【A4】Delphi/C++**テクニカルセッション**

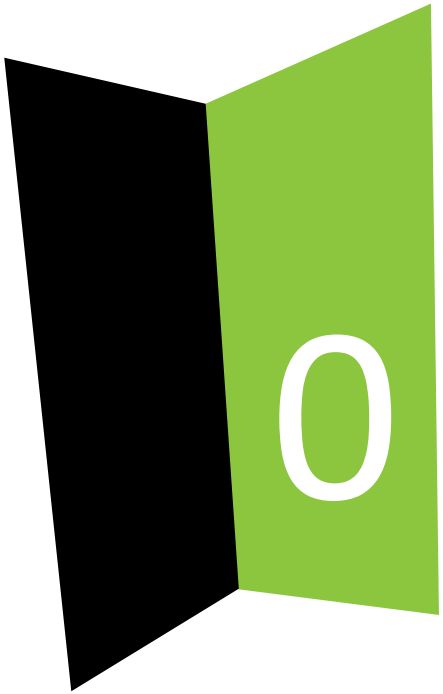
FireMonkey の仕組み

慶應義塾大学 藤代研究室 & 玉泉山 安国院
中山 雅紀

サンプルプログラム

- 講演内で紹介するプログラムはダウンロード可能
  - Delphi XE7 のプロジェクト

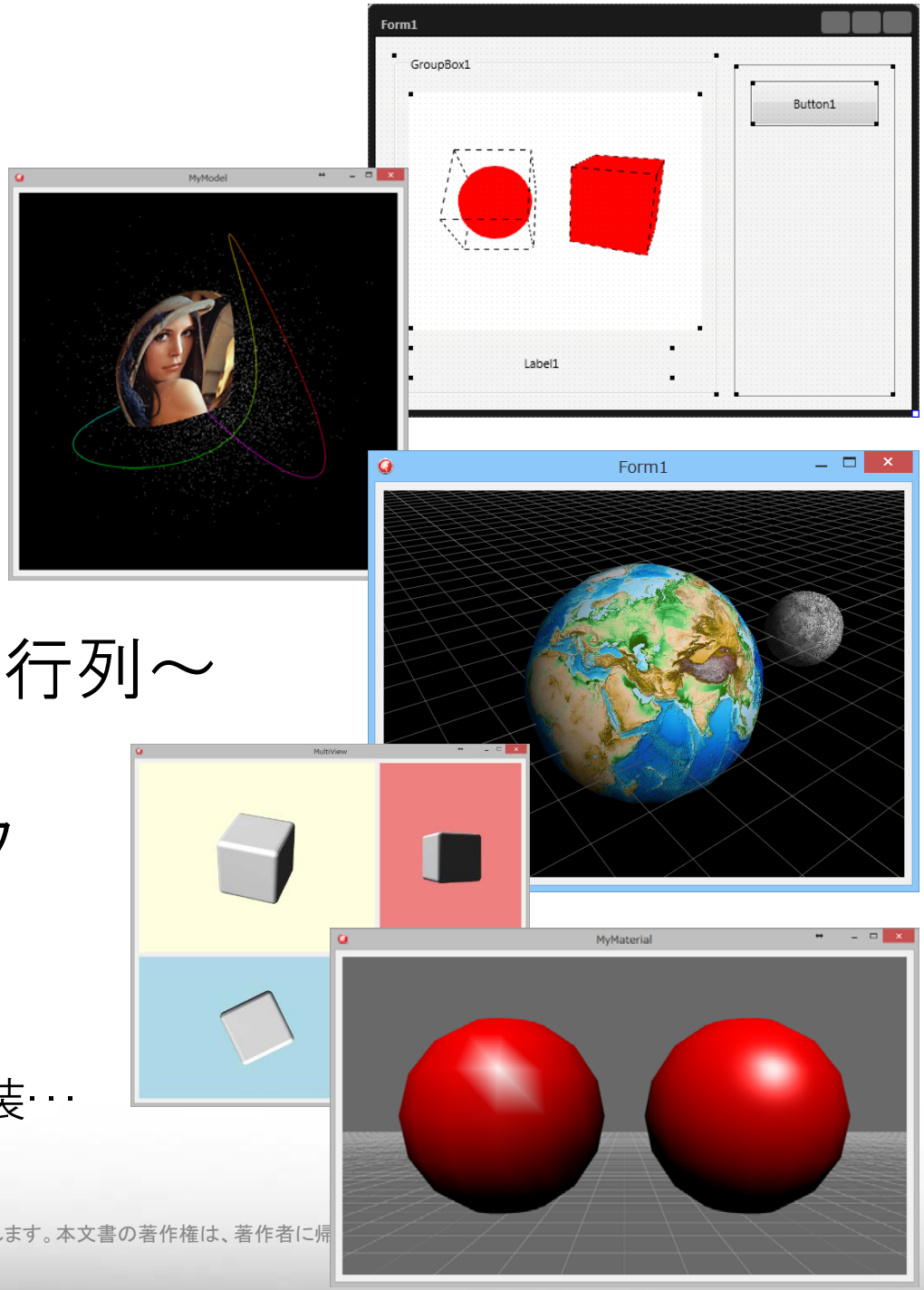
http://www.luxidea.net/XAUYEZPG

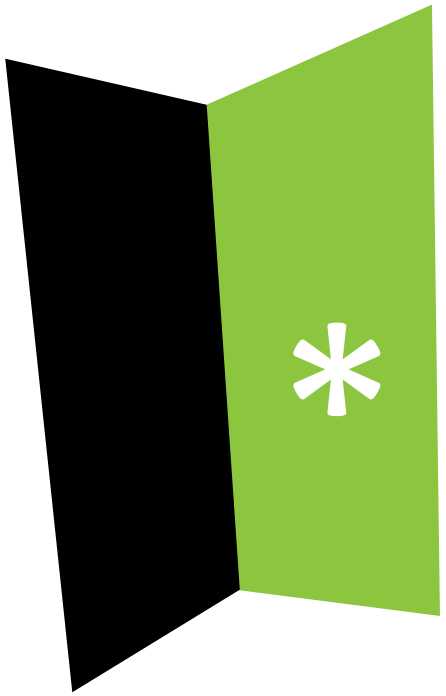


アジェンダ

アジェンダ

- FireMonkey とは？
  - シーンの作り方･･･
- 3Dモデルの作り方
  - 点･線･面 のモデル･･･
- 座標変換～ベクトルと行列～
  - 太陽系･･･
- イレギュラーテクニック
  - 複数視点･･･
- マテリアルの作り方
  - HLSL言語 によるシェーダ実装･･･





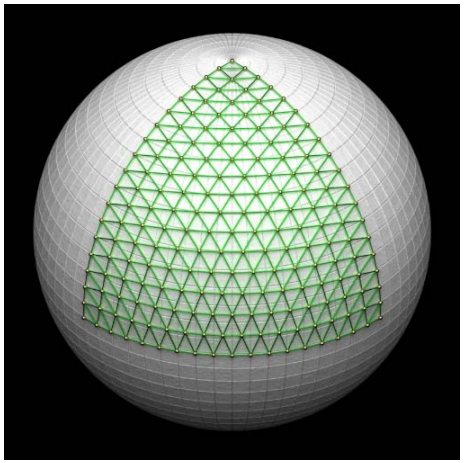
自己紹介

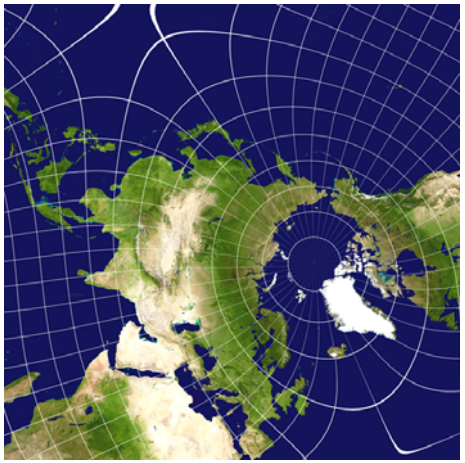
自己紹介. 身分

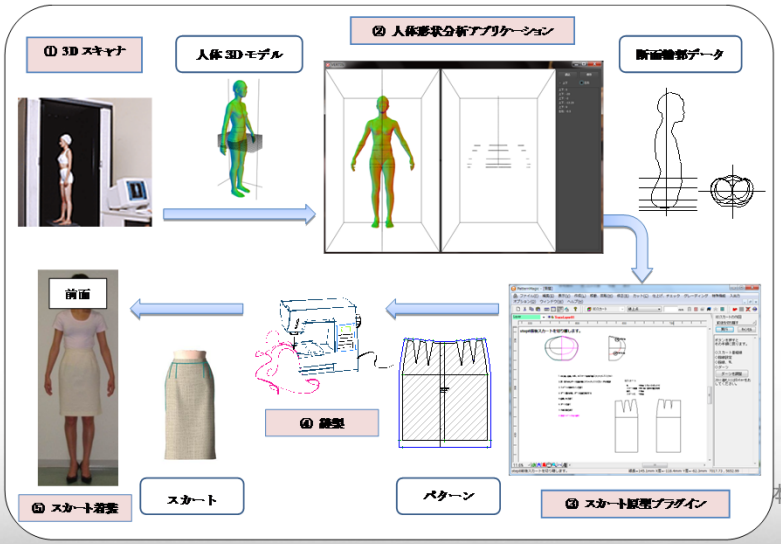
- 慶應義塾大学 藤代研究室
  - 社会人研究生
    - コンピュータ･グラフィックスの研究
- 和洋女子大学 山本研究室
  - 講師＆共同研究員
    - アパレル設計の授業＆研究
- 日蓮宗 玉泉山 安国院
  - 住職
    - 心の平穏の研究

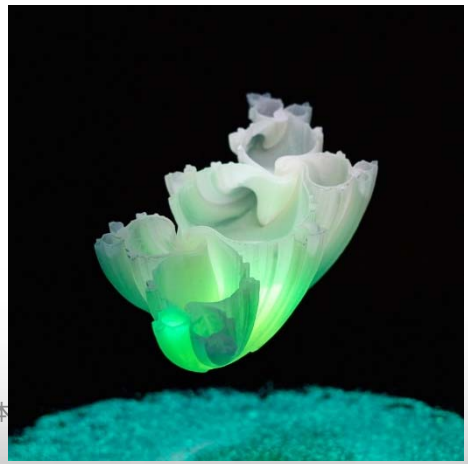
自己紹介. 専門

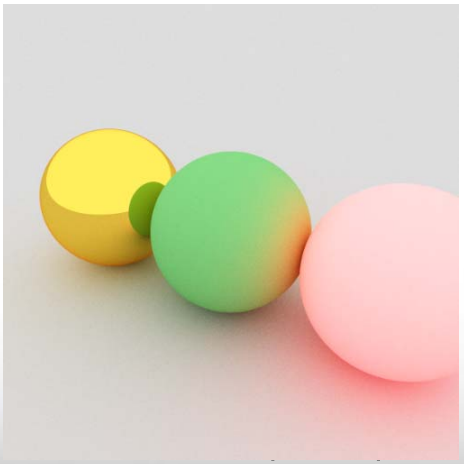
- 球面上の信号処理
- 写実的なレンダリング
- 立体視映像
- 3Dアパレル設計
- 3Dプリンタ





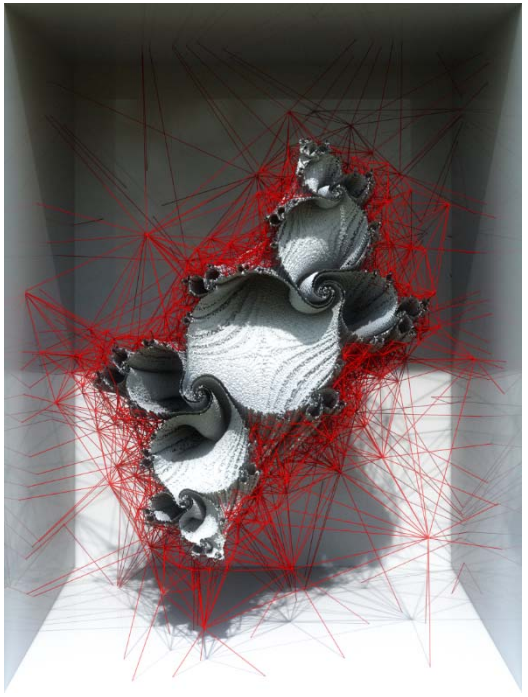


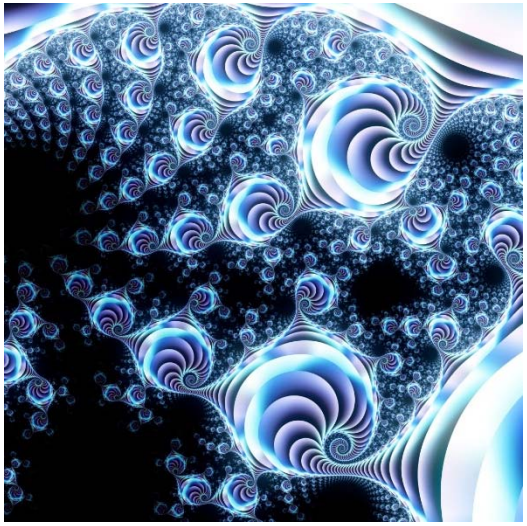


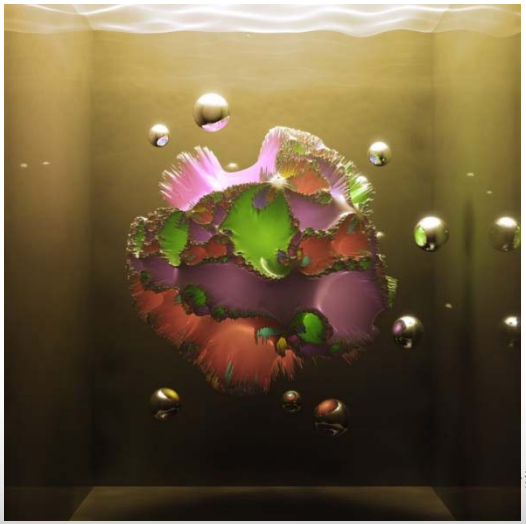


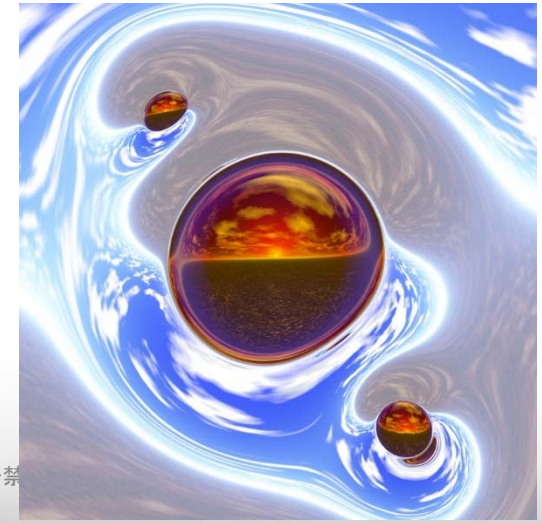
自己紹介. 趣味

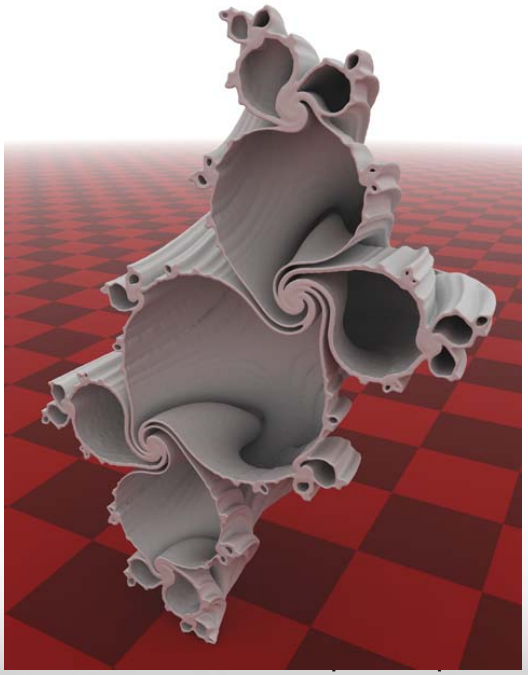
- CGアート( )制作
  - プロシージャル
    - フラクタル幾何学
    - カオス理論



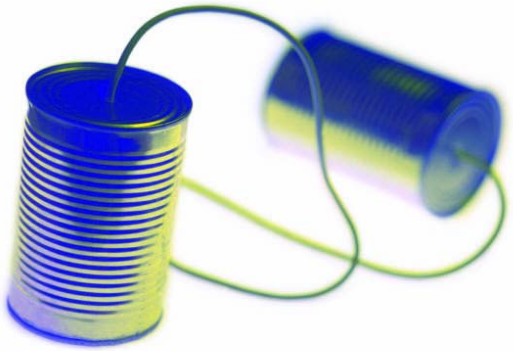








自己紹介. 連絡

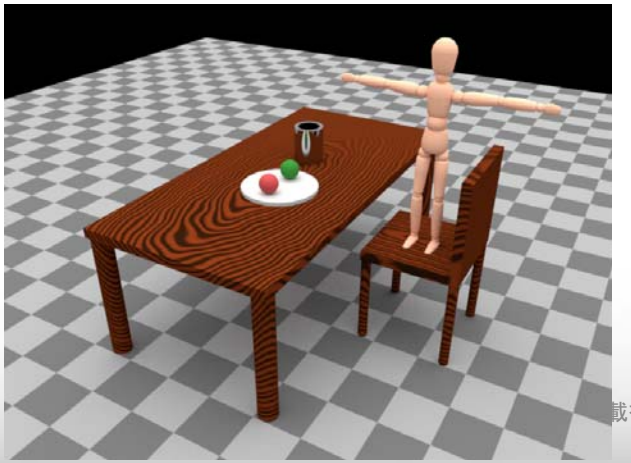


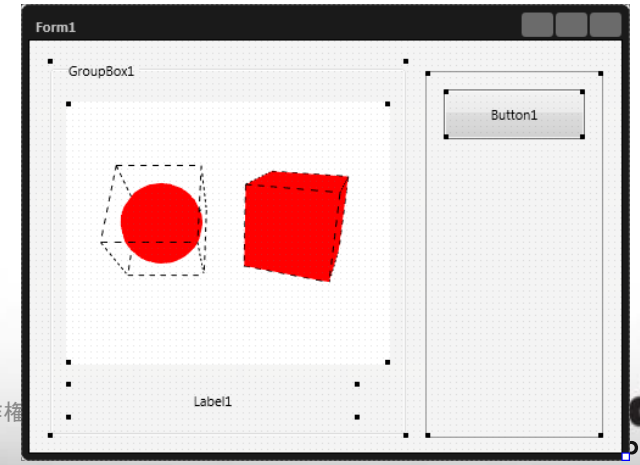
- 個人
  - サイト　：　http://www.luxidea.net
  - メール　：　contact@luxidea.net
  - Facebook：　http://www.facebook.com/luxidea
  - Twitter　：　@luxidea
- 慶應義塾大学 藤代研究室
  - サイト　：　http://www.fj.ics.keio.ac.jp
- 日蓮宗 玉泉山 安国院
  - サイト　：　http://www.ankokuin.or.jp

