Computer Graphics

Model-View Transformation

Konstantin Tretyakov
kt@ut.ee

Sep 25, 2013

In the previous episodes

- Vectors & Tools

Linear transformations

Each linear transformation corresponds to a matrix.

Linear transformations

Each linear transformation corresponds to a matrix.

Columns of a matrix show how it transforms the canonical basis

Orthogonal transformations

To compute the inverse of an orthogonal matrix, simply transpose it.

Quiz

- How does this matrix transform the (canonical) basis?

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

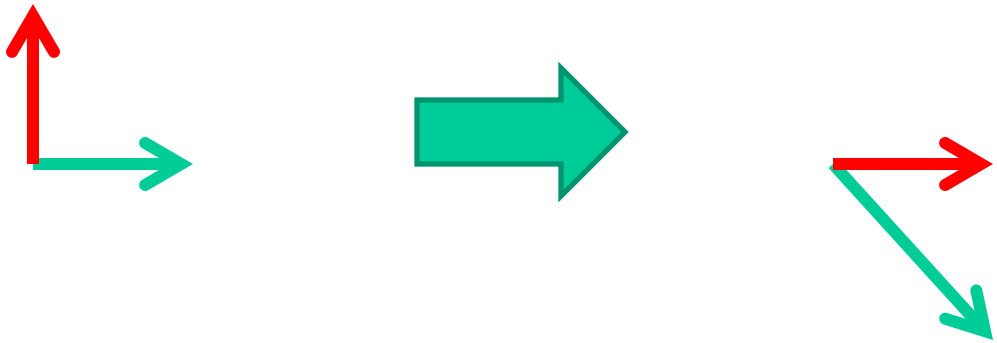
Quiz

- How does this matrix transform the (canonical) basis?

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

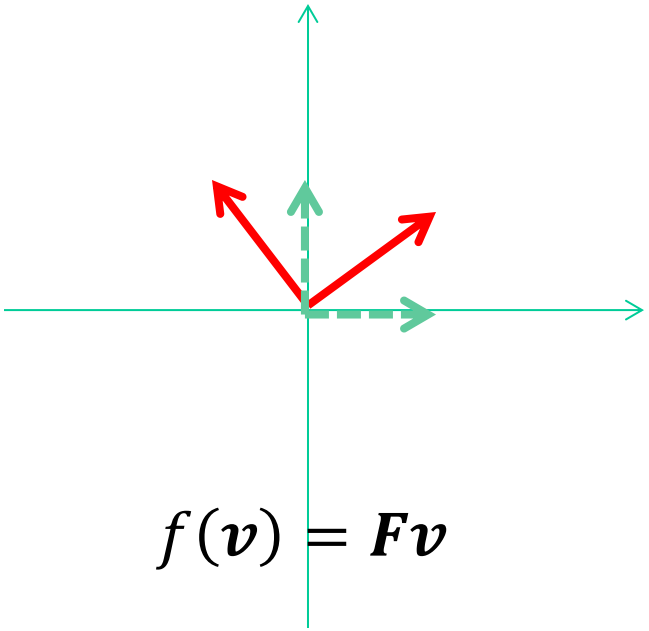
Quiz

- Which matrix does the following?



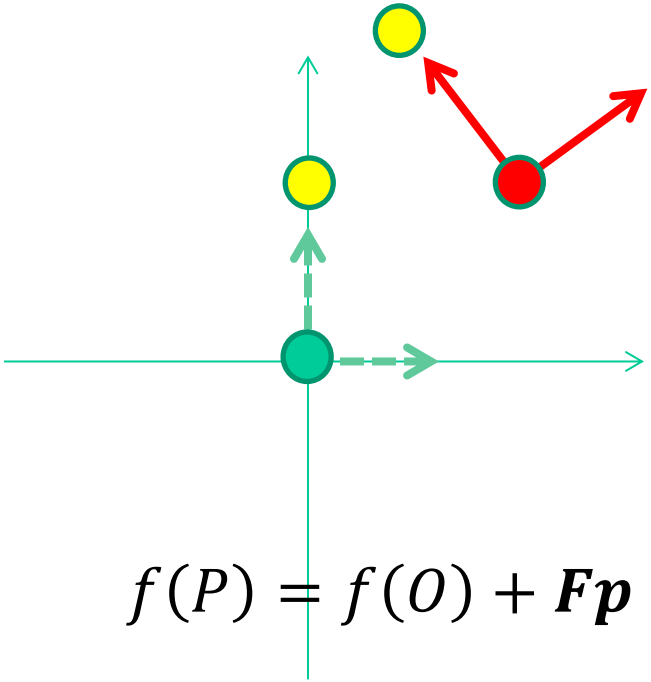
Affine space

- Vector space



$$f(\boldsymbol{v}) = \boldsymbol{F}\boldsymbol{v}$$

- Affine space



$$f(P) = f(O) + \boldsymbol{F}\boldsymbol{p}$$

Quiz

- Represent the point $(1, 2)^T$ in homogeneous coordinates.

Quiz

- Represent the vector $(1, 2)^T$ in homogeneous coordinates.

Quiz

- Which matrix performs a shift by $(1, 1)$ in homogeneous coordinates?

Quiz

- Which matrix performs a rotation by α, followed by a shift by $(1, 1)$ in homogeneous coordinates?

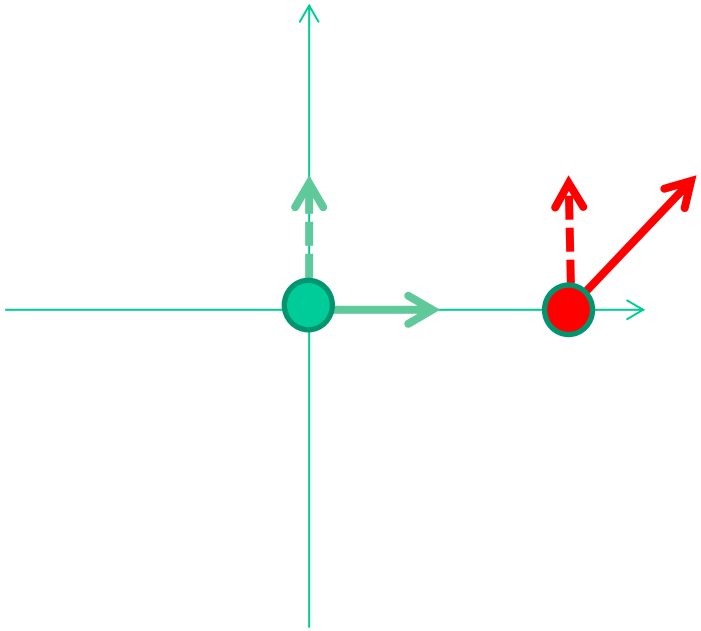
Quiz

- How does this matrix transform the (canonical) frame?

$$\left(\begin{array}{cc|c} 0 & 1 & 0 \\ 1 & 0 & 1 \\ \hline 0 & 0 & 1 \end{array}\right)$$

Quiz

- Which matrix performs this transformation (in homogeneous coordinates)?



