3D Engine Core Documentation

Rendering System Architecture

Overview of the Rendering Pipeline

At its core, 3D rendering is about transforming three-dimensional data into a two-dimensional image that can be displayed on your screen. Our engine's rendering pipeline consists of several stages that data must flow through:

1. **Model Definition**: Objects are defined as collections of vertices and faces in 3D space
2. **Transformation**: Vertices are transformed through various coordinate systems
3. **Rasterization**: 3D geometry is converted into pixels on the 2D screen
4. **Fragment Processing**: Each pixel is assigned appropriate color values
5. **Display**: The final image is presented on screen

Unlike modern GPUs that handle most of these operations in hardware, our software renderer performs all these steps manually, giving you a clear understanding of the process.

Component Interactions and Data Flow

Our renderer follows a data-driven architecture where information flows through distinct components:

1. **Scene Graph**: Contains all objects, lights, and camera information
2. **Transformation System**: Applies matrices to convert vertices between coordinate systems
3. **Rasterizer**: Converts transformed geometry into screen pixels
4. **Frame Buffer**: Stores pixel data until rendering is complete
5. **Display Interface**: Presents the completed frame to the screen

Data flows primarily in one direction, from 3D scene data to 2D screen pixels. Each component performs a specific transformation on the data before passing it to the next stage.

Canvas and Framebuffer Management

Our engine uses HTML5 Canvas for display, but the actual rendering calculations happen in our own managed buffer:

```
// Creating the canvas and context
const canvas = document.getElementById('renderCanvas');
const ctx = canvas.getContext('2d');

// Creating our framebuffer
const frameBuffer = ctx.createImageData(canvas.width, canvas.height);
const pixels = frameBuffer.data;
```

The framebuffer is essentially a large array where each pixel is represented by four consecutive values (Red, Green, Blue, Alpha). By manipulating this array directly, we achieve faster performance than drawing individual pixels through the Canvas API.

When a frame is complete, we transfer the entire framebuffer to the canvas at once:

```
// After all rendering is complete
ctx.putImageData(frameBuffer, 0, 0);
```

This approach gives us direct control over every pixel while leveraging the canvas for efficient display.

Pixel and Primitive Operations

`setPixel()`: Individual Pixel Manipulation

At the most fundamental level, 3D rendering is about determining the color of individual pixels. Our `setPixel()` function handles this core operation:

```
function setPixel(x, y, r, g, b) {
    // Convert coordinates to array index
    const index = (y * canvas.width + x) * 4;

    // Bounds checking to prevent buffer overflows
    if (index >= 0 && index < pixels.length - 3) {
```

```
        pixels[index] = r;     // Red
        pixels[index+1] = g;   // Green
        pixels[index+2] = b;   // Blue
        pixels[index+3] = 255; // Alpha (fully opaque)
    }
}
```

This function translates the 2D coordinates (x, y) into the appropriate position in our 1D array. The multiplication by 4 accounts for the four values (RGBA) that define each pixel.

Understanding this function is crucial because it's the final step in our rendering pipeline - every visual element ultimately calls `setPixel()` to become visible on screen.

`clearScreen()`: Frame Initialization

Before drawing a new frame, we need to clear the previous one. The `clearScreen()` function resets our framebuffer:

```
function clearScreen(r = 0, g = 0, b = 0) {
    // Fill the entire buffer with the specified color
    for (let i = 0; i < pixels.length; i += 4) {
        pixels[i] = r;     // Red
        pixels[i+1] = g;   // Green
        pixels[i+2] = b;   // Blue
        pixels[i+3] = 255; // Alpha
    }
}
```

This function iterates through every pixel in our framebuffer, setting them all to the specified color (default black). In a more advanced renderer, this would also reset the depth buffer used for hidden surface removal.

`mixColors()`: Color Interpolation Technique

When drawing lines or filling triangles, we often need to smoothly blend between colors. Our `mixColors()` function implements linear interpolation (lerp) between two color values:

```
function mixColors(color1, color2, factor) {
    // Ensure factor is between 0 and 1
    const t = Math.max(0, Math.min(1, factor));

    // Linear interpolation between colors
    return {
        r: Math.round(color1.r * (1 - t) + color2.r * t),
        g: Math.round(color1.g * (1 - t) + color2.g * t),
        b: Math.round(color1.b * (1 - t) + color2.b * t)
    };
}
```

The `factor` parameter (between 0 and 1) determines how much of each color contributes to the result. When `factor` is 0, the result is identical to `color1`; when it's 1, the result matches `color2`; values in between create a smooth gradient.

This interpolation is essential for creating smooth color transitions and is used extensively in our line drawing and triangle filling algorithms.

