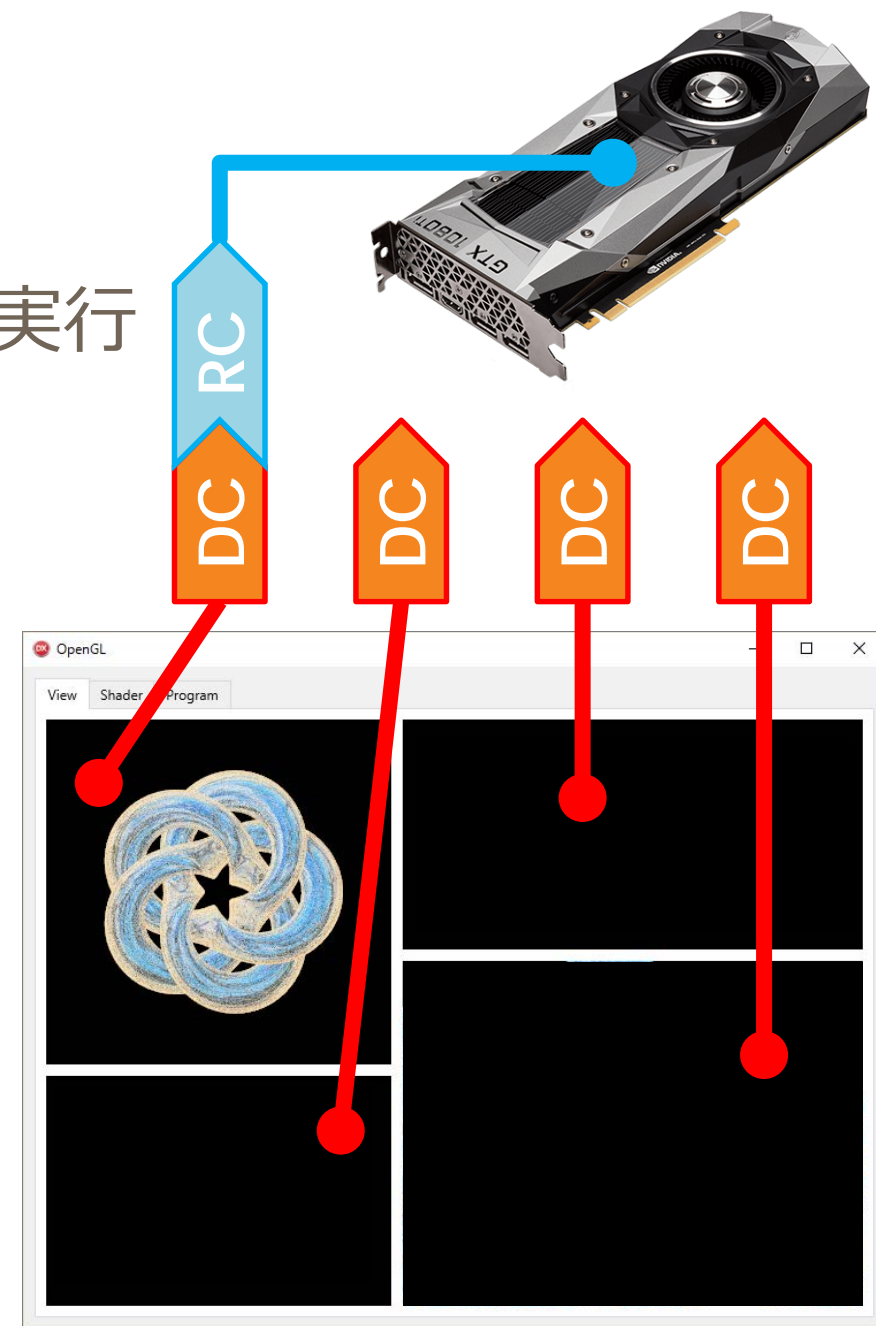
# OpenGLとは？.実行

## ■ 描画領域のD CにR Cを結合してA P Iを実行

- D C : Device Context
  - 描画領域のハンドル
  - HWND : ウィンドウハンドルを元に生成
- R C : Rendering Context
  - OpenGL実行環境のハンドル
  - D Cを元に生成
  - 異なるR Cの間では各種データは共有されない
    - wglShareLists 関数によって共有可能
  - 単一のR Cを複数のD Cへ使い回すことは可能？

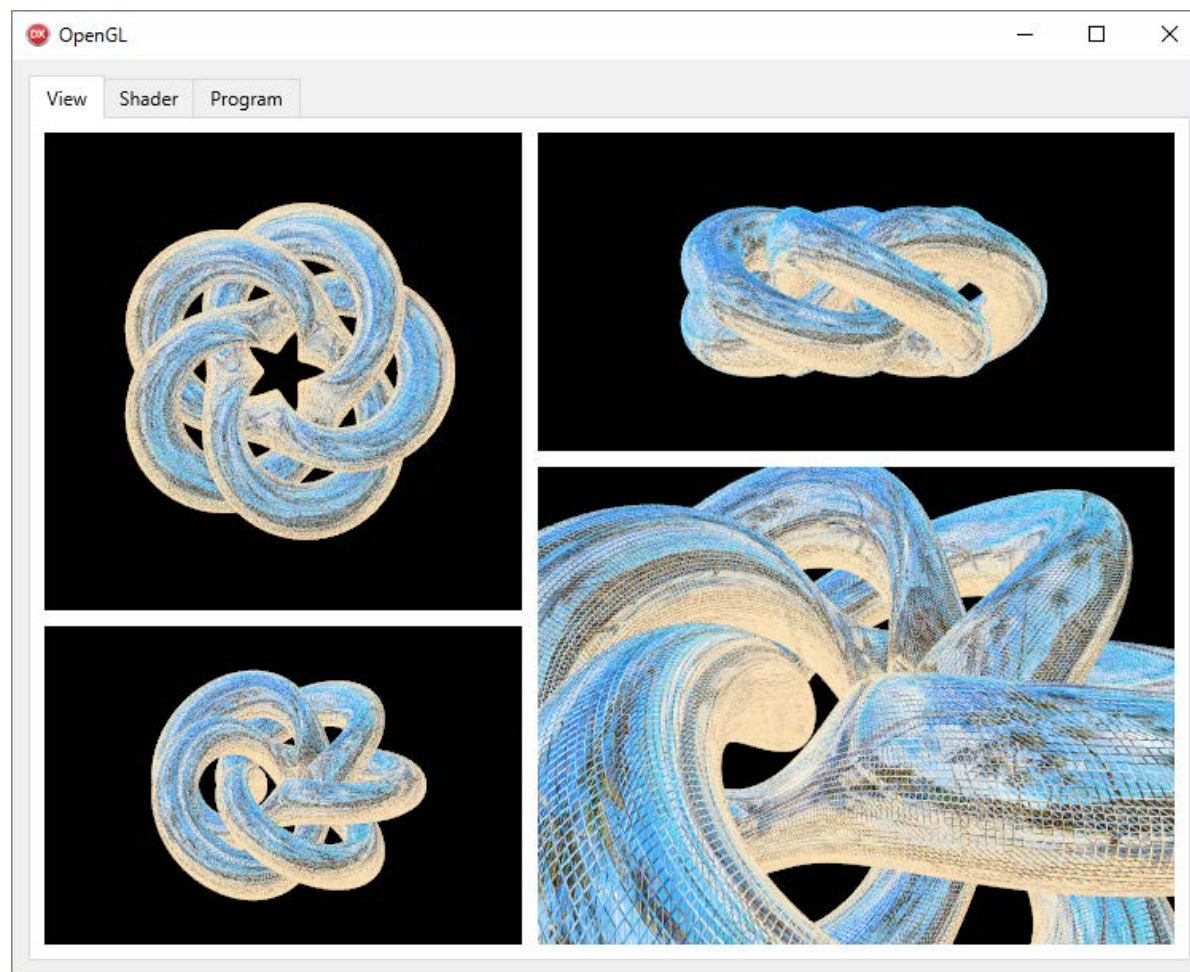
## ■ wglMakeCurrent 関数により結合

- 描画系以外のA P I 実行にも必須
- プロセス毎に有効化できる結合は1つ
  - マルチスレッド化できない





## ■ OpenGLの初期化



# OpenGLの初期化.手順

- ① D C の取得
  - FMX と VCL で異なる
- ② 理想のピクセル形式を定義
  - TPixelFormatDescriptor 構造体
- ③ 利用可能なピクセル形式を検索
  - ChoosePixelFormat 関数
- ④ D C へ現実ピクセル形式を適用
  - SetPixelFormat 関数
- ⑤ R C の生成
  - wglCreateContext 関数
- ⑥ 拡張 A P I の有効化
  - Winapi.OpenGLExt.InitOpenGLExt

[tokoik.github.io/opengl/delphi.html](https://tokoik.github.io/opengl/delphi.html)

The screenshot shows a web browser window with the address bar displaying `tokoik.github.io/opengl/delphi.html`. The page title is **Delphiによる最低限のOpenGL制御**. Below the title, the author is listed as 和歌山大学経済学部産業工学科47期2104番 中山礼児. The article is divided into sections: 1.はじめに, 2.OpenGLとは, 3.問題点, and 4.前準備. The text in the 'はじめに' section explains that the document is for people with some knowledge of programming who want to learn the minimum OpenGL control in Delphi. The '2.OpenGLとは' section explains that OpenGL is a standard graphics library and that the document will focus on the minimum control. The '3.問題点' section discusses the challenges of using OpenGL in Delphi, such as the lack of a standard library and the need for manual management of resources. The '4.前準備' section is partially visible at the bottom.

## Delphiによる最低限のOpenGL制御

和歌山大学経済学部産業工学科47期2104番 中山礼児

### 1.はじめに

この文書は、Borland社の高性能RADツールDelphiを使用しているプログラミングについて少量の知識を持っている人が、最低限のOpenGL制御を行うことを目的に書かれています。しかし、最低限の制御ということで話を進めたいと思いますが、本当に制御ができたかどうかを視覚的に確認できる方がよいので、『DelphiでOpenGLを使って3次元空間に板を回転させる』という事を具体的な最終目標にしましょう。

### 2.OpenGLとは

制御する前にOpenGLを詳しく知らないという人のための説明を入れておこうと思います。OpenGLは3次元アプリケーションを作成するための代表的な標準のグラフィックス関数ライブラリです。通常、3次元グラフィックスを扱う場合、複雑な数値計算を処理しなければなりません。しかし、OpenGLを使用すると、それらの複雑な計算をOpenGLが行ってくれるため、プログラマ側はそれらの計算を全く意識せずに作成することができます。さらに、OpenGLは、多数の異なるプラットフォームでの互換性に優れているため、他のマシンやオペレーティングシステムに比較的簡単に移植することができます。

### 3.問題点

Delphiを使ってがOpenGLを制御する上で、知っておかなければならない問題が最低でも2つ存在します。1つ目の問題は、OpenGLが通常C/C++上で制御される事を前提に創られているという事です。これは、Delphiユーザーなら常につきまとう問題なのですが、これは逆に、ある程度の技術を持っているユーザーや、CのプログラムをDelphiへ移植するスキルを持っているならば、それらの大量なソースファイル等の資源を有効利用できる可能性が出てくることを意味します。

さて、2つ目の問題ですが、互換性の問題です。OpenGLのプログラムが異なるプラットフォームでの互換性が優れている理由の1つに、そのほとんどのプログラムが同じC/C++で書かれているからに違いありません。しかし、DelphiをWindows環境以外のプラットフォームで実行するのは事実上無理でしょう(一部のエミュレータなどにより擬似的な実行は可能かもしれませんが、100%のパフォーマンスは出せないのではないのでしょうか?)。従って、他のプラットフォームに移植する場合、Object PascalからC/C++に移植した上で、元と先のプラットフォーム固有の部分を変更しなければなりません。そのため、DelphiでUNIX等の環境への移植性を高めたい場合、Object PascalからC/C++への移植という作業の事も考慮に入れて創る必要が生じてしまい、細心の注意を払ったコーディングを行わなければなりません。しかも、その際のソースコードは極めてCライクなソースになりやすく、Delphiの一部の特徴を発揮させることが困難になります。その様な理由と、私自身の技術と、OpenGLに関しての知識が未熟なため、この文書では他のプラットフォームや言語への互換性はほぼ考えていません。

### 4.前準備

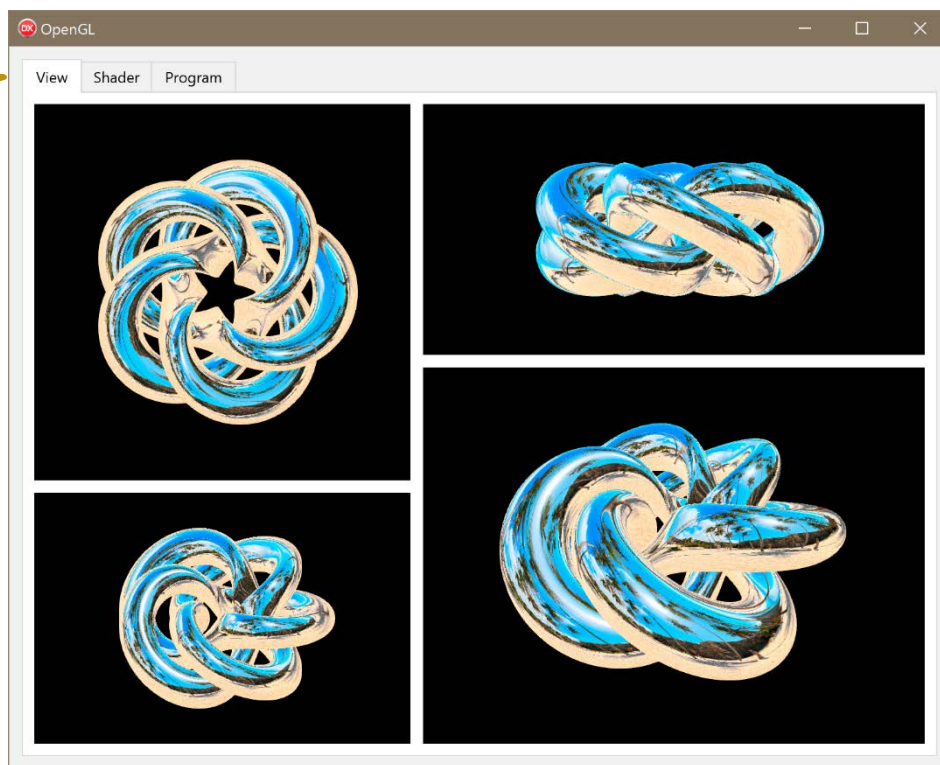


# OpenGLの初期化.① D Cの取得

- FMX のコンポーネントはビットマップ
  - ウィンドウ全体の D C を F M X が管理
  - コンポーネントの H W N D を個別に取得できない

HWND

FMX



HWND

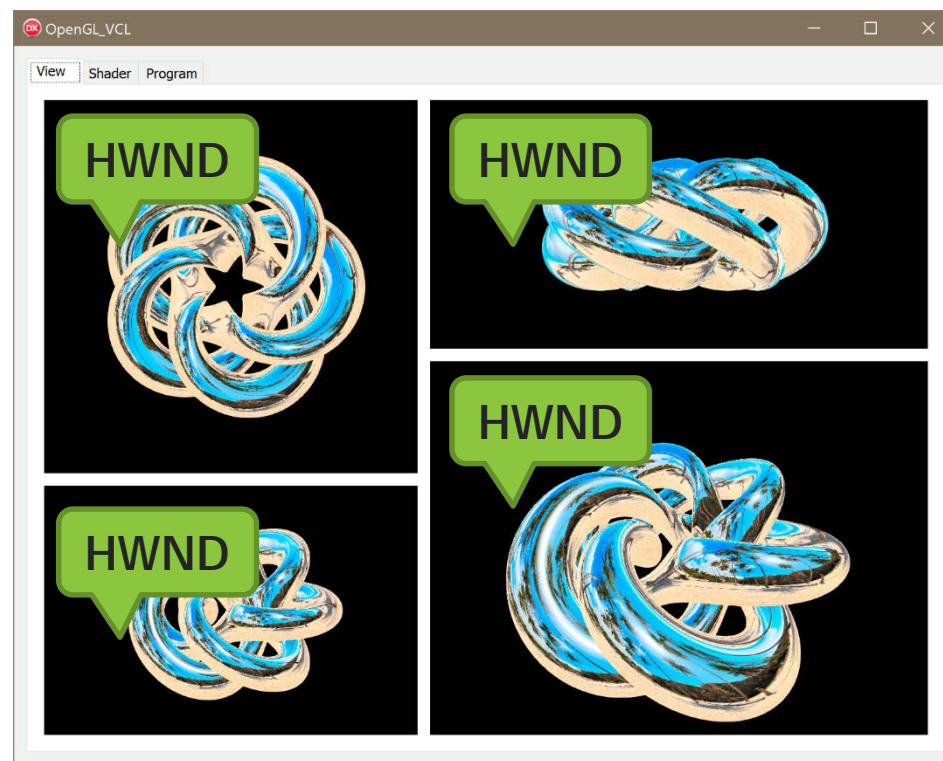
HWND

HWND

HWND

HWND

VCL



# OpenGLの初期化.① D Cの取得. V C Lの場合

- TFrame クラスを利用してビューワーコンポーネントを作成
  - VCL / LUX.GPU.OpenGL.Viewer.pas
- Vcl.Controls.TWinControl を継承するコンポーネント
  - 必ず HWND を持つ
    - Handle プロパティにより HWND を取得

```
constructor TGLViewer.Create( AOwner_:TComponent );
begin
    inherited;
    _OnPaint := procedure begin end;
    CreateDC;
    _Viewer := TGLUnifor<TSingleM4>.Create( GL_DYNAMIC_DRAW );
    _Viewer.Count := 1;
end;

destructor TGLViewer.Destroy;
begin
    _Viewer.DisposeOf;
    DestroyDC;
    inherited;
end;
```

```
procedure TGLViewer.CreateDC;
begin
    _DC := GetDC( Handle );
    _OpenGL_.ApplyPixelFormat( _DC );
end;
```

