Core Mathematics for CS405 Project 1

This document explains the mathematical foundations needed for your 3D graphics project, focusing on the concepts implemented in your `math.js`, `camera.js`, and `bezier.js` files.

1. 4x4 Homogeneous Coordinates and Transformations (Part A)

In 3D graphics, we use 4x4 matrices and 4-component vectors (homogeneous coordinates) to represent and combine transformations like translation, rotation, and scaling. This allows us to represent all transformations as a single matrix multiplication: $\mathbf{p}_{\text{transformed}} = \mathbf{M} \times \mathbf{p}_{\text{object}}$.

A point $\mathbf{p}$ in 3D space (x, y, z) is represented as a 4D vector $(x, y, z, 1)^T$. The $w = 1$ component makes translations possible via matrix multiplication.

The General 4x4 Matrix Structure (Column-Major)

Your JavaScript uses **column-major** order, which is standard for WebGL/OpenGL. The matrix is laid out in memory as columns:

$$\mathbf{M} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

A. Translation Matrix (T)

To translate a point by (t_x, t_y, t_z), the values are placed in the fourth column (m_{12}, m_{13}, m_{14}).

$$\mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implementation in `math.js` (`translate`): The code sets `m[12] = tx; m[13] = ty; m[14] = tz;`. This matches the column-major indices for t_x, t_y, t_z.

B. Scaling Matrix (S)

To scale by (s_x, s_y, s_z), the values are placed on the diagonal.

$$\mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implementation in `math.js` (`scale`): The code sets `m[0] = sx; m[5] = sy; m[10] = sz;`. This matches the column-major indices for s_x, s_y, s_z.

C. Rotation Matrices ($\mathbf{R}_x, \mathbf{R}_y, \mathbf{R}_z$)

Rotations use Sine ($\sin\theta$) and Cosine ($\cos\theta$) of the rotation angle (θ).

Rotation around X-axis ($\mathbf{R}_x$):

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation around Y-axis ($\mathbf{R}_y$):

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation around Z-axis ($\mathbf{R}_z$):

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note on Order: In `main.js`, your Model matrix is calculated as $\mathbf{M} = \mathbf{T} \times \mathbf{R}_z \times \mathbf{R}_y \times \mathbf{R}_x \times \mathbf{S}$. Since matrix multiplication is applied right-to-left to the point, this means the transformations are applied in the order: **Scale** $\rightarrow$ **Rotate X** $\rightarrow$ **Rotate Y** $\rightarrow$ **Rotate Z** $\rightarrow$ **Translate**. This is the standard order for many graphics systems (Scale, then Rotate, then Translate, or SRT).

2. The View Matrix (`lookAt`) (Part B)

The **View Matrix** ($\mathbf{V}$) transforms vertices from the fixed **World Space** to the **Camera Space**. It is effectively the inverse of the camera's Model matrix. The `lookAt` function calculates this matrix based on three vectors:

1. **Eye Position ($\mathbf{e}$):** Where the camera is located.
2. **Target Position ($\mathbf{t}$):** Where the camera is looking (the center of the scene).
3. **Up Vector ($\mathbf{u}$):** The direction that is "up" for the camera (usually $(0, 1, 0)$).

The first step is to establish the Camera's Local Coordinate System (a basis):

1. **Z-axis ($\mathbf{z}_{\text{cam}}$):** The direction the camera is looking. For a right-handed system (like WebGL), this is $\mathbf{z}_{\text{cam}} = \text{normalize}(\mathbf{e} - \mathbf{t})$.
2. **X-axis ($\mathbf{x}_{\text{cam}}$):** The camera's local "right" direction. This must be perpendicular to the up vector and the view direction: $\mathbf{x}_{\text{cam}} = \text{normalize}(\mathbf{u} \times \mathbf{z}_{\text{cam}})$.
3. **Y-axis ($\mathbf{y}_{\text{cam}}$):** The camera's local "up" direction. It must be perpendicular to both $\mathbf{x}_{\text{cam}}$ and $\mathbf{z}_{\text{cam}}$: $\mathbf{y}_{\text{cam}} = \mathbf{z}_{\text{cam}} \times \mathbf{x}_{\text{cam}}$.

