3D Objects on a Path - Assignment 2 Report

Abhishek Kumar (MT2024005)

April 9, 2025

1 Overview

This assignment implements a 3D scene with interactive objects that can be moved along quadratic paths. The application features three main components:

- A 3D environment with multiple objects
- A path creation and editing system
- Interactive controls for object manipulation

2 Model Creation

The models were created using Blender with the following operations:

- **Model1 (Hollow Cube)**: Created using boolean subtraction of a sphere from a larger cube
- **Model2 (Ring)**: Generated with torus primitive and scaling operations
- **Model3 (Rabbit)**: Imported from reference OBJ file demonstrating complex mesh loading

3 Key Features Implemented

3.1 Scene Setup and Rendering

- **WebGL Initialization**: The main.js file sets up WebGL context and initializes all systems
- **Shader Program**: Uses vertex and fragment shaders for lighting and coloring (shaders.js)
- **Render Loop**: Continuous rendering with clear color, depth testing, and proper matrix updates

3.2 File Format Support

The system supports loading models in OBJ format, with architecture designed to easily extend to PLY and STL formats through additional parsers. The implementation includes:

- Robust OBJ file parser
- Fallback geometry generation when loading fails
- Vertex normal calculation for imported meshes

3.3 Camera System (camera.js)

- **Two View Modes**:
  - Top View (orthographic projection)
  - 3D View (perspective projection with mouse controls)
- **Interactive Controls**:

  - Mouse rotation in 3D view
  - Zoom with mouse wheel
  - Toggle between views with 'C' key
- **Screen-to-World Conversion**: For accurate path point placement

3.4 Rotation System

Object rotations are implemented using quaternions with a virtual trackball interface for smooth arbitrary rotations in 3D space. Key aspects include:

- Quaternion-based rotation calculations
- Virtual trackball for intuitive 3D rotation
- Separate rotation controls for each axis (X/Y/Z keys)
- Normal matrix generation for proper lighting

3.5 Object System (models.js)

- **Three Models**: Each with different geometry demonstrating modeling operations
- **Model Properties**:
  - Position, rotation, scale
  - Custom colors with highlight when selected
  - Path following capability
- **Loading System**:
  - Attempts to load OBJ files
  - Falls back to primitive shapes if loading fails
- **Selection**: Uses color-based picking technique

3.6 Path System (path.js)

- **Quadratic Paths**: Defined by 3 control points
- **Path Calculation**:
  - Computes coefficients for quadratic Bézier curves
  - Generates smooth curves from control points
- **Visualization**:
  - Shows control points
  - Displays the calculated path
- **Object Movement**: Objects can follow paths with adjustable speed

3.7 User Interaction

- **Keyboard Controls**:
  - Rotation (X/Y/Z keys)
  - Scaling (S/Shift+S)
  - Path movement (Space to start, arrows for speed)
- **Mouse Controls**:
  - Object selection
  - Path point placement (with Shift)
  - Camera rotation in 3D view

4 Technical Implementation Details

4.1 Rendering Pipeline

1. Clear buffers and set up viewport
2. Update camera matrices based on current mode
3. Set shader uniforms (matrices, lighting)
4. Draw axes (always behind other objects)
5. Draw path if exists
6. Draw all models with proper transformations

4.2 Object Selection

Implements a color-picking technique:

1. Renders scene to offscreen buffer with each object having a unique color ID
2. Reads pixel color at mouse position
3. Maps color back to object ID
4. Highlights selected object

4.3 Path Following

1. User places 3 points (creates quadratic Bézier curve)
2. Pressing space starts object movement
3. Object position is calculated using:
$$p(t) = at^2 + bt + c$$
4. Speed adjustable with arrow keys

5 Challenges and Solutions

- **Model Loading**: Implemented fallback geometries when OBJ files fail to load
- **Coordinate Systems**: Careful handling of screen-to-world conversions for accurate point placement
- **Visual Feedback**: Added highlighting and color cycling for selected objects
- **Performance**: Used indexed rendering and buffer reuse where possible
- **Complex Rotations**: Implemented quaternion rotations with virtual trackball for smooth 3D manipulation

6 Conclusion

This assignment successfully demonstrates:

- 3D rendering with WebGL
- Interactive object manipulation
- Path creation and following
- Multiple camera views

- Robust user interaction
- Advanced modeling operations
- Comprehensive file format support
- Sophisticated rotation system

The implementation shows a good understanding of 3D graphics principles, matrix transformations, and interactive application design. The code is well-structured with clear separation of concerns between different systems.