DCを取得

```
procedure TGLViewer.DestroyDC;
begin
    ReleaseDC( Handle, _DC );
end;
```

DCを廃棄



# OpenGLの初期化.① D Cの取得. F M Xの場合

- TFrame クラスを利用してビューワーコンポーネントを作成
  - FMX / LUX.GPU.OpenGL.Viewer.pas
- コンポーネントに縁のないサブウィンドウをはめる
  - TCommonCustomForm クラスから生成
    - FMX.Platform.Win.WindowHandleToPlatform 関数で HWND を取得

ルートWND : TForm

```
procedure TGLViewer.CreateWindow;  
begin  
  _Form := TCommonCustomForm.CreateNew( Self );  
  with _Form do  
  begin  
    BorderStyle := TFmxFormBorderStyle.None;  
    OnMouseClicked := GoMouseClicked;  
    OnMouseDown := GoMouseDown;  
    OnMouseMove := GoMouseMove;  
    OnMouseUp := GoMouseUp;  
    OnMouseWheel := GoMouseWheel;  
    HandleNeeded;  
    _WND := WindowHandleToPlatform( Handle ).Wnd;  
  end;  
  SetWindowLong( _WND, GWL_STYLE, WS_CHILD or WS_CLIPSIBLINGS );  
  Winapi.Windows.SetParent( _WND, GetRootWND );  
end;
```

サブWNDを生成

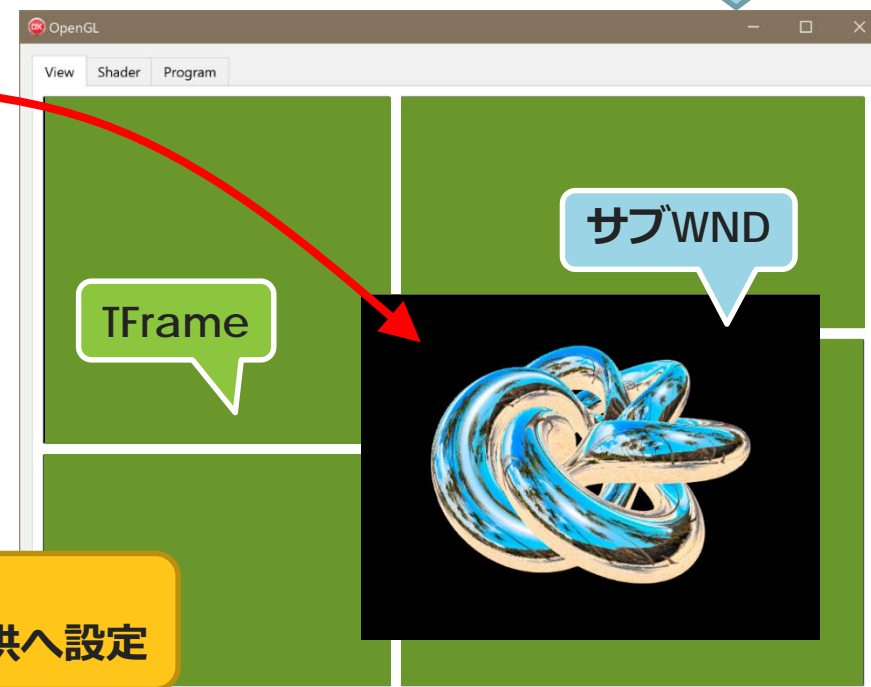
縁を除去

HWNDを強制生成

HWNDを取得

サブWNDのスタイルに  
チャイルド属性を付与

サブWNDを  
ルートWNDの子供へ設定



# OpenGLの初期化.① D Cの取得. F M Xの場合.変更検知

- サブWND は TFrame コンポーネントの配下ではない
  - 単に TFrame と同じ位置に**重ねている**だけ
    - TFrame が移動/変形しても サブWND は**追従しない**
  - サイズ変化を検知するには以下のメソッドの override が必須
    - TControl.DoAbsoluteChanged
    - TControl.Resize

TFrame の絶対位置と絶対サイズを取得

サブWNDのサイズを変更

```
procedure TGLViewer.FitWindow;  
var  
  R : TRectF;  
begin  
  R := TRectF.Create( LocalToAbsolute( TPointF.Zero ) * Scene.GetSceneScale, Width, Height );  
  _Form.Bounds := R.Round;  
  if Height < Width then _Viewer[ 0 ] := TSingleM4.Scale( Height / Width, 1, 1 )  
  else  
    if Width < Height then _Viewer[ 0 ] := TSingleM4.Scale( 1, Width / Height, 1 )  
    else _Viewer[ 0 ] := TSingleM4.Identity;  
end;
```

高解像度モニタのスケールを乗算  
Scene.GetSceneScale

- 可視性変化を検知するには以下のメソッドの override が必須
  - TFmxObject.ParentChanged
  - TControl.AncestorVisibleChanged

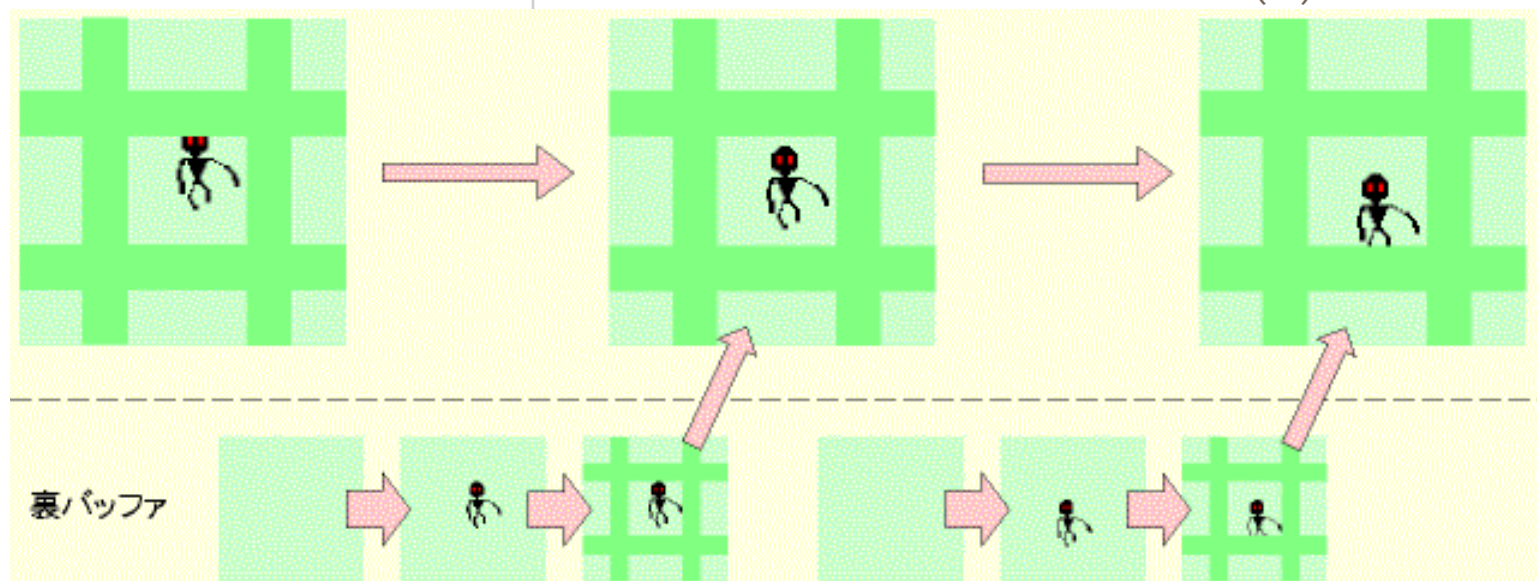


# OpenGLの初期化. ②理想のピクセル形式を定義

- 色のビット数や描画方法を設定
- Winapi.Windows.TPixelFormatDescriptor 構造体で定義
  - 実行時のGPUがサポートしている必要はない

```
class function TOpenGL.DefaultPFD :TPixelFormatDescriptor;  
begin  
  with Result do  
  begin  
    nSize      := SizeOf( TPixelFormatDescriptor );  
    nVersion   := 1;  
    dwFlags    := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL or PFD_DOUBLEBUFFER;  
    iPixelFormat := PFD_TYPE_RGBA;  
    cColorBits  := 24;  
    cRedBits    := 0;  
    cRedShift   := 0;  
    cGreenBits  := 0;  
    cGreenShift := 0;  
    cBlueBits   := 0;  
    cBlueShift  := 0;  
    cAlphaBits  := 0;  
    cAlphaShift := 0;  
    cAccumBits  := 0;  
    cAccumRedBits := 0;  
    cAccumGreenBits := 0;  
    cAccumBlueBits := 0;  
    cAccumAlphaBits := 0;  
    cDepthBits  := 32;  
    cStencilBits := 0;  
    cAuxBuffers := 0;  
    iLayerType  := PFD_MAIN_PLANE;  
    bReserved   := 0;  
    dwLayerMask := 0;  
    dwVisibleMask := 0;  
    dwDamageMask := 0;  
  end;  
end;  
end;
```

ダブルバッファ



(C)鹿児島大学

## OpenGLの初期化.③利用可能なピクセル形式を検索

- Winapi.Windows.ChoosePixelFormat 関数で検索
  - DCが扱える範囲で 理想 PFD に最も近い現実 PFD を検索
  - 実行デバイスで有効な PFD は ID で管理されている

```
procedure TOpenGL.ValidatePFD( const PFD_:TPixelFormatDescriptor );  
var  
    I :Integer;  
begin  
    _PFD := PFD_;  
    I := ChoosePixelFormat( _DC, @_PFD );  
    Assert( I > 0, 'Not found the PixelFormat with a close setting!' );  
    ValidatePFI( I );  
end;
```

DC

理想 PFD

現実 PFD の ID

ID から現実 PFD を取得する

## OpenGLの初期化.④ D Cへピクセル形式を適用

- Winapi.Windows.SetPixelFormat 関数で設定
  - 一度設定すると変更できない
  - 一旦 D C を廃棄し新規に作成する必要

```
procedure TOpenGL.ApplyPixelFormat( const DC_:HDC );  
begin  
    Assert( SetPixelFormat( DC_, _PFI, @_PFD ), 'SetPixelFormat() is failed!' );  
end;
```

DC

ID

PFDの実データが返る

- Winapi.Windows.DescribePixelFormat 関数で取得
  - 設定せずに PFD の ID から実データのみを取得

```
procedure TOpenGL.ValidatePFI( const PFI_:Integer );  
begin  
    _PFI := PFI_;  
    Assert( DescribePixelFormat( _DC, _PFI, SizeOf( TPixelFormatDescriptor ), _PFD ),  
        'Not found the PixelFormat of the index!' );  
end;
```

DC

ID

PFDの実データが返る



## OpenGLの初期化.⑤ R Cの生成

- Winapi.Windows.wglCreateContext 関数で生成
- Winapi.Windows.wglDeleteContext 関数で廃棄

```
procedure TOpenGL.CreateRC;
```

```
begin
```

```
    ApplyPixelFormat( _DC );
```

DCへPFDを設定

```
    _RC := wglCreateContext( _DC );
```

DCからRCを生成

```
end;
```

```
procedure TOpenGL.DestroyRC;
```

```
begin
```

```
    wglDeleteContext( _RC );
```

RCを廃棄

```
end;
```

# OpenGLの初期化.⑥拡張 A P I の有効化

- Winapi.**OpenGLex** ユニットで定義されている拡張 A P I
  - デフォルトで有効化されていない
- Winapi.OpenGLex.**InitOpenGLex** 関数で初期化する必要あり
  - wglGetProcAddress 関数で各 A P I の関数ポインタを取得
- **wglGetProcAddress** 関数
  - OpenGLの A P I が実行可能な状態でないと実行できない
    - D C と R C を結合させてから実行

```

- procedure InitOpenGLExt;
.
.
- implementation
.
- procedure InitOpenGLExt;
00 begin
.   glDrawRangeElements := wglGetProcAddress('glDrawRangeElements');
.   glTexImage3D := wglGetProcAddress('glTexImage3D');
.   glTexSubImage3D := wglGetProcAddress('glTexSubImage3D');
.   glCopyTexSubImage3D := wglGetProcAddress('glCopyTexSubImage3D');
-   glActiveTexture := wglGetProcAddress('glActiveTexture');
.   glSampleCoverage := wglGetProcAddress('glSampleCoverage');
.   glCompressedTexImage3D := wglGetProcAddress('glCompressedTexImage3D');

```

## OpenGLの初期化.②～⑤

- LUX.GPU.OpenGL.TOpenGL クラスで行う
- 一時的にコマンド実行を可能とするためのダミー D Cを用意する
  - 隠しウィンドウを作る

```
□ constructor TOpenGL.Create;  
begin  
    inherited;  
    CreateWindow; ①隠しウィンドウ生成  
    CreateDC;  
    ValidatePFD( DefaultPFD ); ②③  
    CreateRC; ④⑤  
    BeginGL;  
    InitOpenGL;  
    EndGL;  
    if wglGetCurrentContext = 0 then BeginGL;  
end;
```

```
□ destructor TOpenGL.Destroy;  
begin  
    if wglGetCurrentContext = _RC then EndGL;  
    DestroyRC;  
    DestroyDC;  
    DestroyWindow; 隠しウィンドウ廃棄  
    inherited;  
end;
```



# OpenGLの初期化.①～⑥

- メインフォームが表示される前に初期化を済ませる
  - F M X の場合

```
procedure TOpenGL_FMX.CreateWindow;  
begin  
    _Form := TCommonCustomForm.Create( nil );  
    _WND := WindowHandleToPlatform( _Form.Handle ).Wnd;  
end;  
  
procedure TOpenGL_FMX.DestroyWindow;  
begin  
    _Form.DisposeOf;  
end;
```

```
initialization //$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  
    _OpenGL_ := TOpenGL_VCL.Create;  
    InitOpenGLext;
```

- V C L の場合

```
procedure TOpenGL_VCL.CreateWindow;  
begin  
    _Form := TCustomForm.CreateNew( nil );  
    _WND := _Form.Handle;  
end;  
  
procedure TOpenGL_VCL.DestroyWindow;  
begin  
    _Form.Free;  
end;
```

```
initialization //$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  
    _OpenGL_ := TOpenGL_FMX.Create;  
    InitOpenGLext;
```

## ■ コマンドの実行

```
GLViewer1.OnPaint := procedure
begin
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity;
    glOrtho( -2, +2, -2, +2, _N, _F );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity;
    glTranslatef( 0, 0, -5 );
    glRotatef( +90, 1, 0, 0 );
    glRotatef( _Angle, 0, 1, 0 );
    DrawShaper;
end;
```



# コマンドの実行.開始/終了

- 描画したいDCとRCを結合
  - wglMakeCurrent 関数

```
procedure TGLViewer.BeginRender;  
begin  
    BeginGL; ●  
    glClearColor( 0, 0, 0, 0 );  
    glClear( GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT );  
    _Viewer.Use( 0{BinP} );  
end;  
  
procedure TGLViewer.EndRender;  
begin  
    _Viewer.Unuse( 0{BinP} );  
    glFlush; ●  
    EndGL; ●  
    SwapBuffers( _DC );  
end;
```

描画領域のクリア

コマンドキュー強制実行

ダブルバッファ交換

```
procedure TOpenGL.BeginGL;  
begin  
    wglMakeCurrent( _DC, _RC );  
end;  
  
procedure TOpenGL.EndGL;  
begin  
    wglMakeCurrent( _DC, 0 );  
end;
```

隠しWNDのDCを外す

```
procedure TGLViewer.BeginGL;  
begin  
    _OpenGL_.EndGL; ●  
    wglMakeCurrent( _DC, _OpenGL_.RC );  
end;  
  
procedure TGLViewer.EndGL;  
begin  
    wglMakeCurrent( _DC, 0 );  
    _OpenGL_.BeginGL; ●  
end;
```

TGLViewerのDCと  
隠しWNDのRCを結合

TGLViewerのDCを外す



# コマンドの実行.OnPaint イベント

- F M X の場合
  - Paint メソッドで描画
- V C L の場合
  - WMPaint メソッドで描画
  - WMeraseBkgnd メソッドを無効化

```
GLViewer1.OnPaint := procedure
begin
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity;
    glOrtho( -2, +2, -2, +2, _N, _F );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity;
    glTranslatef( 0, 0, -5 );
    glRotatef( +90, 1, 0, 0 );
    glRotatef( _Angle, 0, 1, 0 );
    DrawShaper;
end;
```

OpenGLコマンド

高画質モニタの  
スケールを考慮した実ピクセル数

```
procedure TGLViewer.Paint;
begin
    BeginRender;

    with GetPixSiz do glViewport( 0, 0, Width, Height );
    if Assigned( _Camera ) then _Camera.Render;
    if Assigned( _OnPaint ) then _OnPaint;
    EndRender;
end;
```