FireMonkey とは？

FireMonkey とは？．意義

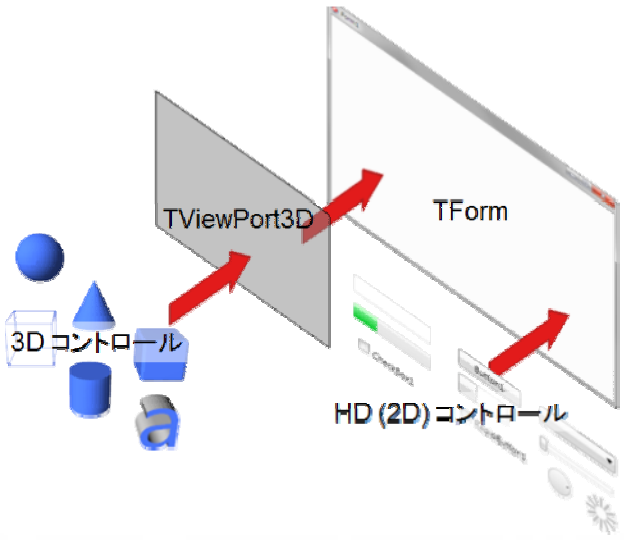
- 2D/3D のコンポーネント･フレームワーク
  - 主にGUIの表示
- マルチプラットフォーム
  - OS固有APIからの脱却
- CG と GUI の融合
  - グラフィックス関連のアプリ制作が容易に！



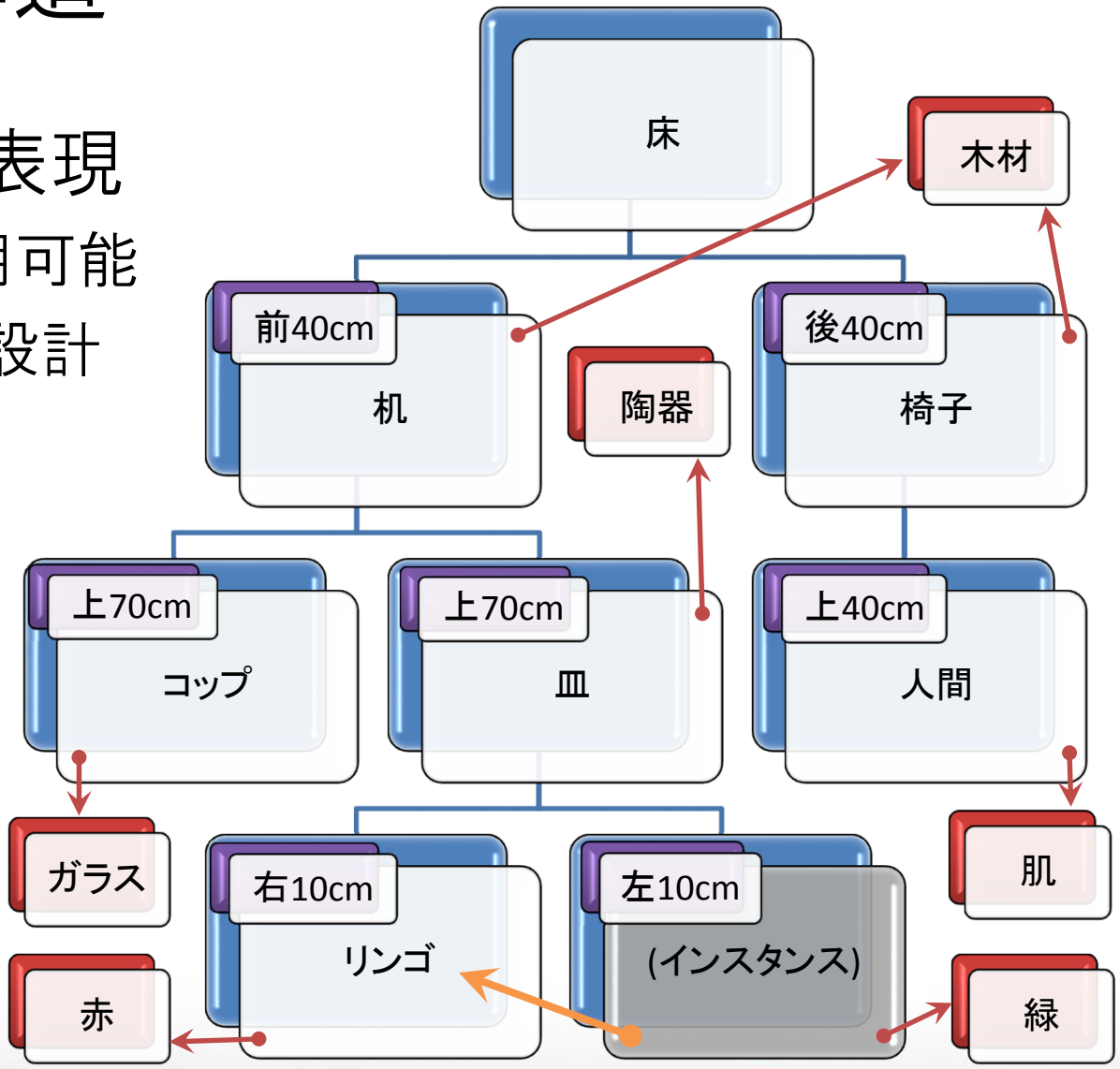


FireMonkey とは？. 構造

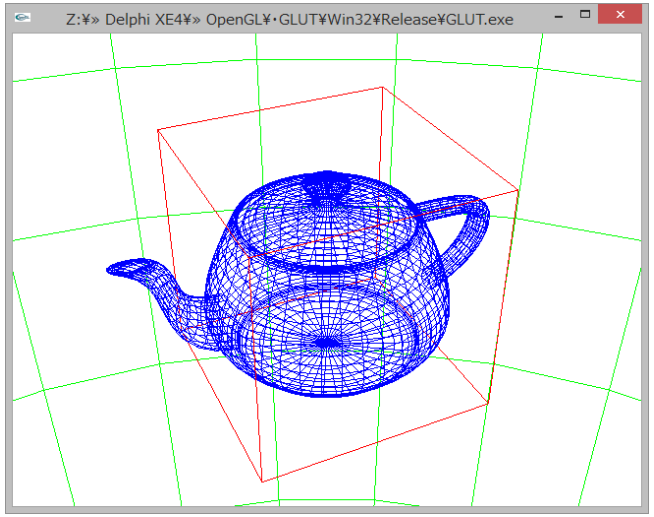
- シーングラフで表現
  - CG シーンへ転用可能
  - CG の概念で UI設計



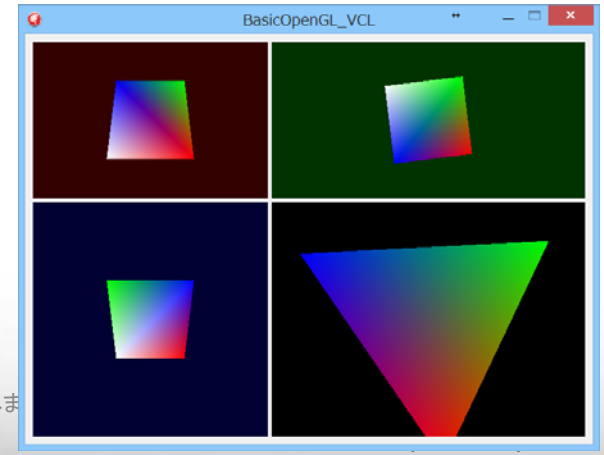
出典「FireMonkeyファーストインプレッション」



FireMonkey とは？. 解放

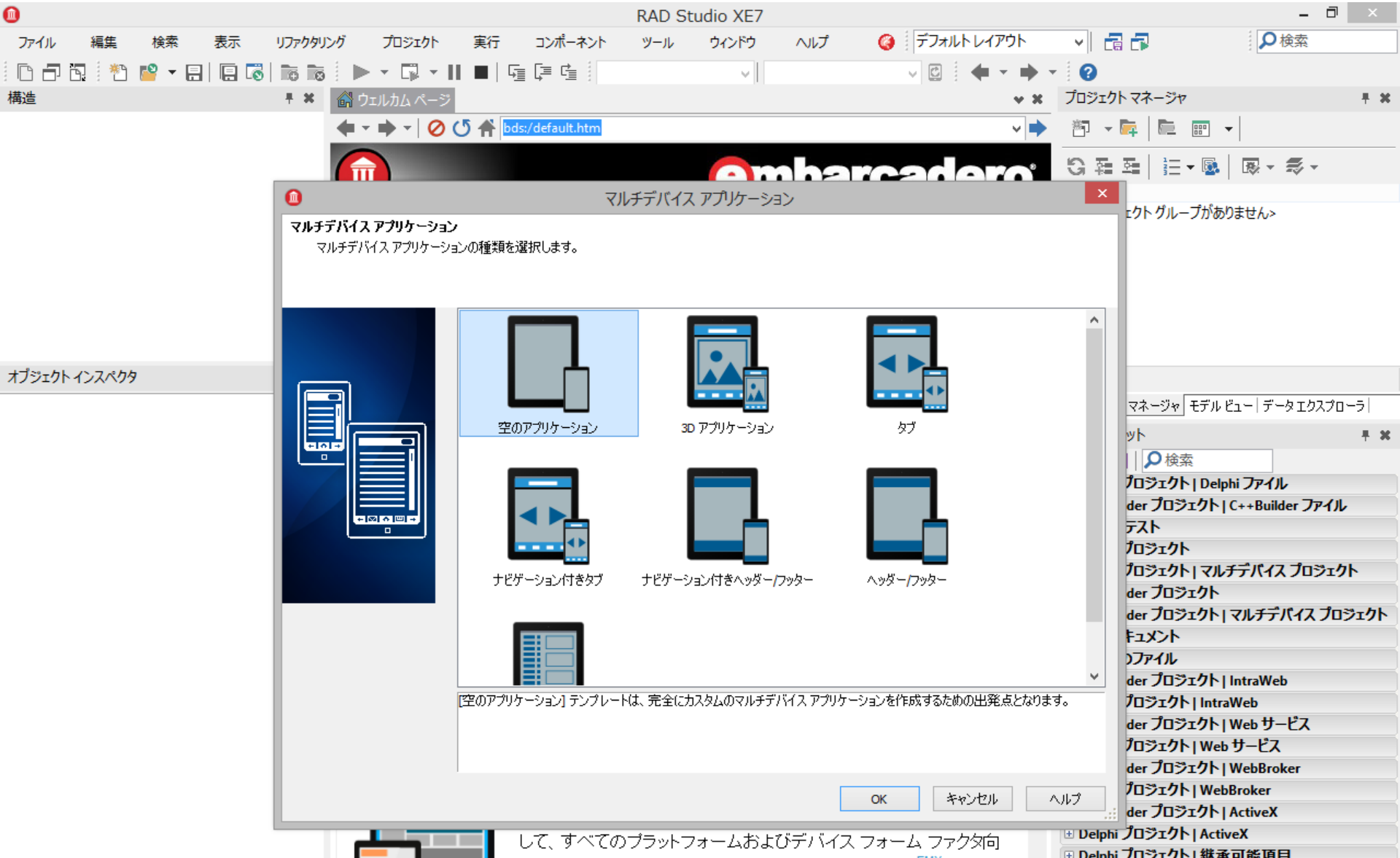


- リアルタイムCG の定番は GLUT
  - GUI が作れない
  - 低レベル実装が辛い
- 高機能ゲームライブラリ
  - 大規模すぎてブラックボックス
  - 細かいところを弄れない
- 俺々ライブラリ
  - マルチプラットフォーム化辛い
  - API バージョンの差異吸収辛い
  - GPU の対応レベルの確認辛い