Mathematical background

- Matrices:
  - Linear transformations
  - Invertibility, rank, determinant
  - Orthogonal transformations
  - Affine transformations
  - Homogeneous coordinates

OpenGL

- Graphics API

B

glBegin
glBeginQuery
glBindAttribLocation
glBindBuffer
glBindTexture
glBitmap
glBlendColor
glBlendEquation
glBlendEquationSeparate
glBlendFunc
glBlendFuncSeparate
glBufferData
glBufferSubData

OpenGL

- Graphics API
- Hardware (driver) interface

OpenGL

- Graphics API
- Hardware (driver) interface
- Lingua-franca of computer graphics

OpenGL

- Created in **1992** by  (now )
- Still in active development by the Khronos consortium (ATI, SGI, Intel, NVIDIA, Sun, + a 100 others)

- Current version: OpenGL **4.4**

OpenGL pre-3.0 and post 3.1

- Revision 3.0 in 2008 deprecated a lot of the core features

- Application-generated object names
- Color index mode
- Shading language 1.10 and 1.20
- Begin/End primitive specification
- Edge flags
- Fixed function vertex processing
- Client-side vertex arrays
- Rectangles
- Current raster position
- Two-sided color selection
- Non-sprite points
- Wide lines and line stripple
- Quadrilateral and polygon primitives
- Separate polygon draw mode
- Polygon stripple
- Pixel transfer modes and operations
- Pixel drawing
- Bitmaps
- Legacy OpenGL 1.0 pixel formats
- Legacy pixel formats
- Depth texture mode
- Texture wrap mode CLAMP
- Texture borders
- Automatic mipmap generation
- Fixed function fragment processing
- Alpha test
- Accumulation buffers
- Context framebuffer size queries
- Evaluators
- Selection and feedback mode
- Display lists
- Hints
- Attribute stacks
- Unified extension string

OpenGL pre-3.0 and post 3.1

- Revision 3.0 in 2008 deprecated a lot of the core features

- Applicatio
- Color inde
- Shading la
- Begin/End
- Edge flags
- Fixed func
- Client-side
- Rectangle
- Current ra
- Two-sided
- Non-sprite

eneration
nt processing
ize queries
ck mode
ng

Disclaimer

Nearly all of the following examples will be purely pre-3.0 stuff.

Later we shall gently turn to the modern features.

OpenGL pre-3.0 and post 3.1

- Revision 3.0 in 2008 deprecated a lot of the core features

- Applicatio
- Color inde
- Shading la
- Begin/End
- Edge flags
- Fixed func
- Client-side
- Rectangle
- Current ra
- Two-sided
- Non-sprite

eneration
nt processing
ize queries
ck mode
ng

Disclaimer

Nearly all of the following examples will be purely pre-3.0 stuff.

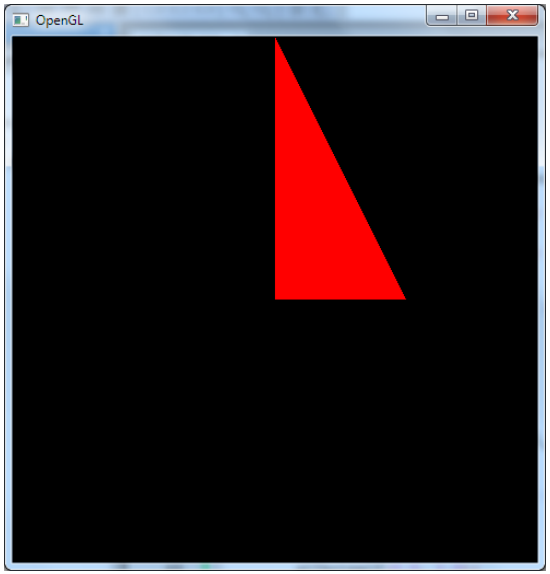
Later we shall gently turn to the modern features.

OpenGLES and **WebGL** resemble the newer specification

OpenGL Example

- The following code draws a triangle with vertices $(0, 0)^T$, $(0.5, 0)^T$, $(0, 1)^T$.

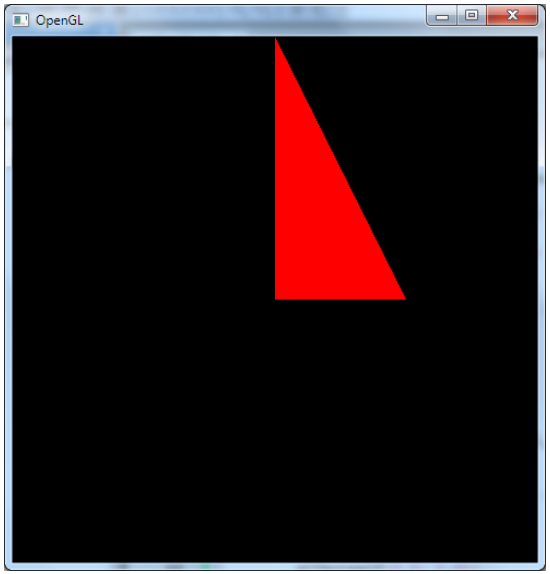
```
glBegin(GL_TRIANGLES);
     glVertex2f(0.0, 0.0);
     glVertex2f(0.5, 0.0);
     glVertex2f(0.0, 1.0);
glEnd();
```



OpenGL Example

- The following code draws a triangle with vertices $(0,0)^T$, $(0.5,0)^T$, $(0,1)^T$.

```
glBegin(GL_TRIANGLES);
     glVertex2f(0.0, 0.0);
     glVertex2f(0.5, 0.0);
     glVertex2f(0.0, 1.0);
glEnd();
```