描画範囲の設定

OnPaint 実行

```
procedure TGLViewer.WMEraseBkgnd( var Message_:TWMEraseBkgnd );
begin
    ///// 背景描画を無効化
end;

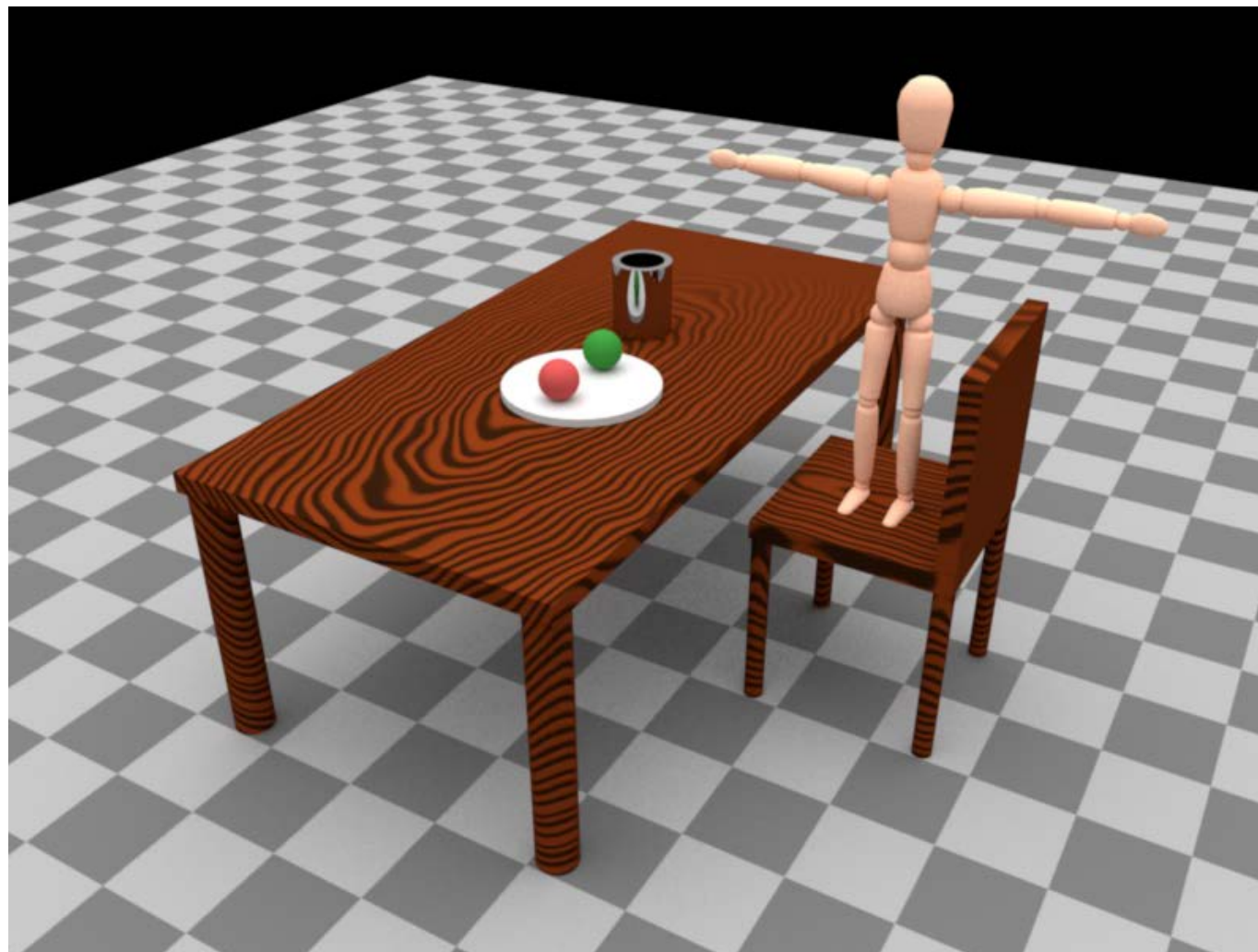
procedure TGLViewer.WMPaint( var Message_:TWMPaint );
begin
    inherited;
    BeginRender;

    glViewport( 0, 0, ClientWidth, ClientHeight );
    if Assigned( _Camera ) then _Camera.Render;
    if Assigned( _OnPaint ) then _OnPaint;
    EndRender;
end;
```

描画範囲の設定

OnPaint 実行

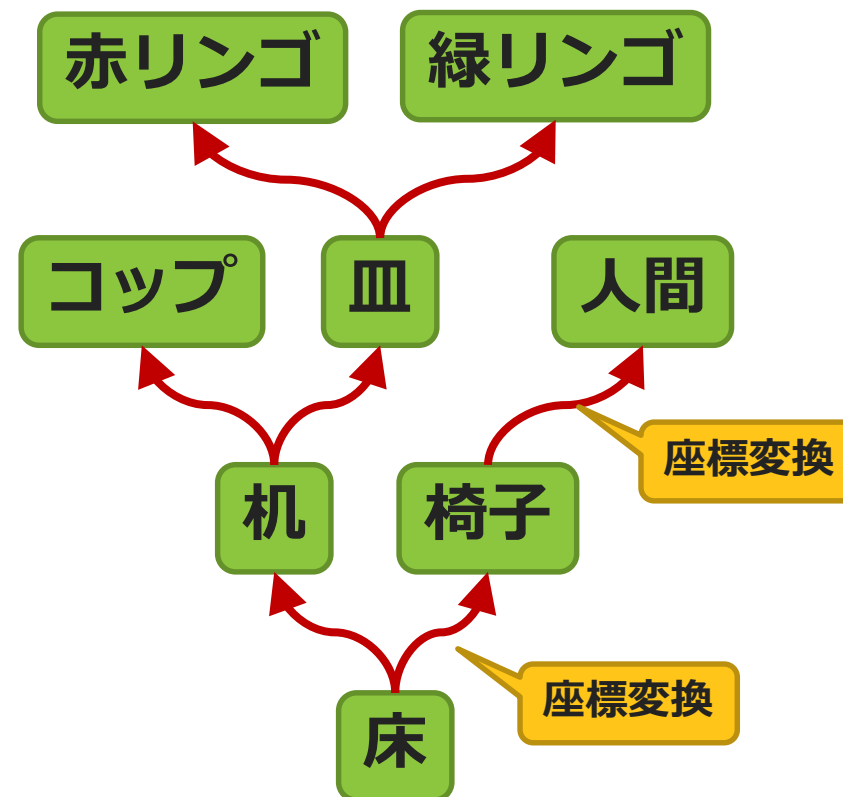
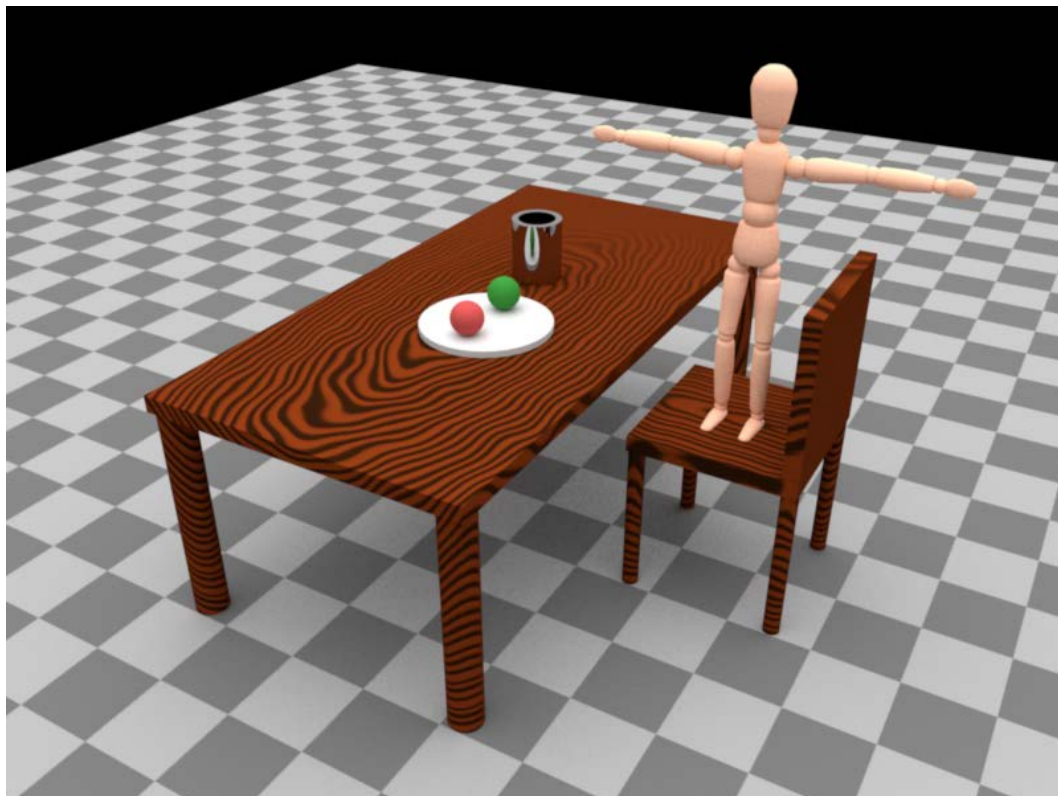
## ■シーンの構築



