Computer Graphics

MTAT.03.015

Raimond Tunnel

Study IT in .ee
The Road So Far...

Last week & This week
Frames of Reference

• Can you name different spaces (frames of reference) we use?
Frames of Reference

- Can you name different spaces (frames of reference) we use?
Object Space → World Space

- We model our objects in object space
Object Space → World Space

- We model our objects in object space
  - Symmetrically from the origin
Object Space $\rightarrow$ World Space

- We model our objects in object space
  - Symmetrically from the origin
  - Up from the origin
Object Space → World Space

- We model our objects in object space
  - Symmetrically from the origin
  - Up from the origin
- We position, orient and scale our object with the model matrix, thus creating the world space!
Object Space $\rightarrow$ World Space

- We model our objects in object space
  - Symmetrically from the origin
  - Up from the origin
- We position, orient and scale our object with the model matrix, thus creating the world space!
- World space is like the root node in the scene graph
Object Space → World Space

- We model our objects in object space
  - Symmetrically from the origin
  - Up from the origin
- We position, orient and scale our object with the model matrix, thus creating the world space!
- World space is like the root node in the scene graph:
  - Origin defined by the identity transformation
Object Space $\rightarrow$ World Space

- We model our objects in object space
  - *Symmetrically* from the origin
  - Up from the origin
- We position, orient and scale our object with the *model matrix*, thus creating the world space!
- World space is like the root node in the scene graph:
  - Origin defined by the identity transformation
  - Every child transformed relative to it
Object Space → World Space

This is what you did last week. :)

Object_1

Object_0

Object_2
Object Space $\rightarrow$ World Space

\[
\text{projectionMatrix} \cdot \text{viewMatrix} \cdot \text{modelMatrix} \cdot v
\]

\[
P \cdot V \cdot M \cdot v
\]

This is what you did last week. :)
World Space ➞ Camera Space

- We want to represent everything related to the camera (to make projection easier)

Transform so that this is the origin + basis
World Space $\rightarrow$ Camera Space

- We want to represent everything related to the camera (to make projection easier)
- We can think of the camera as another object in the scene.
World Space → Camera Space

- We want to represent everything related to the camera (to make projection easier)
- We can think of the camera as another object in the scene.
  - It has its own rotation and position.
World Space $\rightarrow$ Camera Space

- We want to represent everything related to the camera (to make projection easier).
- We can think of the camera as another object in the scene.
  - It has its own rotation and position.
  - Scale is not really relevant for the camera.
World Space → Camera Space

- Assume that we have a camera's model transformation matrix:
World Space $\rightarrow$ Camera Space

- Assume that we have a camera's model transformation matrix:

$$ M_{camera} = \begin{pmatrix} \text{right}_x & \text{up}_x & \text{back}_x & \text{pos}_x \\ \text{right}_y & \text{up}_y & \text{back}_y & \text{pos}_y \\ \text{right}_z & \text{up}_z & \text{back}_z & \text{pos}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} $$
World Space $\rightarrow$ Camera Space

- Assume that we have a camera's model transformation matrix:

\[
M_{\text{camera}} = \begin{pmatrix}
\text{right}_x & \text{up}_x & \text{back}_x & \text{pos}_x \\
\text{right}_y & \text{up}_y & \text{back}_y & \text{pos}_y \\
\text{right}_z & \text{up}_z & \text{back}_z & \text{pos}_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

- Remember that the columns are the transformed standard basis...
World Space → Camera Space

- Assume that we have a camera's model transformation matrix:

\[
M_{\text{camera}} = \begin{pmatrix}
right_x & up_x & back_x & pos_x \\
right_y & up_y & back_y & pos_y \\
right_z & up_z & back_z & pos_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

- Remember that the columns are the transformed standard basis...

- Can you come up with a matrix that describes our world relative to the camera?
World Space → Camera Space

- **View matrix** can be found like this:
World Space → Camera Space

- **View matrix** can be found like this:

  1) Camera's linear transform. is an orthonormal matrix

\[
M_{\text{camera}} = \begin{pmatrix}
\text{right}_x & \text{up}_x & \text{back}_x & \text{pos}_x \\
\text{right}_y & \text{up}_y & \text{back}_y & \text{pos}_y \\
\text{right}_z & \text{up}_z & \text{back}_z & \text{pos}_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
World Space → Camera Space