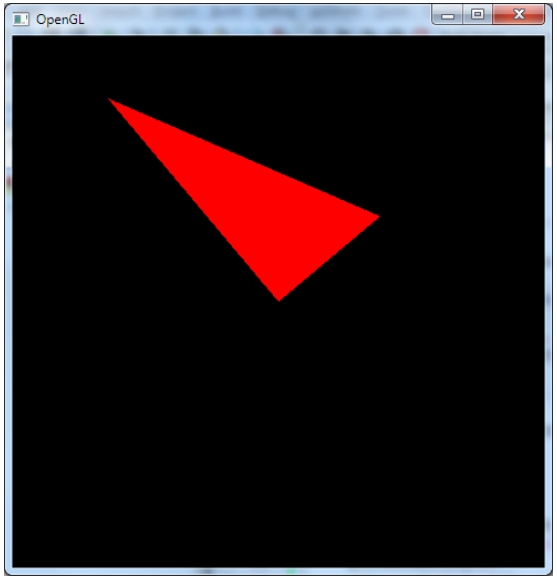



```
glVertex{2,3,4}{f,d,i}[v]
```

OpenGL Example

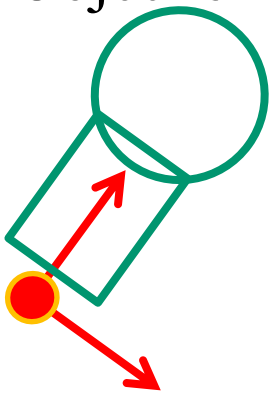
- The following code draws the same triangle, rotated by 40 degrees.

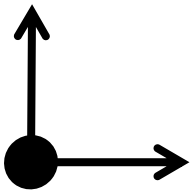
```
glRotatef(40, 0.0, 0.0, 1.0);
glBegin(GL_TRIANGLES);
      glVertex2f(0.0, 0.0);
      glVertex2f(0.5, 0.0);
      glVertex2f(0.0, 1.0);
glEnd();
```



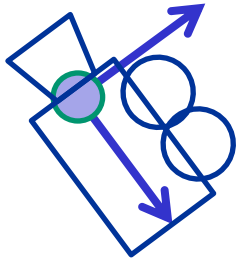
World/Object/Camera frames

Object's frame





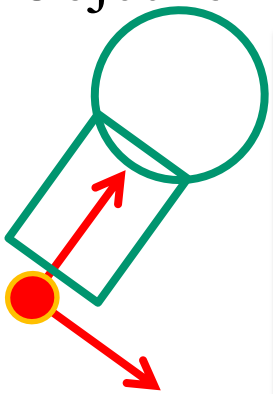
World frame



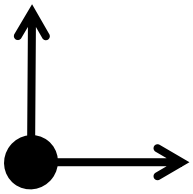
Camera frame

World/Object/Camera frames

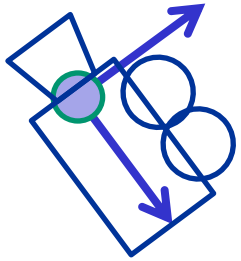
Object's frame



The object is described in its own frame using vertices $p_1, p_2, \ldots$



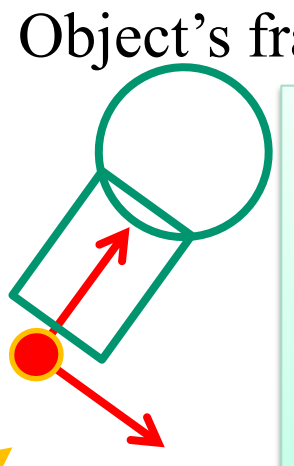
World frame



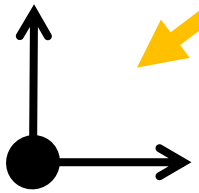
Camera frame

World/Object/Camera frames

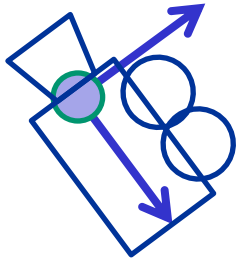
The position of the object's frame wrt the world frame is given by the (affine) *modeling transform* matrix $\boldsymbol{M}$.



The object is described in its own frame using vertices $\boldsymbol{p_1}, \boldsymbol{p_2}, \ldots$



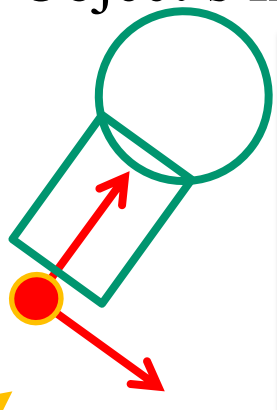
World frame



Camera frame

World/Object/Camera frames

Object's frame

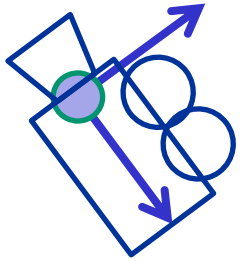


The position of the object's frame wrt the world frame is given by the (affine) *modeling transform* matrix $\boldsymbol{M}$.

The object is described in its own frame using vertices $\boldsymbol{p_1}, \boldsymbol{p_2}, \ldots$.

In world coordinates, the object's vertices are therefore $\boldsymbol{Mp_1}, \boldsymbol{Mp_2}, \ldots$

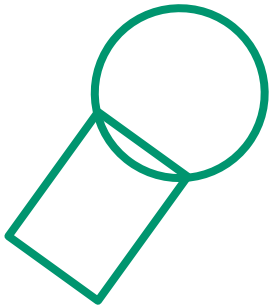
World frame

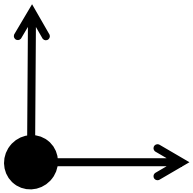


Camera frame

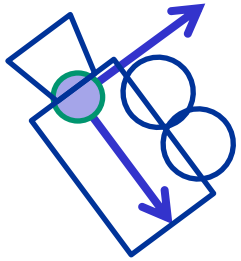
World/Object/Camera frames

In world coordinates, the object's vertices are therefore
$\boldsymbol{Mp_1}, \boldsymbol{Mp_2}, \ldots$





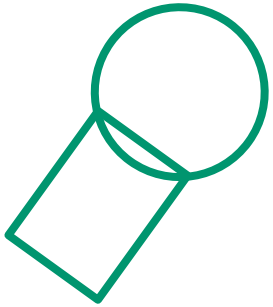
World frame



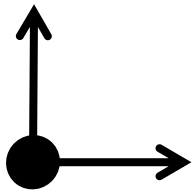
Camera frame

World/Object/Camera frames

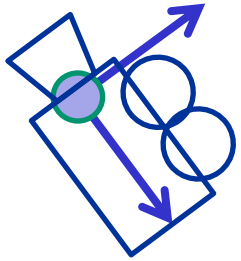
In world coordinates, the object's vertices are therefore
$\boldsymbol{Mp_1}, \boldsymbol{Mp_2}, \ldots$



The world is observed via a camera.