**embarcadero**  
DEVELOPER CAMP

# シーンの構築.シーングラフ

- シーン構造をツリー構造で表現
  - 物体はノード
  - パスを経る旅に位置/姿勢が変更されると考える

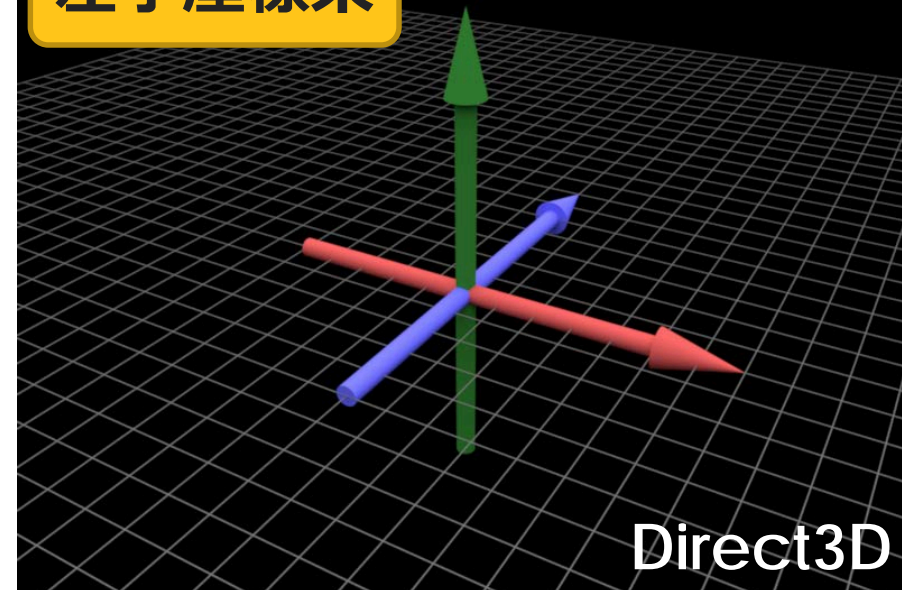




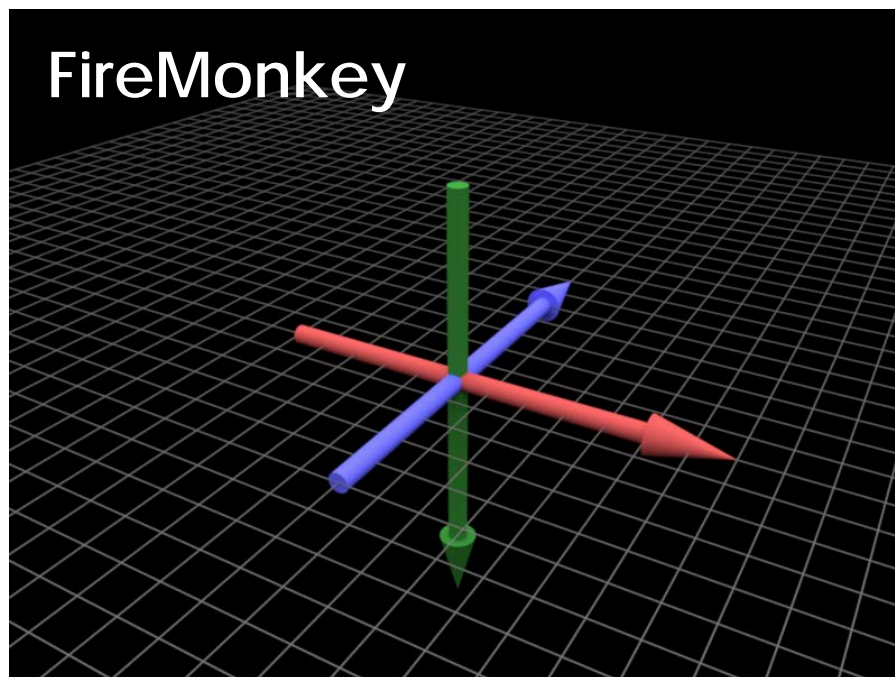
# シーンの構築.座標系

- OpenGL は右手座標系
  - X : 親指
  - Y : 人差し指
  - Z : 中指

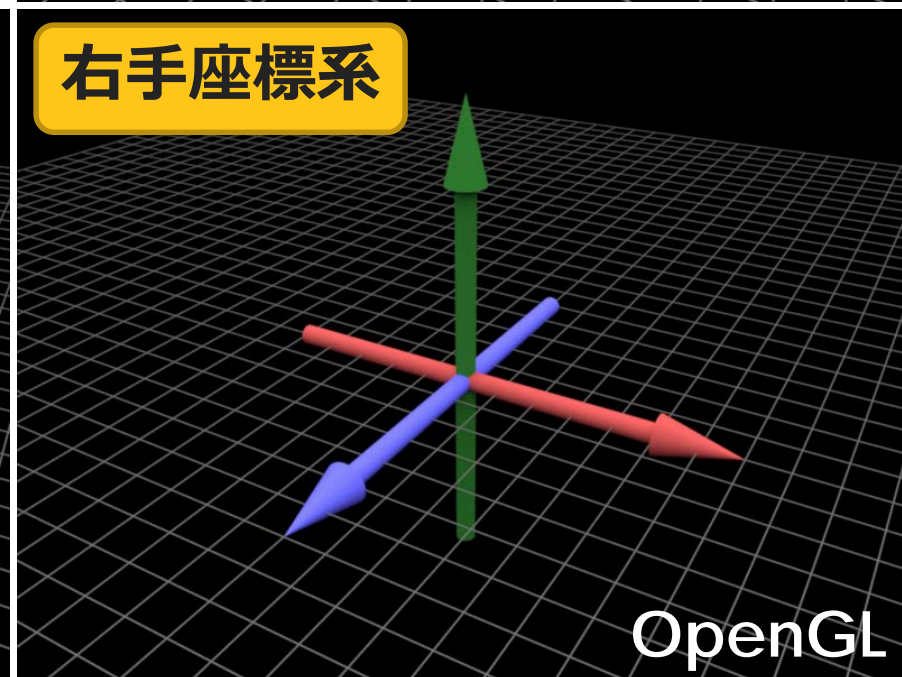
左手座標系



FireMonkey

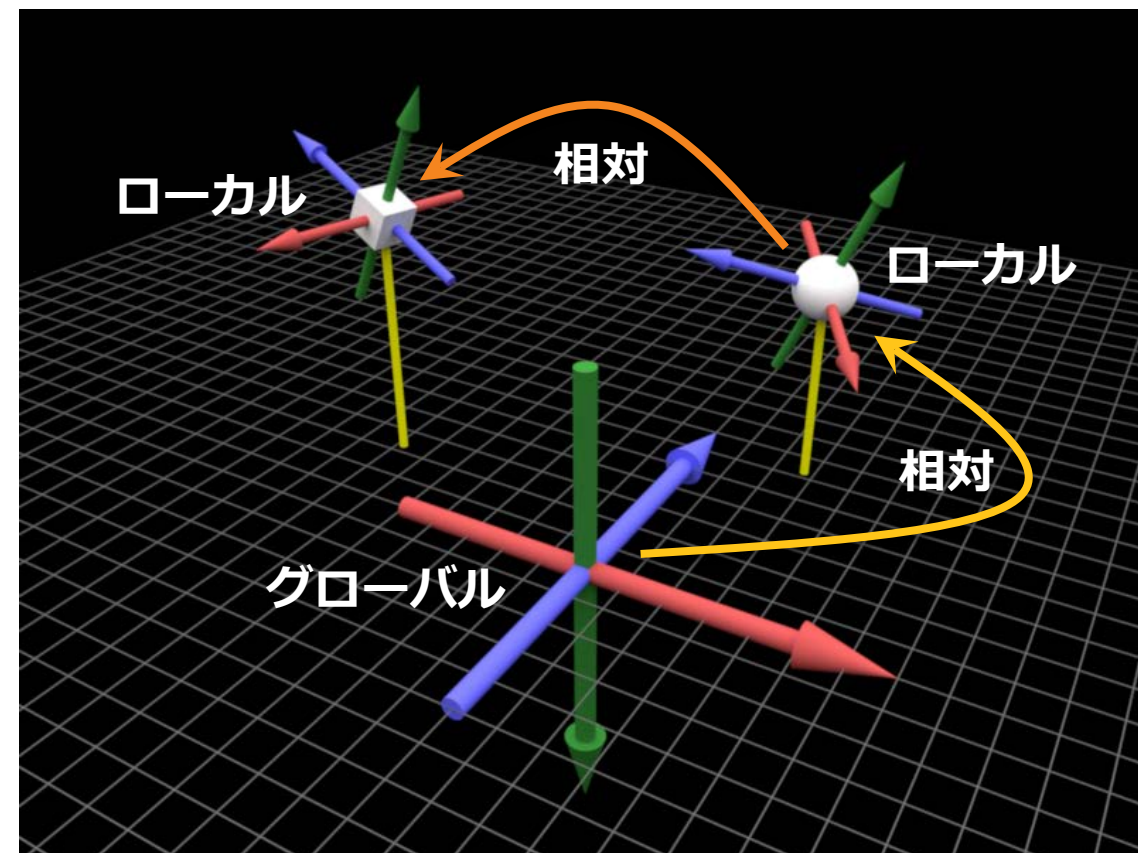


右手座標系



# シーンの構築.座標

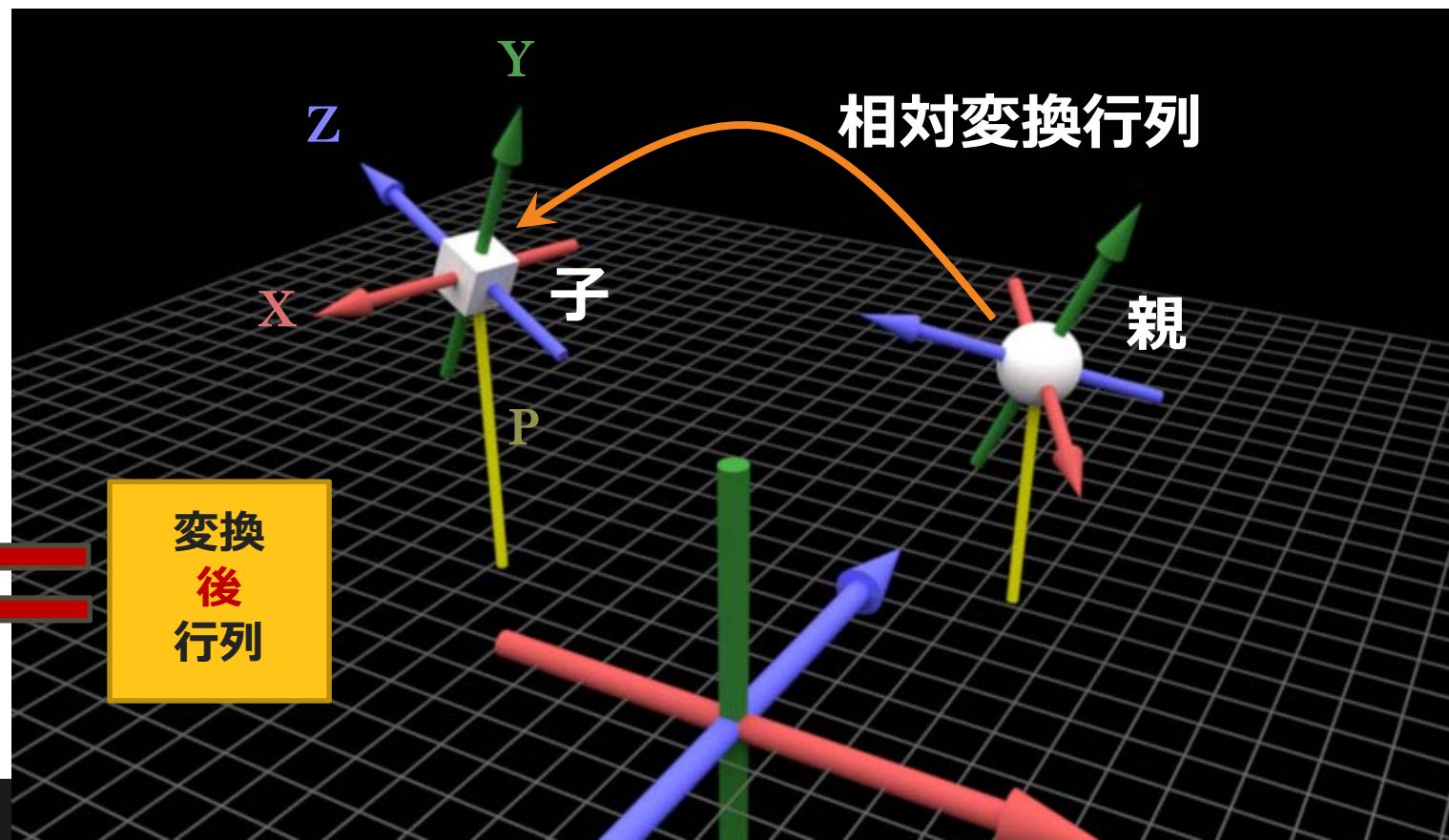
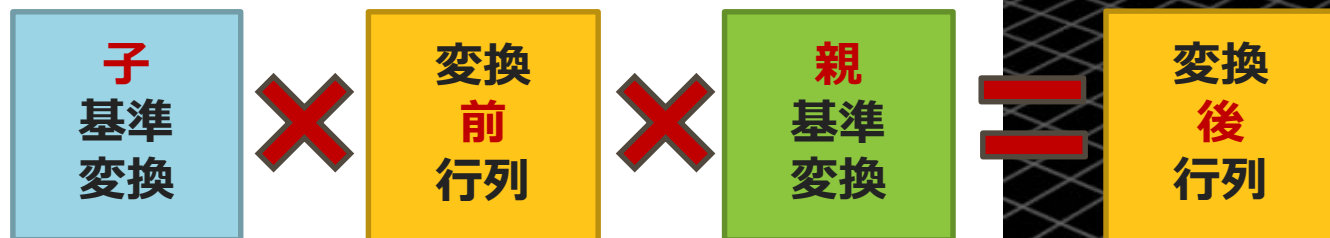
- グローバル（ワールド）座標系
  - ただひとつの絶対基準
- ↕
- ローカル（オブジェクト）座標系
  - 物体の姿勢を表す
- 絶対座標
  - グローバル座標系から見た絶対位置
- ↕
- 相対座標
  - ローカル座標系から見た相対位置



# シーンの構築. 同時座標系

- 座標変換は行列（ $4 \times 4$ ）の演算で表現できる
  - ローカル座標系の基底ベクトルを並べたもの
  - 複数の変換を合成して1つの行列で表せる

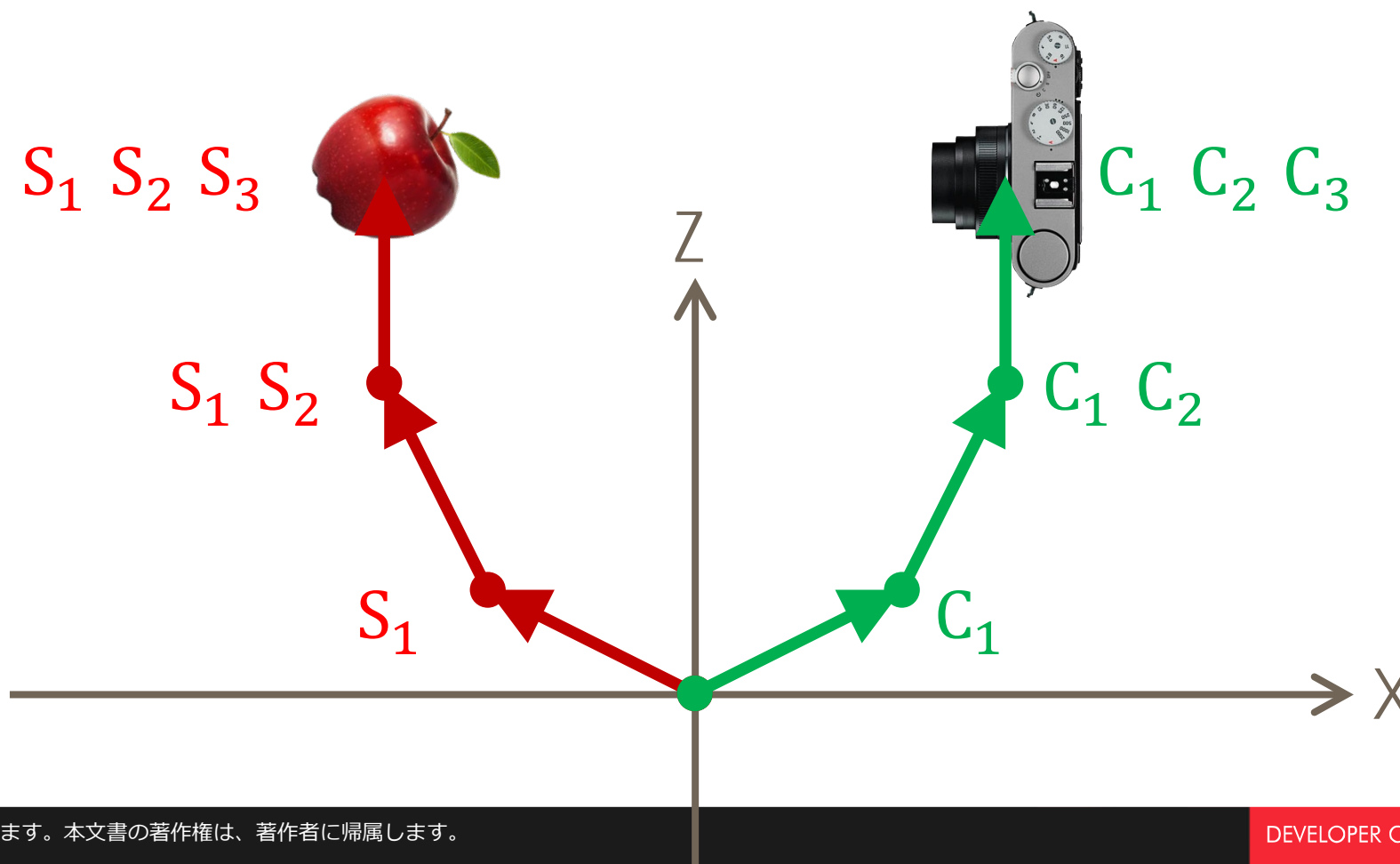
$$\begin{bmatrix} \mathbf{X}_X & \mathbf{Y}_X & \mathbf{Z}_X & \mathbf{P}_X \\ \mathbf{X}_Y & \mathbf{Y}_Y & \mathbf{Z}_Y & \mathbf{P}_Y \\ \mathbf{X}_Z & \mathbf{Y}_Z & \mathbf{Z}_Z & \mathbf{P}_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_X \\ A_Y \\ A_Z \\ 1 \end{bmatrix} = \begin{bmatrix} B_X \\ B_Y \\ B_Z \\ 1 \end{bmatrix}$$





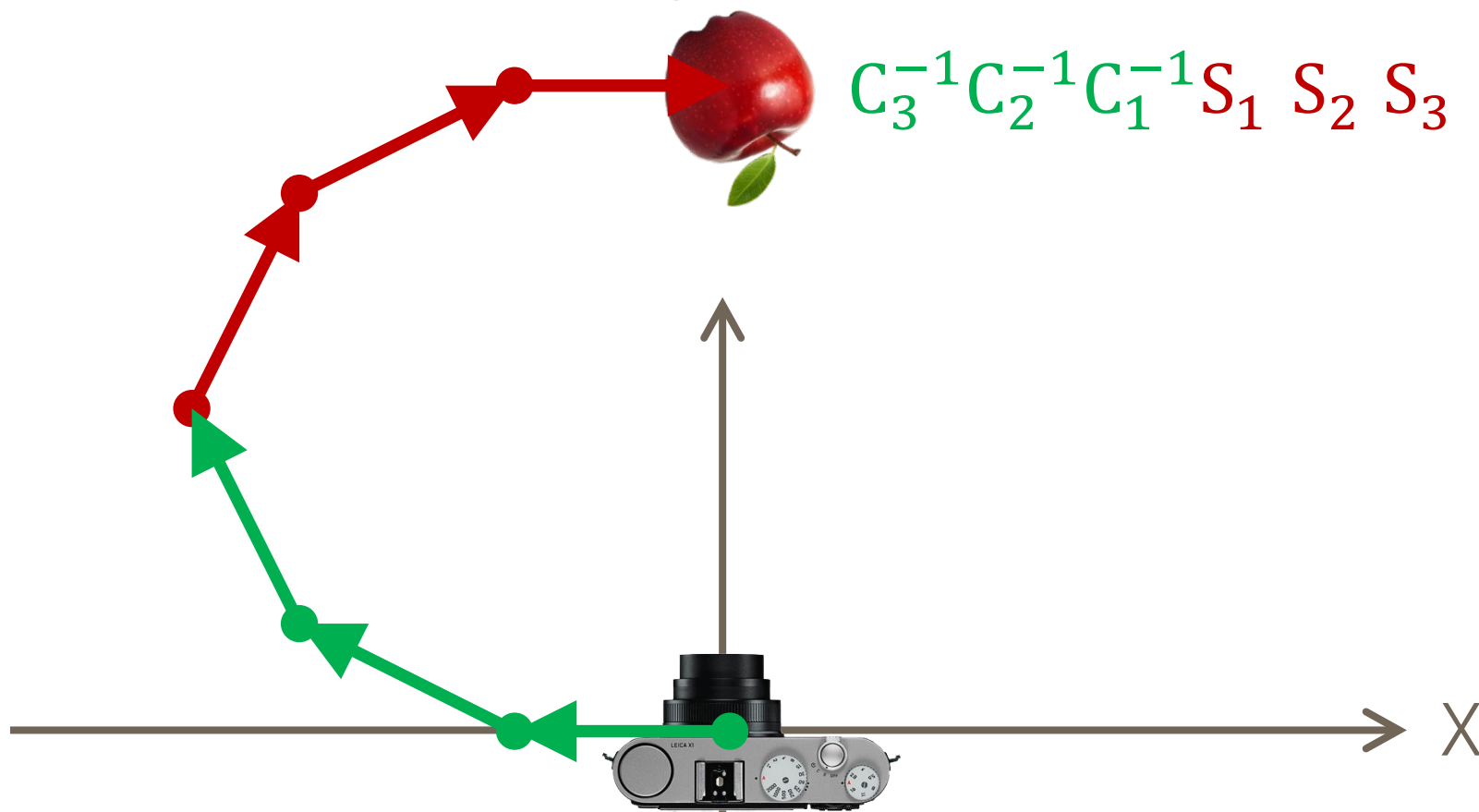
# シーンの構築.カメラ

- カメラも位置と姿勢を持つ物体の一種である



# シーンの構築.カメラ.視界

- OpenGL はカメラから見た風景しか描けない
  - 物体にカメラの**逆行列**を左から掛ける



# シーンの構築.カメラ.設定

## ■ 現在は非推奨

```
GLViewer4.OnPaint := procedure
begin
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity;
    glFrustum( -4/4*_N, +4/4*_N,
               -3/4*_N, +3/4*_N, _N, _F );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity;
    glTranslatef( 0, +0.3, 0 );
    glTranslatef( 0, 0, -3 );
    glRotatef( +30, 1, 0, 0 );
    glRotatef( _Angle, 0, 1, 0 );
    DrawShaper;
end;
```

単位行列をロード

座標変換行列指定モード

単位行列をロード

投影行列指定モード

投影行列を乗算

Y方向へ移動

Z方向へ移動

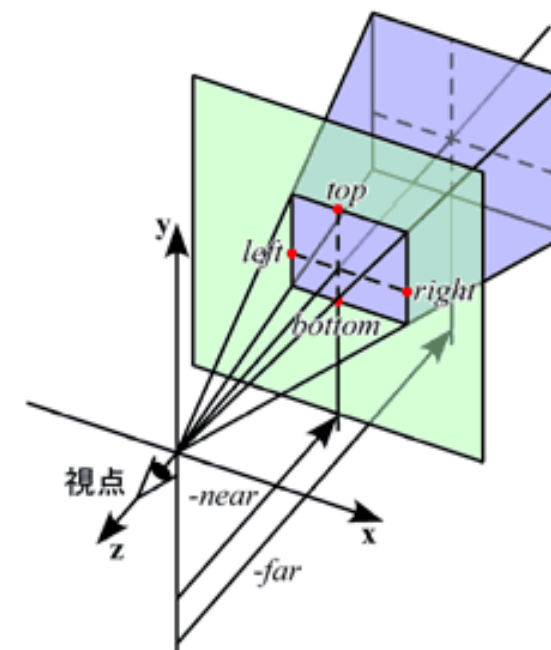
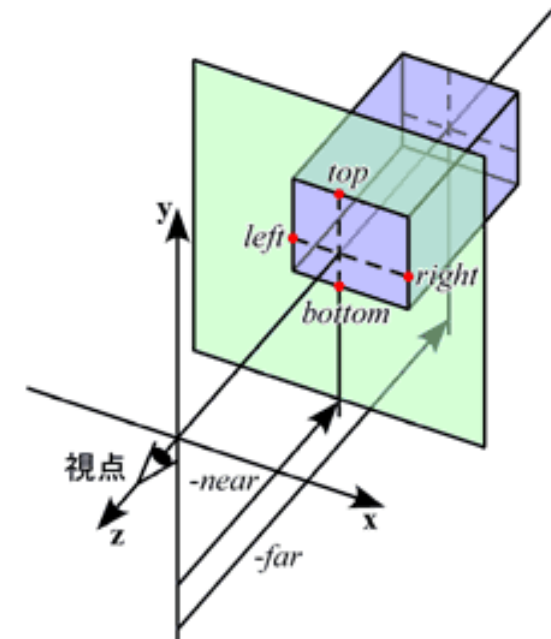
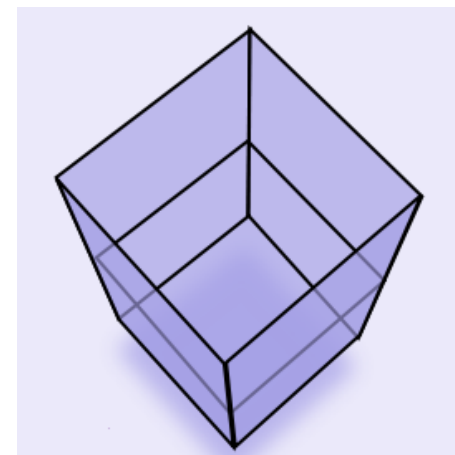
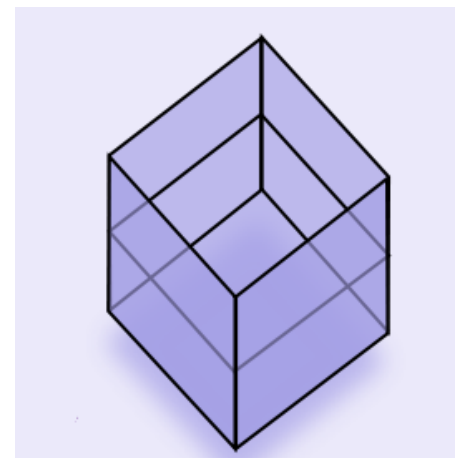
X軸で回転