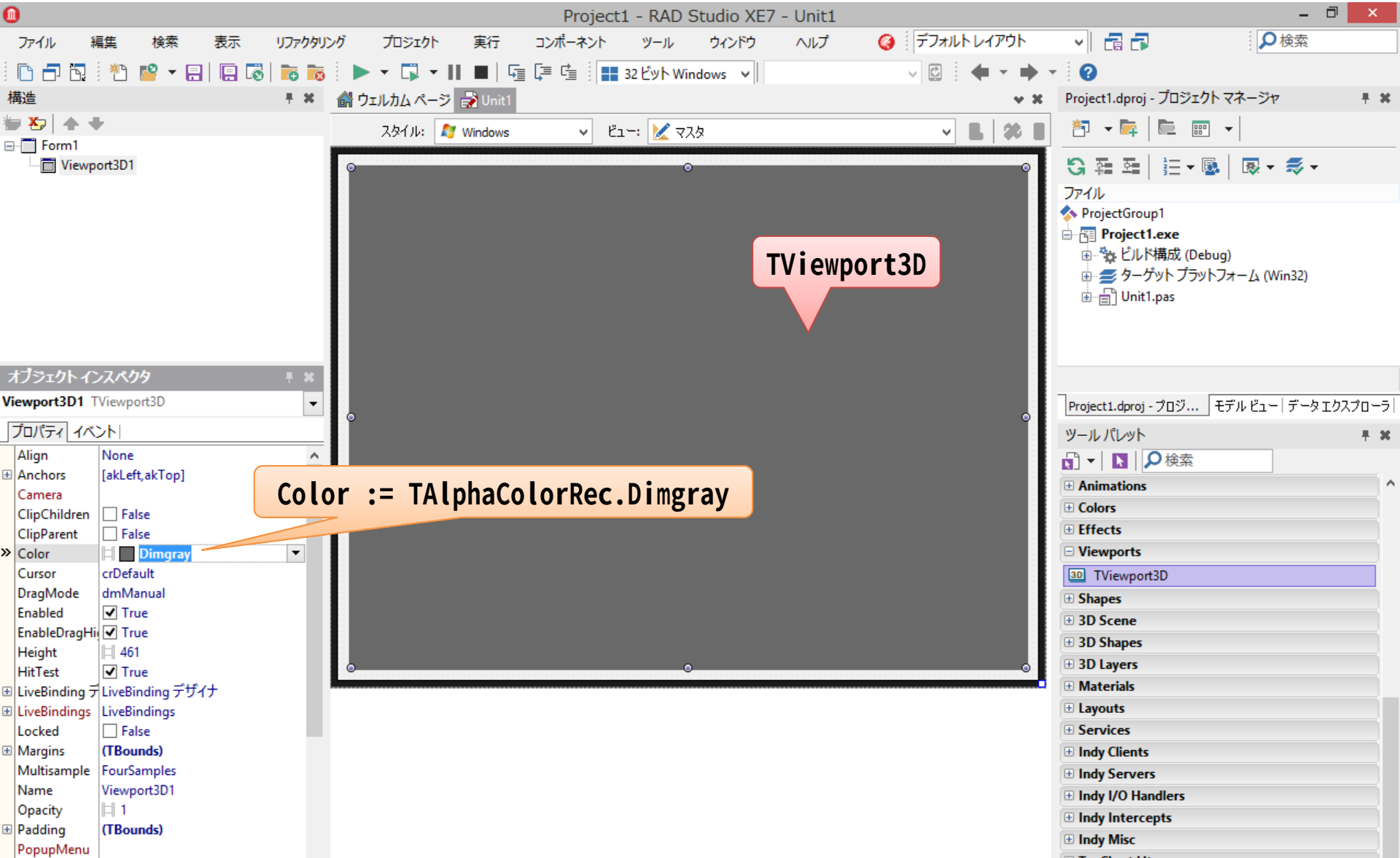


FireMonkey とは？. シーンの作り方.プロジェクト

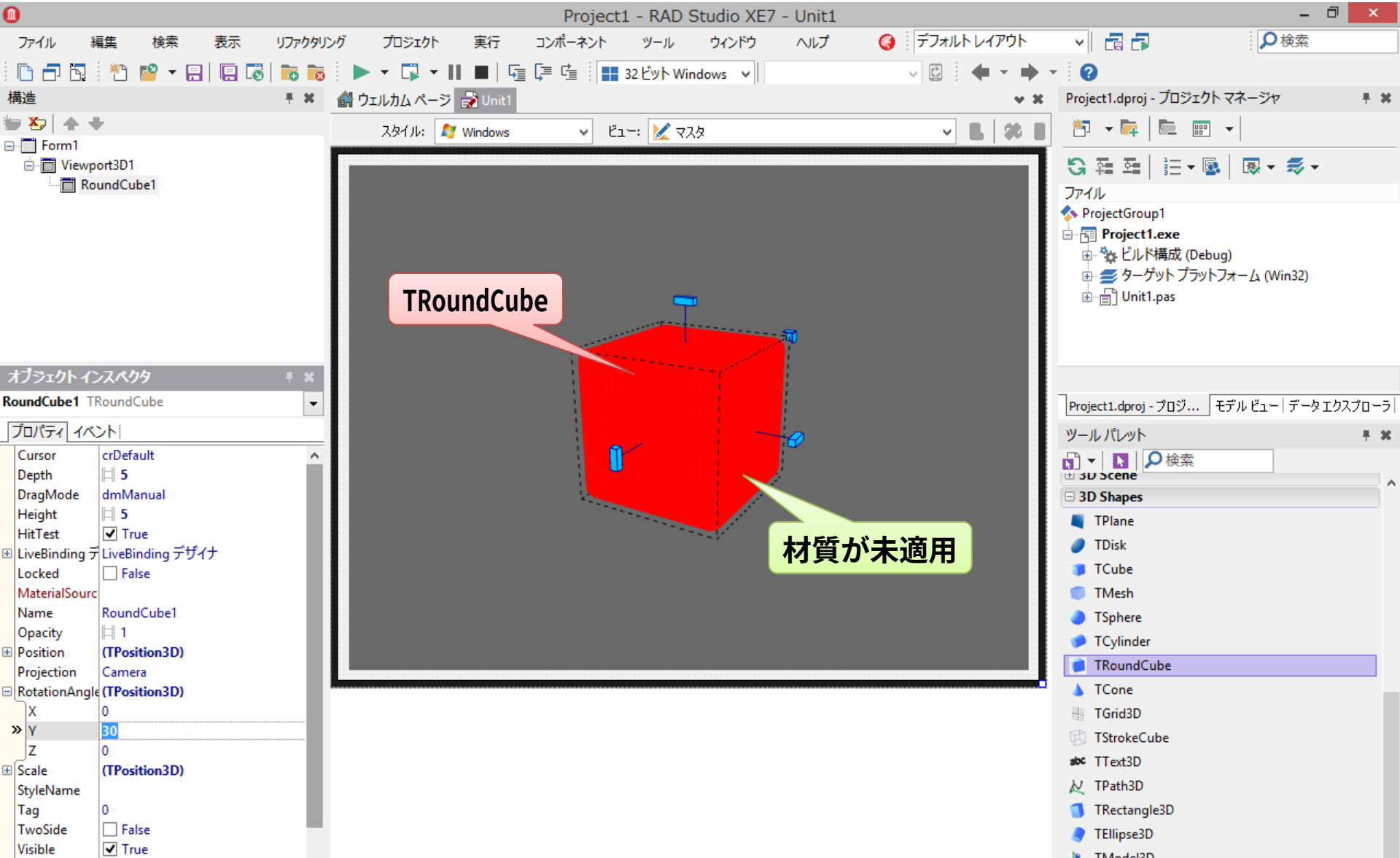
FireMonkey とは？. シーンの作り方.テンプレート



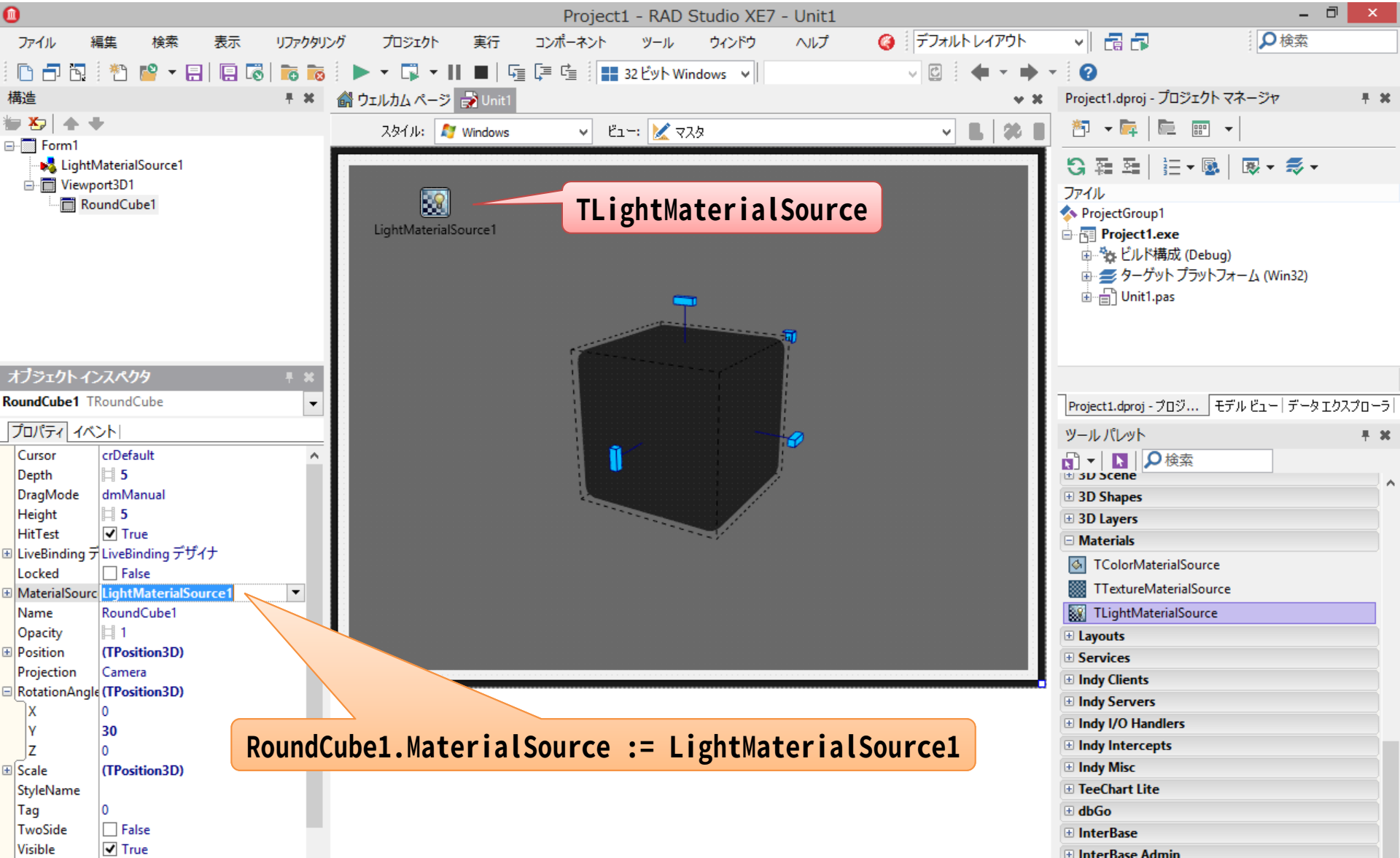
FireMonkey とは？. シーンの作り方.表示窓



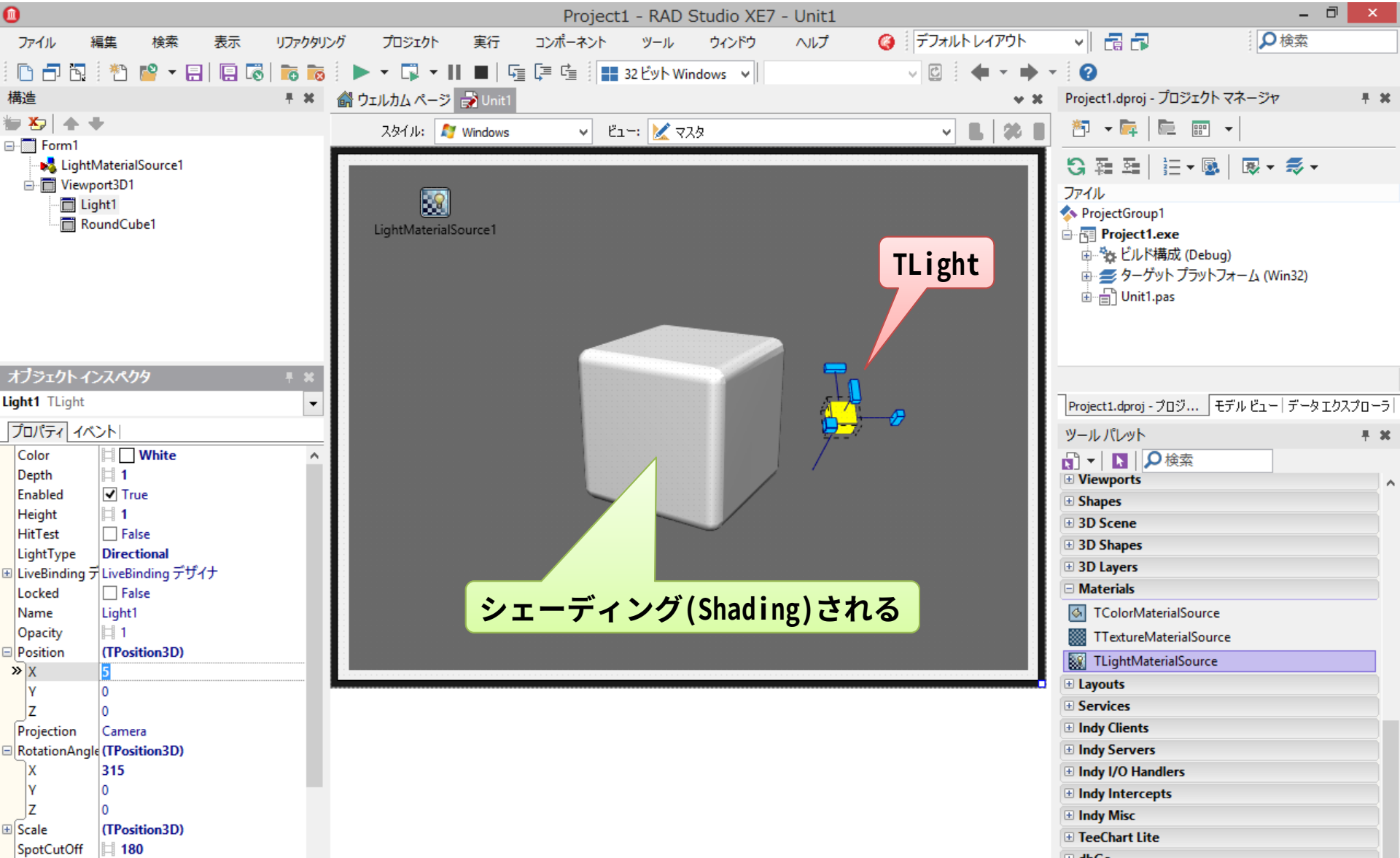
FireMonkey とは？. シーンの作り方.モデル



FireMonkey とは？. シーンの作り方.マテリアル

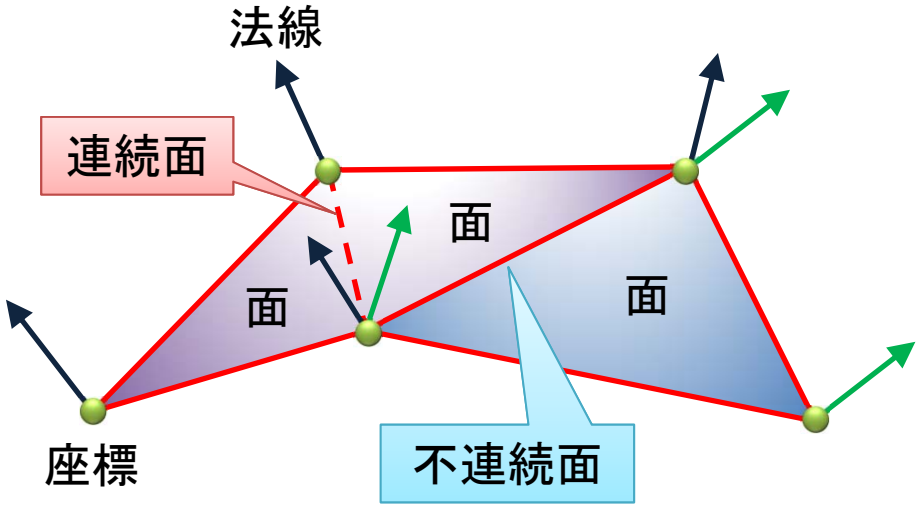


FireMonkey とは？. シーンの作り方.照明

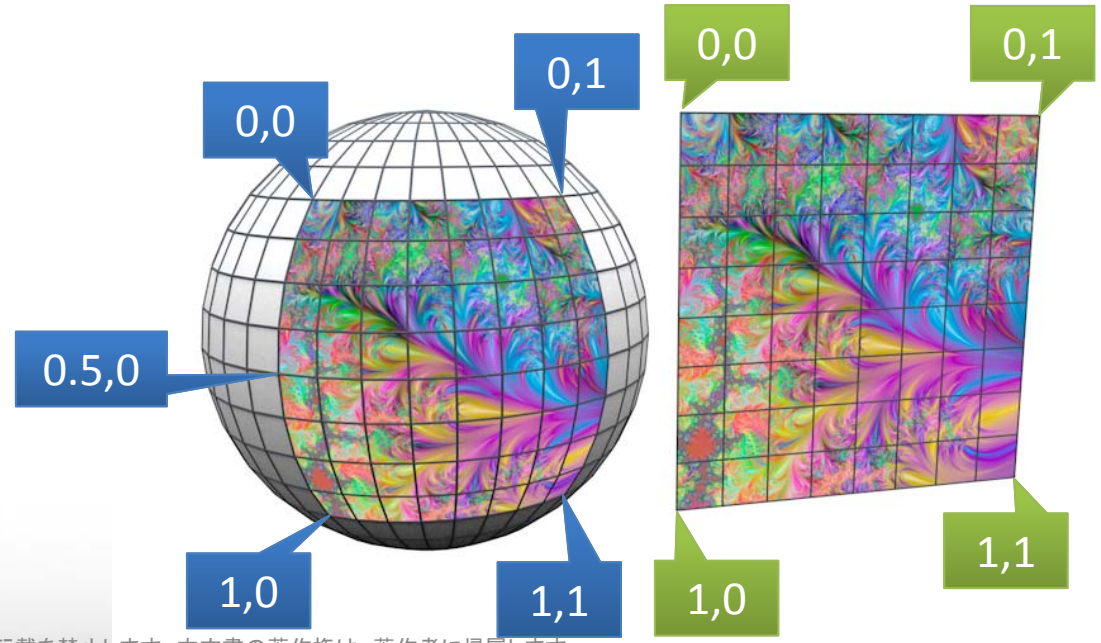


3Dモデルの作り方

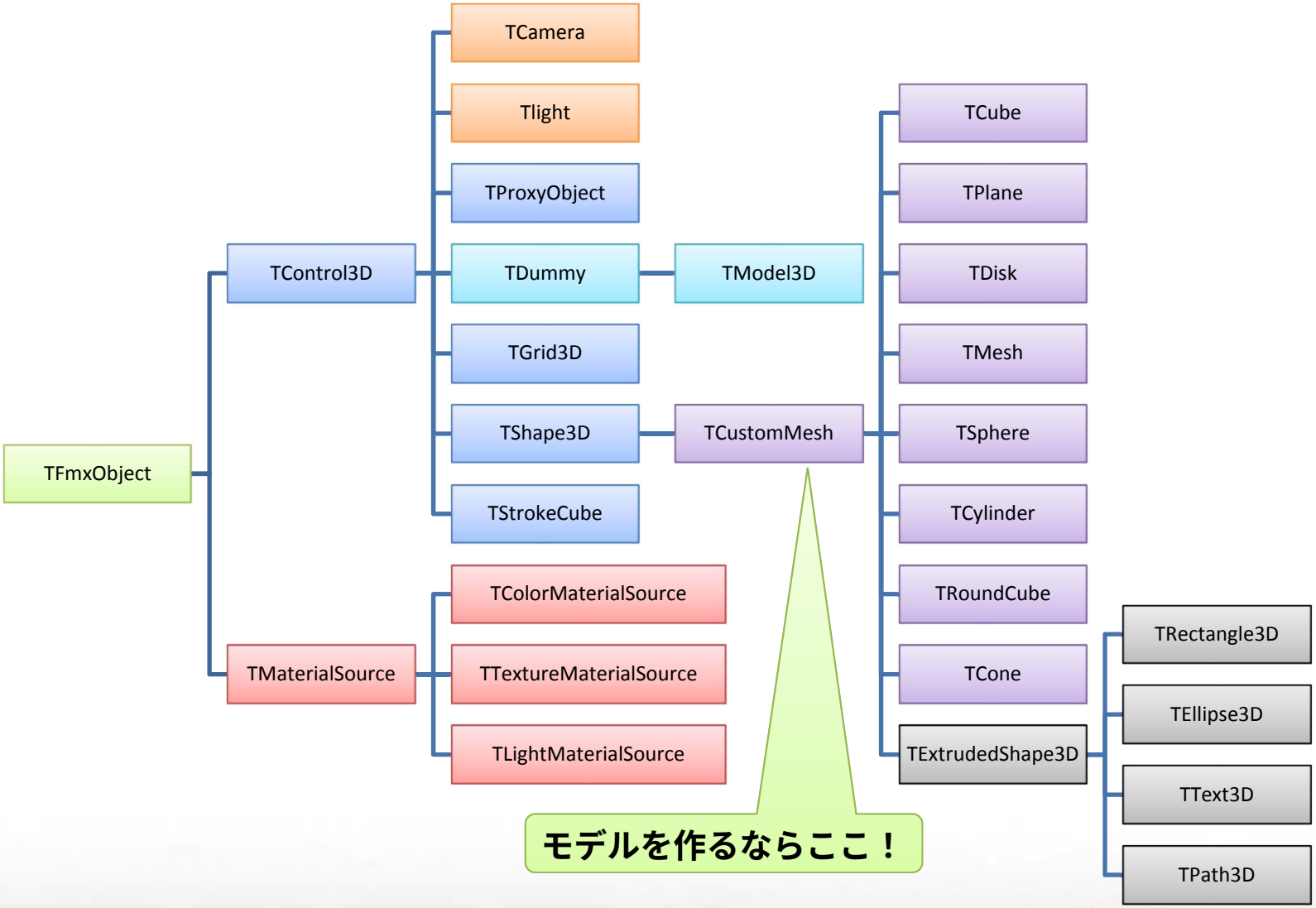
3Dモデルの作り方. ポリゴン



- 三角面の集合
  - GPU は 点･線･三角形しか描けない
- 構成要素
  - 頂点
    - 位置座標
    - 法線
      - 面の垂直ベクトル
    - テクスチャ座標
  - 面
    - 3つの頂点への参照



3Dモデルの作り方. クラス構造



3Dモデルの作り方. 転送バッファ

- TShape3D = class(TControl3D)
  - property MaterialSource :TMaterialSource
    - マテリアルコンポーネントへの参照
- TCustomMesh = class(TShape3D)
  - property Data :TMeshData
    - ポリゴンデータを保持する
- TMeshData = class(TPersistent)
  - property VertexBuffer :TVertexBuffer
    - 頂点配列
  - property IndexBuffer :TIndexBuffer
    - 面配列

3Dモデルの作り方. 頂点バッファ

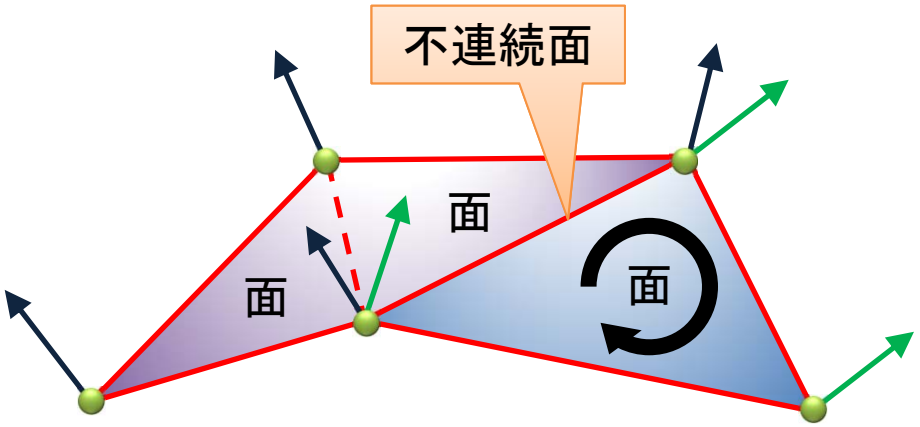
- TVertexBuffer = class(TPersistent)
  - property Length :Integer
    - 頂点数
  - property Vertices[AIndex:Integer] :TPoint3D
    - 頂点の位置座標
  - property Normals[AIndex:Integer] :TPoint3D
    - 頂点の法線ベクトル
  - property TexCoord0[AIndex:Integer] :TPointF
    - 頂点のテクスチャ座標

3Dモデルの作り方. 面バッファ

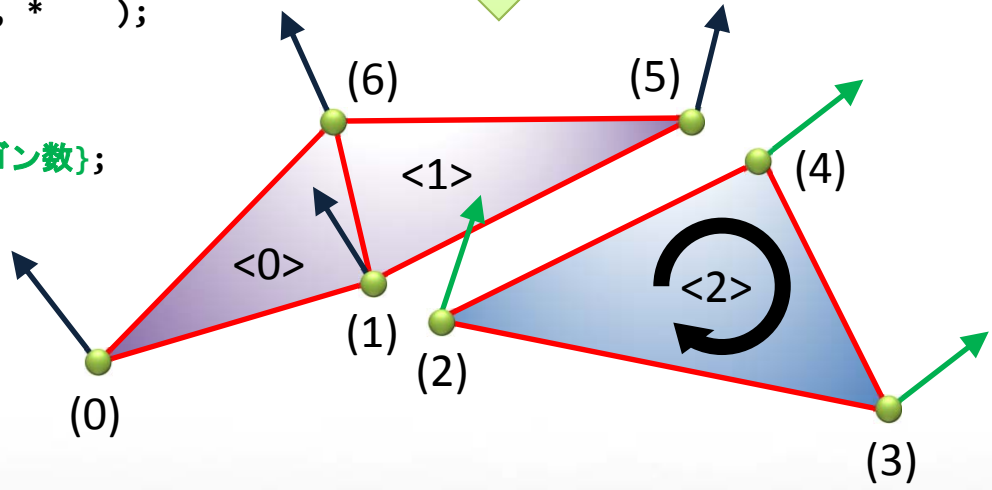
- TIndexBuffer = class(TPersistent)
  - property Length ：Integer
    - 面数×３：ポリゴン
    - 線数×２：ワイヤーフレーム
    - 点数×１：ポイントクラウド
  - property Indices[AIndex:Integer] :Integer
    - N番目のポリゴンの属する頂点のインデックス番号
    - Indices[3×N＋0] = 頂点①
    - Indices[3×N＋1] = 頂点②
    - Indices[3×N＋2] = 頂点③