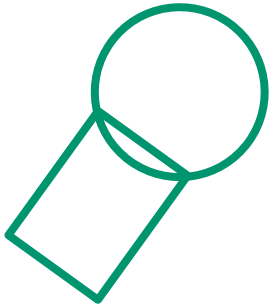
World frame



Camera frame

World/Object/Camera frames

In world coordinates, the object's vertices are therefore $\boldsymbol{Mp_1}, \boldsymbol{Mp_2}, \ldots$



The world is observed via a camera.

Transformation from world coordinates to camera coordinates is given by the *view matrix* $\boldsymbol{V}$

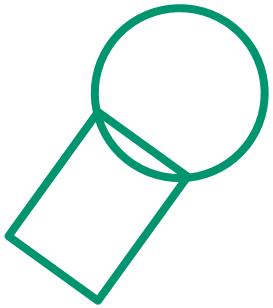
World frame

Camera frame

World/Object/Camera frames

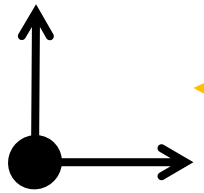
In world coordinates, the object's vertices are therefore
$\boldsymbol{Mp_1, Mp_2, \ldots.}$

Object's vertices in camera coordinates are
$\boldsymbol{VMp_1, VMp_2, \ldots.}$

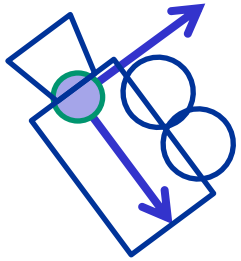


The world is observed via a camera.

Transformation from world coordinates to camera coordinates is given by the *view matrix* $\boldsymbol{V}$

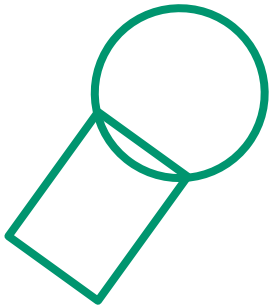


World frame



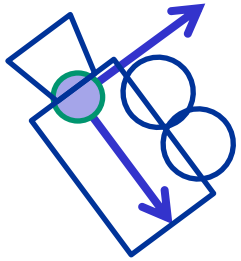
Camera frame

World/Object/Camera frames



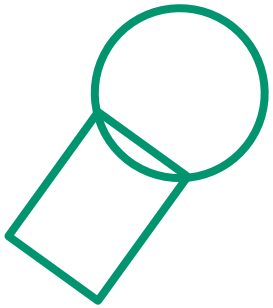
Object's vertices in camera coordinates are