X軸で回転

図形を描画

# シーンの構築.カメラ.投影方法

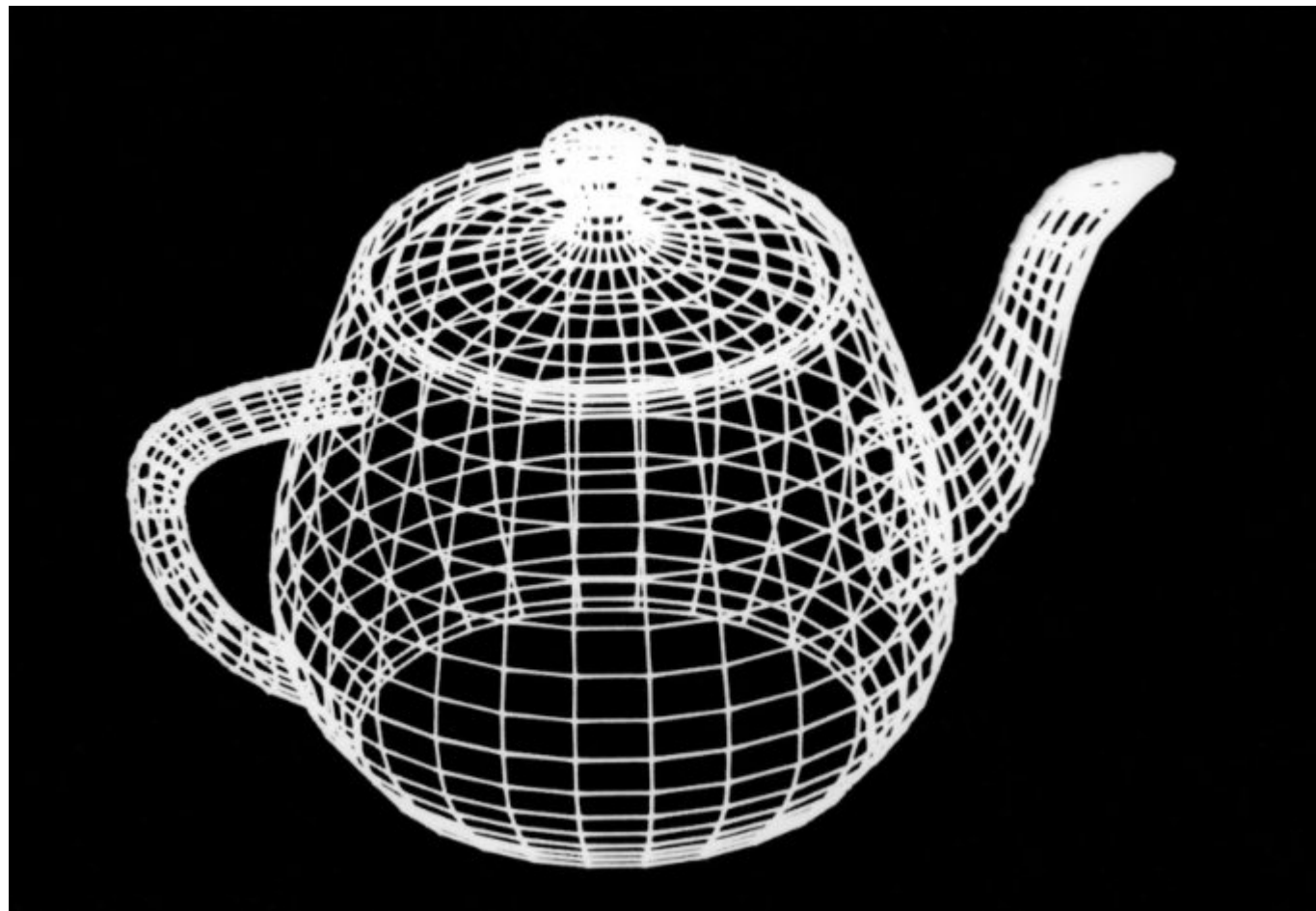
- 風景をスクリーンへ投影するのも座標変換
  - **投影行列**によって指定可能
- 平行投影
  - 寸法や平行線が歪まない図面的表示
  - `glOrtho` 関数で行列生成
- 透視投影
  - 遠近法を加味したリアルな表示
  - `glFrustum` 関数で行列生成



[marina.sys.wakayama-u.ac.jp/~tokoi/?date=20090829](http://marina.sys.wakayama-u.ac.jp/~tokoi/?date=20090829)



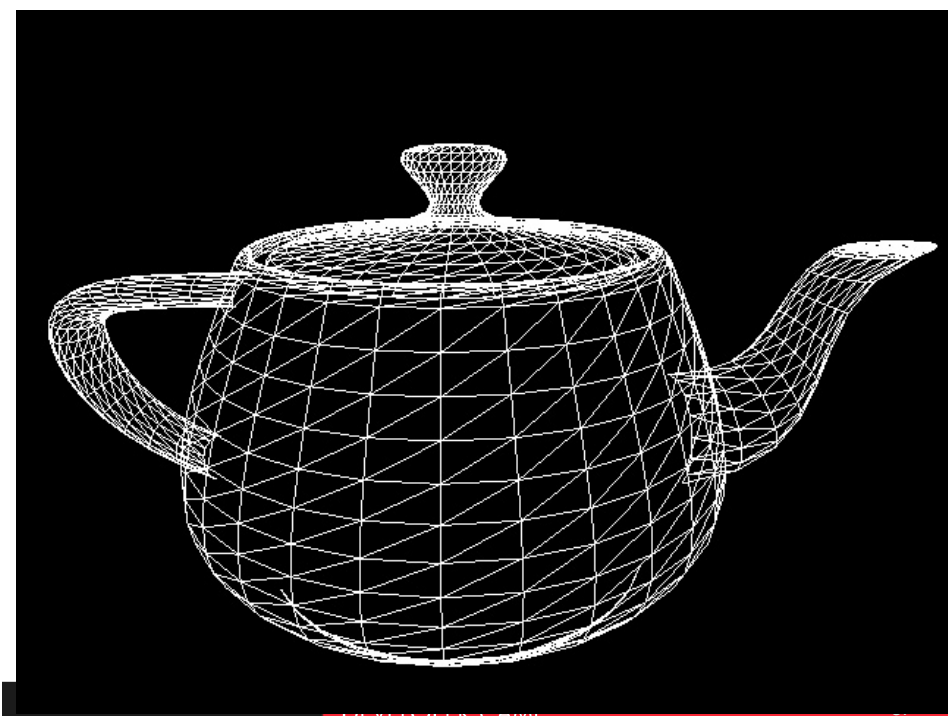
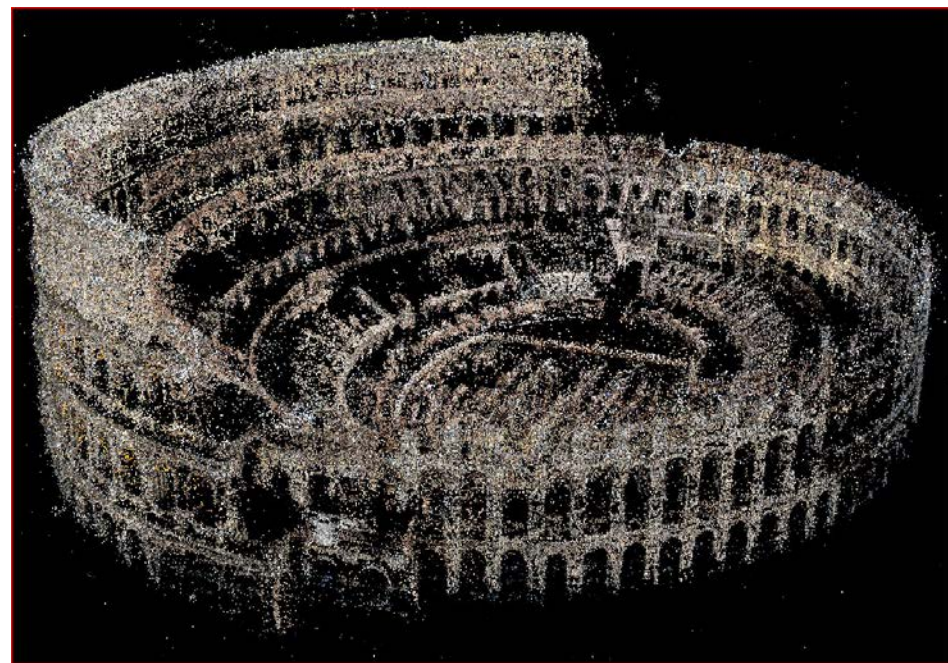
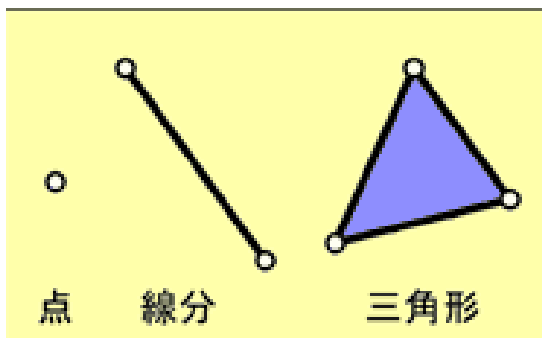
## ■ 物体の定義



**e**mbarcadero®  
DEVELOPER CAMP

# 物体の定義.構成要素

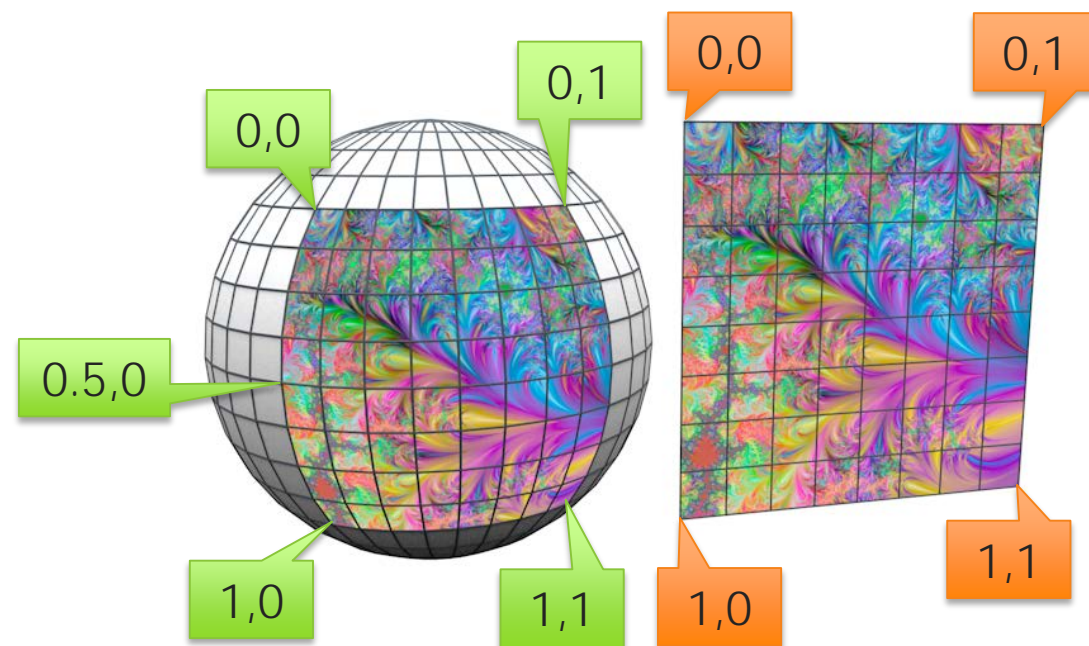
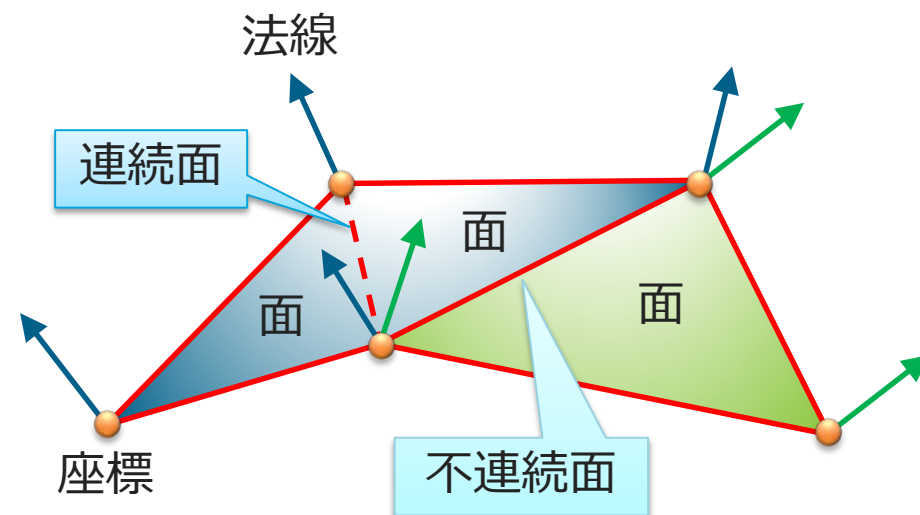
- 大量に描くことで様々な形状を表現  
点 と 線分 と 三角形 しか描けない
  - Point Cloud モデル
  - Wireframe モデル
  - Polygon Mesh モデル