3Dモデルの作り方. 実装

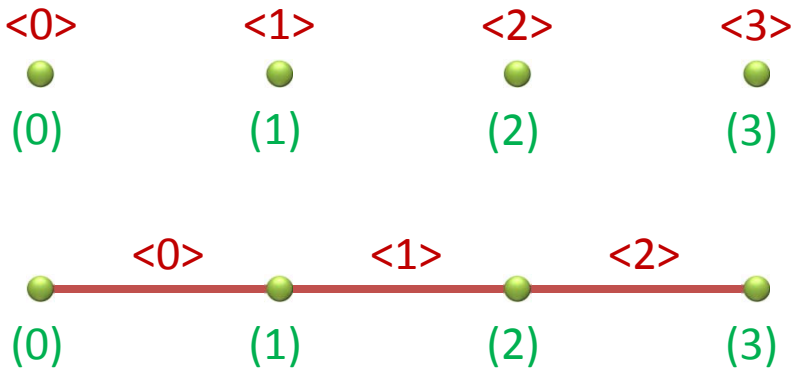
```
with Data do
begin
  with VertexBuffer do
  begin
    Length := 7{頂点数};
    Vertices [ 0 ] := TPoint3D.Create( *, *, * ){位置座標};
    Normals  [ 0 ] := TPoint3D.Create( *, *, * ){法線ベクトル};
    TexCoord0[ 0 ] := TPointF .Create( *, *    ){テクスチャ座標};
      ～
    Vertices [ 6 ] := TPoint3D.Create( *, *, * );
    Normals  [ 6 ] := TPoint3D.Create( *, *, * );
    TexCoord0[ 6 ] := TPointF .Create( *, *    );
  end;
  with IndexBuffer do
  begin
    Length := 3{頂点数/ポリゴン} * 3{ポリゴン数};
    Indices[ 3*0+0 ] := 6;
    Indices[ 3*0+1 ] := 1;
    Indices[ 3*0+2 ] := 0;
    Indices[ 3*1+0 ] := 6;
    Indices[ 3*1+1 ] := 5;
    Indices[ 3*1+2 ] := 1;
    Indices[ 3*2+0 ] := 4;
    Indices[ 3*2+1 ] := 3;
    Indices[ 3*2+2 ] := 2;
  end;
end;
```



分解



3Dモデルの作り方. 点と線

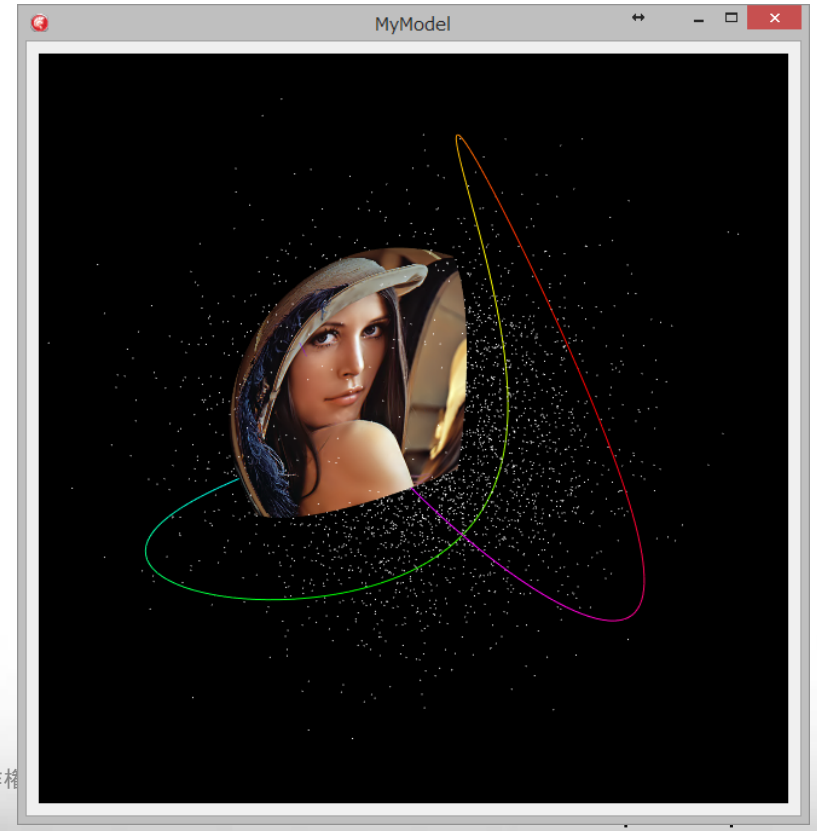


- 点

```
with IndexBuffer do
begin
  Length := 1{頂点数/点} * 4{点数};
  Indices[ 0 ] := 0;
  Indices[ 1 ] := 1;
  Indices[ 2 ] := 2;
  Indices[ 3 ] := 3;
    ～
end;
```

- 線

```
with IndexBuffer do
begin
  Length := 2{頂点数/線} * 3{線数};
  Indices[ 2*0+0 ] := 0;
  Indices[ 2*0+1 ] := 1;
  Indices[ 2*1+0 ] := 1;
  Indices[ 2*1+1 ] := 2;
  Indices[ 2*2+0 ] := 2;
  Indices[ 2*2+1 ] := 3;
    ～
end;
```



3Dモデルの作り方. レンダリング

- Render メソッドによって描画される。

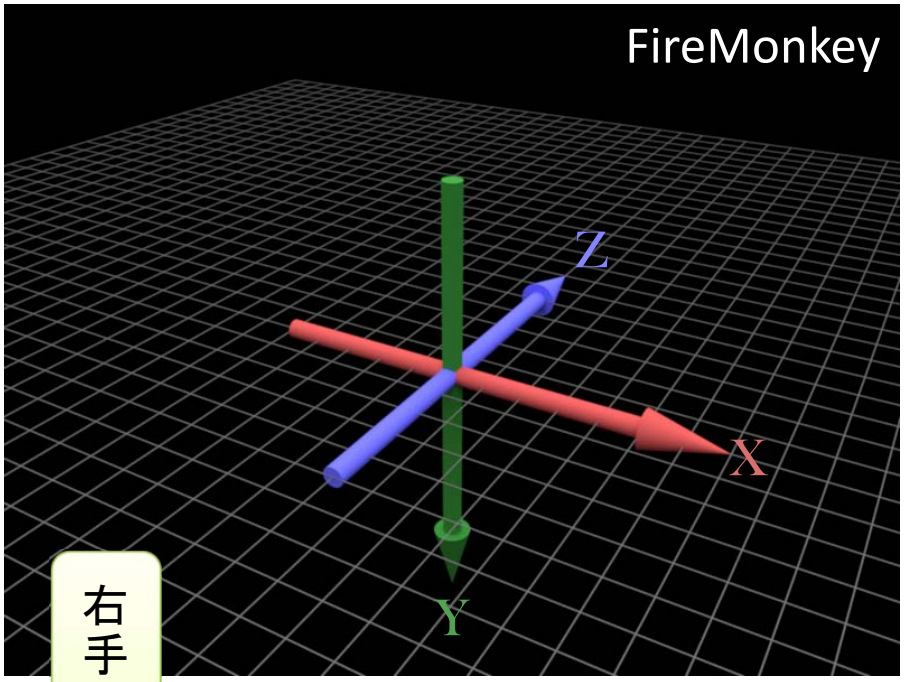
```
procedure TCustomMesh.Render;
begin
  Context.SetMatrix( TMatrix3D.CreateScaling( TPoint3D.Create( Width, Height, Depth ) ) * AbsoluteMatrix );

  Context.DrawTriangles( Data.VertexBuffer,
                         Data.IndexBuffer,
                         TMaterialSource.ValidMaterial( FMaterialSource ),
                         AbsoluteOpacity );
end;
```

- 点や線を描く場合は override による変更が必要
  - Context.DrawPoints　　：点描画
  - Context.DrawLines　　：線描画
  - Context.DrawTriangles　：面描画

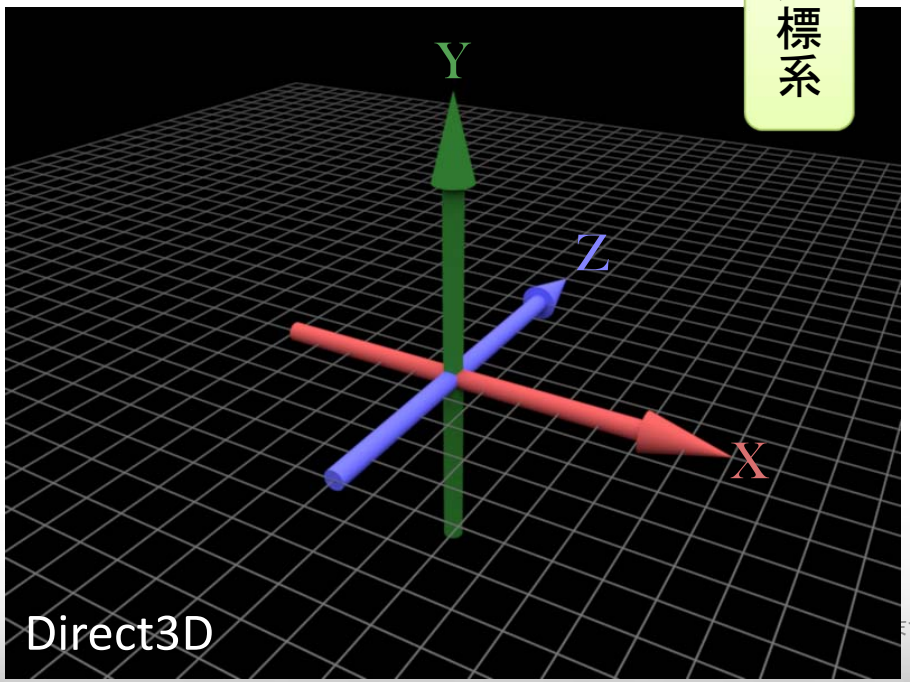
座標変換
～ベクトルと行列～

座標変換. 座標系

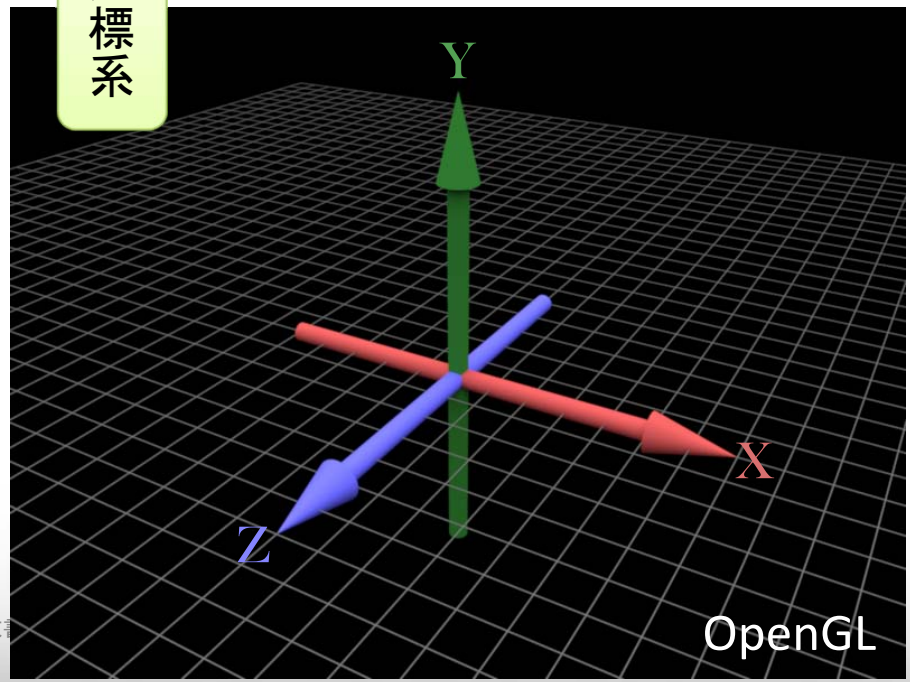
- 反転した右手座標系
  - X：親指
  - Y：人差し指
  - Z：中指



右手座標系

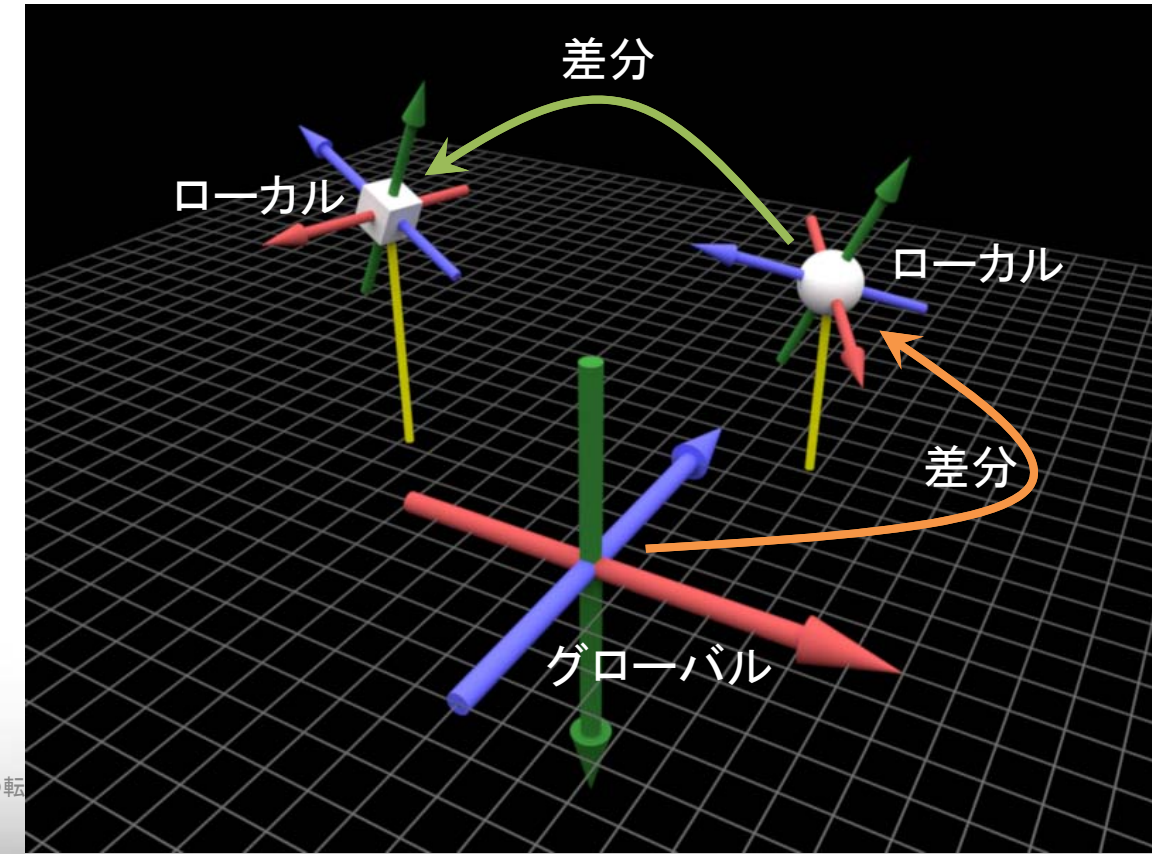


左手座標系



座標変換. 物と世界

- グローバル座標
  - ワールド座標
  - 絶対座標
- ローカル座標
  - オブジェクト座標
  - 相対座標
  - 差分



座標変換. 同時座標系

- 座標変換は行列(4×4)の演算で表現できる
  - 基底ベクトルを並べたもの
  - 複数の変換を合成して一つの行列で表せる

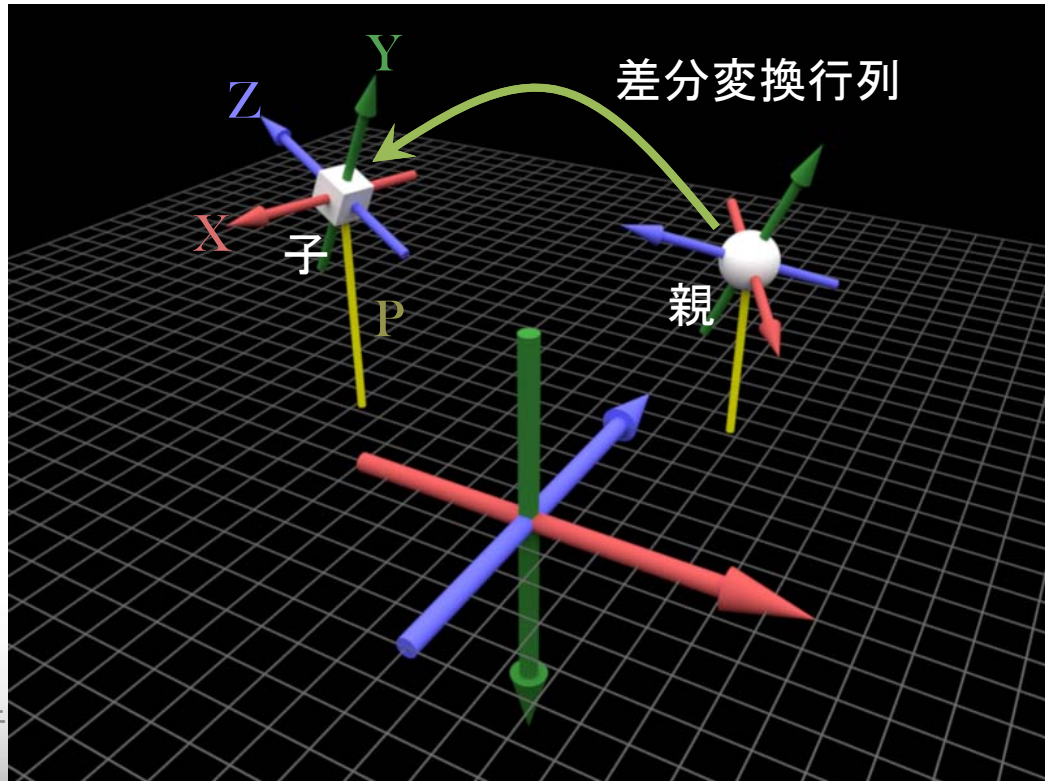
TVector3D

$$[A_X \quad A_Y \quad A_Z \quad 1] \times \begin{bmatrix} X_X & X_Y & X_Z & 0 \\ Y_X & Y_Y & Y_Z & 0 \\ Z_X & Z_Y & Z_Z & 0 \\ P_X & P_Y & P_Z & 1 \end{bmatrix}$$

$$= [B_X \quad B_Y \quad B_Z \quad 1]$$

TMatrix3D

子基準変換 × 変換前行列 × 親基準変換 = 変換後行列



座標変換. TControl3D.LocalMatrix

- 親に対する差分変換行列を保持
- クラスヘルパーを用いて入力可能とする
  - Position, RotationAngle などのプロパティが不整合化

```
interface

type
  HControl3D = class helper for TControl3D
  protected
    function GetLocalMatrix :TMatrix3D;
    procedure SetLocalMatrix( const LocalMatrix_:TMatrix3D ); virtual;
  public
    property LocalMatrix :TMatrix3D read GetLocalMatrix write SetLocalMatrix;
  end;

implementation

function HControl3D.GetLocalMatrix :TMatrix3D;
begin
  Result := FLocalMatrix;
end;

procedure HControl3D.SetLocalMatrix( const LocalMatrix_:TMatrix3D );
begin
  FLocalMatrix := LocalMatrix_;  RecalcAbsolute;  Repaint;
end;
```

座標変換. 行列系関数

・TPoint3D = [1 2 3]
・TVector3D = [1 2 3 4]

- System.Math.Vectors.TMatrix3D
  - 生成
    - const Identity :TMatrix3D
      - 無変換(単位行列)
    - class function CreateScaling(～) :TMatrix3D;
    - class function CreateTranslation(～) :TMatrix3D;
    - class function CreateRotationX/Y/Z(～) :TMatrix3D;
    - class function CreateRotation(～) :TMatrix3D;
      - 任意軸での回転
  - 演算
    - class operator Multiply(TMatrix3D×TMatrix3D) :TMatrix3D;
    - class operator Multiply(TPoint3D×TMatrix3D) :TPoint3D;
    - class operator Multiply(TVector3D×TMatrix3D) :TVector3D;