$$\boldsymbol{VMp_1}, \boldsymbol{VMp_2}, \ldots.$$



Camera frame

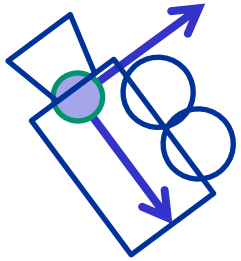
World/Object/Camera frames



Object's vertices in camera coordinates are

$VMp_1, VMp_2, \ldots.$

Model-view transform



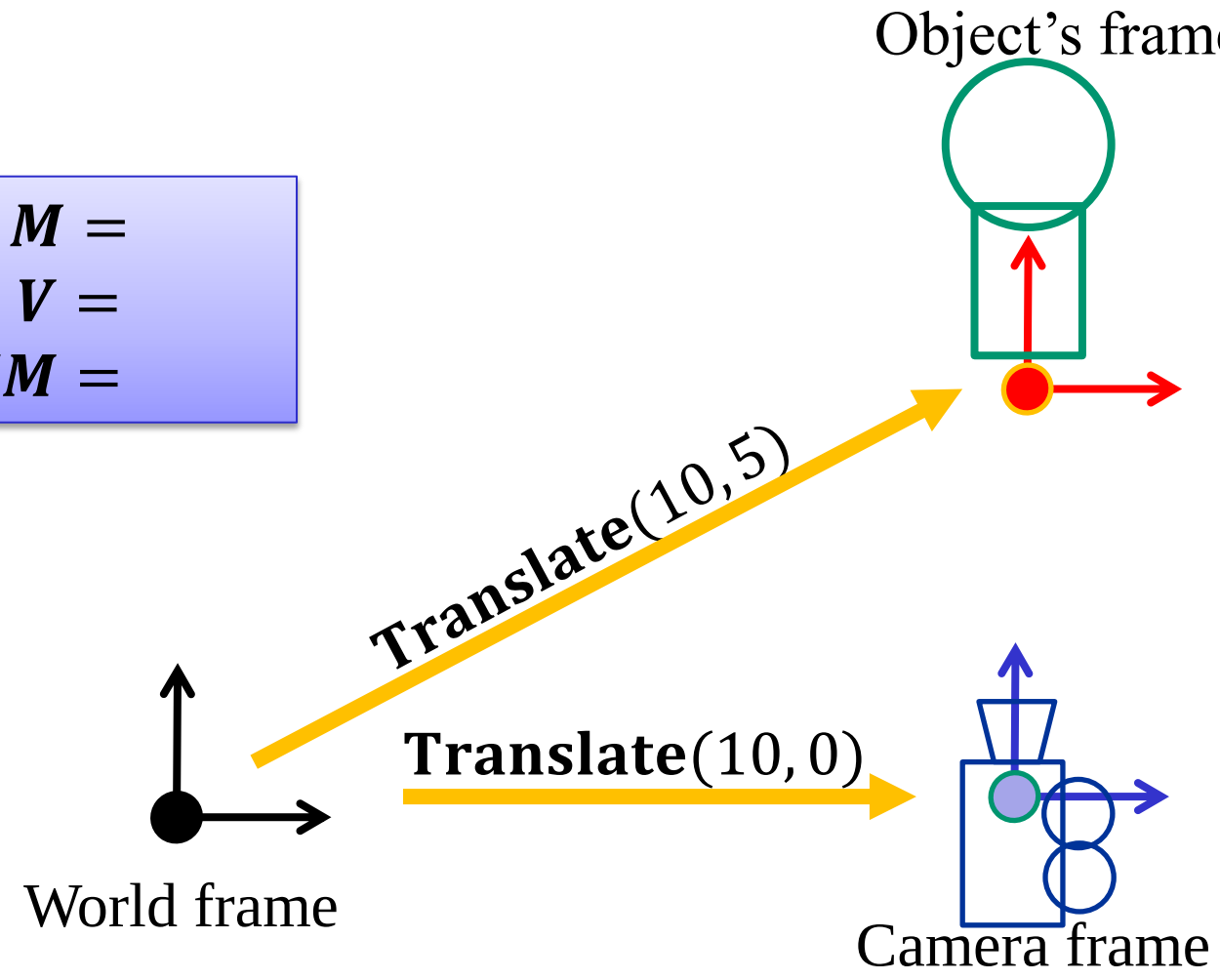
Camera frame

Quiz

$M =$

$V =$

$VM =$


Object's frame
Translate(10,5)
Translate(10,0)
World frame
Camera frame

Model-view matrix

- Whenever you write
  - `glVertex**(x, y, z)`
- The following conceptually takes place:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} := \mathcal{M} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

The point (x', y') is then used for 2D rasterization*

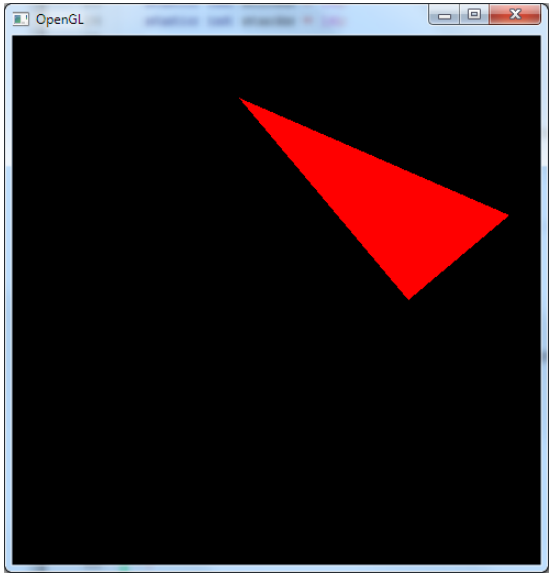
Model-view matrix

- The Model-view matrix can be provided explicitly
  - `glLoadMatrix*(. . . . . .)`
- Or, more commonly, constructed by multiplying with elementary matrices **on the right**.
  - `glLoadIdentity();` $\mathcal{M} = I$
  - `glTranslatef(…);` $\mathcal{M} = IT$
  - `glRotatef(…);` $\mathcal{M} = ITR$

OpenGL Example

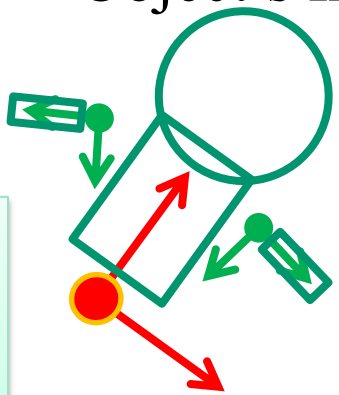
- The following code draws the same triangle, rotated by 40 degrees **and then** translated by 0.5 along x axis.

```
glTranslatef(0.5, 0.0, 0.0);
glRotatef(40, 0.0, 0.0, 1.0);
glBegin(GL_TRIANGLES);
     glVertex2f(0.0, 0.0);
     glVertex2f(0.5, 0.0);
     glVertex2f(0.0, 1.0);
glEnd();
```

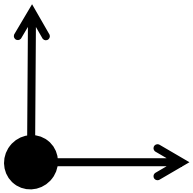


Scene graph

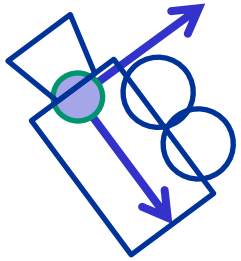
Object's frame



Objects usually consist of multiple parts, each described in their own frame



World frame



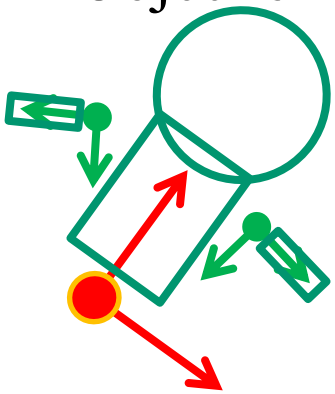
Camera frame

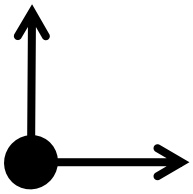
Scene graph

Each sub-object is first transformed to the frame of its parent:

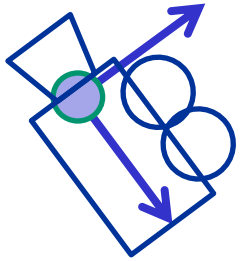
- Left arm: $\boldsymbol{M}_{left \rightarrow body}$
- Right arm: $\boldsymbol{M}_{right \rightarrow body}$

Object's frame





World frame



Camera frame

Scene graph

Each sub-object is first transformed to the frame of its parent:

- Left arm: $\boldsymbol{M}_{left \rightarrow body}$
- Right arm: $\boldsymbol{M}_{right \rightarrow body}$

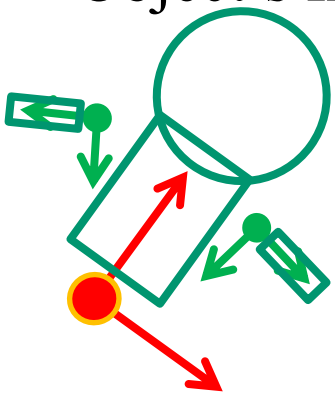
The whole object is then transformed to world

