CS405 Project 1 — Transformations, Projection, and Curves

İde Melis Yılmaz- 32400

November 6, 2025

1 Introduction

In this project, I implemented a 3D WebGL application demonstrating the mathematical foundations of computer graphics: rigid transformations, camera setup, projection, and Bézier curves by using WebGL and JavaScript. The aim is to understand the complete transformation chain.

2 Methodology

2.1 Rigid Transformations

I used affine transformations in order to transform the object - a cube in my implementation- in 3D space. Here are the three transformations:

$$\textbf{Translation}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \textbf{Scale}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\textbf{Rotation}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \textbf{Rotation}_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \textbf{Rotation}_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

I implemented this transformations in math.js file with column major order as WebGL uses.I also implemented 6 different orders for these affine transformations since the order of application will result in different. Additionally, I have defined some helper functions such as matrix multiplication in my file. These transformaitons cna be controlled and their parameters can be edited from the interface.
Here is an experiment with my interface to understand the effect of transformations to the object.

Figure 1: The Effect of Transformation Order to the End Position $TRS \neq RTS$

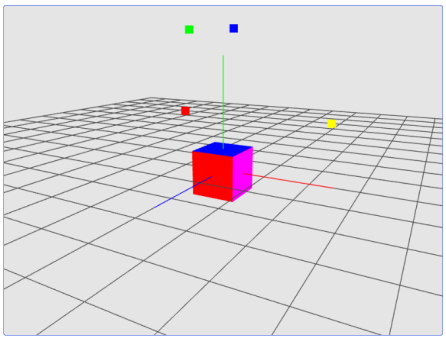

Figure 2: *
Initial Position: (T=0, R=0, S=1)

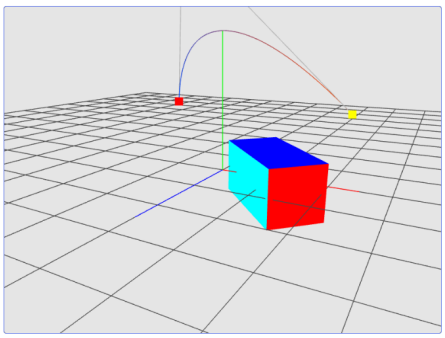

Figure 3: *
RTS $R_y : 45 \rightarrow T_x : 2 \rightarrow S_z : 2.$

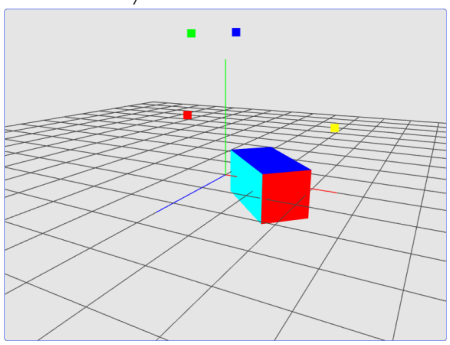

Figure 4: *
SRT $S_z : 2 \rightarrow R_y : 45 \rightarrow T_x : 2.$

2.2 Camera and View Matrix

The View Matrix **V** transforms world coordinates into the camera's local coordinate system. My implementation uses an orbital camera for navigation and the standard **LookAt** function to construct the view matrix.
I constructed the **LookAt** matrix using three key vectors: the Forward vector (**f**), the Right vector (**r**), and the Up vector (**u**), all normalized. This defines the camera's local coordinate system.

$$\mathbf{f} = \text{normalize}(\mathbf{eye} - \mathbf{target}) \qquad \mathbf{r} = \text{normalize}(\mathbf{up} \times \mathbf{f}) \qquad \mathbf{u} = \mathbf{f} \times \mathbf{r}$$

The resulting view matrices **V**: **Row-Major** (for report) and **Column-Major** (as WebGL).

