Computer Graphics

MTAT.03.015

Raimond Tunnel
Last week & this week

Vertex Transformations

- Construct geometry
- Define transformations
- Assign material properties

Vertex Shader
Object's local space → viewport space

Culling & Clipping
- Determine front-facing triangles
- Determine which vertices are visible

Rasterization
- Fill the triangle with fragments

Fragment Shading
- Calculate correct color values

Visibility Tests
- Is the fragment visible?

Blending
- Blend together multiple fragments
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 $\rightarrow$ World Space

- We model our objects in object space
Object Space $\rightarrow$ World Space

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

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

\[
P \cdot V \cdot M \cdot v
\]
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
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
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
  - Every child transformed relative to it
Object Space → World Space

This is what you did last week. :)}
Object Space → World Space

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

P \cdot V \cdot M \cdot \mathbf{v}
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)
- 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)
- 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)
- Think of the camera as another object in the scene.
  - It has its own \textit{rotation} and \textit{position}.
  - Scale is not 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:

$$M_{\text{camera}} = \begin{pmatrix}
right_x & up_x & back_x & pos_x \\
right_y & up_y & back_y & pos_y \\
right_z & up_z & back_z & pos_z \\
0 & 0 & 0 & 1
\end{pmatrix}$$
World Space → Camera Space

- Assume that we have a camera's model 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 from last week that the columns are the transformed standard basis...
World Space $\rightarrow$ Camera Space

- Assume that we have a camera's model 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 from last week that the columns are the transformed standard basis...

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

- **View matrix** can be found like this:
World Space $\rightarrow$ Camera Space

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

0) 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:
  0) Camera's linear transform is an orthonormal matrix
  1) Transpose it to find the inverse

\[
\begin{pmatrix}
\text{right}_x & \text{up}_x & \text{back}_x \\
\text{right}_y & \text{up}_y & \text{back}_y \\
\text{right}_z & \text{up}_z & \text{back}_z \\
\end{pmatrix}
\begin{array}{c}
^T \\
= \\
\end{array}
\begin{pmatrix}
\text{right}_x & \text{right}_y & \text{right}_z \\
\text{up}_x & \text{up}_y & \text{up}_z \\
\text{back}_x & \text{back}_y & \text{back}_z \\
\end{pmatrix}
\]

This was hinted at in the Preliminary Math Tasks 12-14.
World Space $\rightarrow$ Camera Space

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

\[
\begin{pmatrix}
  \text{right}_x & \text{right}_y & \text{right}_z \\
  \text{up}_x & \text{up}_y & \text{up}_z \\
  \text{back}_x & \text{back}_y & \text{back}_z \\
\end{pmatrix}
- \begin{pmatrix}
  \text{pos}_x \\
  \text{pos}_y \\
  \text{pos}_z \\
\end{pmatrix}
= \begin{pmatrix}
  - \text{pos}_x \\
  - \text{pos}_y \\
  - \text{pos}_z \\
\end{pmatrix}
\]
World Space $\rightarrow$ Camera Space

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

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

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

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

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

1) Transpose the rotation to inverse it
2) Negate the translation to inverse it
3) Multiply together in the reverse order
World Space → 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 view matrix?
- What do we have and what do we need?
World Space $\rightarrow$ Camera Space

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

\[
P \cdot V \cdot M \cdot \mathbf{v}
\]
Camera Space → ND Space

- 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 $[-1, 1]^3$.

Orthographic

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
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 $[-1, 1]^3$.
- 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

\[
\text{projectionMatrix} \cdot \text{viewMatrix} \cdot \text{modelMatrix} \cdot \mathbf{v} \\
\mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{v}
\]
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.

```
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.
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.

`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 \textit{left}, \textit{right}, \textit{top} and \textit{bottom} on the near plane, when the projection is \textit{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

- First find and then map the x and y coordinates of the projected point to the correct range using similar triangles.
- Next map the values to the [-1, 1] range.
Perspective Projection

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

- Next map the values to the [-1, 1] range.

- Lastly,
  - Take out the scaling parts for the matrix's diagonal.
  - Move the division with z to the homogeneous division by changing the last row of the matrix.
Perspective Projection

\[
P = \begin{pmatrix}
\text{near} & 0 & 0 & 0 \\
\text{right} & 0 & 0 & 0 \\
\text{top} & 0 & \text{near} & 0 \\
0 & 0 & -1 & 0
\end{pmatrix}
\]

- If the third row is \((0, 0, 1, 0)\), 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 $[\text{near, far}]$ 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...

\[
\begin{align*}
\frac{s \cdot \text{near} + t}{\text{near}} &= -1 \\
\frac{s \cdot \text{far} + t}{\text{far}} &= 1
\end{align*}
\]

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

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

Hint
Perspective Projection

- Applying this matrix and doing the point normalization (dividing with \(w\)), you get 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 ($ar$).

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

- **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 $\rightarrow$ 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.
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.

This will not happen!
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.
- 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)
Object Space
Object Space

World Space

\( M \)
Object Space

$$M$$

World Space

Camera (View) Space

$$V$$
Object Space

Camera (View) Space

World Space

Light calculations are usually in this space!
Camera (View) Space

Clip Space
Camera (View) Space

Normalized Device Space

Clip Space

\[
P \rightarrow \begin{pmatrix} x & y & z \\ w' & w' & w \end{pmatrix}
\]
Camera (View) Space

Normalized Device Space

Sceen Space

Clip Space

Normalized Device Space

\[
\begin{pmatrix}
\frac{x}{w} \\
\frac{y}{w} \\
\frac{z}{w}
\end{pmatrix}
\]
Vertex Shader

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

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

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

  $\text{gl\_Position} = \text{projection} \ast \text{view} \ast \text{model} \ast \text{vec4(position, 1.0)}$;

  $\text{gl\_Position} = \text{projectionMatrix} \ast \text{modelViewMatrix} \ast \text{vec4(position, 1.0)}$;

  $\text{gl\_Position} = \text{modelViewProjectionMatrix} \ast \text{vec4(position, 1.0)}$;

- 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:
What was interesting for you today?

What more would you like to know?

Next time:
Shading and Lighting