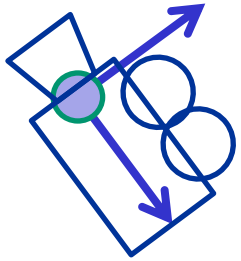
$$\boldsymbol{M}_{body \rightarrow world}$$

and finally to camera frame:

$$\boldsymbol{V}$$

Object's frame





Camera frame

Scene graph

Each sub-object is first transformed to the frame of its parent:

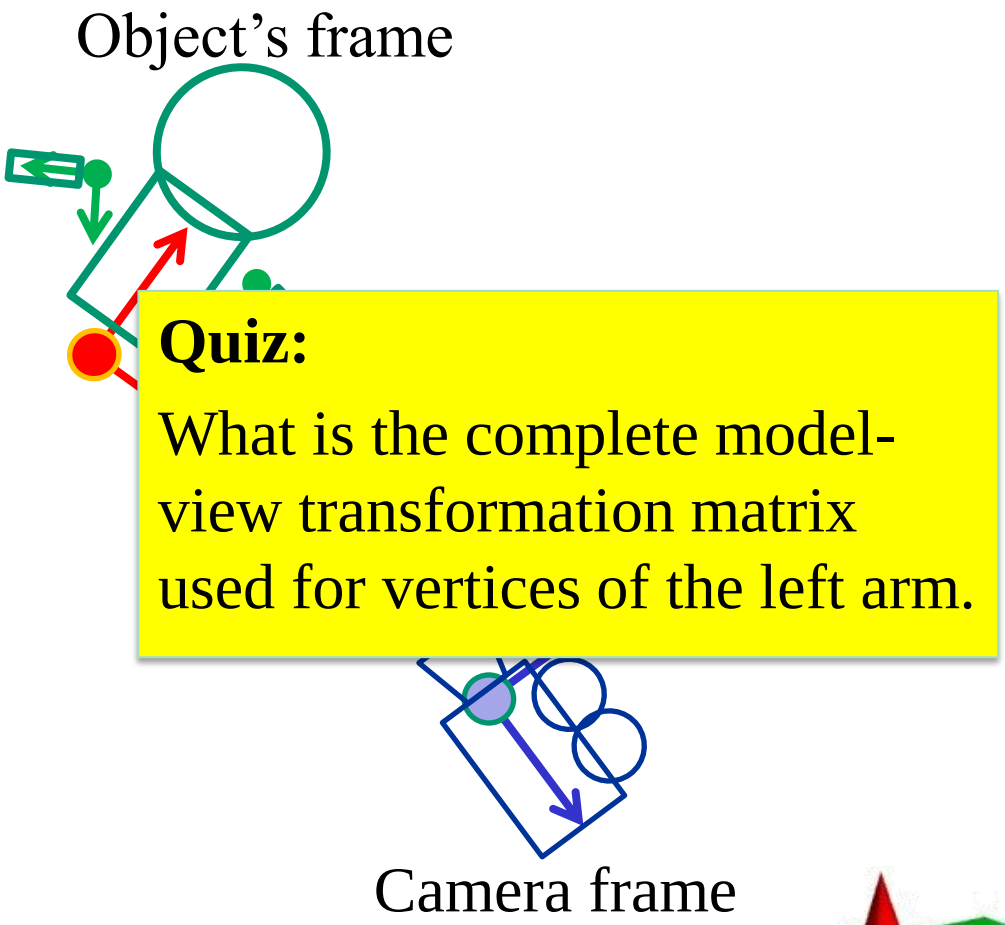
- Left arm: $\boldsymbol{M}_{left \rightarrow body}$
- Right arm: $\boldsymbol{M}_{right \rightarrow body}$

The whole object is then transformed to world

$$\boldsymbol{M}_{body \rightarrow world}$$

and finally to camera frame:

$$\boldsymbol{V}$$



Scene graph

Each sub-object is first transformed to the frame of its parent:

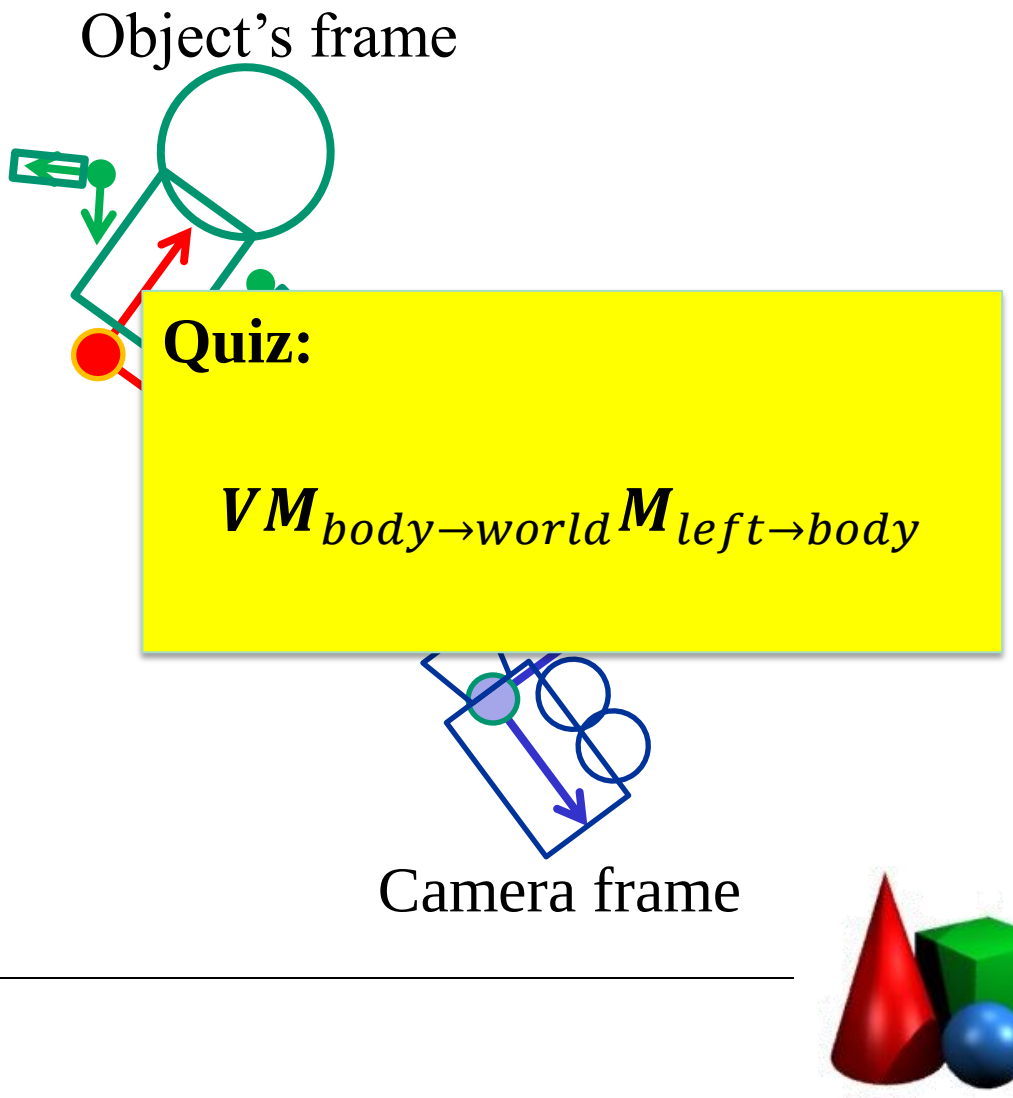
- Left arm: $\boldsymbol{M}_{left \rightarrow body}$
- Right arm: $\boldsymbol{M}_{right \rightarrow body}$