$$\mathbf{V}_{\text{Row-Major}} = \begin{pmatrix} r_x & r_y & r_z & -\mathbf{r}\cdot\mathbf{eye} \\ u_x & u_y & u_z & -\mathbf{u}\cdot\mathbf{eye} \\ f_x & f_y & f_z & -\mathbf{f}\cdot\mathbf{eye} \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{V}_{\text{Column-Major}} = \begin{pmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ -\mathbf{r}\cdot\mathbf{eye} & -\mathbf{u}\cdot\mathbf{eye} & -\mathbf{f}\cdot\mathbf{eye} & 1 \end{pmatrix}$$

The **Camera** class in camera.js implements this logic. The view can be edited by mouse for rotating zooming and dragging. I showed the camera's local coordinate system with colored axes (X=red, Y=green, Z=blue) and a magenta cube at the camera's **eye** position.

Figure 5: Observation of Near & Far Planes and Field of View Changes

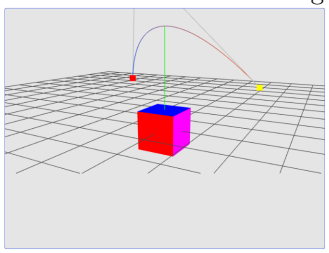

Figure 6: Near plane (10)

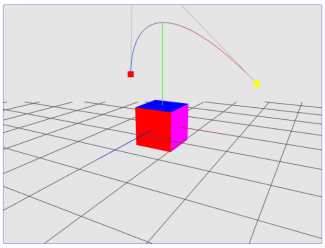

Figure 7: Far plane (20)

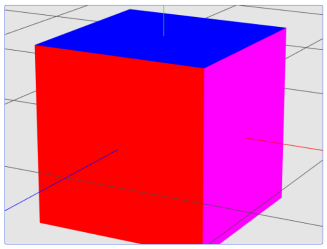

Figure 8: Field of View: 10°

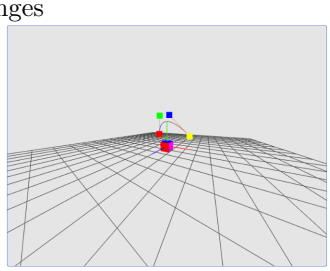

Figure 9: Field of View:120°

We can observe from figure 6 that if we increase near plane from it's initial value 0.1 to 10 grid starts to disappear showing that camera's visual area went deeper.Where as if we decrease the far plane value from 100 to 20 we can see from figure 7 that grid at the back of the cube started to disappear showing camera's visual area went shallower. In figure 8 camera has a tunnel vision because of 10° field of view whereas in figure 9 camera has a wide angle 120°.

2.3 Projections

The Projection Matrix **P** transforms vertices from the camera's view space into the clip space, preparing them for the perspective divide and final screen mapping. The Perspective matrix creates the illusion of depth by mapping a frustum (a truncated pyramid) to a normalized viewing cube. The parameters are the **Field of View** (**FOV**), **Aspect Ratio** (**aspect**), **Near Plane** (**near**), and **Far Plane** (**far**). The scale factor $f = 1/\tan(\text{fov}/2)$.

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

Orthographic projection maintains object proportions by mapping a rectangular view volume directly to the viewing cube, resulting in parallel projection. The parameters are **left**, **right**, **top**, **bottom**, **near**, and **far**.

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

Frustum Visualization The camera's view volume is visualized using a frustum geometry, which can be toggled on. This wireframe visualization helps in understanding the effect of modifying the **FOV** and the **Near**/**Far** clipping planes. However it is very hard to see the frustrum of the camera from that camera so i showed the edges of frustrums near plane to the camera with blue lines.