# 物体の定義. 頂点の情報

## ■ 頂点の情報で物体は定義される

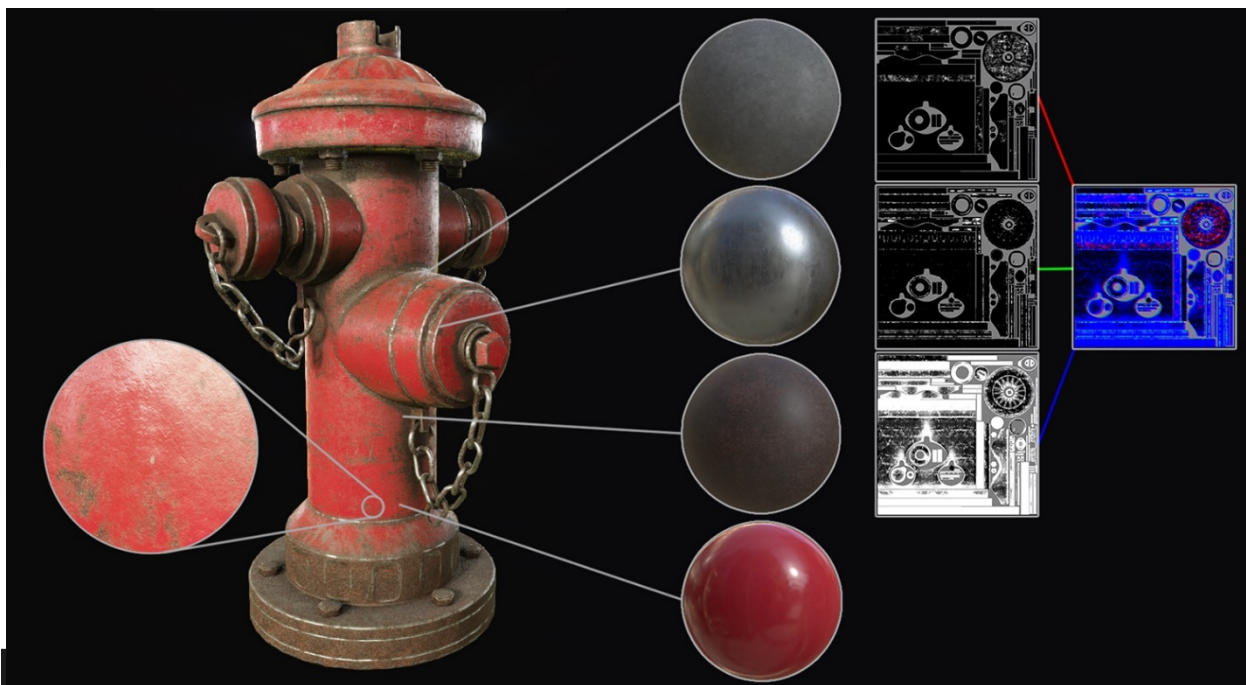
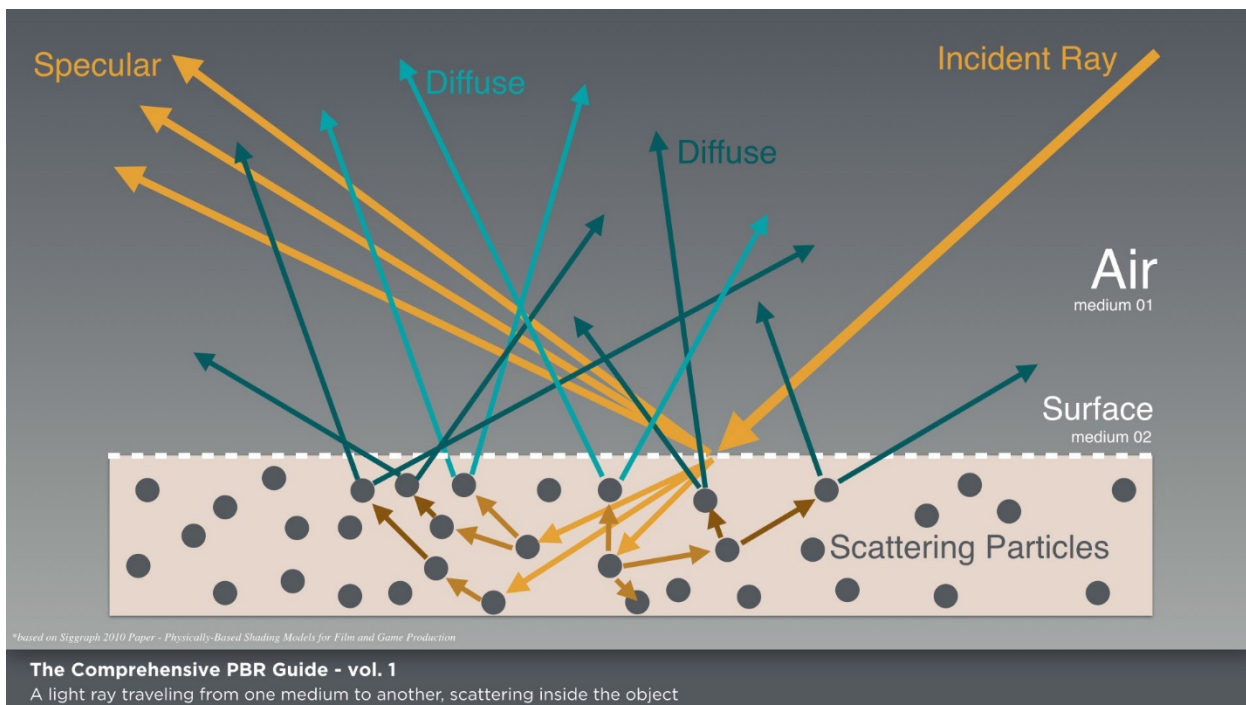
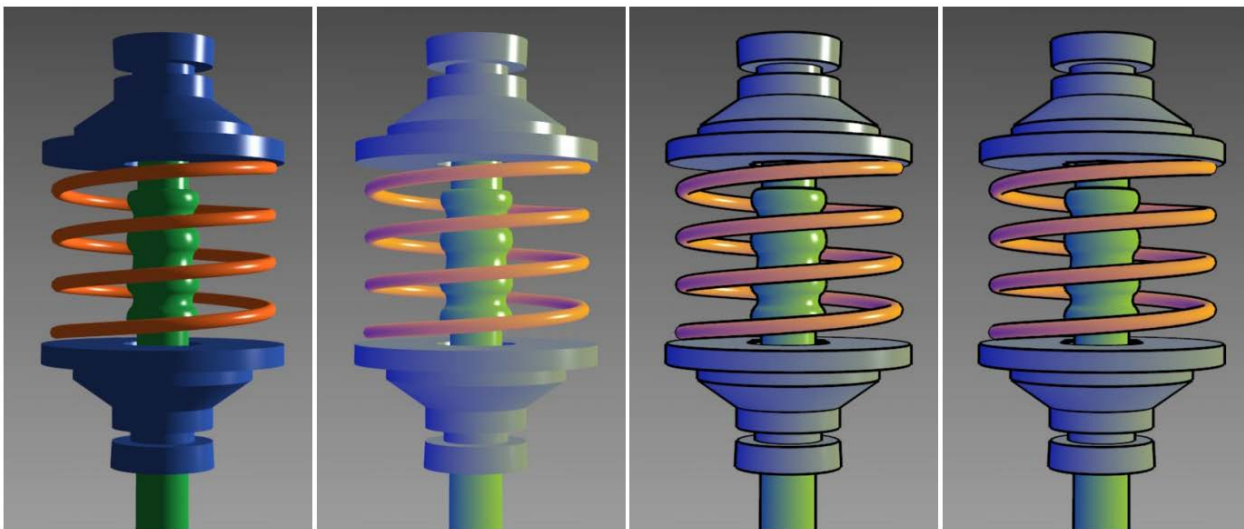
- 位置
  - 3次元座標
- 法線ベクトル
  - 面に垂直な3次元ベクトル
- 色
  - 基本的にテクスチャで代替される
- テクスチャ座標
  - 貼り付ける画像上の2次元座標





# 物体の定義.質感表現

- 図形を彩色することで様々な質感を表現
  - リアルに描く
    - 光学特性を正確に再現する
  - アニメに描く
    - 人間の視覚特性や感性を考慮する
    - 図面などに相応しい





- F M X 版 : [github.com/LUX0PHIA/OpenGL](https://github.com/LUX0PHIA/OpenGL)
- V C L 版 : [github.com/LUX0PHIA/OpenGL\\_VCL](https://github.com/LUX0PHIA/OpenGL_VCL)

## ■ OpenGL の歴史

📖 README.md

### 歴史に学ぶ OpenGL

OpenGL のバージョンを辿りながら実装していくことで、新しい API が追加された意匠を把握していきます。

- [OpenGL 1.0](#)
- [OpenGL 1.1](#)
- [OpenGL 1.5](#)
- [OpenGL 2.1](#)
- [OpenGL 3.0](#)



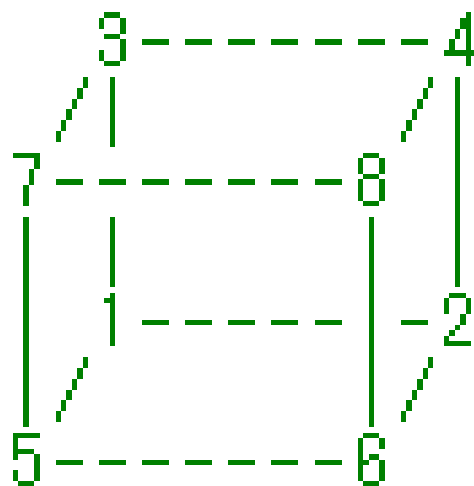
ここ見て！

### OpenGL

FMX : FireMonkey フレームワークのコンポーネントとして [OpenGL](#) の描画領域を埋

# OpenGL 1.0 (1992)

- glBegin ~ glEnd 関数で括る
- 頂点の属性毎に異なる関数
  - glVertex : 位置
  - glColor : 色
  - glNormal : 法線
  - glTexCoord : テクスチャ座標



```
Ps :array [ 1..8 ] of TSingle3D
    = (
        { X:-1; Y:-1; Z:-1 }, { X:+1; Y:-1; Z:-1 },
        { X:-1; Y:+1; Z:-1 }, { X:+1; Y:+1; Z:-1 },
        { X:-1; Y:-1; Z:+1 }, { X:+1; Y:-1; Z:+1 },
        { X:-1; Y:+1; Z:+1 }, { X:+1; Y:+1; Z:+1 } );

Cs :array [ 1..8 ] of TAlphaColorF
    = (
        { R:0; G:0; B:0; A:1 }, { R:1; G:0; B:0; A:1 },
        { R:0; G:1; B:0; A:1 }, { R:1; G:1; B:0; A:1 },
        { R:0; G:0; B:1; A:1 }, { R:1; G:0; B:1; A:1 },
        { R:0; G:1; B:1; A:1 }, { R:1; G:1; B:1; A:1 } );

Es :array [ 1..6, 1..4 ] of Cardinal
    = (
        { 1, 5, 7, 3 }, { 8, 6, 2, 4 },
        { 1, 2, 6, 5 }, { 8, 4, 3, 7 },
        { 1, 3, 4, 2 }, { 8, 7, 5, 6 } );
```

```
glBegin( GL_QUADS );
```

```
for N := 1 to 6 do
begin
    for K := 1 to 4 do
    begin
        E := Es[ N, K ];

        with Cs[ E ] do glColor3f( R, G, B );
        with Ps[ E ] do glVertex3f( X, Y, Z );
    end;
end;
```

1 頂点ずつ送る

```
glEnd;
```

一気に描画

# OpenGL 1.1 (1997)

## ■ 頂点配列

- 属性毎の配列を一気に転送可能

```
Ps :array [ 0..8-1 ] of TSingle3D
= (
  { X:-1; Y:-1; Z:-1 }, { X:+1; Y:-1; Z:-1 },
  { X:-1; Y:+1; Z:-1 }, { X:+1; Y:+1; Z:-1 },
  { X:-1; Y:-1; Z:+1 }, { X:+1; Y:-1; Z:+1 },
  { X:-1; Y:+1; Z:+1 }, { X:+1; Y:+1; Z:+1 } );

Cs :array [ 0..8-1 ] of TAlphaColorF
= (
  { R:0; G:0; B:0; A:1 }, { R:1; G:0; B:0; A:1 },
  { R:0; G:1; B:0; A:1 }, { R:1; G:1; B:0; A:1 },
  { R:0; G:0; B:1; A:1 }, { R:1; G:0; B:1; A:1 },
  { R:0; G:1; B:1; A:1 }, { R:1; G:1; B:1; A:1 } );

Es :array [ 0..12-1, 0..3-1 ] of Cardinal
= (
  { 0, 4, 6 }, { 6, 2, 0 }, { 7, 5, 1 }, { 1, 3, 7 },
  { 0, 1, 5 }, { 5, 4, 0 }, { 7, 3, 2 }, { 2, 6, 7 },
  { 0, 2, 3 }, { 3, 1, 0 }, { 7, 6, 4 }, { 4, 5, 7 } );
```

位置配列を有効化

```
glEnableClientState( GL_VERTEX_ARRAY );
```

色配列を有効化

```
glEnableClientState( GL_COLOR_ARRAY );
```

```
glVertexPointer( 3, GL_FLOAT, 0, @Ps[ 0 ] );
```

位置配列を転送

```
glColorPointer ( 4, GL_FLOAT, 0, @Cs[ 0 ] );
```

色配列を転送

インデックス配列で描画

```
glDrawElements( GL_TRIANGLES, 3{Poin} * 12{Face}, GL_UNSIGNED_INT, @Es[ 0, 0 ] );
```

```
glDisableClientState( GL_VERTEX_ARRAY );
```

色配列を無効化

```
glDisableClientState( GL_COLOR_ARRAY );
```

位置配列を無効化

データ型

更新頻度

# OpenGL 1.5 (2003)

```
_VerterP := TGLVerterS<TSingle3D> .Create( GL_STATIC_DRAW );  
_VerterC := TGLVerterS<TAlphaColorF>.Create( GL_STATIC_DRAW );  
_Elemer := TGLElemerFace32 .Create( GL_STATIC_DRAW );
```

- V B O : Verttex Buffer Object
  - 予めG P U側へ頂点配列を転送し保持可能
    - 描画速度の大幅な向上

頂点配列を転送

```
_VerterP.Import( Ps );  
_VerterC.Import( Cs );  
_Elemer .Import( Es );
```



```
glEnableClientState( GL_VERTEX_ARRAY );  
glEnableClientState( GL_COLOR_ARRAY );
```

```
with _VerterP do  
begin  
Bind;  
glVertexPointer( 3, GL_FLOAT, 0, nil );  
Unbind;  
end;
```

バイト

転送不要

利用登録

```
with _VerterC do  
begin  
Bind;  
glColorPointer( 4, GL_FLOAT, 0, nil );  
Unbind;  
end;
```

バイト

転送不要

利用登録

```
with _Elemer do  
begin  
Bind;  
glDrawElements( GL_TRIANGLES, 3{Poin} * 12{Face}, GL_UNSIGNED_INT, nil );  
Unbind;  
end;
```

バイト

転送不要

描画

```
glDisableClientState( GL_VERTEX_ARRAY );  
glDisableClientState( GL_COLOR_ARRAY );
```



# OpenGL 1.5. V B O

- 生成/使用方法は皆同じ
  - 種類フラグが異なるだけ

## V B Oの有効化

```
procedure TGLBuffer<_TYPE_>.Bind;  
begin  
    glBindBuffer( GetKind, _ID );  
end;  
  
procedure TGLBuffer<_TYPE_>.Unbind;  
begin  
    glBindBuffer( GetKind, 0 );  
end;
```

## V B Oの無効化

```
procedure TGLBuffer<_TYPE_>.Import( const Array_:array of _TYPE_ );  
begin  
    _Count := Length( Array_ );  
  
    Bind;  
  
    glBufferData( GetKind, SizeOf( Array_ ), @Array_[ 0 ], _Usage );  
  
    Unbind;  
end;
```

## 種類フラグ

```
constructor TGLBuffer<_TYPE_>.Create( const Usage_:GLenum );  
begin  
    inherited Create;  
  
    _Align := InitAlign;  
    _Strid := InitStrid;  
  
    glGenBuffers( 1, @_ID );  
  
    _Usage := Usage_;  
    _Count := 0;  
end;  
  
destructor TGLBuffer<_TYPE_>.Destroy;  
begin  
    glDeleteBuffers( 1, @_ID );  
  
    inherited;  
end;
```

## 生成

## 廃棄

## 頂点配列の転送

# OpenGL 2.1 (2006)

- プログラマブルシェーダーが実現
  - S O : シェーダーオブジェクト
    - 質感を計算するプログラム
  - P O : プログラムオブジェクト
    - S O をまとめる実行主体

```
with _Elemen do
begin
  Bind;
```

```
  with _Progra do
  begin
    Use;
```

POのバインド

```
    glDrawElements( GL_TRIANGLES, 3{Poin} * 12{Face}, GL_UNSIGNED_INT, nil );
```

```
    Unuse;
```

```
  end;
```

```
  Unbind;
```

```
end;
```

形状の描画

```
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );
```

```
with _VerterP do
begin
  Bind;
  glVertexPointer( 3, GL_FLOAT, 0, nil );
  Unbind;
end;
```

```
with _VerterC do
begin
  Bind;
  glColorPointer( 4, GL_FLOAT, 0, nil );
  Unbind;
end;
```

```
with _Elemen do
begin
  Bind;

  with _Progra do
  begin
    Use;

    glDrawElements( GL_TRIANGLES, 3{Poin} * 12{Face}, GL_UNSIGNED_INT, nil );

    Unuse;
  end;

  Unbind;
end;
```

```
glDisableClientState( GL_VERTEX_ARRAY );
glDisableClientState( GL_COLOR_ARRAY );
```

# OpenGL 2.1. **G L S L**

## ■ **G L S L** 言語でシェーダーを記述

- **V S** : バーテックスシェーダー
  - 頂点毎に呼ばれるイベント
- **F S** : フラグメントシェーダー
  - ピクセル毎に呼ばれるイベント

```
with _ShaderV.Source do
begin
    BeginUpdate;
    Add( '#version 120' );
    Add( 'void main()' );
    Add( '{' );
    Add( '    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;' );
    Add( '    gl_FrontColor = gl_Color;' );
    Add( '}' );
    EndUpdate;
end;
```

**V S のソース**

```
with _Program do
begin
    Attach( _ShaderV );
    Attach( _ShaderF );
    Link;
end;
```

**V S の登録**

**F S の登録**

**リンク**

```
with _ShaderF.Source do
begin
    BeginUpdate;
    Add( '#version 120' );
    Add( 'void main()' );
    Add( '{' );
    Add( '    gl_FragColor = gl_Color;' );
    Add( '}' );
    EndUpdate;
end;
```

**F S のソース**

# OpenGL 3.0 (2008)

- V A O : Vertex Array Object
  - V B Oの**有効/無効状態**をまとめて登録しておく辞書
    - 描画の度に複数のV B Oをバインドする手間が省ける
- Core Profile 利用時はV A Oの利用が必須

```
with _Progra do
begin
    Use;
    with _Varray do
    begin
        Bind;
        glDrawElements( GL_TRIANGLES, 3{Poin} * 12{Face}, GL_UNSIGNED_INT, nil );
        Unbind;
    end;
    Unuse;
end;
```