The whole object is then transformed to world

$$\boldsymbol{M}_{body \rightarrow world}$$

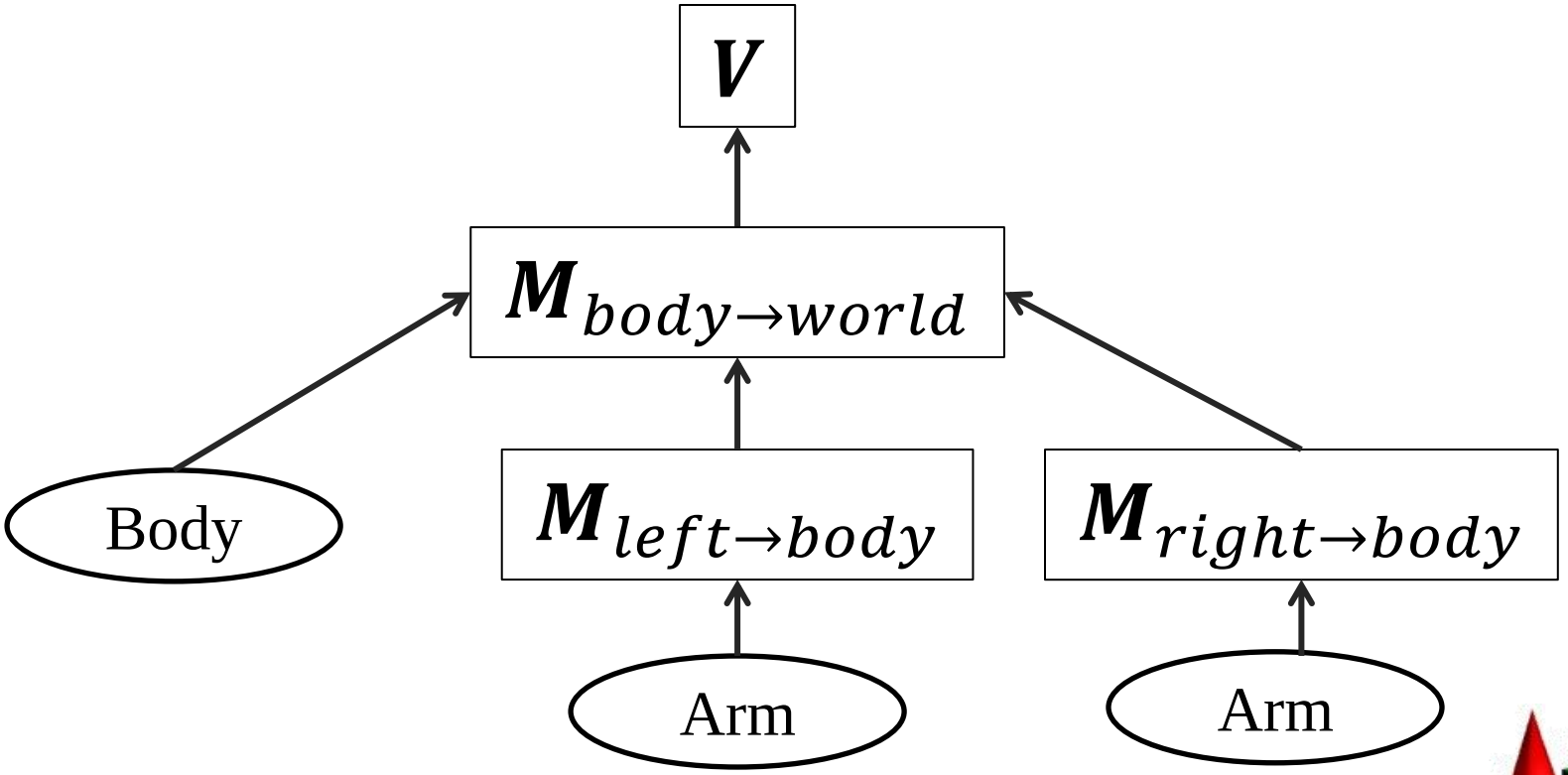
and finally to camera frame:

$$\boldsymbol{V}$$



Scene graph

- The whole scene is thus described as a tree – the **scene graph**.



Example

```
gluLookAt(...);                          // View transform

    glRotatef(1.0, 0.0, 0.0, 1.0);       // Body → world
    draw_body();

        glTranslatef(0.0, 0.5, 2.0);     // Left arm → body
        glRotatef(0.5, 0.0, 1.0, 0.0);
        draw_arm();
        "unmultiply" left -> body

        glTranslate(0.0, -0.5, -2.0);    // Right arm → body
        draw_arm();
        "unmultiply" right -> body
    "unmultiply" body -> world
```

Example

```
gluLookAt(eyeX,    eyeY,    eyeZ,
          centerX, centerY, centerZ,
          upX,     upY,     upZ);
```

```
        glTranslatef(0.0, 0.5, 2.0);   // Left arm → body
        glRotatef(0.5, 0.0, 1.0, 0.0);
        draw_arm();
        "unmultiply" left -> body

        glTranslate(0.0, -0.5, -2.0);  // Right arm → body
        draw_arm();
        "unmultiply" right -> body
    "unmultiply" body -> world
```