Figure 10: Perspective, Orthagonal Projections and Frustrum visualisation

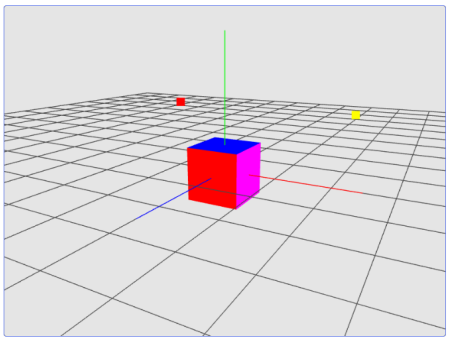

Figure 11: Perspective

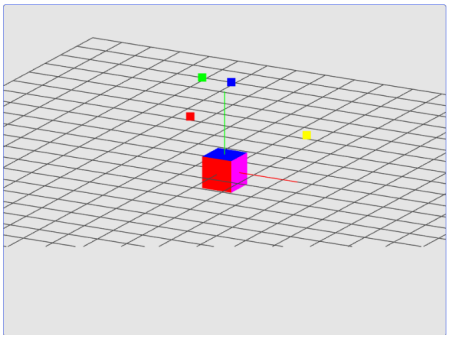

Figure 12: Orthagonal

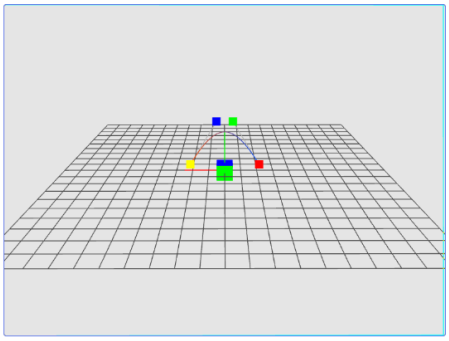

Figure 13: Frustrum

2.4 Bezier Curves

In the project I used a cubic Bézier curve defined by four control points: P_0, P_1, P_2, and P_3. The curve is calculated using the parametric formula, based on the fourth-degree Bernstein polynomials $\mathbf{B}_{i,3}(t)$:

$$\mathbf{B}(t) = \sum_{i=0}^{3} \mathbf{P}_i \mathbf{B}_{i,3}(t) \quad \text{for } 0 \leq t \leq 1$$

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

The first derivative $\mathbf{B}'(t)$ provides the tangent vector at any point t along the curve, which is used to orient an object moving along the path12.

$$\mathbf{B}'(t) = -3(1-t)^2\mathbf{P}_0 + 3(1-t)^2\mathbf{P}_1 - 6(1-t)t\mathbf{P}_1 + 6(1-t)t\mathbf{P}_2 - 3t^2\mathbf{P}_2 + 3t^2\mathbf{P}_3$$

This derivative is implemented in the **getBezierTangent** function in bezier.js.
The **BezierCurve** class manages four 3D control points with a default shape designed for visibility. The curve itself is rendered by calculating it in **100** points, creating a smooth visual path with a color gradient (blue to red). The control polygon (gray lines connecting the points) is also visualized. An animated cube moves along the curve based on the parameter t. The cube in **updateAnimation** function goes backward and forward where the parameter t moves from 0 to 1 and then back from 1 to 0. The cube is oriented using the tangent vector $\mathbf{B}'(t)$ at the current position t. Control points are interactive and can be dragged by the user.

Bézier Curve Validation:

- Endpoint Interpolation: The formula ensures that the curve passes through the first and last control points: $\mathbf{B}(0) = \mathbf{P}_0$ and $\mathbf{B}(1) = \mathbf{P}_3$.
- Tangent at Endpoints: The tangents at the endpoints are correctly determined by the adjacent control points: $\mathbf{B}'(0) = 3(\mathbf{P}_1 - \mathbf{P}_0)$ and $\mathbf{B}'(1) = 3(\mathbf{P}_3 - \mathbf{P}_2)$. The animated object's orientation confirms this tangent direction.