POを有効化

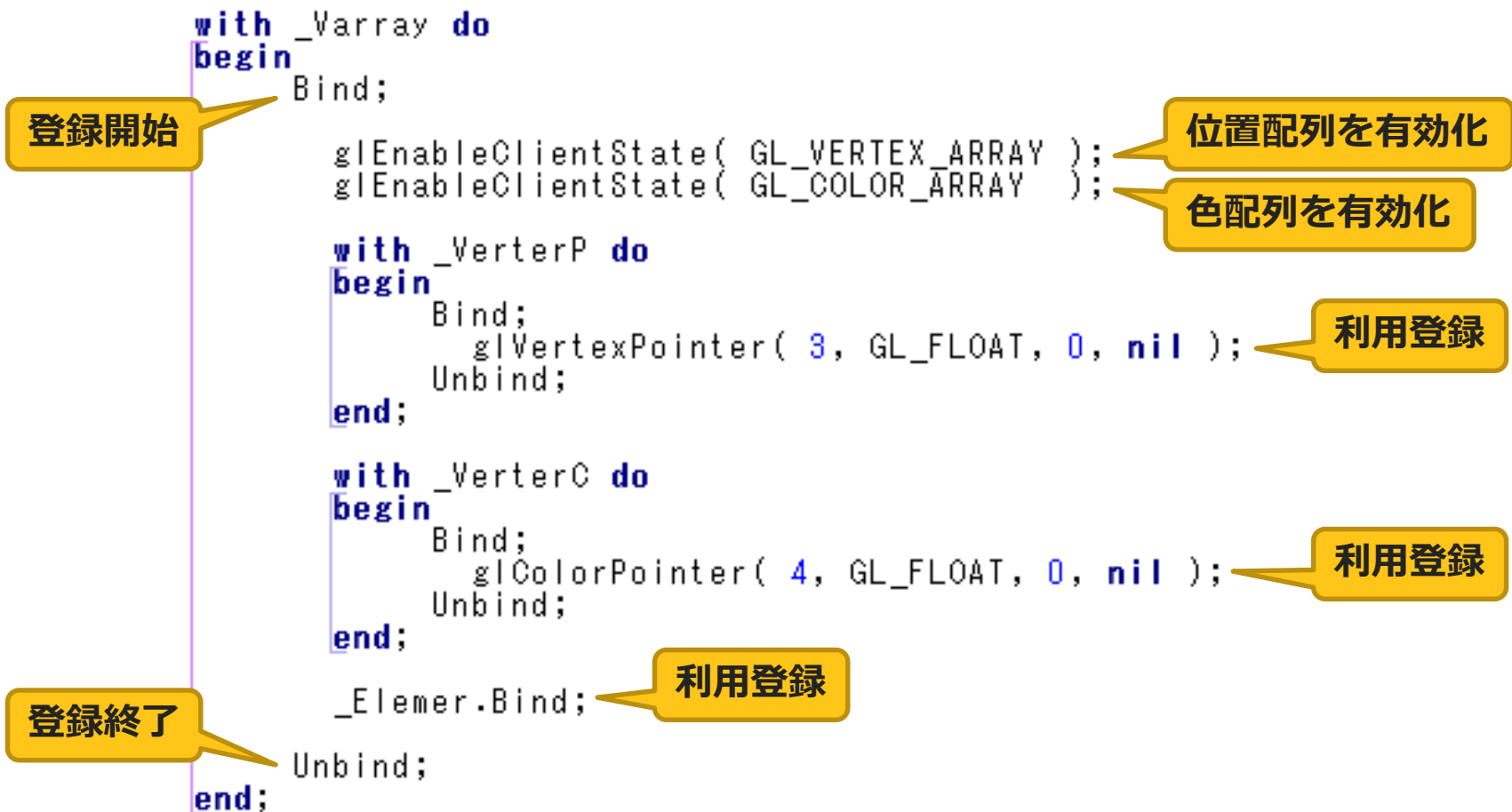
V A Oを有効化

形状の描画



# OpenGL 3.0. **V A O**

## ■ 予め利用する V B O を登録する



# Post OpenGL 3.0

- OpenGL 3.0
  - シェーダで代替できる古い関数は非推奨にするよ
    - [www.khronos.org/opengl/wiki/Legacy\\_OpenGL](http://www.khronos.org/opengl/wiki/Legacy_OpenGL)
- OpenGL 3.1
  - 非推奨な関数はすべて削除（拡張機能扱い）したよ
- OpenGL 3.2
  - 互換モードを選べば使えるようにしたよ
    - Core Profile：モダン関数のみモード
    - Compatible Profile：レガシー関数含有モード



～ ～ ～ ～ ～
- OpenGL 4.3
  - モダンな機能がほぼ出そろう
    - 今から勉強するならここから！

## Contents [hide]

- 1 Overview
- 2 Summary of version changes
  - 2.1 OpenGL 4.5 (2014)
  - 2.2 OpenGL 4.4 (2013)
  - 2.3 OpenGL 4.3 (2012)
  - 2.4 OpenGL 4.2 (2011)
  - 2.5 OpenGL 4.1 (2010)
  - 2.6 OpenGL 4.0 (2010)
  - 2.7 OpenGL 3.3 (2010)
  - 2.8 OpenGL 3.2 (2009)
  - 2.9 OpenGL 3.1 (2009)
  - 2.10 OpenGL 3.0 (2008)
    - 2.10.1 Deprecation Model
  - 2.11 OpenGL 2.1 (2006)
  - 2.12 OpenGL 2.0 (2004)
  - 2.13 OpenGL 1.5 (2003)
  - 2.14 OpenGL 1.4 (2002)
  - 2.15 OpenGL 1.3 (2001)
  - 2.16 OpenGL 1.2.1 (1998)
  - 2.17 OpenGL 1.2 (1998)
    - 2.17.1 Imaging subset (optional)
  - 2.18 OpenGL 1.1 (1997)
  - 2.19 OpenGL 1.0 (1992)

# OpenGL Extensions Viewer

- [realtech-vr.com/admin/glview](http://realtech-vr.com/admin/glview)
- OpenGL のバージョンを調べられるアプリ
  - 各種 A P I の実装率も分かる
- マルチプラットフォーム
  - Windows, macOS, Android, iOS



### Core features

- 3.0 (100 % - 23/23)
- 3.1 (100 % - 8/8)
- 3.2 (100 % - 10/10)
- 3.3 (100 % - 10/10)
- 4.0 (100 % - 14/14)
- 4.1 (100 % - 7/7)
- 4.2 (100 % - 13/13)
- 4.3 (100 % - 23/23)
- 4.4 (100 % - 10/10)
- 4.5 (27 % - 3/11)

### Supported

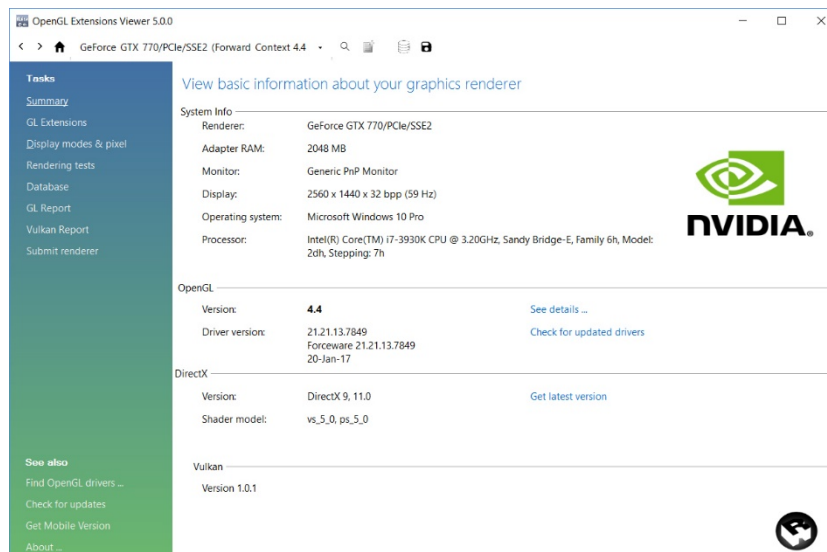
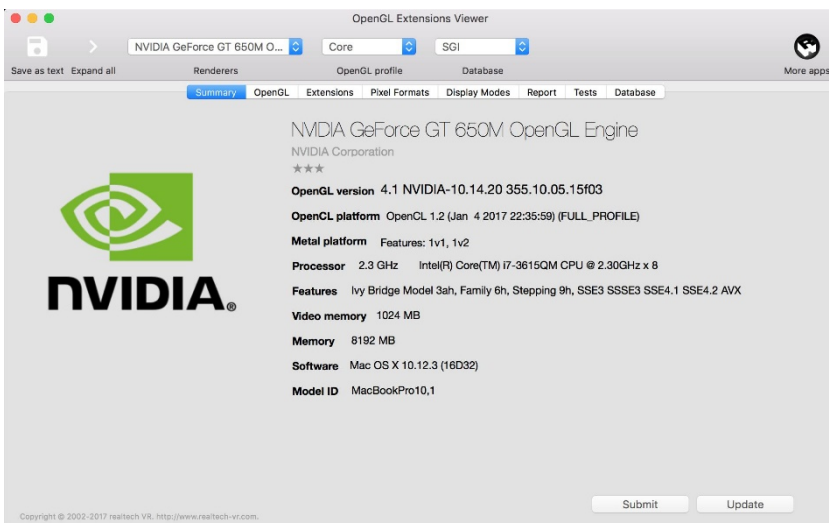
- GL\_ARB\_derivative\_control
- GL\_ARB\_texture\_barrier
- GL\_ARB\_clip\_control

### Unsupported

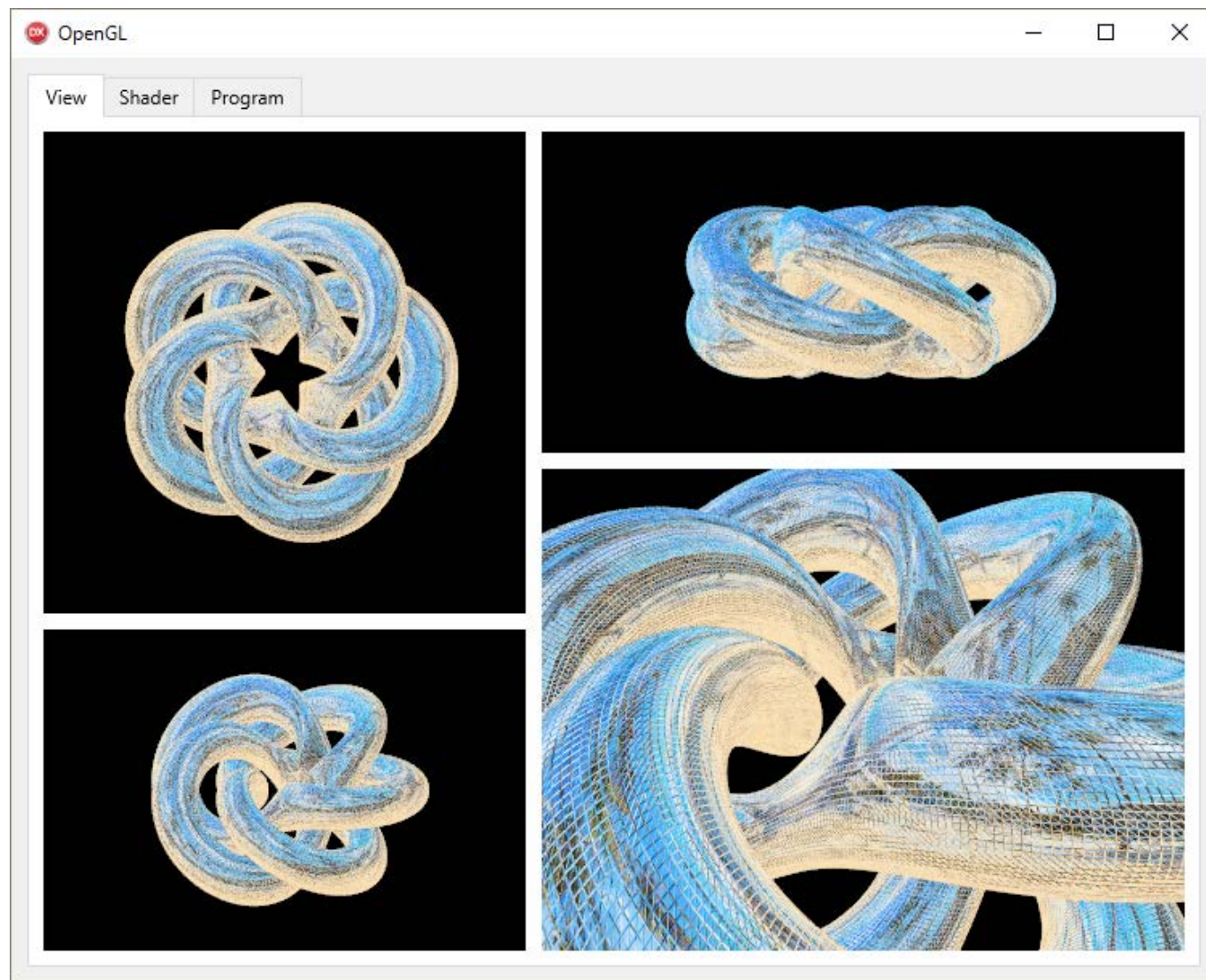
- Shading language version: 4.50
- GL\_ARB\_get\_texture\_sub\_image
- GL\_KHR\_robustness
- GL\_ARB\_conditional\_render\_inverted
- GL\_ARB\_cull\_distance
- GL\_ARB\_shader\_texture\_image\_samples
- GL\_ARB\_ES3\_1\_compatibility
- GL\_ARB\_direct\_state\_access

### ARB 2015 (15 % - 2/13)

### ARB 2016 (0 % - 0/1)



## ■モダンな実装





## モダンな実装.座標変換

- 従来の行列系関数はすべて使えない
  - すべて自前で計算する
- 頂点の座標変換はV Sで代替する

```
GLViewer1.OnPaint := procedure
begin
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity;
    glOrtho( -2, +2, -2, +2, 1, 1 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity;
    glTranslatef( 0, 0, -5 );
    glRotatef( +90, 1, 0, 0 );
    glRotatef( _Angle, 0, 1, 0 );
    DrawShaper;
end;
```

```
gl_Position = _ViewerScal * _Camera.Proj * inverse( _Camera.Pose ) * _Shaper.Pose * _VertexPos;
```

ウィンドウ比率

カメラの投影行列

カメラの姿勢行列

物体の姿勢行列

頂点座標

逆行列

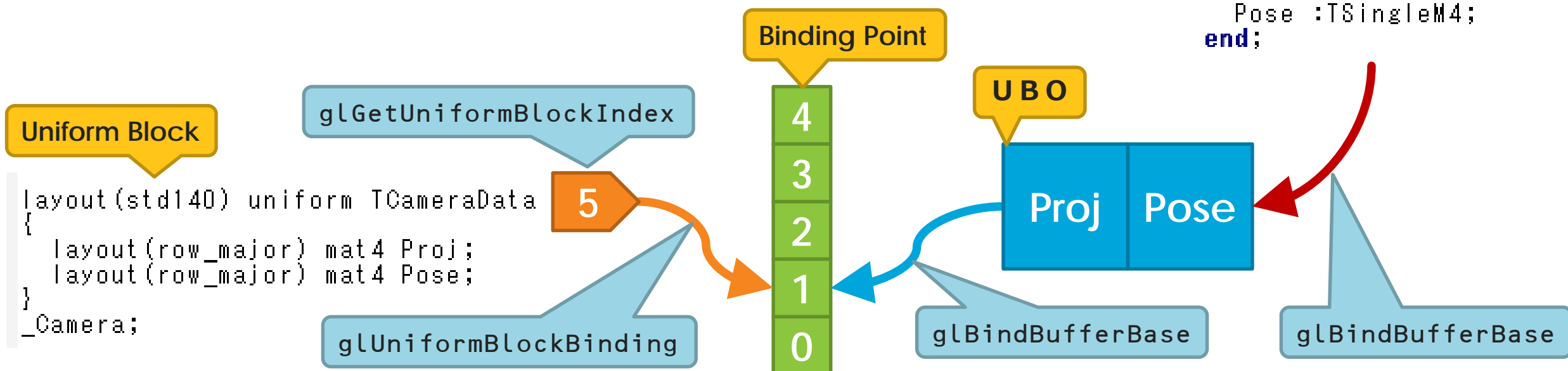
スクリーン座標

- 物体の姿勢行列は変数としてシェーダに読み込ませる

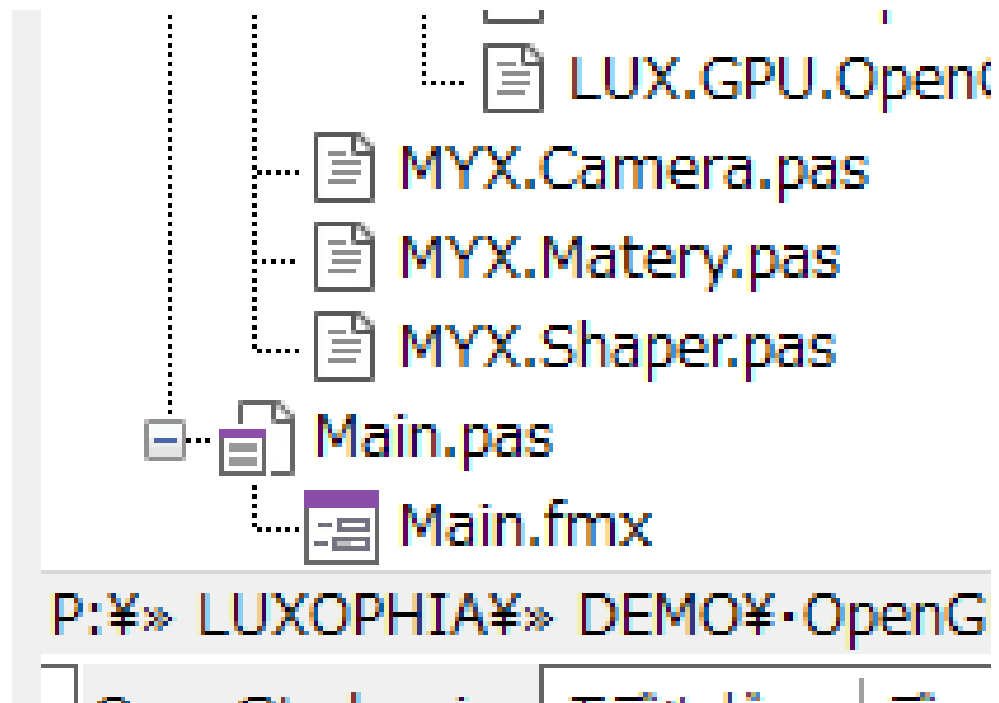
# モダンな実装.UBO

## ■ UBO : Uniform Buffer Object

- すべてのシェーダから共通の定数として参照できる配列
- 生成/使用方法はVBOと同じ
  - 種類フラグに GL\_UNIFORM\_BUFFER を指定
- 任意の構造体をシェーダへ渡すことが可能
  - メンバ変数の型と順番を一致させる必要あり