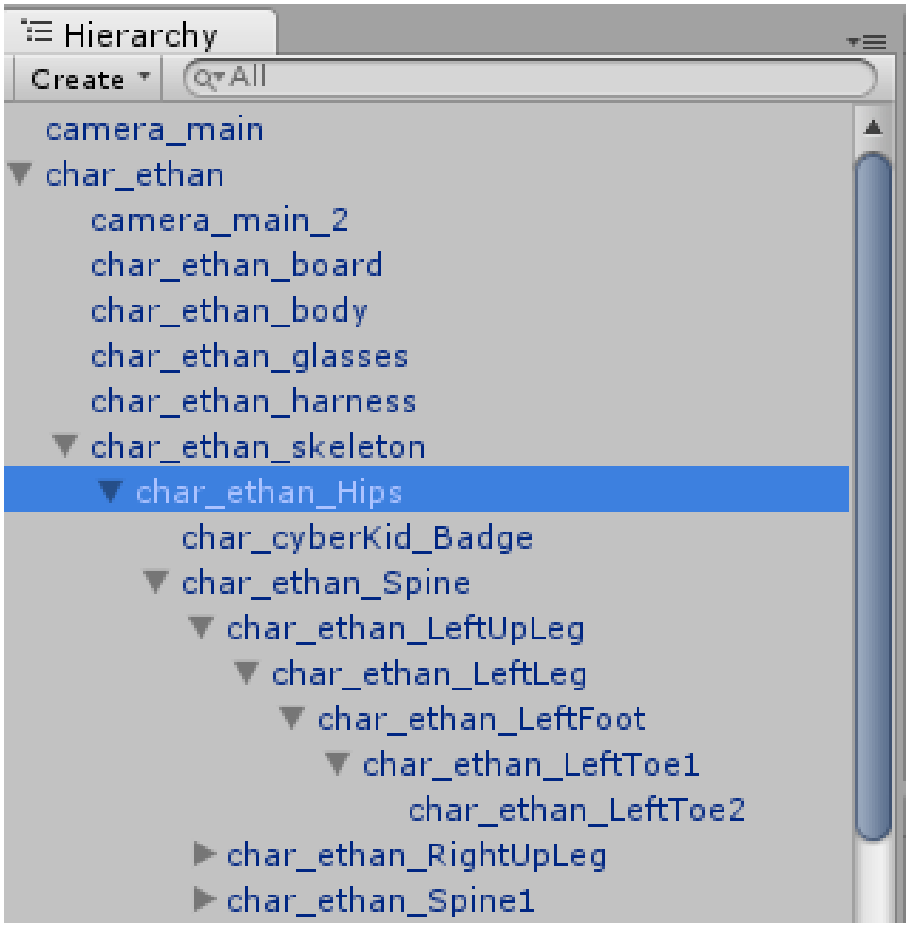
Example

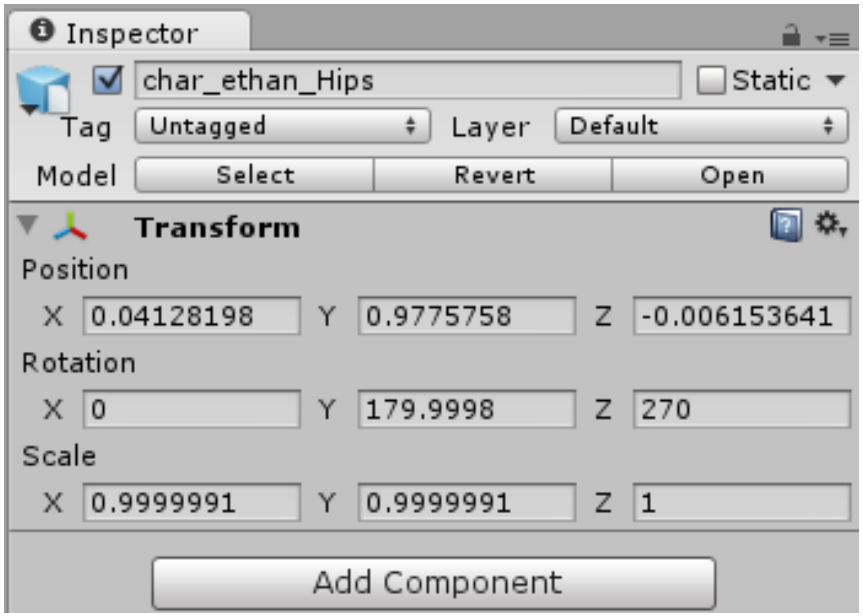
```
gluLookAt(...);                          // View transform

    glRotatef(1.0, 0.0, 0.0, 1.0);       // Body → world
    draw_body();

        glTranslatef(0.0, 0.5, 2.0);     // Left arm → body
        glRotatef(0.5, 0.0, 1.0, 0.0);
        draw_arm();
    "unmultiply" left -> body

        glTranslate(0.0, -0.5, -2.0);    // Right arm → body
        draw_arm();
    "unmultiply" right -> body
"unmultiply" body -> world
```

Example

```
gluLookAt(...);                          // View transform
glPushMatrix();
   glRotatef(1.0, 0.0, 0.0, 1.0);        // Body → world
   draw_body();
   glPushMatrix();
      glTranslatef(0.0, 0.5, 2.0);       // Left arm → body
      glRotatef(0.5, 0.0, 1.0, 0.0);
      draw_arm();
   glPopMatrix();
   glPushMatrix();
      glTranslate(0.0, -0.5, -2.0);      // Right arm → body
      draw_arm();
   glPopMatrix();
glPopMatrix();
```

Unity3D Example





VRML example

```
Transform {
   translation 0 0 -5
   children Transform {
      rotation 0 0 1 1.0
      children [
         USE BODY
         Transform {
            translation 0 0.5 2.0
            rotation 0 1 0 0.5
            children USE ARM
         }
         Transform {
            translation 0 -0.5 -2.0
            children USE ARM
         }
      ]
   }
}
```

Projection transform

- After the vertices are transformed to the camera frame, they are *projected* to the camera plane using a *projection transform* $\boldsymbol{P}$.
- Thus, the total transformation pipeline for each object vertex $\boldsymbol{p_i}$ is:

$$\boldsymbol{PVMp_i}$$

- This is the topic of the next lecture