- **View matrix** can be found like this:
  1) Camera's linear transform is an orthonormal matrix
  2) Transpose it to find the inverse

\[
\begin{pmatrix}
\text{right}_x & \text{up}_x & \text{back}_x \\
\text{right}_y & \text{up}_y & \text{back}_y \\
\text{right}_z & \text{up}_z & \text{back}_z
\end{pmatrix}^T = \begin{pmatrix}
\text{right}_x & \text{right}_y & \text{right}_z \\
\text{up}_x & \text{up}_y & \text{up}_z \\
\text{back}_x & \text{back}_y & \text{back}_z
\end{pmatrix}
\]
World Space → Camera Space

- **View matrix** can be found like this:
  1) Camera's linear transform is an orthonormal matrix
  2) Transpose it to find the inverse
  3) Camera's translation can be inverted by negation

\[
\begin{pmatrix}
\text{right}_x & \text{right}_y & \text{right}_z \\
\text{up}_x & \text{up}_y & \text{up}_z \\
\text{back}_x & \text{back}_y & \text{back}_z
\end{pmatrix}
\]

\[
\begin{pmatrix}
pos_x \\
pos_y \\
pos_z
\end{pmatrix} = \begin{pmatrix}
- pos_x \\
- pos_y \\
- pos_z
\end{pmatrix}
\]
World Space $\rightarrow$ Camera Space

- **View matrix** can be found like this:

  4) Put the two inverse transformations together in the opposite order

\[
V = \begin{pmatrix}
  \text{right}_x & \text{right}_y & \text{right}_z & 0 \\
  \text{up}_x & \text{up}_y & \text{up}_z & 0 \\
  \text{back}_x & \text{back}_y & \text{back}_z & 0 \\
  0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
  1 & 0 & 0 & -pos_x \\
  0 & 1 & 0 & -pos_y \\
  0 & 0 & 1 & -pos_z \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]
World Space $\rightarrow$ Camera Space

- **View matrix** can be found like this:

$$V = \begin{pmatrix}
  \text{right}_x & \text{right}_y & \text{right}_z & 0 \\
  \text{up}_x & \text{up}_y & \text{up}_z & 0 \\
  \text{back}_z & \text{back}_y & \text{back}_z & 0 \\
  0 & 0 & 0 & 1
\end{pmatrix} \cdot \begin{pmatrix}
  1 & 0 & 0 & -\text{pos}_x \\
  0 & 1 & 0 & -\text{pos}_y \\
  0 & 0 & 1 & -\text{pos}_z \\
  0 & 0 & 0 & 1
\end{pmatrix}$$

1. Transpose the rotation to inverse it
2. Negate the translation to inverse it
3. Multiply together in the reverse order
World Space $\rightarrow$ Camera Space

- Usually it is more intuitive to specify the camera by its **position**; **point it is looking at**; and the **up-vector**
World Space → Camera Space

- Usually it is more intuitive to specify the camera by its position; point it is looking at; and the up-vector

Three.js:

camera.position.set(x, y, z);
camera.up.set(upX, upY, upZ);
camera.lookAt(point);
World Space $\rightarrow$ Camera Space

- Usually it is more intuitive to specify the camera by its position; point it is looking at; and the up-vector

**OpenGL:**

```cpp
glm::mat4 view = glm::lookAt(
    glm::vec3(x, y, z),
    glm::vec3(pX, pY, pZ),
    glm::vec3(upX, upY, upZ)
);
```
World Space → Camera Space

- Usually it is more intuitive to specify the camera by its position; point it is looking at; and the up-vector.

- The up-vector may not be the same as the y-direction of camera's space. It just gives a rough orientation.
World Space → Camera Space

- Using the lookAt() command parameters, how to find the correct matrix?
- What do we have and what do we need?
World Space $\rightarrow$ Camera Space

\[
\text{projectionMatrix} \cdot \text{viewMatrix} \cdot \text{modelMatrix} \cdot v
\]

\[
P \cdot V \cdot M \cdot v
\]
Camera Space $\rightarrow$ ND Space

- For the normalized device space, we transform the view frustum into a cube $[-1, 1]^3$. 
For the normalized device space, we transform the view frustum into a cube \([-1, 1]^3\).
Camera Space → ND Space

• For the normalized device space, we transform the view frustum into a cube $[-1, 1]^3$.

Slices from $x=0$ plane
Camera Space → ND Space

- For the **normalized device space**, we transform the view frustum into a cube \([-1, 1]^3\).