## ■ ミニマムライブラリ



# ミニマムライブラリ.クラス構造

## ■ 3つのクラスに整理

- カメラ
  - TMyCamera
- 材質（質感）
  - TMyMatery
- 物体（形状）
  - TMyShaper

## ■ 描画は3行

```
GLViewer1.OnPaint := procedure
begin
    _Camera1.Use;
    _Matery .Use;
    _Shaper .Draw;
end;
```

カメラを有効化

材質を有効化

物体を描画

投影行列を設定

```
with C do
begin
    Proj := TSingleM4.ProjPers( -_N/2, +_N/2, -_N/2, +_N/2, _N, _F );
    Pose := TSingleM4.RotateX( DegToRad( -45 ) )
            * TSingleM4.Translate( 0, 0, +2 );
end;
_Camera4.Data := C;
```

カメラの  
姿勢行列を設定

カメラのUBOへ入力

シェーダソースのロード

```
with _Matery do
begin
    ShaderV.Source.LoadFromFile( '..\..\_DATA\ShaderV.glsl' );
    ShaderF.Source.LoadFromFile( '..\..\_DATA\ShaderF.glsl' );
    Imager.LoadFromFile( '..\..\_DATA\Spherical_1024x1024.png' );
end;
```

テクスチャのロード

```
with _Shaper do
begin
    LoadFormFunc( BraidedTorus, 1300, 100 );
    with S do
    begin
        Pose := TSingleM4.Identify;
    end;
    Data := S;
end;
```

ポリゴン生成

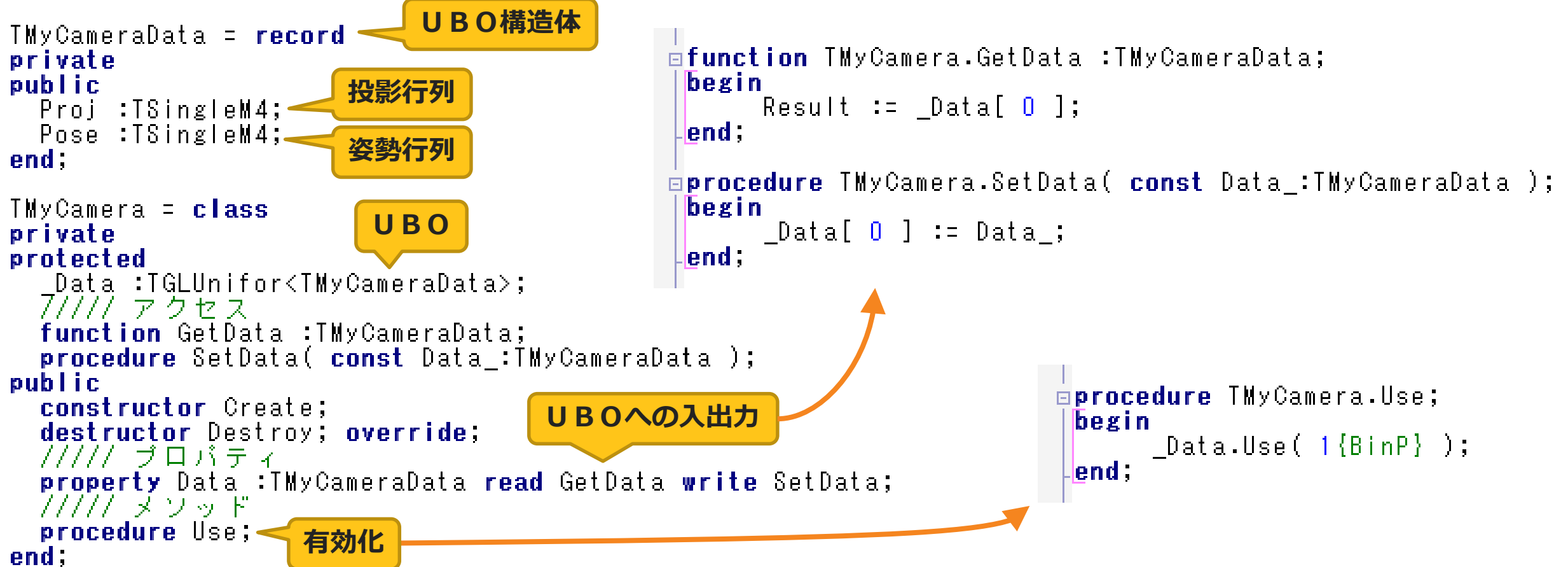
物体の姿勢行列を設定

物体のUBOへ入力



# ミニマムライブラリ.TMyCamera

- 投影行列と姿勢行列をU B Oで保持



# ミニマムライブラリ.TMyMaterly

- シェーダー関連オブジェクトを内包
- テクスチャオブジェクトを保持

```
TMyMaterly = class
private
protected
    _ShaderV :TGLShaderV;
    _ShaderF :TGLShaderF;
    _Engine :TGLEngine;
    _Sample :TGLSample;
    _Imager :TGLImager2D_RGBA;
public
    constructor Create;
    destructor Destroy; override;
    ///// プロパティ
    property ShaderV :TGLShaderV read _ShaderV;
    property ShaderF :TGLShaderF read _ShaderF;
    property Engine :TGLEngine read _Engine;
    property Sample :TGLSample read _Sample;
    property Imager :TGLImager2D_RGBA read _Imager;
    ///// メソッド
    procedure Use;
end;
```

頂点SO

断片SO

PO

サンプラーオブジェクト

テクスチャオブジェクト

有効化

```
procedure TMyMaterly.Use;
begin
    _Engine.Use;
    _Sample.Use( 0{BinP} );
    _Imager.Use( 0{BinP} );
end;
```

PO有効化

テクスチャ有効化

# ミニマムライブラリ.TMyMaterly.Create

## ■ 変数⇒B Pの紐付けを自動化

- Verters : V B O用
  - 新しいA P IではV B Oも  
U B OのようにB P経由で接続可能
- Uniforms : U B O用
- Imagers : テクスチャ用
  - U B Oのようにテクスチャユニット  
という単位で接続可能
- Framers : フレーム用
  - ウィンドウへだけでなく  
内部的に確保した複数の画像へ  
レンダリングすることも可能

```
constructor TMyMaterly.Create;  
begin  
    inherited;  
  
    _ShaderV := TGLShaderV      .Create;  
    _ShaderF := TGLShaderF      .Create;  
    _Engine  := TGLEngine        .Create;  
    _Sample  := TGLSample        .Create;  
    _Imager  := TGLImager2D_RGBA.Create;  
  
    with _Engine do  
    begin  
        Attach( _ShaderV{Shad} );  
        Attach( _ShaderF{Shad} );  
  
        with Verters do  
        begin  
            Add( 0{BinP}, '_VerterPos', {Name}, 3{EleN}, GL_FLOAT{EleT} );  
            Add( 1{BinP}, '_VerterNor', {Name}, 3{EleN}, GL_FLOAT{EleT} );  
            Add( 2{BinP}, '_VerterTex', {Name}, 2{EleN}, GL_FLOAT{EleT} );  
        end;  
  
        with Uniforms do  
        begin  
            Add( 0{BinP}, 'TViewerScal', {Name} );  
            Add( 1{BinP}, 'TCameraData', {Name} );  
            Add( 2{BinP}, 'TShaperData', {Name} );  
        end;  
  
        with Imagers do  
        begin  
            Add( 0{BinP}, '_Imager', {Name} );  
        end;  
  
        with Framers do  
        begin  
            Add( 0{BinP}, '_FramerCol', {Name} );  
        end;  
    end;  
end;
```

POへのSO登録

変数名⇒B P  
の登録

# ミニマムライブラリ. TMyShaperBase

- TMyShaper の基底クラス
  - 姿勢行列をU B Oで保持

```
TMyShaperData = record
private
public
    Pose :TSingleM4;
end;
```

U B O構造体

姿勢行列

```
TMyShaperBase = class
private
protected
    _Data :TGLUnifor<TMyShaperData>;
    ///// アクセス
    function GetData :TMyShaperData;
    procedure SetData( const Data_:TMyShaperData );
public
    constructor Create;
    destructor Destroy; override;
    ///// プロパティ
    property Data :TMyShaperData read GetData write SetData;
    ///// メソッド
    procedure Draw; virtual;
end;
```

U B O

U B Oへの入出力

描画

# ミニマムライブラリ. TMyShaper

## ■ 各種 V B O を内包

```
TMyShaper = class( TMyShaperBase )
private
protected
    _PosBuf :TGLVertex<TSingle3D>;
    _NorBuf :TGLVertex<TSingle3D>;
    _TexBuf :TGLVertex<TSingle2D>;
    _EleBuf :TGLElementFace32;
public
    constructor Create;
    destructor Destroy; override;
    ///// プロパティ
    property PosBuf :TGLVertex<TSingle3D> read _PosBuf;
    property NorBuf :TGLVertex<TSingle3D> read _NorBuf;
    property TexBuf :TGLVertex<TSingle2D> read _TexBuf;
    property EleBuf :TGLElementFace32 read _EleBuf;
    ///// メソッド
    procedure Draw; override;
    procedure LoadFormFunc( const Func_:TConstFunc<TdSingle2D,TdSingle3D>; const DivX_,DivY_:Integer );
end;
```

位置 V B O

法線 V B O

テクスチャ座標 V B O

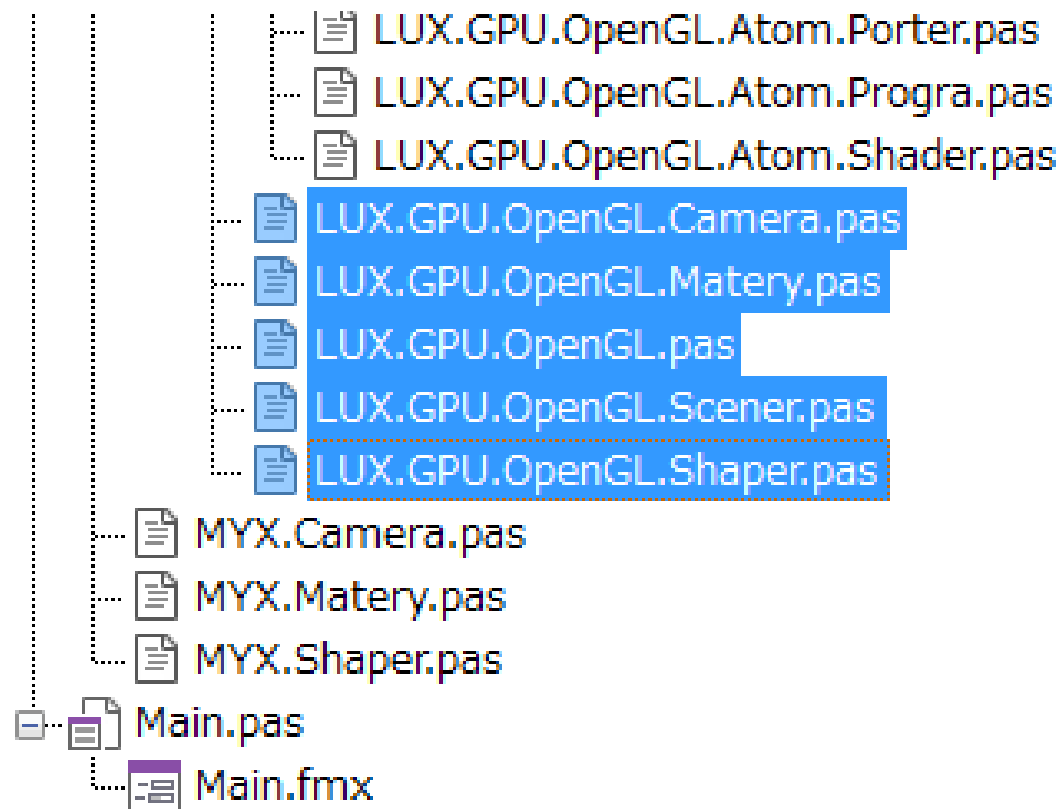
インデックス V B O

描画

形状のロード

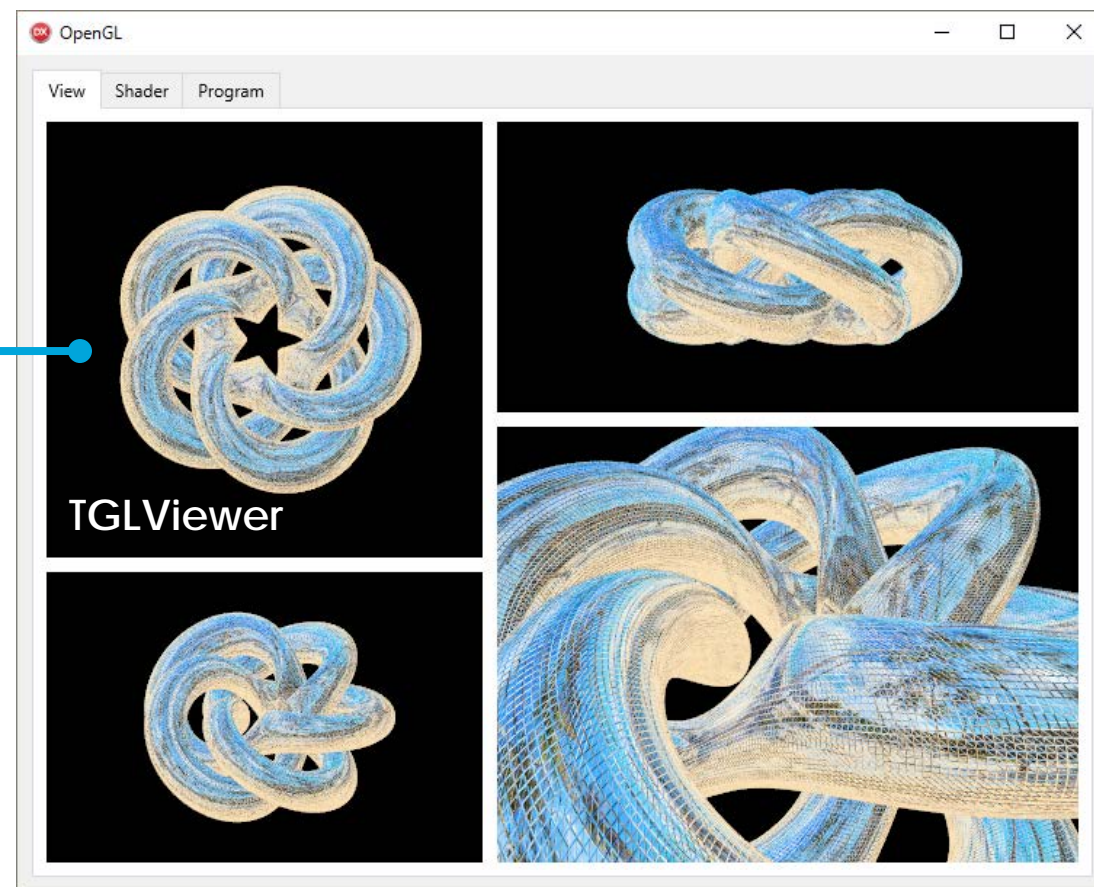
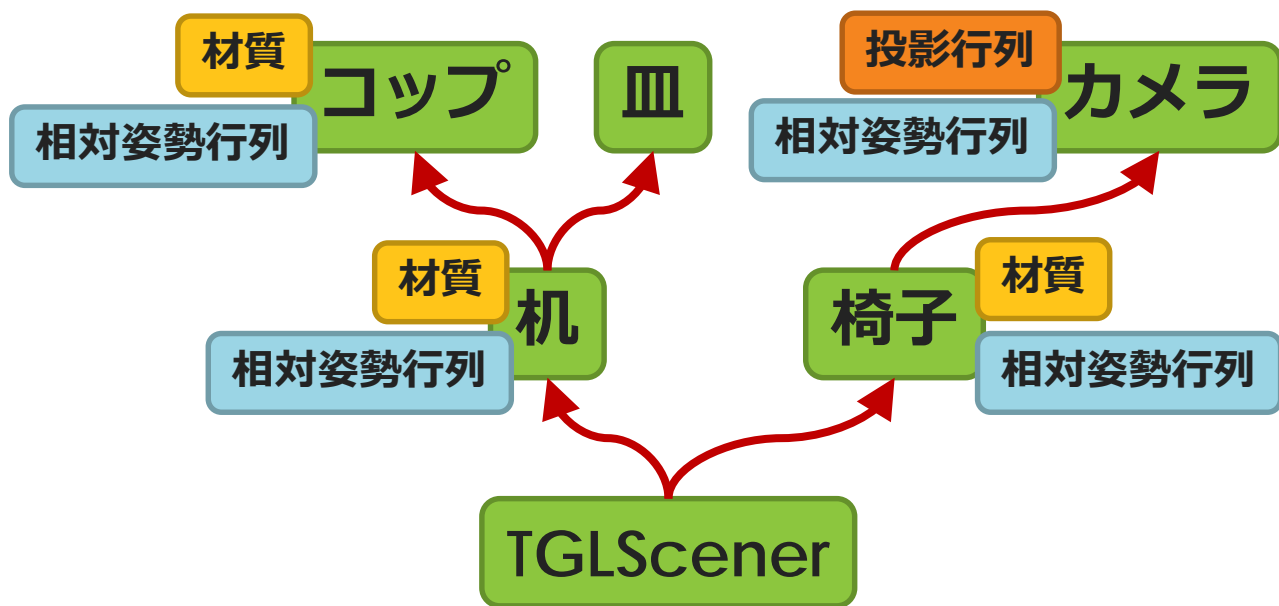


# ■マキシマムライブラリ



# マキシマムライブラリ

- ツリー構造でシーン構築できるノードベースライブラリ
- ジオメトリシェーダに対応
- TGLViewer をカメラに紐付け
  - Repaint メソッドによる直感的な描画



# THANKS!

[www.embarcadero.com/jp](http://www.embarcadero.com/jp)

第34回 エンバカデロ・デベロッパーキャンプ