Line Drawing

`drawLine()`: Bresenham-Inspired Algorithm

Line drawing is one of the most fundamental operations in computer graphics. Our engine uses a Bresenham-inspired algorithm that efficiently determines which pixels should be illuminated to form a straight line between two points:

```
function drawLine(x0, y0, x1, y1, color1, color2) {
    // Calculate differences and steps
    const dx = Math.abs(x1 - x0);
    const dy = Math.abs(y1 - y0);
    const sx = x0 < x1 ? 1 : -1;
    const sy = y0 < y1 ? 1 : -1;
    let err = dx - dy;

    // Current position
    let x = x0;
```

```
    let y = y0;

    // Total distance for interpolation
    const totalDist = Math.sqrt(dx*dx + dy*dy);
    let currentDist = 0;

    while (true) {
        // Calculate interpolation factor
        const t = totalDist === 0 ? 0 : currentDist / totalDist;

        // Interpolate color and draw pixel
        const color = mixColors(color1, color2, t);
        setPixel(x, y, color.r, color.g, color.b);

        // Exit condition
        if (x === x1 && y === y1) break;

        // Calculate next position
        const e2 = 2 * err;
        if (e2 > -dy) {
            err -= dy;
            x += sx;
            currentDist += Math.abs(sy);
        }
        if (e2 < dx) {
            err += dx;
            y += sy;
            currentDist += Math.abs(sx);
        }
    }
}
```

The Bresenham algorithm works by choosing optimal pixels to represent a line without using floating-point calculations. It determines which pixels to illuminate by tracking an error value that represents how far the ideal line passes from each pixel center.

Color Interpolation Across Line Segments

When drawing a line between two differently colored endpoints, we need to smoothly transition the color. Our implementation calculates a parameter `t` based on the current distance along the line, then uses this to interpolate between the two colors.

The color at any point is determined by:

- When `t = 0` (start of line): 100% of color1, 0% of color2
- When `t = 1` (end of line): 0% of color1, 100% of color2
- When `t = 0.5` (middle of line): 50% of each color

This creates a smooth gradient effect along the length of the line.

Boundary Handling and Clipping

Our line drawing function includes bounds checking in the `setPixel()` function to ensure we never attempt to draw outside the valid canvas area. However, a more efficient approach is to clip lines against the canvas boundaries before processing them.

Line clipping uses algorithms like Cohen-Sutherland to determine which portions of a line are visible and which can be discarded. This increases efficiency by avoiding calculations for invisible segments.

Triangle Rendering

`drawTriangle()`: Rasterization Algorithm

Triangles are the fundamental building blocks of 3D graphics. Our `drawTriangle()` function converts a 3D triangle into colored pixels on the 2D screen:

```
function drawTriangle(v1, v2, v3, color1, color2, color3) {
    // Sort vertices by y-coordinate (top to bottom)
    let vertices = [v1, v2, v3].sort((a, b) => a.y - b.y);
    const [top, middle, bottom] = vertices;

    // Match colors to sorted vertices
    const colors = [color1, color2, color3];
    const sortedColors = [
```

```
        colors[vertices.indexOf(top)],
        colors[vertices.indexOf(middle)],
        colors[vertices.indexOf(bottom)]
    ];

    // Calculate slopes
    const slopeTop2Mid = (middle.x - top.x) / Math.max(1, middle.y - top.y);
    const slopeTop2Bottom = (bottom.x - top.x) / Math.max(1, bottom.y - top.y);
    const slopeMid2Bottom = (bottom.x - middle.x) / Math.max(1, bottom.y - middle.y);

    // Draw the upper part of the triangle
    for (let y = Math.ceil(top.y); y <= middle.y; y++) {
        const dy1 = (y - top.y) / Math.max(1, middle.y - top.y);
        const dy2 = (y - top.y) / Math.max(1, bottom.y - top.y);

        const x1 = top.x + slopeTop2Mid * (y - top.y);
        const x2 = top.x + slopeTop2Bottom * (y - top.y);

        const startX = Math.min(x1, x2);
        const endX = Math.max(x1, x2);

        // Draw horizontal scanline
        for (let x = Math.ceil(startX); x <= endX; x++) {
            // Calculate barycentric coordinates for color interpolation
            const bary = calculateBarycentricCoords(x, y, top, middle, bottom);

            // Interpolate color using barycentric coordinates
            const color = interpolateColor(sortedColors[0], sortedColors[1], sortedColors[2], bary);

            setPixel(x, y, color.r, color.g, color.b);
        }
    }

    // Draw the lower part of the triangle (similar logic)
    // ...
}
```

Scanline Conversion Technique

Our triangle rasterization uses a scanline approach, which processes the triangle one horizontal line at a time. For each scanline:

1. We calculate where the scanline intersects the triangle edges
2. We fill all pixels between those intersection points
3. We interpolate colors across the scanline based on the triangle's vertex colors

This approach is efficient because it minimizes the number of edge calculations and takes advantage of horizontal memory access patterns.

