The View Transformation

CS 418: Interactive Computer Graphics

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Eric Shaffer
Graphics Pipeline and WebGL

From WebGL Beginner’s Guide by Cantor and Jones
Which of the following mean the same thing?

• See if you can guess...
  • Camera transformation
  • Eye transformation
  • View transformation
  • Camera space
  • Eye space
  • Clip space
  • Normalized device coordinates
  • Viewport transformation
  • Windowing transformation
  • Screen space
  • Pixel coordinates
  • Viewport coordinates
Computer Graphics has Non-standardized Vocabulary

• Camera transformation
• Eye transformation
• View transformation
• Camera space
• Eye space
• Clip space
• Normalized device coordinates
• Viewport transformation
• Windowing transformation
• Screen space
• Pixel coordinates
• Viewport coordinates

• So don’t be afraid to ask someone what something means
Viewing

We often will want to allow the view of our 3D scene to change

We can do so using by applying affine transformations to the geometry

A **view matrix** is functionally equivalent to a camera

It is a transformation matrix like the Model matrix, but

- Happens after the modeling transformation
- It applies the same transformations equally to every object
  - Moving the whole world 5 units towards us = walking 5 units forwards

The engines don’t move the ship at all. The ship stays where it is and the engines move the universe around it.

--- Futurama
Example

From WebGL Beginner’s Guide by Cantor and Jones
Graphics Pipeline

\[
\begin{bmatrix}
x_s \\
y_s \\
0 \\
1
\end{bmatrix} = \begin{bmatrix} W2V & \text{Persp} & \text{View} & \text{Model} \end{bmatrix} \begin{bmatrix} x_m \\
y_m \\
z_m \\
1 \end{bmatrix}
\]
Graphics Pipeline

\[
\begin{bmatrix}
x_s \\
y_s \\
0 \\
1
\end{bmatrix} = \begin{bmatrix}
x_m \\
y_m \\
z_m \\
1
\end{bmatrix}
\]
Viewing Transformation

\[
\begin{bmatrix}
    x_s \\
    y_s \\
    0 \\
    1
\end{bmatrix}
= \begin{bmatrix}
    W2V & \text{Persp} & \text{View} & \text{Model}
\end{bmatrix}
\begin{bmatrix}
    x_m \\
    y_m \\
    z_m \\
    1
\end{bmatrix}
\]
Viewing Transformation

- **W2V**: Viewing Model
- **Viewing Coords**: Model Coords
- **Screen Coords**: W2V
- **Clip Coords**: Persp
- **Persp**: View
- **View Coords**: World Coords
- **World Coords**: Model Coords
- **Model Coords**: Screen Coords

Diagram showing the transformation process from screen coordinates to model coordinates.
Creating a Camera Function

Suppose we want to implement a function that sets up view...think of it as setting up a camera

There are lots of possible ways to do this...we’ll choose a simple lookat camera

The API we create will require a someone using the function to specify:

- The **eyepoint** (or camera location)
- The **lookat point** (a point in the view direction)
- An ”up” **vector** that we use to specify rotation around the view vector
Deriving the Viewing Transformation

One way to think about what you are doing

• Translate the eyepoint to the origin
• Rotate so that
  • lookat vector aligns with \(-z\) axis
  • up aligns with \(y\)

We move all objects (the world) this way...

• Another way to think of it
  • Create an orthonormal basis with eye at the origin
  • And vectors \(u, v, w\) as the basis vectors
  • ...and then align \(u, v, w\) with \(x, y, z\)
Constructing a Local Frame

A frame has an origin point and set of basis vectors.

Any point can be expressed as coordinates in such a frame.

For example (0,0,0) and <1,0,0>, <0,1,0>, <0,0,1>

- And an example of a point in that space:
  \((4,0,0) = (0,0,0) + 4 <1,0,0> + 0 <0,1,0> + 0 <0,0,1>\)
Example in 2D

To convert coordinates from (u,v) space to (x,y) we can:

\[
\begin{bmatrix}
x_p \\
y_p \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & x_e \\
0 & 1 & y_e \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_u & x_v & 0 \\
y_u & y_v & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_p \\
v_p \\
1
\end{bmatrix} =
\begin{bmatrix}
x_u & x_v & x_e \\
y_u & y_v & y_e \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_p \\
v_p \\
1
\end{bmatrix}
\]

This can be written as

\[
P_{xy} = \begin{bmatrix} u & v & e \\ 0 & 0 & 1 \end{bmatrix} P_{uv}
\]
Example in 2D

Imagine we want to see the point \( p \) as it would be seen in \((u,v)\) space

That means we convert the point to \((u,v)\) coordinates...pretend those are the \((x,y)\)

Alternatively, think of the matrix as rotating & translating \( p \) so it is seen as if in \((u,v)\) space

To convert coordinates from \((u,v)\) space to \((x,y)\) we can:

\[
\begin{bmatrix}
x_p \\
y_p \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & x_e \\
0 & 1 & y_e \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_u & x_v & 0 \\
y_u & y_v & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_p \\
v_p \\
1
\end{bmatrix} =
\begin{bmatrix}
x_u & x_v & x_e \\
y_u & y_v & y_e \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_p \\
v_p \\
1
\end{bmatrix}
\]

This can be written as

\[
P_{xy} =
\begin{bmatrix}
u & v & e \\
0 & 0 & 1
\end{bmatrix}
P_{uv}
\]
Cross Product of Two Vectors

**Important Property:**

The cross product yields a vector orthogonal to the original two vectors.

\[ \mathbf{a} = \langle a_1, a_2, a_3 \rangle \]
\[ \mathbf{b} = \langle b_1, b_2, b_3 \rangle \]
\[ \mathbf{a} \times \mathbf{b} = \langle a_2b_3 - b_2a_3, a_3b_1 - b_3a_1, a_1b_2 - b_1a_2 \rangle \]
The Orthonormal Basis for View Space

- Let $l$ be the lookat vector...then $w = -\frac{l}{\|l\|}$
- If $t$ is the up direction $u = \frac{t \times w}{\|t \times w\|}$
- And then $v = w \times u$
- The view matrix is then:

$$M_{\text{view}} = M_{\text{cam}}^{-1} = \begin{bmatrix} u & v & w & e \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Why is the matrix inverted?
View Transformation

You can now look at your scene from any

- Position
- Orientation (almost)
  - What lookat and up vector pair won’t work?

...just uses a matrix multiplication

$$M_{view} = M_{cam}^{-1} = \begin{bmatrix} u & v & w & e \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
glMatrix lookAt transform

{mat4} mat4.lookAt(out, eye, center, up)
Generates a look-at matrix with the given eye position, focal point, and up axis

Parameters:
(mat4) out
mat4 matrix will be written into

(vec3) eye
Position of the viewer

(vec3) center
Point the viewer is looking at

(vec3) up
vec3 pointing up

Returns:
(mat4) out

glMatrix can compute a lookAt transformation for you

How would you implement a moving camera?

What key difficulties would you need to handle?
Let’s review viewing....

We start out by setting up our geometry in world coordinates.

Which transformation does this?
The view transformation

We pick a specific viewing position and direction in worldspace
The view transformation

We transform the world so the view position is at the origin. The view direction is down the –z axis.
The projection transformation

We pick a viewing volume.
This is specified in *viewing coordinates*. 
The projection transformation

Our view volume is transformed to fit in the WebGL view volume
The WebGL view volume is a box with clip planes at -1, +1
The z coordinates are negated to flip the z-axis