Ray tracing is a pixel-first approach to rendering.

1.) set up view and image plane
2.) for each pixel:
   a.) create ray
   b.) for each object:
      does ray hit object?
      keep closest
   c.) calculate color
Rasterization is an object-first approach to rendering.

1) set up view + image plane
2) for each object:
   a) project to screen
   b) for each pixel covered by object:

   if depth of the object-pixel pair is less than current depth
   c) calculate color
We need to first transform into the CAMERA frame of reference.

\[ C = \begin{bmatrix} u_x & v_x & w_x & e_x \\ u_y & v_y & w_y & e_y \\ u_z & v_z & w_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = TB \]
View Matrix ($M_v$): transforming from world space \textit{into} camera space.

\[
C = \begin{bmatrix}
u_x & v_x & w_x & e_x \\
v_y & v_y & w_y & e_y \\
u_z & v_z & w_z & e_z \\
0 & 0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & e_x \\
0 & 1 & 0 & e_y \\
0 & 0 & 1 & e_z \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
u_x & v_x & w_x & 0 \\
v_y & v_y & w_y & 0 \\
u_z & v_z & w_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} = TB.
\]

Given the transformation from camera to world space ($C = T \ast B$), which matrix will transform from world to camera space?

- [ ] inverse($T$) $\ast$ inverse($B$)
- [ ] inverse($T$) $\ast$ transpose($B$)
- [x] inverse($B$) $\ast$ inverse($T$)
- [ ] transpose($B$) $\ast$ inverse($T$)
- [ ] None of these.

Voting as Anonymous

slido.com (event #3097031)
Then we will project vertices onto the image plane.

Really want to use a matrix.

**Similar triangles:**

\[
\frac{d_n}{-z} = \frac{x_p}{X} \\
\frac{d_n}{-z} = \frac{y_p}{Y}
\]

\[
x_p = x \left( \frac{-d_n}{z} \right) \\
y_p = y \left( \frac{-d_n}{z} \right)
\]

\[
M_p = \begin{bmatrix} x & y & z \end{bmatrix}
\]

\[\approx \frac{1}{z} \text{ not linear}\]
Oh wait, let me tell you one more thing about homogeneous coordinates: *homogenization*.

Position vectors

\[
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

homogenization

\[
\begin{bmatrix}
x/w \\
y/w \\
z/w \\
1
\end{bmatrix}
\]

divide by fourth component

equivalent 3D cartesian points

homogeneous coordinate
If we remember to always homogenize the result, then we can write a perspective projection matrix.

\[ M_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & d_n \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ -\frac{z}{d_n} \\ -\frac{z}{d_n} \end{bmatrix} \]

\[
x_p = x \left( -\frac{d_n}{z} \right), \quad y_p = y \left( -\frac{d_n}{z} \right), \quad z_p = d_n
\]

homogenize
Nomenclature: Model-View Matrix & MVP matrix.

model-view matrix: $M_v M_m$  
MVP matrix
model-view-projection matrix: $M_p M_v M_m$
Introduction to **WebGL**: GPU-based rasterization API.

Example **GLSL** syntax with vectors & matrices:
Moving forward: always ask which frame of reference you're in.

When doing a lighting calculation in camera space, the normal vector (originally defined in object space), should be transformed by:

How should normals be transformed?

- inverse(transpose(Mv * Mm)) 69%
- inverse(transpose(Mm)) 15%
- Mv * Mm 8%
- inverse(transpose(Mp * Mv * Mm)) 8%
- Mm 0%
- Mv 0%
- inverse(transpose(Mv)) 0%

points transformed by model-view (MvMm)