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

座標変換. 例

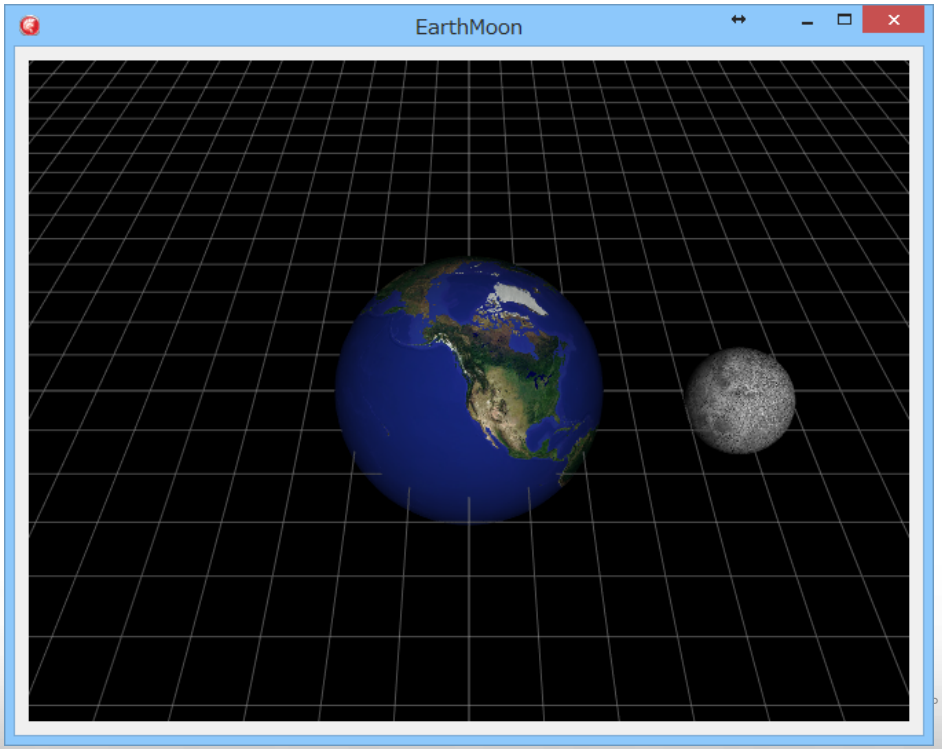
位置を初期化

```
月.LocalMatrix := TMatrix3D.CreateTranslation( TPoint3D.Create( 5, 0, 0 ) );
```

```
月.LocalMatrix := TMatrix3D.CreateRotationX( DegToRad( 2 ) )
                  * 月.LocalMatrix
                  * TMatrix3D.CreateRotationY( DegToRad( 1 ) )
```

ローカル座標系で
X軸回転

ワールド座標系で
Y軸回転



座標変換. 絶対座標変換行列

$$M \times M^{-1} = M^{-1} \times M = I \text{（単位行列）}$$

- TControl3D
  - LocalMatrix :TMatrix3D;
    - 親のローカル座標系に対する座標変換行列
  - AbsoluteMatrix :TMatrix3D;
    - ワールド座標系に対する座標変換行列

    自分.LocalMatrix × 親.AbsoluteMatrix = 自分.AbsoluteMatrix
- 絶対座標変換行列を直接指定するには？
  - 親の位置から逆算して自分の LM を求める

    自.LM × 親.AM ＝ 自.AM

    自.LM ＝ 自.AM ÷ 親.AM

    自.LM ＝ 自.AM × 親.AM.Inverse

行列に割り算はない

逆行列：行列の逆数

座標変換. TControl3D.AbsoluteMatrix

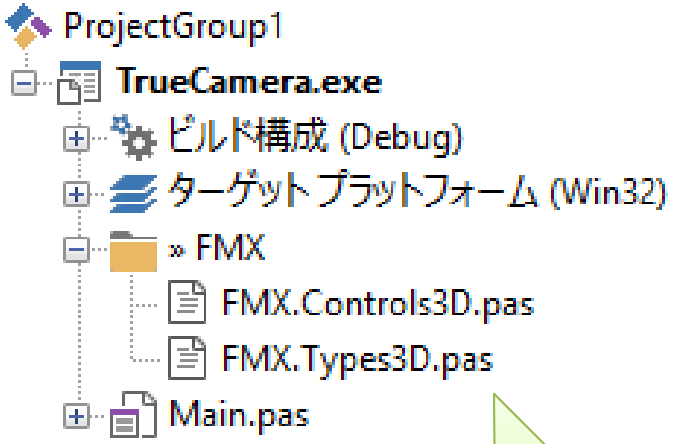
- ワールド座標系に対する差分変換行列を保持
- クラスヘルパーを用いて入力可能とする
  - Position, RotationAngle などのプロパティが不整合化

```
interface

type
  HControl3D = class helper for TControl3D
  protected
    function GetAbsolMatrix :TMatrix3D;
    procedure SetAbsoluteMatrix( const AbsoluteMatrix_:TMatrix3D );
  public
    property AbsoluteMatrix :TMatrix3D read GetAbsolMatrix write SetAbsoluteMatrix;

implementation

function HControl3D.GetAbsoluteMatrix :TMatrix3D;
begin
  Result := Self.GetAbsoluteMatrix;
end;

procedure HControl3D.SetAbsoluteMatrix( const AbsoluteMatrix_:TMatrix3D );
begin
  LocalMatrix := AbsoluteMatrix_ * TControl3D( Parent ).AbsoluteMatrix.Inverse;
end;
```

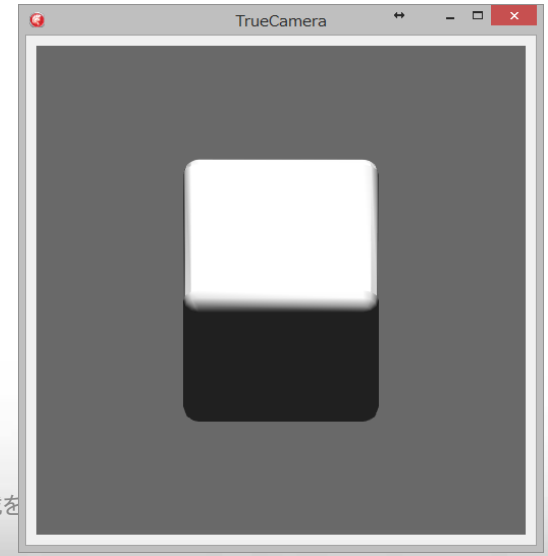
イレギュラーテクニック

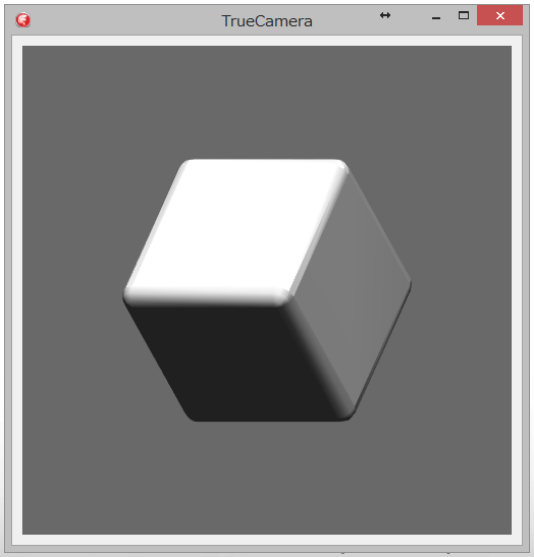
イレギュラーテクニック. 平行投影

- 遠近法を廃した平行線の交わらない表示
  - CADなどの製図用途
  - 2次元を意識したアニメ表現



オリジナルライブラリの
書き換え



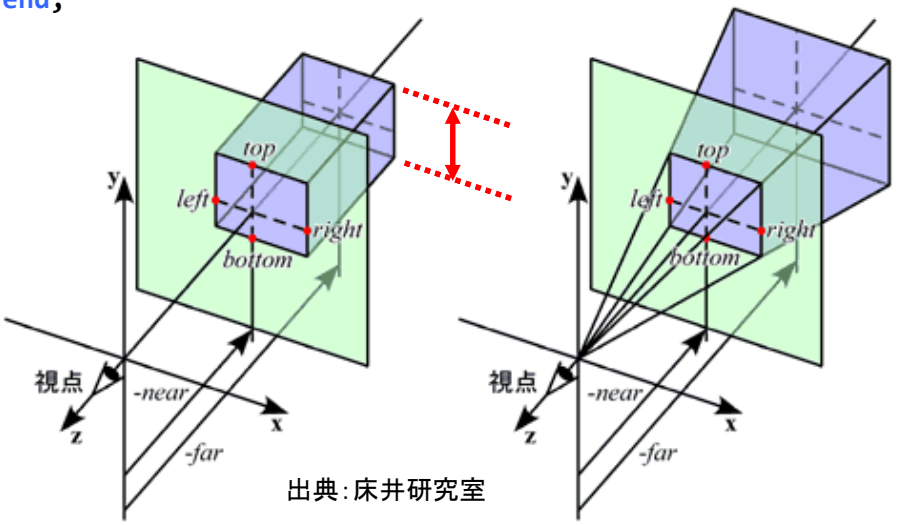


イレギュラーテクニック. 平行投影. 実装

- 投影変換行列を変更
  - TMatrix3D に生成関数
    - CreateOrthoRH(～)
    - CreateOrthoOffCenterRH(～)
      - 中心をずらした平行投影
    - CreatePerspectiveFovRH(～)
      - 透視投影

```
procedure TCamera.SetAngleOfView(const Value: Single);
begin
  if FAngleOfView <> Value then
  begin
    FAngleOfView := Value;
    if FAngleOfView <   0 then FAngleOfView :=   0;
    if FAngleOfView > 179 then FAngleOfView := 179;
    ～
end;
```

画角ゼロを許可



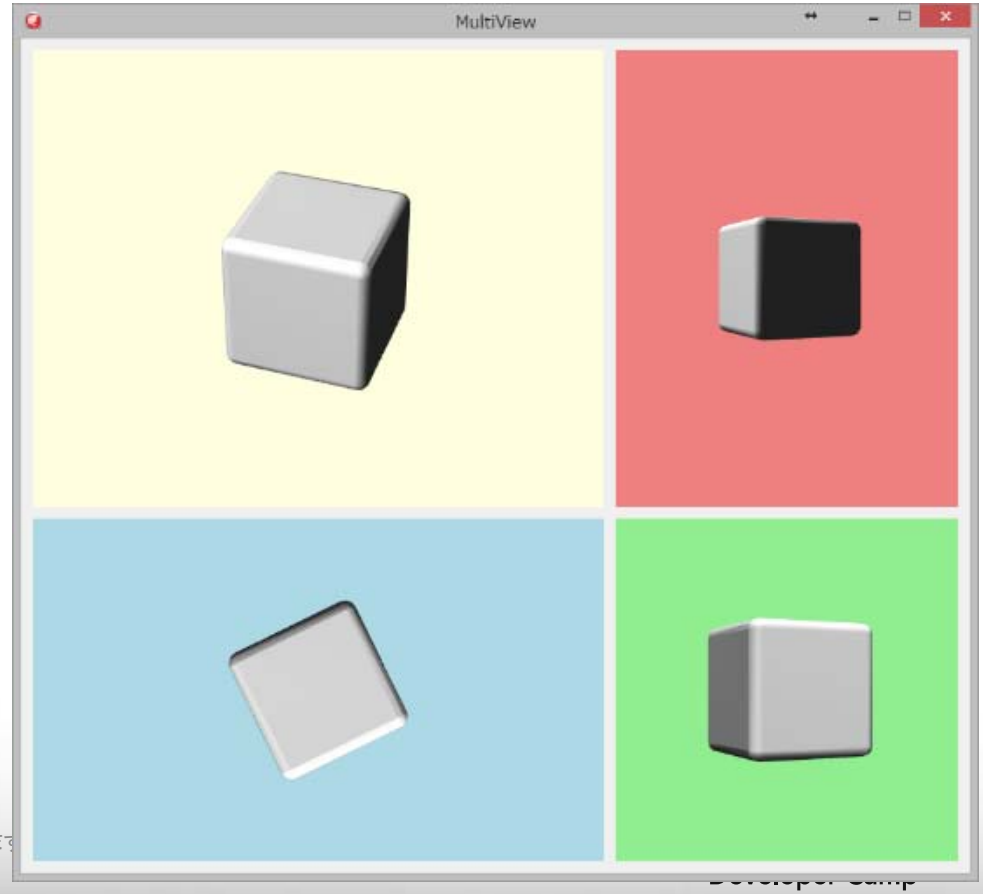
出典:床井研究室

```
function TContext3D.GetProjectionMatrix: TMatrix3D;
var
  M: TMatrix3D;
begin
  if FRecalcProjectionMatrix then
  begin
    if FCurrentAngleOfView = 0 then
       Result := TMatrix3D.CreateOrthoRH( {表示高}, {表示高} * FWidth / FHeight, 1.0, 1000.0 )
    else
    if SameValue(FHeight, 0.0, Epsilon) then
      Result := TMatrix3D.CreatePerspectiveFovRH(DegToRad(FCurrentAngleOfView), 1.0, 1.0, 1000.0)
    else
      Result := TMatrix3D.CreatePerspectiveFovRH(DegToRad(FCurrentAngleOfView), FWidth / FHeight, 1.0, 1000.0);
    ～
end;
```

平行投影

イレギュラーテクニック. 複数視点

- 一つのシーンを複数のカメラで撮影
  - それぞれのカメラは画角も解像度も個別に変更可能
  - 三面図などへの応用



イレギュラーテクニック. 複数視点. 実装. 1

- 任意サイズの TBitmap へレンダリングするクラス

```
TBitmapRender = class
Protected
  _Camera      :TCamera;
  _BMP         :TBitmap;
  _Texture     :TTexture;
  _Context     :TContext3D;
  _Color       :TAlphaColor;
  _Multisample :TMultisample;
public
  constructor Create( const Camera_:TCamera; const BMP_:TBitmap );
  destructor Destroy; override;
  property Color       :TAlphaColor  read _Color       write _Color;
  property Multisample :TMultisample read _Multisample write _Multisample;
  procedure Render;
end;

HViewport3D = class helper for TViewport3D
protected
  function GetLights :TList<TLight>;
  function GetRenderingList :TList<TControl3D>;
public
  property Lights        :TList<TLight>     read GetLights;
  property RenderingList :TList<TControl3D> read GetRenderingList;
end;
```

カメラ

描画先

private のフィールドへ強制アクセス

イレギュラーテクニック. 複数視点. 実装. 2

- 任意サイズの TTexture から TContext3D を生成

```
constructor TBitmapRender.Create(～);
begin
     _Camera := Camera_;
     _BMP    := BMP_;

     _Texture := TTexture.Create;
     _Texture.Style := [ TTextureStyle.RenderTarget ];
     _Texture.SetSize( _BMP.Width, _BMP.Height );
     ITextureAccess( _Texture ).TextureScale := 1;

     _Multisample := TMultisample.FourSamples;
     _Color       := TAlphaColors.White;

     _Context := TContextManager.CreateFromTexture( _Texture, _Multisample, True );
end;

destructor TBitmapRender.Destroy;
begin
     FreeAndNil( _Context );
     FreeAndNil( _Texture );
end;
```

描画先ビットマップと同サイズ

背景色

アンチエイリアシングのサンプル数

イレギュラーテクニック. 複数視点. 実装. 3

- 描画は TViewport3D に任せる

```
procedure TBitmapRender.Render;
var
   W, H :Integer;
   C :TControl3D;
begin
     with _Camera.Viewport.Context do
     begin
          W := Width;
          H := Height;
          SetSize( _BMP.Width, _BMP.Height );
          SetCameraMatrix( _Camera.CameraMatrix );
          SetCameraAngleOfView( _Camera.AngleOfView );
     end;
     with _Context do
     begin
          BeginScene;
          SetContextState( TContextState.csScissorOff );
          Clear( [ TClearTarget.Color, TClearTarget.Depth ], _Color, 1.0, 0 );
          for C in _Camera.Viewport.RenderingList do C.RenderInternal;
          EndScene;
          CopyToBitmap( _BMP, Rect( 0, 0, _BMP.Width, _BMP.Height ) );
     end;
     _Camera.Viewport.Context.SetSize( W, H );
end;
```

参照元の元サイズを保存

参照元のサイズを描画先のサイズへ

BeginScene を呼んだ Context が描画対象

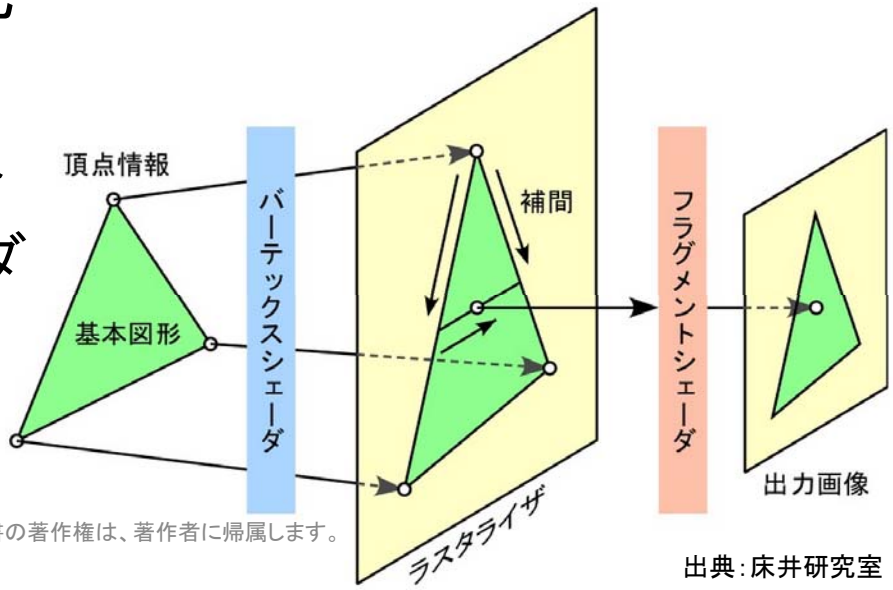
結果を描画先へ転写

参照元のサイズを復元

マテリアルの作り方

マテリアルの作り方. 原理

- FireMonkey は Direct3D と OpenGL により描画
  - Direct3D：Microsoft が策定する API
    - Windows 環境でのみ動作
  - OpenGL　：Khronosグループが策定する API
    - MacOSX / iOS / Android / Linux 環境で動作
- 質感はシェーダにより表現
  - バーテックス（頂点）シェーダ
    - モデルの頂点毎に呼ばれるイベント
  - フラグメント（ピクセル）シェーダ
    - ピクセル毎に呼ばれるイベント