The final **View Matrix** ($\mathbf{V}$) is constructed using these basis vectors and the camera's position:

$$\mathbf{V} = \begin{pmatrix} x_{\text{cam},x} & x_{\text{cam},y} & x_{\text{cam},z} & -\mathbf{x}_{\text{cam}} \cdot \mathbf{e} \\ y_{\text{cam},x} & y_{\text{cam},y} & y_{\text{cam},z} & -\mathbf{y}_{\text{cam}} \cdot \mathbf{e} \\ z_{\text{cam},x} & z_{\text{cam},y} & z_{\text{cam},z} & -\mathbf{z}_{\text{cam}} \cdot \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implementation in `camera.js` (`lookAt`): The code calculates $\mathbf{z}_{\text{cam}}$, $\mathbf{x}_{\text{cam}}$, and $\mathbf{y}_{\text{cam}}$ correctly using vector subtraction, normalization, and cross-products (handled by `Vec3` in `math.js`). The matrix is then built using the basis vectors in the first three columns, and the negative dot product of the basis with the eye position for the translation components (m_{12}, m_{13}, m_{14}).

3. Projection Matrices (Part C)

The **Projection Matrix** ($\mathbf{P}$) transforms vertices from **Camera Space** into **Clip Space** (a $4 \times 4 \times 4$ cube centered at the origin, with normalized coordinates).

A. Perspective Projection

This matrix creates the illusion of depth by mapping the viewing frustum (a truncated pyramid shape defined by FOV, Aspect, Near, Far) into the clip space cube.

Let $f = \cot(\text{FOV}/2)$. The Perspective Projection Matrix is:

$$\mathbf{P}_{\text{persp}} = \begin{pmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{near}-\text{far}} & \frac{2 \cdot \text{far} \cdot \text{near}}{\text{near}-\text{far}} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Implementation in `math.js` (`perspective`): The implementation uses $nf = 1/(\text{near} - \text{far})$ to simplify the last row/column terms:

- $m_0 = f/\text{aspect}$
- $m_5 = f$
- $m_{10} = (\text{far} + \text{near}) \cdot nf$
- $m_{11} = -1$ (This is m_3 in the fourth row of the matrix above, which is index 11 in column-major.)
- $m_{14} = (2 \cdot \text{far} \cdot \text{near}) \cdot nf$

B. Orthographic Projection

This matrix maps an axis-aligned bounding box (defined by l, r, b, t, n, f) into the clip space cube, preserving parallel lines and ignoring perspective.

$$\mathbf{P}_{\text{ortho}} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implementation in `math.js` (`ortho`): The implementation correctly places the scaling terms on the diagonal (m_0, m_5, m_{10}) and the translation terms in the last column (m_{12}, m_{13}, m_{14}).

4. Bézier Curves (Part D)

A cubic Bézier curve is defined by four control points: $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$. It is a parametric curve where a point $\mathbf{B}(t)$ is calculated for any parameter t between 0 and 1.

The formula uses **Bernstein Polynomials** as blending functions:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3$$

The terms $3(1-t)^2t$ and $3(1-t)t^2$ come from the binomial expansion, ensuring the weights sum to 1.

- When $t = 0$, $\mathbf{B}(0) = 1^3\mathbf{P}_0 = \mathbf{P}_0$. (Starts at $\mathbf{P}_0$)
- When $t = 1$, $\mathbf{B}(1) = 1^3\mathbf{P}_3 = \mathbf{P}_3$. (Ends at $\mathbf{P}_3$)

Implementation in `bezier.js` (`samplePoint`):

The code uses:

- $u = (1-t)$
- $uu = u^2$, $uuu = u^3$
- $tt = t^2$, $ttt = t^3$

The calculation for the X-coordinate is:

$$x = uuu \cdot p_0.x + 3 \cdot uu \cdot t \cdot p_1.x + 3 \cdot u \cdot tt \cdot p_2.x + ttt \cdot p_3.x$$