Figure 14: Bezier Curve

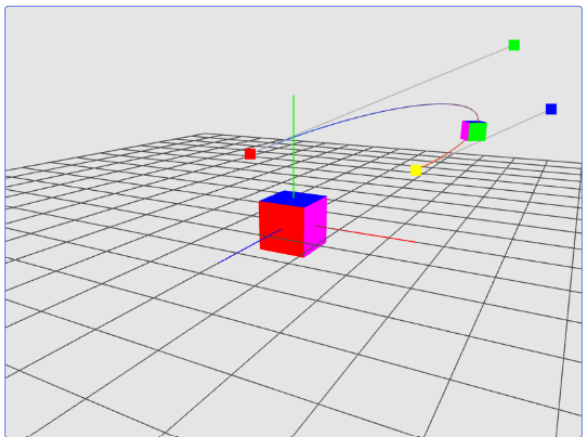

Figure 15: Bezier Curve Position 1

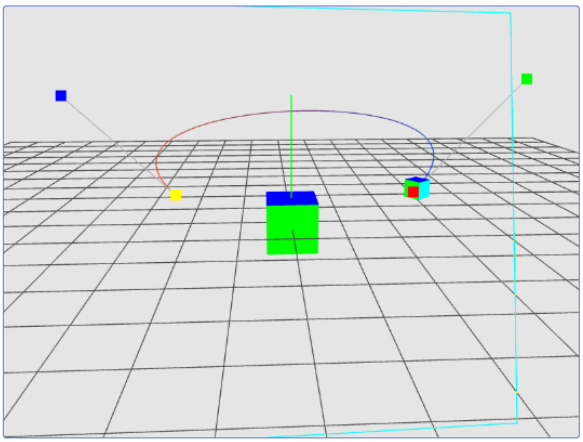

Figure 16: Bezier Curve Position 2

2.5 The Complete Transformation Pipeline

The rendering loop in render.js successfully implements the complete transformation chain, applying all matrices in the correct order in the vertex shader:

$$\mathbf{p}_{\text{clip}} = \mathbf{P} \times \mathbf{V} \times \mathbf{M} \times \mathbf{p}_{\text{object}}$$

Where:$\mathbf{P}$ is the Projection Matrix .$\mathbf{V}$ is the View Matrix .$\mathbf{M}$ is the Model Matrix.$\mathbf{p}_{\text{object}}$ is the vertex in object space.

2.6 Interface

I combined transformations, camera, projection, and Bézier elements into a single WebGL scene. I also added controls to toggle projection mode, start/stop animation, and reset. Full interface is in the appendix.

2.7 Conclusion

In this project I developed an interactive 3D WebGL application demonstrating the core mathematical principles of computer graphics.

The implementation of the Model Matrix ($\mathbf{M}$) handled rigid-body transformations (Translation, Rotation, and Scale) across six different composition orders, with experimental validation showing the order dependence of matrix multiplication.Also matrix can be exported from the interface as a JSON format. The View Matrix ($\mathbf{V}$) was constructed using the **LookAt** function, enabling orbital camera control and providing visualization of the camera's local coordinate system.

The project provides a comparison between Perspective and Orthographic Projection, allowing users to adjust parameters like **FOV** and clipping planes while visualizing the camera frustum. Finally, the Bézier Curve implementation demonstrated curve generation, visualization of the control polygon, and applied the first derivative to smoothly animate an object following the path and maintaining tangent-based orientation.

The modular structure, using separate JavaScript files for math, camera, Bézier curves, and rendering, provides a foundation for understanding given topics in 3D graphics.

A Interface

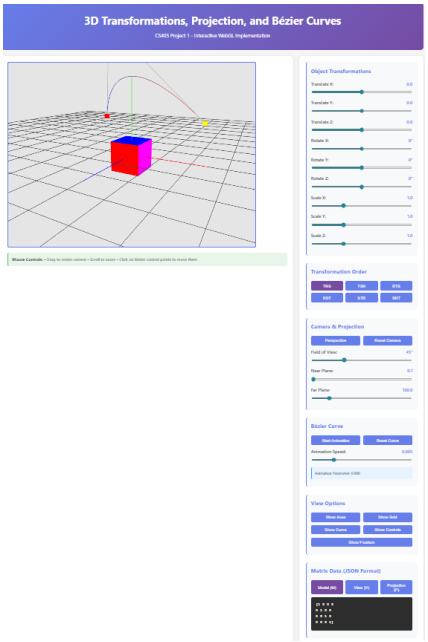

Figure 17: Full view of the interface