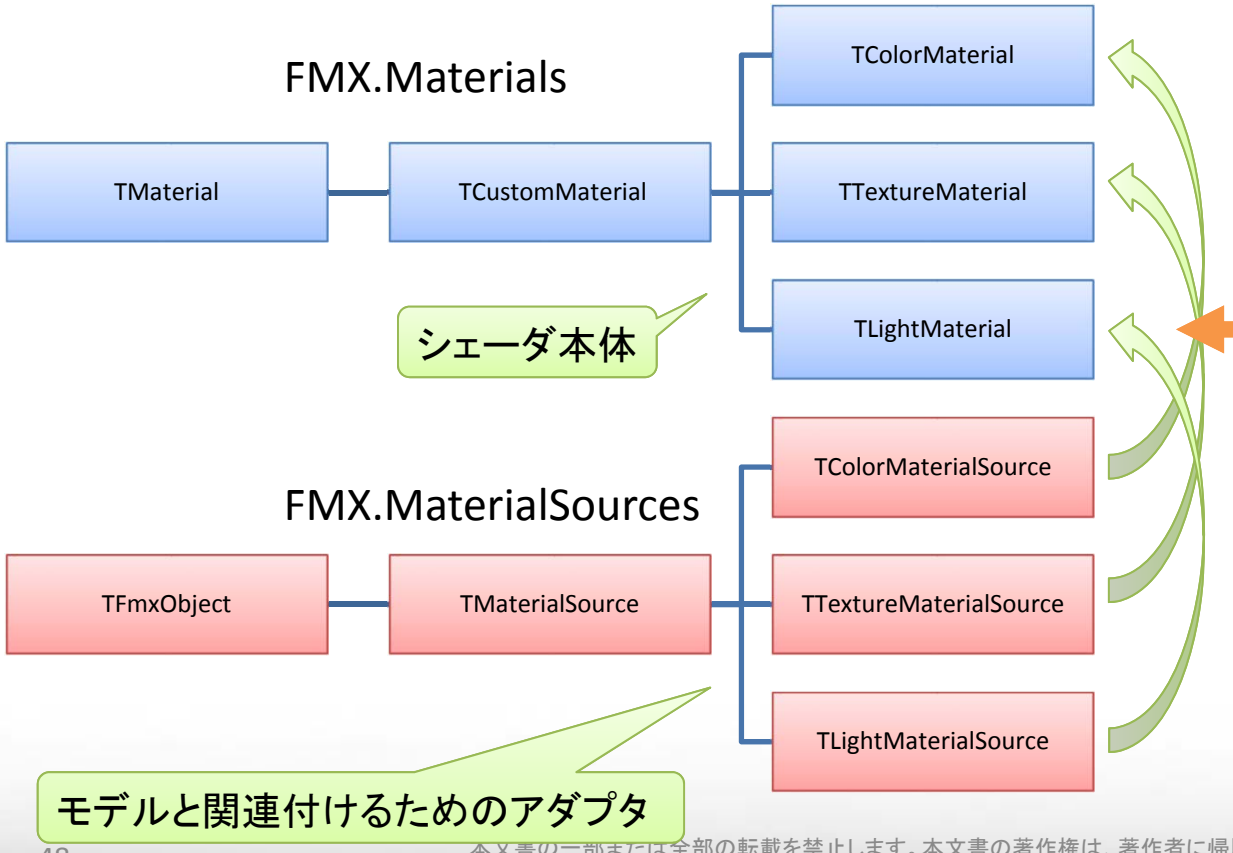
出典：床井研究室

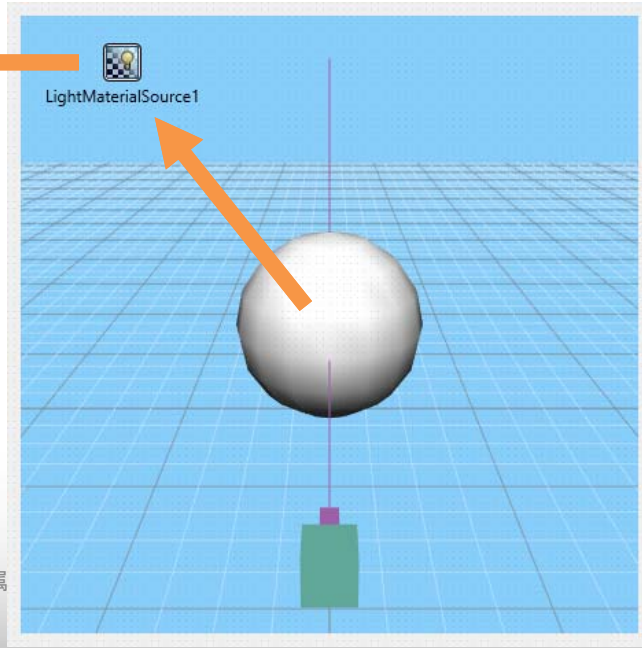
マテリアルの作り方. プログラママブル

- シェーダ は シェーディング言語 により記述
  - Direct3D：HLSL (High Level Shader Language) 言語
    - Windows SDK によりプリコンパイルしたバイナリを実行時にロード
      - Windows 8.1 用 Windows ソフトウェア開発キット (Windows SDK)
        http://msdn.microsoft.com/ja-jp/windows/desktop/bg162891.aspx
  - OpenGL　：GLSL (OpenGL Shading Language) 言語
    - ソースを直接ロードし実行時にコンパイル
      - ソースを秘匿できない
      - プリコンパイルに辛うじて対応：GL_ARB_get_program_binary
        - » GPU ベンダ依存の互換性問題

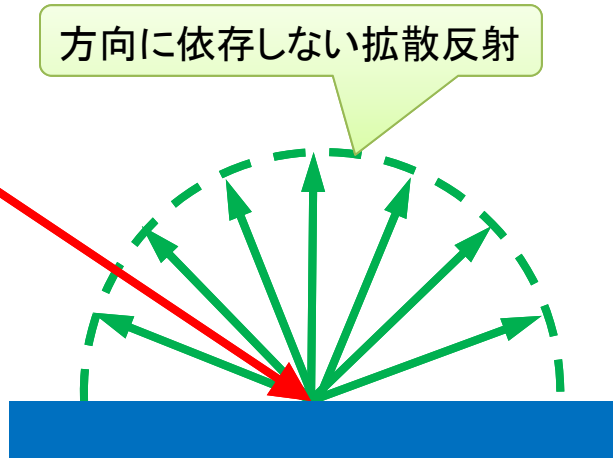
マテリアルの作り方. クラス構造

- TMaterial が TMaterialSource に内包される
  - 同じシェーダを複数のモデルに適用可能



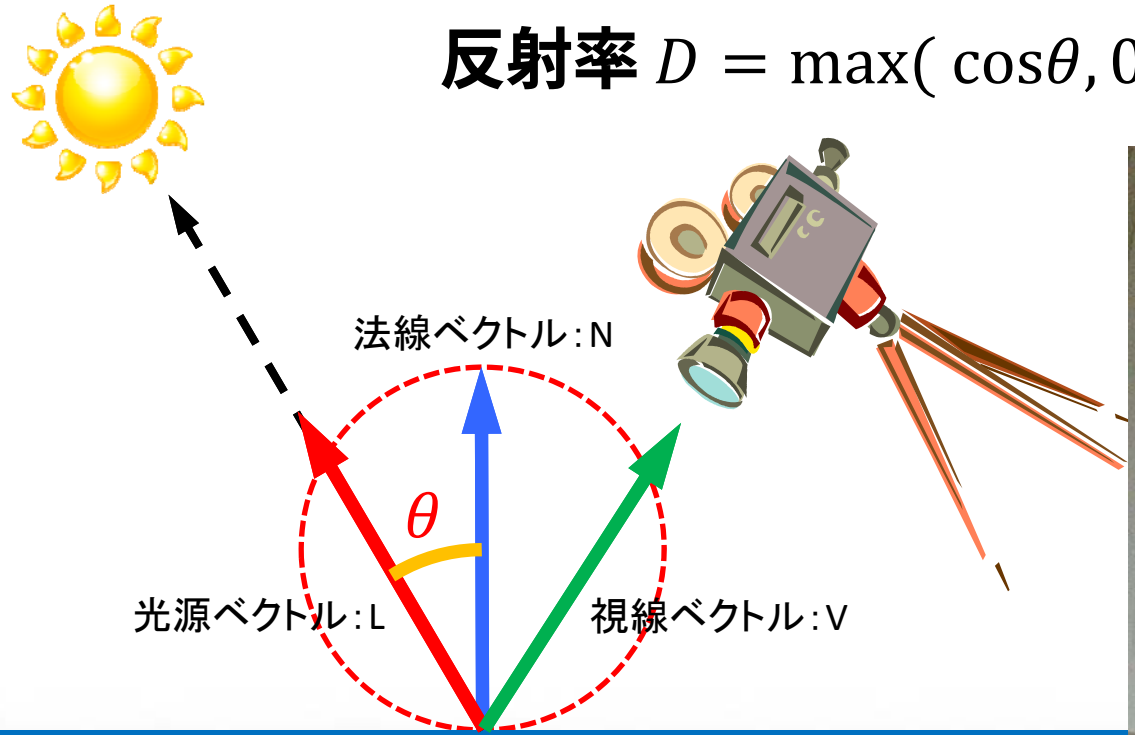


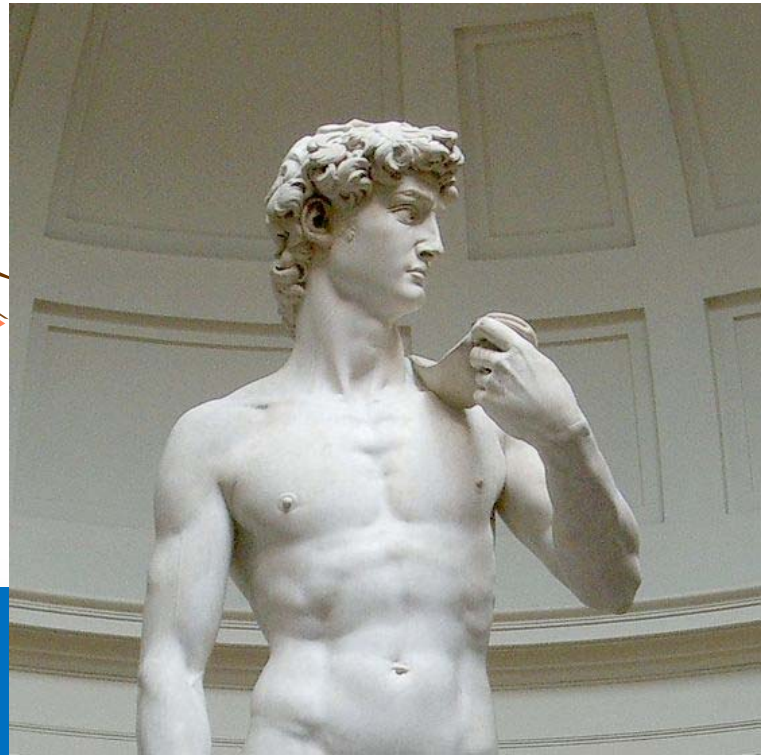
マテリアルの作り方. 初級



- ディフューズ反射のシェーダ
  - 石膏のような光沢のない滑らかな質感

反射率 $D = \max(\cos\theta, 0) = \max(\mathrm{N} \cdot \mathrm{L}, 0)$





マテリアルの作り方. 初級. 実装. クラス

- TMaterialSource にも同じフィールドを持たせる
  - 同じシェーダを複数のパラメータで使い回せる

```
TMyMaterial = class( TCustomMaterial )
protected
  _DiffColor :TAlphaColor;
  procedure SetDiffColor( const DiffColor_:TAlphaColor );
  procedure InitShaderV;
  procedure InitShaderP;
  procedure DoInitialize; override;
  procedure DoApply( const Context_:TContext3D ); override;
  class function DoGetMaterialProperty( const Prop_:TMaterial.TProperty ): string; override;
public
  constructor Create; override;
  property DiffColor: TAlphaColor read _DiffColor write SetDiffColor;
end;

TMyMaterialSource = class( TMaterialSource )
protected
  function GetDiffColor :TAlphaColor;
  procedure SetDiffColor( const DiffColor_:TAlphaColor );
  function CreateMaterial: TMaterial; override;
public
  property DiffColor: TAlphaColor read GetDiffColor write SetDiffColor;
end;
```

マテリアルの作り方. 初級. 実装. 初期化

- シェーダのロード

```
procedure TMyMaterial.DoInitialize;
begin
     inherited;
     InitShaderV;
     InitShaderP;
end;
```

シェーダのステージ毎に分けた（後述）

- デフォルト変数の命名

```
class function TMyMaterial.DoGetMaterialProperty( const Prop_:TMaterial.TProperty ): string;
begin
     case Prop_ of
       TProperty.ModelViewProjection       : Result := 'VS_FMatrixMVP';
       TProperty.ModelView                 : Result := 'VS_FMatrixMV';
       TProperty.ModelViewInverseTranspose : Result := 'VS_TIMatrixMV';
     else Result := '';
     end;
end;
```

- プロパティ

```
procedure TMyMaterial.SetDiffColor( const DiffColor_:TAlphaColor );
begin
     _DiffColor := DiffColor_;  DoChange;
end;
```

マテリアルの作り方. 初級. 実装. 初期化. バーテックスシェーダ

- プラットフォーム別のシェーダを用意

```
function FileToBytes( const FileName_:string ) :TBytes;
begin
     with TMemoryStream.Create do
     begin
          LoadFromFile( FileName_ );
          SetLength( Result, Size );
          Read( Result, Size );
          Free;
     end;
end;
```

```
procedure TMyMaterial.InitShaderV;
var
   CSS :array [ 0..1 ] of TContextShaderSource;
begin
     CSS[ 0 ] := TContextShaderSource.Create
       ( TContextShaderArch.DX9,
         FileToBytes( '..¥..¥》SHADER¥HLSL¥3.0¥MyMaterial.cvs' ),
         [ TContextShaderVariable.Create( 'VS_FMatrixMVP'   , TContextShaderVariableKind.Matrix,  00,  4 ),
           TContextShaderVariable.Create( 'VS_FMatrixMV'    , TContextShaderVariableKind.Matrix,  04,  4 ),
           TContextShaderVariable.Create( 'VS_TIMatrixMV'   , TContextShaderVariableKind.Matrix,  08,  4 ) ] );

     CSS[ 1 ] := TContextShaderSource.Create
       ( TContextShaderArch.DX11_level_9,
         FileToBytes( '..¥..¥》SHADER¥HLSL¥5.0¥MyMaterial.cvs' ),
         [ TContextShaderVariable.Create( 'VS_FMatrixMVP'   , TContextShaderVariableKind.Matrix, 000, 64 ),
           TContextShaderVariable.Create( 'VS_FMatrixMV'    , TContextShaderVariableKind.Matrix, 064, 64 ),
           TContextShaderVariable.Create( 'VS_TIMatrixMV'   , TContextShaderVariableKind.Matrix, 128, 64 ) ] );

     FVertexShader := TShaderManager.RegisterShaderFromData( 'MyMaterial.vs', TContextShaderKind.VertexShader, '', CSS );
end;
```

シェーダのロード

シェーダに必要な入力変数を登録する

変数名　変数型　入力位置　変数サイズ

マテリアルの作り方. 初級. 実装. 初期化. ピクセルシェーダ

```
procedure TMyMaterial.InitShaderP;
var
   CSS :array [ 0..1 ] of TContextShaderSource;
begin
     CSS[ 0 ] := TContextShaderSource.Create
       ( TContextShaderArch.DX9,
         FileToBytes( '..¥..¥》 SHADER¥HLSL¥3.0¥MyMaterial.cps' ),
         [ TContextShaderVariable.Create( 'PS_FMatrixMVP'   , TCo      tShaderVariableKind.Matrix,  00,  4 ),
           TContextShaderVariable.Create( 'PS_FMatrixMV'    , T      ntextShaderVariableKind.Matrix,  04,  4 ),
           TContextShaderVariable.Create( ‘PS_TIMatrixMV’   , TContextShaderVariableKind.Matrix,  08,  4 ),
           TContextShaderVariable.Create( 'PS_EyePos'       , TContextShaderVariableKind.Vector,  12,  1 ),
           TContextShaderVariable.Create( 'PS_DiffColor'    , TContextShaderVariableKind.Vector,  13,  1 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Opt', TContextShaderVariableKind.Vector,  14,  1 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Pos', TContextShaderVariableKind.Vector,  15,  1 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Dir', TContextShaderVariableKind.Vector,  16,  1 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Col', TContextShaderVariableKind.Vector,  17,  1 ) ] );

     CSS[ 1 ] := TContextShaderSource.Create
       ( TContextShaderArch.DX11_level_9,
         FileToBytes( '..¥..¥》 SHADER¥HLSL¥5.0¥MyMaterial.cps' ),
         [ TContextShaderVariable.Create( 'PS_FMatrixMVP'   , TContextShaderVariableKind.Matrix, 000, 64 ),
           TContextShaderVariable.Create( 'PS_FMatrixMV'    , TContextShaderVariableKind.Matrix, 064, 64 ),
           TContextShaderVariable.Create( ‘PS_TIMatrixMV’   , TContextShaderVariableKind.Matrix, 128, 64 ),
           TContextShaderVariable.Create( 'PS_EyePos'       , TContextShaderVariableKind.Vector, 192, 16 ),
           TContextShaderVariable.Create( 'PS_DiffColor'    , TContextShaderVariableKind.Vector, 208, 16 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Opt', TContextShaderVariableKind.Vector, 224, 16 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Pos', TContextShaderVariableKind.Vector, 240, 16 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Dir', TContextShaderVariableKind.Vector, 256, 16 ),
           TContextShaderVariable.Create( 'PS_Lights[0].Col', TContextShaderVariableKind.Vector, 272, 16 ) ] );

     FPixelShader := TShaderManager.R      rShaderFromData( 'MyMaterial.ps', TContextShaderKind.PixelShader, '', CSS );
end;
```

バーテックスシェーダと同名の変数は渡せない

DX9 ではレジスタ単位

DX11 ではバイト単位

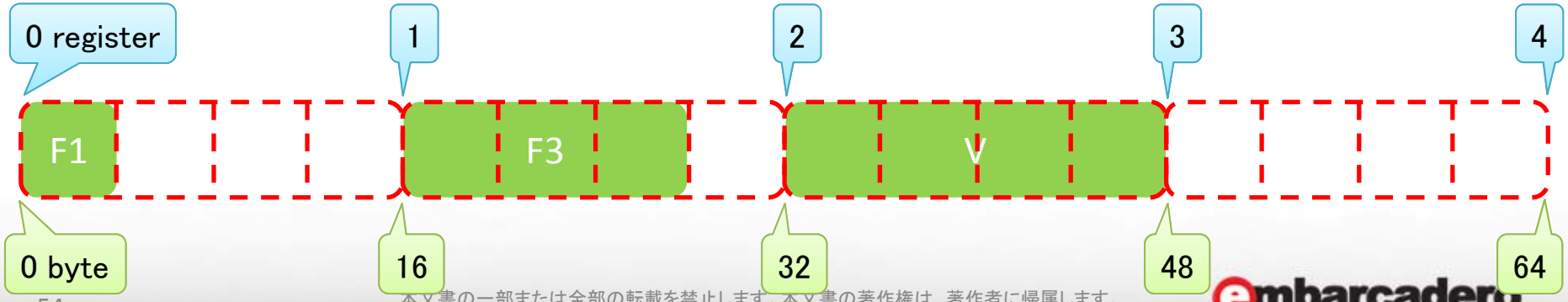
レコード型の配列も小分けにして渡せる

マテリアルの作り方. 初級. 実装. 変数型

- TContextShaderVariableKind

HLSLでの型名

  - Float : 4 byte : 1 register … float
  - Float2 : 8 byte : 1 register … float2
  - Float3 : 12 byte : 1 register … float3
  - Vector : 16 byte : 1 register … float4
  - Matrix : 64 byte : 4 register … float4x4
  - Texture : * : 1 register … Texture2D<float4>



マテリアルの作り方. 初級. 実装. 変数入力

- シェーダ内で用いる定数を外部から提供
  - VS_FMatrixMVP, VS_FMatrixMV, VS_TIMatrixMV は標準で渡される

```
procedure TMyMaterial.DoApply( const Context_:TContext3D );
begin
     inherited;
     with Context_ do
     begin
          SetShaderVariable( 'PS_FMatrixMVP', CurrentModelViewProjectionMatrix );
          SetShaderVariable( 'PS_FMatrixMV' , CurrentMatrix );
          SetShaderVariable( 'PS_TIMatrixMV', CurrentMatrix.Inverse.Transpose );
          SetShaderVariable( 'PS_EyePos'    , [ CurrentCameraInvMatrix.M[ 3 ] ] );
          SetShaderVariable( 'PS_DiffColor' , _DiffColor );
          with Lights[ 0 ] do
          begin
               SetShaderVariable( 'PS_Lights[0].Opt',
                                  [ TVector3D.Create( Integer( LightType ) + 1,
                                    Cos( DegToRad( SpotCutoff ) ),
                                    SpotExponent, 0 ) ] );
               SetShaderVariable( 'PS_Lights[0].Pos', [ Position ] );
               SetShaderVariable( 'PS_Lights[0].Dir', [ Direction ] );
               SetShaderVariable( 'PS_Lights[0].Col', Color );
          end;
     end;
end;
```