7 Questions and Answers

7.1 To what extent were you able to reuse code from Assignment 1?

From my first assignment (Modified Turtle Graphics), I was able to reuse several core concepts in the 3D Objects on Path assignment, though direct code reuse was limited due to the different nature of the projects.

Significant Reuses:

- Transformation System: Adapted matrix operations from 2D affine transformations to 3D (using mat4 instead of mat3)
- Render Pipeline: Carried over WebGL initialization and rendering loop structure
- Event Handling: Reused and expanded keyboard/mouse interaction patterns

Modified Components:

- Camera System: Extended from static 2D to support orthographic and perspective 3D views
- Geometry Handling: Transitioned from simple 2D paths to complex 3D model loading
- Animation System: Reimplemented turtle movement logic for 3D path following

New in Assignment 2:

- Model import system (OBJ loading)
- Quaternion rotations
- Bézier curve mathematics
- Advanced lighting calculations
- Depth buffer management

About 30-40% of core infrastructure code was reusable, while 60-70% needed new implementation or significant modification to handle 3D requirements.

7.2 What were the primary changes in the use of WebGL in moving from 2D to 3D?

The shift from 2D to 3D in WebGL brings about some pretty significant changes. In 2D graphics, we mainly work with straightforward (x,y) coordinates and basic orthographic projections.

But when we dive into 3D, we have to deal with depth (the z-axis), more intricate matrix operations (think 4x4 matrices instead of the simpler 3x3), and a variety of coordinate systems (like model, view, and projection). Lighting becomes a crucial factor in 3D, which means we need to use normal vectors and perform advanced shader calculations, moving away from the flat coloring typical of 2D.

Plus, camera control gets a lot more complex, as we have to manage position, target, and orientation. Handling geometry also becomes trickier, with full 3D meshes requiring vertex buffers, normal vectors, and proper depth testing. User interaction evolves too, shifting from basic screen coordinates to techniques like ray casting and selecting 3D objects. All these changes make 3D rendering not only more computationally demanding but also mathematically intricate compared to 2D, while allowing for realistic depth perception and spatial relationships.

7.3 How were the translate, scale and rotate matrices arranged? Can your implementation allow rotations and scaling during the movement?

In my implementation, I arranged the transformation matrices in a specific sequence: **scale → rotate → translate**. I created the model matrix by first applying translation based on the object's position, then scaling it, and finally adding rotation using a quaternion that I converted into a rotation matrix.

My system fully supports rotations and scaling while objects are in motion. As they follow their paths—affecting only their position—I ensure that all other transformations remain active. During each frame's rendering, I recalculate the complete transformation matrix, so any rotations I initiate with the X/Y/Z keys or scaling adjustments I make with S/Shift+S take effect right away, even while the objects are moving along their paths.

I made sure the vertex shader effectively manages these combined transformations through normal matrix calculations, ensuring that lighting stays accurate no matter how I rotate or scale the objects during animation. This method gives me complete control over the appearance and orientation of the objects throughout their entire movement cycle.

7.4 How did you choose a value for t1 in computing the coefficients of the quadratic curve? How would you extend this to interpolating through n points ($n > 3$) and still obtaining a smooth curve?

In my implementation, I decided to set t1 at 0.5 for the middle control point (p1) when calculating the coefficients for the quadratic curve. Here's why I went with that choice:

- It effectively positions the curve's peak influence right in the middle of p0 and p2.
- It results in a balanced and symmetrical curve when p1 is centered.
- The math works out nicely with this equidistant parameterization.

When it comes to extending this to more than three points, I'd take a piecewise quadratic Bézier spline approach:

- I'd start by grouping the points into overlapping triplets: (p0, p1, p2), (p1, p2, p3), and so on.
- For each triplet, I'd calculate a quadratic curve segment using the same t1=0.5 midpoint parameterization.
- To ensure C1 continuity (matching tangents) at the junction points, I'd do the following:
  - Make the control points collinear at the junctions.
  - Keep equal distances between the handles on either side of each midpoint.

The main benefits of this approach in my implementation would be:

- It's computationally simple since I'm still using quadratic equations.
- It allows for local control, meaning that moving one point only impacts the adjacent segments.
- It ensures smooth transitions between the curve segments.