Perspective

Camera Space → ND Space

• For the normalized device space, we transform the view frustum into a cube $[-1, 1]^3$.
• We want to flip the z-axis, because our near and far planes are positive values.
Camera Space → ND Space

- For the normalized device space, we transform the view frustum into a cube \([-1, 1]^3\).
- We want to flip the z axis, because our near and far planes are positive values.
- This is the job for the projection matrix together with the point normalization.
Camera Space $\rightarrow$ ND Space

$$\text{projectionMatrix} \cdot \text{viewMatrix} \cdot \text{modelMatrix} \cdot \mathbf{v}$$

$$P \cdot V \cdot M \cdot \mathbf{v}$$
Orthographic Projection

- We define our view volume with the values for **left**, **right**, **top**, **bottom**, **near** and **far** planes.
Orthographic Projection

- We define our view volume with the values for *left, right, top, bottom, near* and *far* planes.

```javascript
OrthographicCamera( left, right, top, bottom, near, far )
```

- `left` — Camera frustum left plane.
- `right` — Camera frustum right plane.
- `top` — Camera frustum top plane.
- `bottom` — Camera frustum bottom plane.
- `near` — Camera frustum near plane.
- `far` — Camera frustum far plane.

Together these define the camera’s *viewing frustum*.

From Three.js docs.
Orthographic Projection

- We define our view volume with the values for left, right, top, bottom, near and far planes.
- What would be the matrix that transforms the orthographic view volume into the canonical view volume $([-1, 1]^3)$?
Perspective Projection

- Usually defined by the **vertical angle** for the field-of-view (**FOV**), the **aspect ratio** and the **near** and **far** planes.
Perspective Projection

• Usually defined by the **vertical angle** for the field-of-view (**FOV**), the **aspect ratio** and the **near** and **far** planes.

```javascript
PerspectiveCamera( fov, aspect, near, far )
```

- **fov** — Camera frustum vertical field of view.
- **aspect** — Camera frustum aspect ratio.
- **near** — Camera frustum near plane.
- **far** — Camera frustum far plane.

Together these define the camera's **viewing frustum**.

From Three.js docs.
Perspective Projection

- Usually defined by the vertical angle for the field-of-view (FOV), the aspect ratio and the near and far planes.

- Find the **left**, **right**, **top** and **bottom** on the near plane, when the projection is *symmetric*?

  \[
  \text{top} = -\text{bottom} \\
  \text{left} = -\text{right}
  \]
Perspective Projection

- Differently from the orthographic projection, here we have a viewer located in a single point.
- Similarly we want to find the normalized device coordinates for all points inside the view volume.
Perspective Projection

- First **find** and then **map** the x and y coordinates of the **projected point** to the correct range using similar triangles.

Find \( x_p \) and \( y_p \).
Perspective Projection

\[ P = \begin{pmatrix}
\begin{array}{cccc}
\text{near} & 0 & 0 & 0 \\
\text{right} & 0 & 0 & 0 \\
\text{top} & 0 & 0 & 0 \\
\end{array}
\end{pmatrix}
\]

- If the third row would be \((0, 0, 1, 0)\), then all \(z\) coordinates become -1 (because we found the projected coordinates on the near plane)
Perspective Projection

- We want to map the z value from the range [near, far] to the range [-1, 1].

- We can use scale and translation.

\[
P = \begin{pmatrix}
\frac{\text{near}}{\text{right}} & 0 & 0 & 0 \\
0 & \frac{\text{near}}{\text{top}} & 0 & 0 \\
0 & 0 & s & t \\
0 & 0 & -1 & 0
\end{pmatrix}
\]
Perspective Projection

- We want to map the $z$ value from the range $[\text{near}, \text{far}]$ to the range $[-1, 1]$, so...

$$\begin{cases} \frac{s \cdot \text{near} + t}{\text{near}} = -1 \\ \frac{s \cdot \text{far} + t}{\text{far}} = 1 \end{cases}$$

$$P = \begin{pmatrix} \text{near} & 0 & 0 & 0 \\ \text{right} & 0 & 0 & 0 \\ \text{top} & 0 & 0 & \text{s} & \text{t} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Can this be solved for the $s$ and $t$ unknowns?