変数名

入力値

ワールド座標系での視点位置

ライトの数だけ繰り返す

マテリアルの作り方. 初級. 実装. バーテックスシェーダ

- モデルの頂点毎の情報を加工して出力

変数位置（レジスタ単位）

```
float4x4  FMatrixMVP : register( c00 );  static float4x4 _FMatrixMVP = transpose( FMatrixMVP );
float4x4  FMatrixMV  : register( c04 );  static float4x4 _FMatrixMV  = transpose( FMatrixMV  );
float4x4  IMatrixMV  : register( c08 );  static float4x4 _IMatrixMV  = transpose( IMatrixMV  );

struct TSender { float4 Pos :POSITION;
                 float4 Nor :NORMAL;
                 float4 Tex :TEXCOORD; };

struct TResult { float4 Scr :SV_Position;
                 float4 Pos :TEXCOORD0;
                 float4 Nor :NORMAL;
                 float4 Tex :TEXCOORD1; };

TResult main( TSender _Sender ) {
    TResult _Result;
    _Result.Scr = mul( _Sender.Pos, _FMatrixMVP );
    _Result.Pos = mul( _Sender.Pos, _FMatrixMV  );
    _Result.Nor = mul( _Sender.Nor, _IMatrixMV  );
    _Result.Tex =      _Sender.Tex;
    return _Result;
}
```

頂点の情報

ローカル座標系での位置

スクリーン座標系での位置

ピクセルの情報

ワールド座標系での位置

頂点位置をスクリーン座標系へ変換

頂点位置をワールド座標系へ変換

法線ベクトルをワールド座標系へ変換

マテリアルの作り方. 初級. 実装. ピクセルシェーダ

- ピクセル毎の情報を加工して出力

```
struct Tlight {  float4 Opt;  float4 Pos;  float4 Dir;  float4 Col;  };

float4x4  FMatrixMVP : register( c00 );  static float4x4 _FMatrixMVP = transpose( FMatrixMVP );
float4x4  FMatrixMV  : register( c04 );  static float4x4 _FMatrixMV  = transpose( FMatrixMV  );
float4x4  IMatrixMV  : register( c08 );  static float4x4 _IMatrixMV  = transpose( IMatrixMV  );
float4    _EyePos    : register( c12 );
float4    _DiffColor : register( c13 );
TLight    _Light[1]  : register( c14 );

struct TSender {  float4 Scr :SV_Position;  float4 Pos :TEXCOORD0;
                  float4 Nor :NORMAL;       float4 Tex :TEXCOORD1; };
struct TResult {  float4 Col :SV_Target;  };

TResult main( TSender _Sender )
{
    TResult _Result;
    float3 N = normalize( _Sender.Nor.xyz );
    float3 L = -_Light[0].Dir.xyz;
    float  D = max( dot( L, N ), 0.0 );
    _Result.Col.r = _Light[0].Col.r * D * _DiffColor.r;
    _Result.Col.g = _Light[0].Col.g * D * _DiffColor.g;
    _Result.Col.b = _Light[0].Col.b * D * _DiffColor.b;
    _Result.Col.a = 1.0;
    return _Result;
}
```

レコード型も扱える

バーテックスシェーダの出力と同じ内容のレコード型

ピクセルシェーダの出力は色

ピクセルの法線ベクトル：N

ピクセルの光源ベクトル：L

不透明度値

反射率

マテリアルの作り方. 初級. 実装. コンパイル

- Windows SDK 付属の fxc.exe を利用
  - DX11版 のソースから DX9版 のバイナリも出力可能
    - Shader Model 2.0：DirectX 9.0
    - Shader Model 3.0：DirectX 9.0c
    - Shader Model 4.0：DirectX 10
      - *s_4_0_level_9_1：SM2.0 でコンパイル
      - *s_4_0_level_9_3：SM3.0 でコンパイル
    - Shader Model 5.0：DirectX 11

DX9 モードの強制
`FMX.Types.GlobalUseDX := False;`

32bit版 / 64bit版

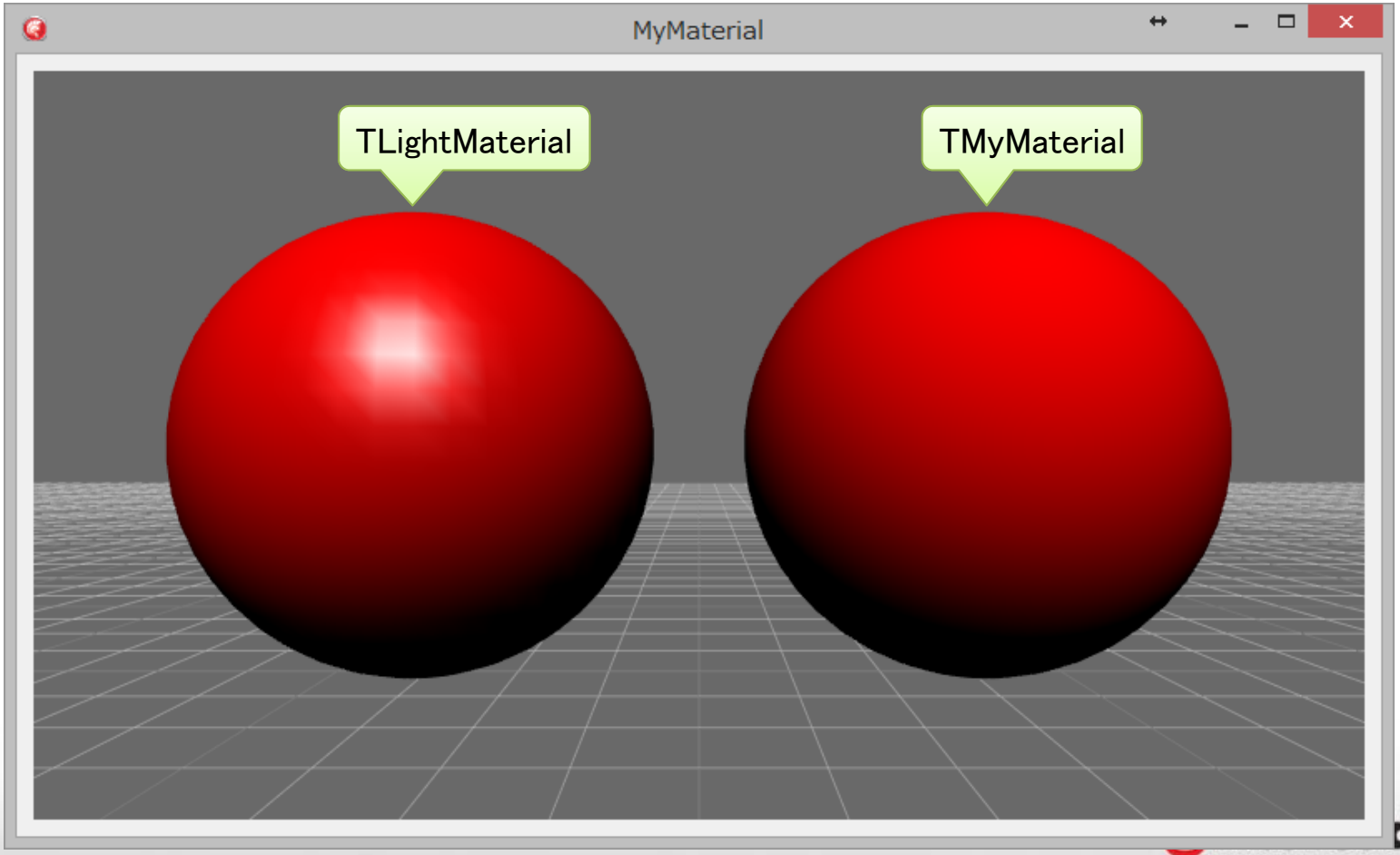
```
rem set FXC="C:¥Program Files (x86)¥Windows Kits¥8.1¥bin¥x86¥fxc.exe"
    set FXC="C:¥Program Files (x86)¥Windows Kits¥8.1¥bin¥x64¥fxc.exe"

for %%i in ( *.vs ) do ( %FXC% %%i /T vs_3_0 /Fo 3.0¥%%~ni.cvs )
for %%i in ( *.ps ) do ( %FXC% %%i /T ps_3_0 /Fo 3.0¥%%~ni.cps )

for %%i in ( *.vs ) do ( %FXC% %%i /T vs_5_0 /Fo 5.0¥%%~ni.cvs )
for %%i in ( *.ps ) do ( %FXC% %%i /T ps_5_0 /Fo 5.0¥%%~ni.cps )
```

Shader Model 3.0

Shader Model 5.0

マテリアルの作り方. 初級. 完成

- 全体的な陰影は TLightMaterialSource と同等



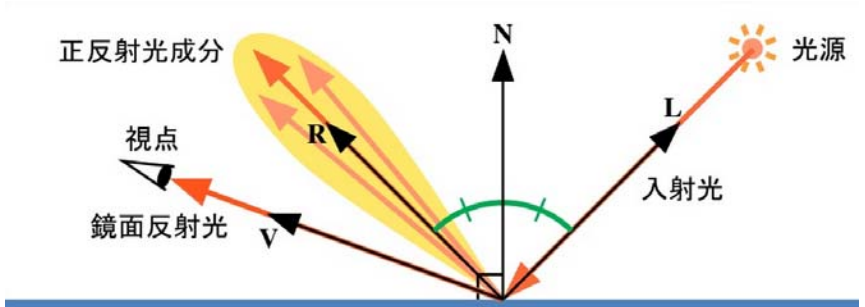
マテリアルの作り方. 中級

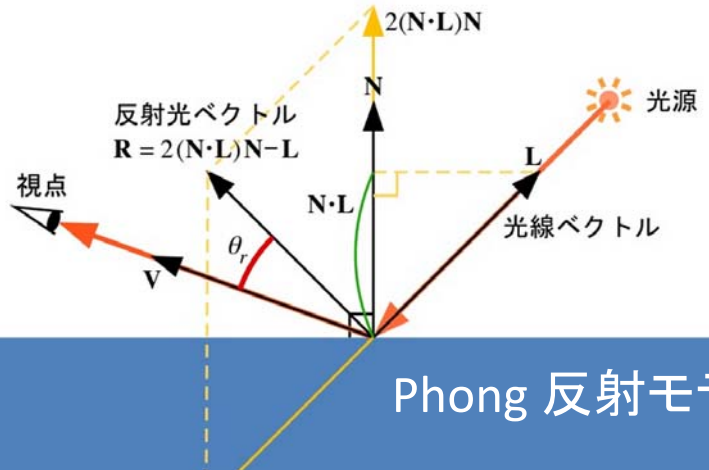
- 光沢(スペキュラ)の表現
  - Phong 反射モデル

    反射率 $= \max(\cos\theta_r, 0)^K$
  - Blinn-Phong 反射モデル

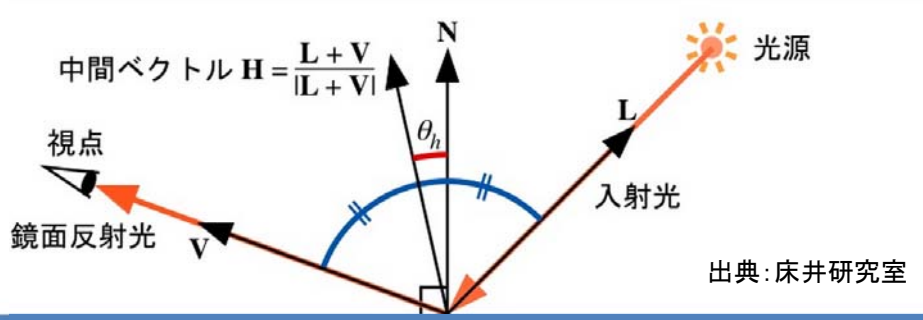
    反射率 $= \max(\cos\theta_h, 0)^K$

※ K：**大きいほど光沢が鋭い**





Phong 反射モデル



Blinn-Phong 反射モデル

マテリアルの作り方. 中級. 実装. ピクセルシェーダ

- ピクセル毎の情報を加工して出力

```
TResult main( TSender _Sender )
{
    TResult _Result;

    float3 N = normalize( _Sender.Nor.xyz );
    float3 L = -_Light[0].Dir.xyz;
    float3 E = normalize( _EyePos.xyz - _Sender.Pos.xyz );
    float3 H = normalize( L + E );

    float  D = max( dot( L, N ), 0.0 );
    float  S = pow( max( dot( H, N ), 0.0 ), _SpecShiny );

    _Result.Col.r = _Light[0].Col.r * D * ( _DiffColor.r + _SpecColor.r * S );
    _Result.Col.g = _Light[0].Col.g * D * ( _DiffColor.g + _SpecColor.g * S );
    _Result.Col.b = _Light[0].Col.b * D * ( _DiffColor.b + _SpecColor.b * S );

    _Result.Col.a = 1.0;

    return _Result;
}
```

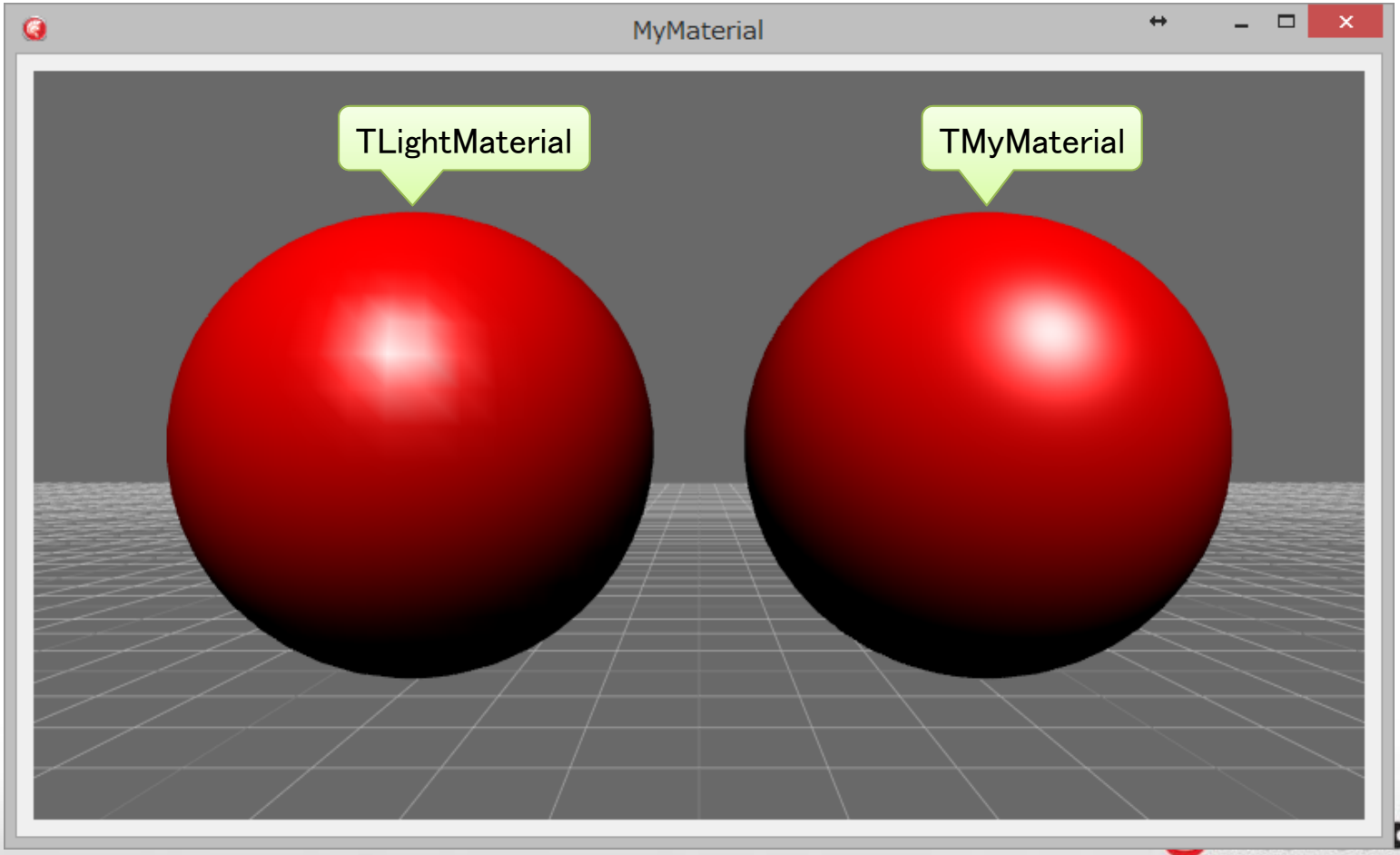
ピクセル位置からの視線ベクトル

スペキュラの反射率

ディフューズ ＋ スペキュラ

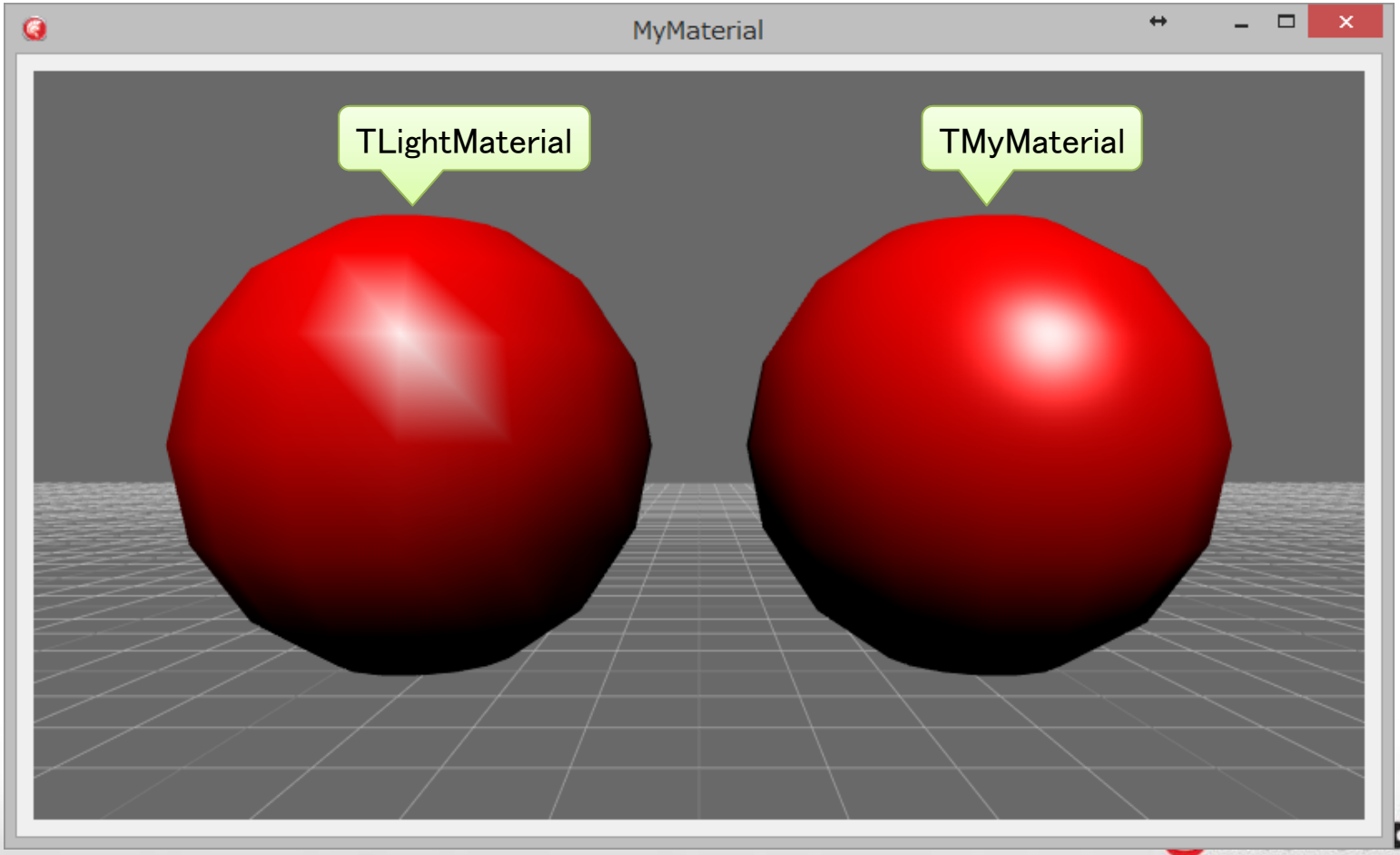
マテリアルの作り方. 中級. 完成

- TMyMaterialSource の方がスペキュラが綺麗？



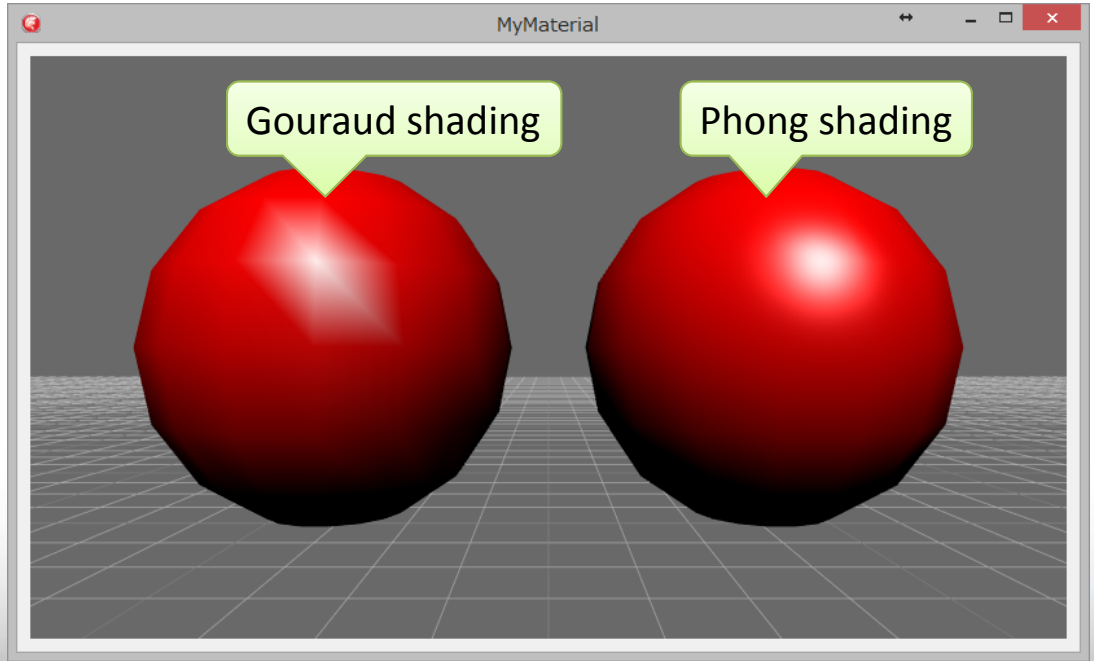
マテリアルの作り方. 中級. ローポリ

- TLightMaterialSource はポリゴン数で品質が変化



マテリアルの作り方. バーテックス vs ピクセル

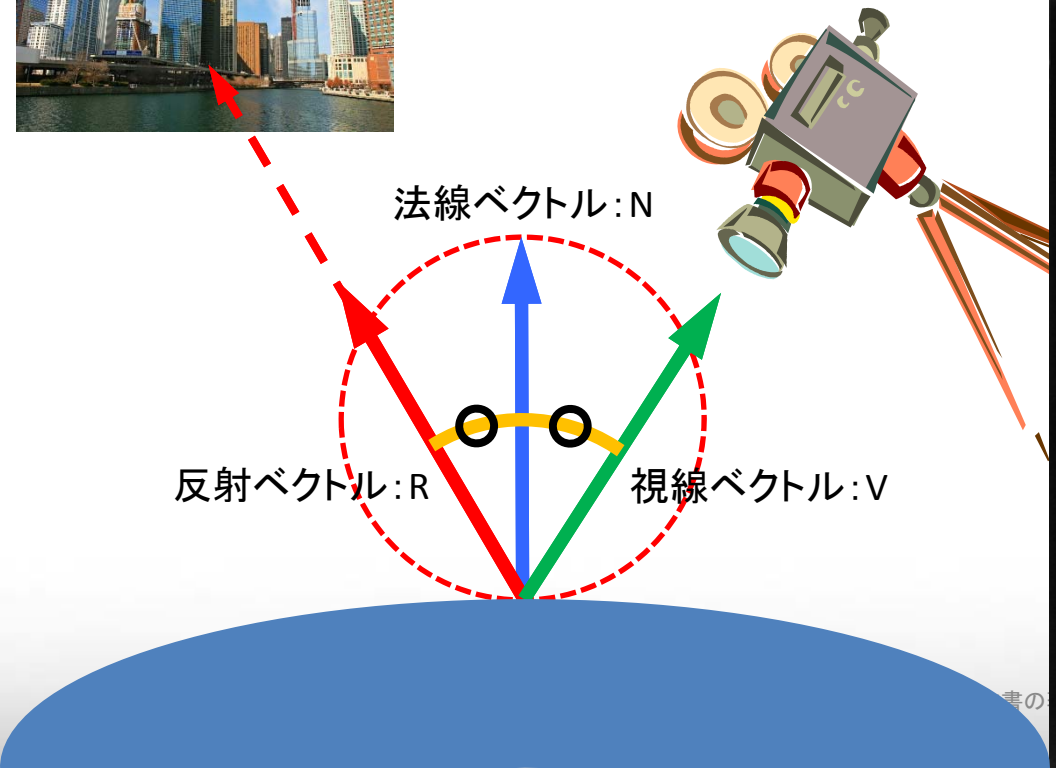
- シェーディングの構成方法は2種類
  - グローシェーディング　…バーテックスシェーダベース
    - 頂点だけで照明計算を行う
  - フォンシェーディング　…ピクセルシェーダベース
    - すべてのピクセルにおいて照明計算を行う

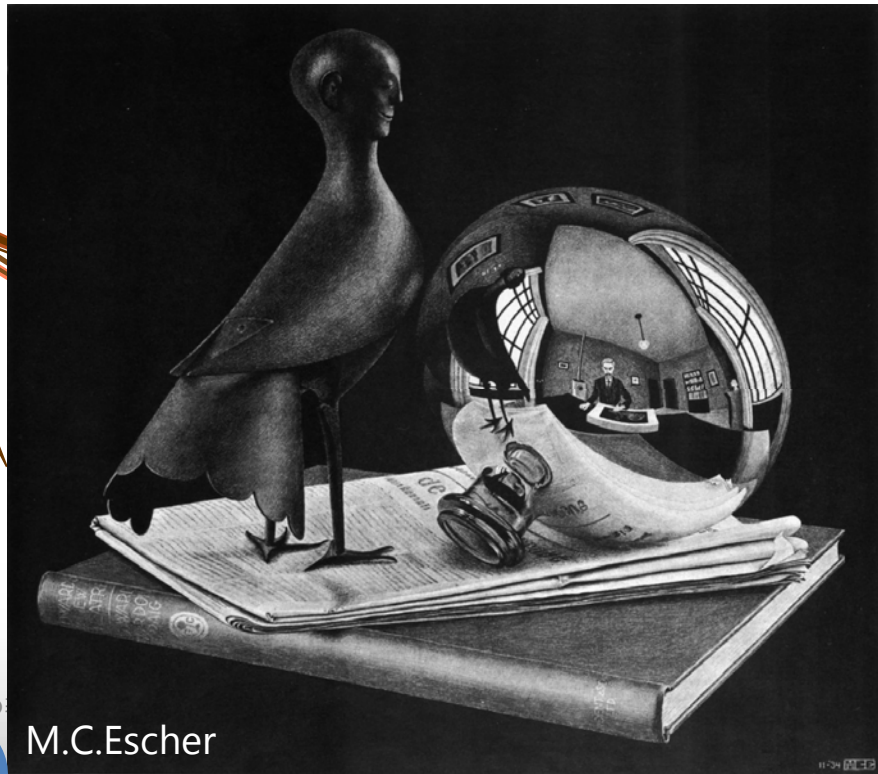


マテリアルの作り方. 上級

- 鏡面反射のシェーダ
  - 磨き上げた金属のような質感

$$R = 2(N \cdot V)N - V$$





マテリアルの作り方. 上級. 実装. 初期化. ピクセルシェーダ

- テクスチャ用のレジスタは定数とは別管理

```
procedure TMyMaterial.InitShaderP;
var
   CSS :array [ 0..1 ] of TContextShaderSource;
begin
     CSS[ 0 ] := TContextShaderSource.Create
       ( TContextShaderArch.DX9,
         FileToBytes( '..¥..¥》 SHADER¥HLSL¥3.0¥MyMaterial.cps' ),
         [ TContextShaderVariable.Create( ‘ PS_FMatrixMVP ’   , TContextShaderVariableKind.Matrix ,  00,  4 ),
           TContextShaderVariable.Create( ‘ PS_FMatrixMV ’    , TContextShaderVariableKind.Matrix ,  04,  4 ),
           TContextShaderVariable.Create( ‘ PS_TIMatrixMV ’   , TContextShaderVariableKind.Matrix ,  08,  4 ),
           TContextShaderVariable.Create( ‘ PS_EyePos ’       , TContextShaderVariableKind.Vector ,  12,  1 ),
           TContextShaderVariable.Create( ‘ PS_Texture ’      , TContextShaderVariableKind.Texture,   0,  0 ) ]  );

     CSS[ 1 ] := TContextShaderSource.Create
       ( TContextShaderArch.DX11_level_9,
         FileToBytes( '..¥..¥》 SHADER¥HLSL¥5.0¥MyMaterial.cps' ),
         [ TContextShaderVariable.Create( ‘ PS_FMatrixMVP ’   , TContextShaderVariableKind.Matrix , 000, 64 ),
           TContextShaderVariable.Create( ‘ PS_FMatrixMV ’    , TContextShaderVariableKind.Matrix , 064, 64 ),
           TContextShaderVariable.Create( ‘ PS_TIMatrixMV ’   , TContextShaderVariableKind.Matrix , 128, 64 ),
           TContextShaderVariable.Create( ‘ PS_EyePos ’       , TContextShaderVariableKind.Vector , 192, 16 ),
           TContextShaderVariable.Create( ‘ PS_Texture ’      , TContextShaderVariableKind.Texture,   0,  0 ) ] );

     FPixelShader := TShaderManager.RegisterShaderFromData( 'MyMaterial.ps', TContextShaderKind.PixelShader, '', CSS );
end;
```

テクスチャには別に番号を振る

マテリアルの作り方. 上級. 実装. 変数入力

- テクスチャとして渡せるのは TTexture のみ

```
procedure TMyMaterial.DoApply( const Context_:TContext3D );
begin
     inherited;
     with Context_ do
     begin
          SetShaderVariable( ‘PS_FMatrixMVP', CurrentModelViewProjectionMatrix );
          SetShaderVariable( 'PS_FMatrixMV' , CurrentMatrix );
          SetShaderVariable( 'PS_TIMatrixMV', CurrentMatrix.Inverse.Transpose );
          SetShaderVariable( 'PS_EyePos'    , [ CurrentCameraInvMatrix.M[ 3 ] ] );
          SetShaderVariable( 'PS_Texture'   , _Texture );
     end;
end;
```

TTexture

  - TBitmap は TTextureBitmap を経由して変換

```
var
  B  :TBitmap;
  TB :TTextureBitmap;
  T  :TTexture;
```

```
begin
  B.LoadFromFile( ‘～’ );
  TB.Assign( B );
  T := TB.Texture;
```

```
begin
  TB.LoadFromFile( ‘～’ );
  T := TB.Texture;
```

マテリアルの作り方. 上級. 実装. ピクセルシェーダ

- ピクセル毎の情報を加工して出力

```
Texture2D<float4> _Texture : register( t0 );

SamplerState _SamplerState {};

TResult main( TSender _Sender )
{
    TResult _Result;
    float3 E = normalize( _EyePos.xyz - _Sender.Pos.xyz );
    float3 N = normalize( _Sender.Nor.xyz );
    float3 R = normalize( reflect( E, N ) );
    _Result.Col.rgb = _Texture.Sample( _SamplerState, VectorToSky( R ) ).rgb;
    _Result.Col.a   = 1.0;
    return _Result;
}

float2 VectorToSky( float3 _Vector )
{
    float2 _Result;
    _Result.x = ( Pi - atan2( _Vector.z, _Vector.x ) ) / Pi2;
    _Result.y = acos( _Vector.y ) / Pi;
    return _Result;
}
```

テクスチャ用レジスタは t* で指定

テクスチャから色を採る際の諸設定

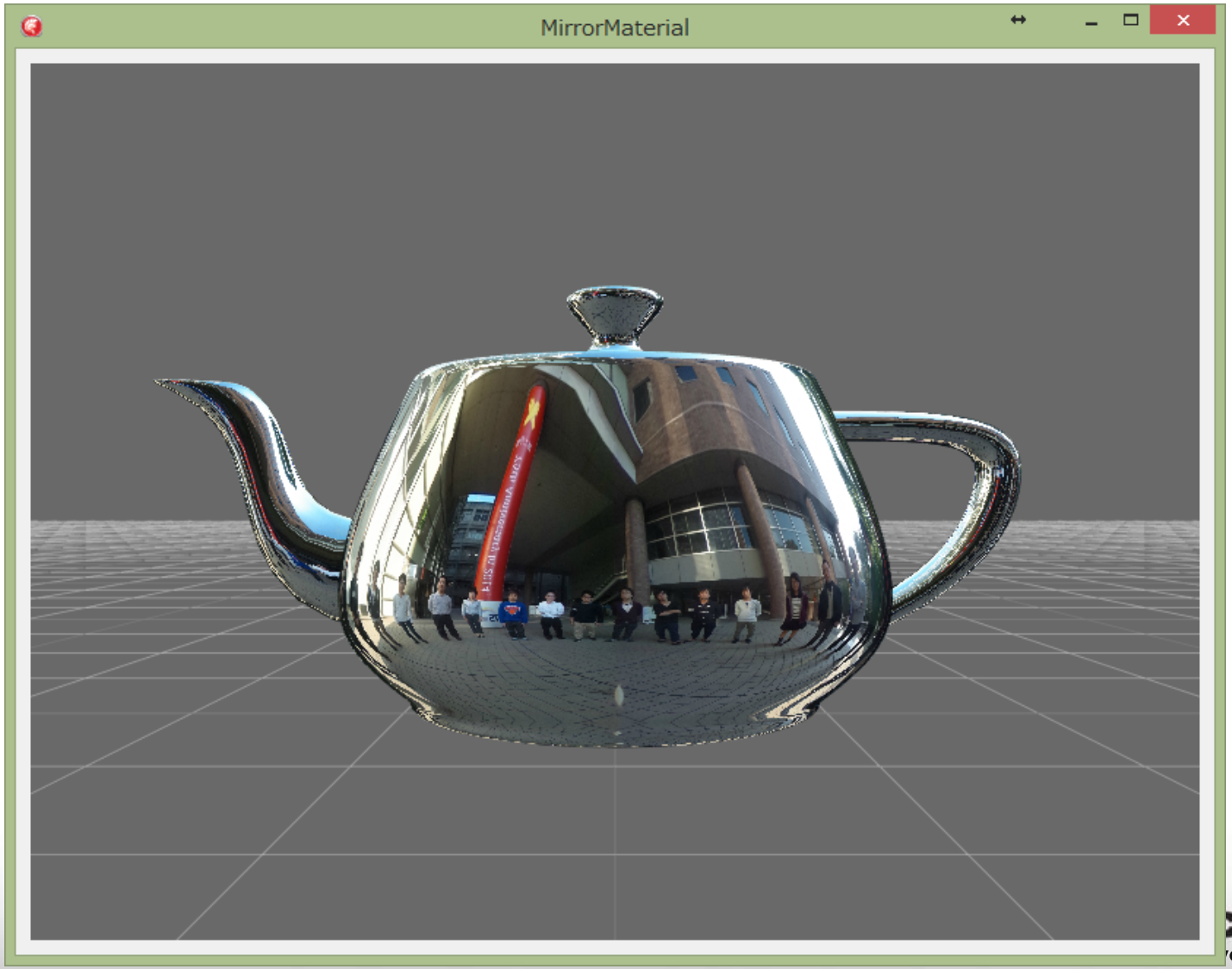
反射ベクトルを算出する標準関数

テクスチャから色を取得するメソッド

RGB をベクトルとして演算することも可能

ベクトルの方向を緯度/経度に換算する関数

マテリアルの作り方. 上級. 完成



謝辞

- シェーダ構造解明の先駆者
  - 個人サイト「全力わはー」
    - http://d.hatena.ne.jp/tales
      - MMD4Delphi 0.0.1
      - LibOVR wrapper for Delphi 0.0.1

Lyna @lynatan · 17 時間
Windows上でもFireMonkeyでOpenGL（とGLSL）を使えるようにする独自TContext3D実装できたー。参考にしたFMX.Context.Macの実装がOpenGL2.0相当で古かったから無理矢理3.3で書いてみた pic.twitter.com/S95ebDIxYd

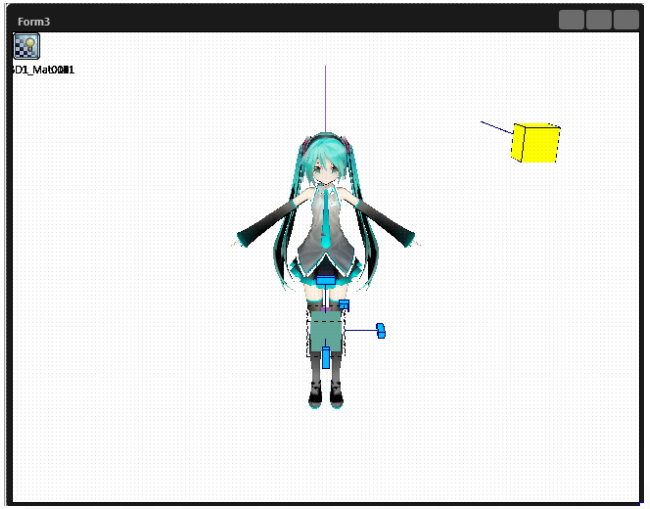
1 1

Lyna
@lynatan フォローされています

Delphiで何かしら作ったり直したり壊したりしてます。CardWirthNextに関するご意見等はこちらへ bit.ly/1bnmD3h

Aichi, Japan
twc.xrea.jp
2009年5月に登録

DelphiでMMD、回転付与とVMDの表情に対応して多少見た目が良くなった。あと再生フレーム数を30FPS固定から時間ベースに変えたから曲流しても同期して踊れるようになった> <




おわり