Division by $\text{near}$ and $\text{far}$ is because of the homogeneous division.
Perspective Projection

- After applying this matrix and doing the point normalization (dividing with $w$), you have the perspective projection.

$$P = \begin{pmatrix}
\frac{\text{near}}{\text{right}} & 0 & 0 & 0 \\
0 & \frac{\text{near}}{\text{top}} & 0 & 0 \\
0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -\frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\
0 & 0 & -1 & 0
\end{pmatrix}$$
Perspective Projection

- When using the FOV ($\alpha$) and aspect ratio ($ar$).

$$P = \begin{bmatrix}
\frac{1}{ar \cdot \tan \left( \frac{\alpha}{2} \right)} & 0 & 0 & 0 \\
0 & \frac{1}{\tan \left( \frac{\alpha}{2} \right)} & 0 & 0 \\
0 & 0 & \frac{ \text{far} + \text{near} }{\text{far} - \text{near}} & \frac{2 \cdot \text{far} \cdot \text{near} }{\text{far} - \text{near}} \\
0 & 0 & -1 & 0
\end{bmatrix}$$
Clip Space

• After the projection matrix multiplication and before the $w$-division, vertices are in a clip space.

Read more here: https://stackoverflow.com/a/21841924/3067608
Clip Space

- After the projection matrix multiplication and before the \( w \)-division, vertices are in a *clip space*.

- That is the space, where it is the most easiest to determine, which triangles need to be clipped or culled.

Read more here: [https://stackoverflow.com/a/21841924/3067608](https://stackoverflow.com/a/21841924/3067608)
Clip Space

- After the projection matrix multiplication and before the \( w \)-division, vertices are in a clip space.
- That is the space, where it is the most easiest to determine, which triangles need to be clipped or culled.
- **Clipping** – performed when some part of the triangle is inside the view volume.

Read more here: [https://stackoverflow.com/a/21841924/3067608](https://stackoverflow.com/a/21841924/3067608)
Clip Space

- After the projection matrix multiplication and before the $w$-division, vertices are in a \textit{clip space}.

- That is the space, where it is the most easiest to determine, which triangles need to be clipped or culled.

- **Clipping** – performed when some part of the triangle is inside the view volume.

- **Culling** – performed when the triangle is not inside the view volume. Or is back-facing.

Read more here: https://stackoverflow.com/a/21841924/3067608
ND Space $\rightarrow$ Screen Space

- We have everything we want to show now in the $[-1, 1]^3$ cube (normalized device space).
ND Space → Screen Space

- We have everything we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.

Before the perspective projection
ND Space $\rightarrow$ Screen Space

- We have everything we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.

After the perspective projection
ND Space $\rightarrow$ Screen Space

- We have everything we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.
ND Space $\rightarrow$ Screen Space

- We have everything we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.
- How to know where to draw on the screen?

Come up with that matrix...
ND Space → Screen Space

• This is done for you, the screen space matrix is constructed when you specify the viewport size.

**Three.js**
renderer = new THREE.WebGLRenderer();
renderer.setSize(width, height);

**OpenGL + GLFW**
win = glfwCreateWindow(width, height,
"Hello GLFW!", NULL, NULL)
Overall

Object Space
Overall

Object Space → World Space
Overall

Object Space → World Space → Camera (View) Space
Overall

Object Space $\rightarrow$ World Space $\rightarrow$

$\rightarrow$ Camera (View) Space

Light calculations are usually in this space!
Overall

Camera (View) Space $\rightarrow$ Normalized Device Space
Overall

→ Normalized Device Space

→ Screen Space
Overall

- Vertex shader must return homogeneous coordinates in the clip space – that is in normalized device space without the w-division.

\[
gl\_Position = \text{projection} \times \text{view} \times \text{model} \times \text{vec4}(\text{position}, 1.0); \\
gl\_Position = \text{projectionMatrix} \times \text{modelViewMatrix} \times \text{vec4}(\text{position}, 1.0); \\
gl\_Position = \text{modelViewProjectionMatrix} \times \text{vec4}(\text{position}, 1.0); \\
\]
Overall

- Vertex shader must return homogeneous coordinates in the clip space – that is in normalized device space without the $w$-division.

```glsl
    gl_Position = projection * view * model * vec4(position, 1.0);
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    gl_Position = modelViewProjectionMatrix * vec4(position, 1.0);
```

- Then GPU does:
  - $w$-division
  - Screen space transformation
Additional Links

- General overview: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
- How to derive the view matrix: http://3dgep.com/understanding-the-view-matrix/
- How to derive the projection matrices: http://www.songho.ca/opengl/gl_projectionmatrix.html
- About transforming the surface normals: http://www.lighthouse3d.com/tutorials/gls-l-tutorial/the-normal-matrix/
What was interesting for you today?

What more would you like to know?

Next time
Shading and Lighting