Vertex Sorting and Edge Walking

Before rasterization begins, we sort the triangle vertices by their y-coordinates (from top to bottom). This simplifies the algorithm by allowing us to divide the triangle into two parts:

- The upper part between the top vertex and the middle vertex
- The lower part between the middle vertex and the bottom vertex

For each part, we can use consistent rules to determine the left and right edges for our scanline algorithm.

Affine Texture Mapping and Color Interpolation

While our basic renderer uses color interpolation, more advanced features include texture mapping. Affine texture mapping assigns texture coordinates to each vertex, then interpolates these coordinates across the triangle face:

```
function interpolateTextureCoords(u1, v1, u2, v2, u3, v3, bary) {
    return {
        u: u1 * bary.a + u2 * bary.b + u3 * bary.c,
        v: v1 * bary.a + v2 * bary.b + v3 * bary.c
    };
}
```

While simple to implement, affine texture mapping can cause distortion in perspective views. More advanced renderers use perspective-correct texture mapping, which accounts for the varying depth across the triangle.

Animation System

Request Animation Frame Usage

Our animation system uses the browser's `requestAnimationFrame` API to synchronize rendering with the display's refresh rate:

```
function animationLoop(timestamp) {
    // Calculate time delta
    const deltaTime = timestamp - lastFrameTime;
    lastFrameTime = timestamp;

    // Update scene (rotate objects, etc.)
    updateScene(deltaTime);

    // Render the scene
    renderScene();

    // Request next frame
    requestAnimationFrame(animationLoop);
}

// Start the animation loop
requestAnimationFrame(animationLoop);
```

This approach offers several advantages:

- The browser optimizes rendering for performance and power efficiency
- Animations automatically pause when the tab is inactive
- Frame rate adapts to the device's display capabilities

Time-Based Animation

Rather than assuming a fixed frame rate, our engine uses the time delta between frames to calculate movements:

```
function updateScene(deltaTime) {
    // Calculate rotation based on time (60 degrees per second)
    const rotationAmount = (60 * Math.PI / 180) * (deltaTime / 1000);

    // Apply rotation to objects
    for (let object of scene.objects) {
        object.rotation.y += rotationAmount;
    }
}
```

This time-based approach ensures consistent animation speeds regardless of the device's performance or frame rate. Objects move the same distance over time whether running at 30fps or 120fps.

Smooth Rotation Implementation

Rotations in 3D space are implemented using rotation matrices, which we'll cover in more detail in the 3D Math Documentation. For smooth animation, we incrementally update rotation angles:

```
function rotateObject(object, deltaTime) {
    // Update rotation angles
    object.rotation.x += object.rotationSpeed.x * deltaTime;
    object.rotation.y += object.rotationSpeed.y * deltaTime;
    object.rotation.z += object.rotationSpeed.z * deltaTime;

    // Recalculate transformation matrix
    object.transformMatrix = calculateWorldMatrix(object);
}
```

By separating rotation speeds from the actual rotation angles, we can easily implement various animation patterns such as oscillation, acceleration, or user-controlled rotation.

Frame Timing and Synchronization

Consistent timing is crucial for smooth animation. Our engine tracks the time between frames and can adapt to varying frame rates:

```
// Track frame times for performance monitoring
const frameTimes = [];
let frameTimeIndex = 0;
const MAX_FRAME_SAMPLES = 60; // Track last 60 frames

function updateFrameMetrics(deltaTime) {
    // Store current frame time
    frameTimes[frameTimeIndex] = deltaTime;
    frameTimeIndex = (frameTimeIndex + 1) % MAX_FRAME_SAMPLES;

    // Calculate average frame time
    const sum = frameTimes.reduce((a, b) => a + b, 0);
    const avgFrameTime = sum / frameTimes.filter(t => t !== undefined).length;

    // Calculate FPS
    const fps = 1000 / avgFrameTime;

    // Update display if needed
    if (showPerformanceMetrics) {
        displayFPS(Math.round(fps));
    }
}
```

This system not only helps maintain smooth animation but also provides valuable metrics for performance optimization.

Culling and Visibility

Backface Culling Implementation

To improve rendering efficiency, we don't draw triangles that face away from the camera. This technique, called backface culling, uses the dot product between the triangle's normal vector and the viewing direction:

```
function isBackface(triangle, cameraPosition) {
    // Calculate triangle normal
    const v1 = vector.subtract(triangle.vertices[1], triangle.vertices[0]);
    const v2 = vector.subtract(triangle.vertices[2], triangle.vertices[0]);
    const normal = vector.normalize(vector.crossProduct(v1, v2));

    // Calculate view direction (from triangle to camera)
    const viewDir = vector.normalize(vector.subtract(cameraPosition, triangle.vertices[0]));

    // Dot product < 0 means triangle is facing away from camera
    return vector.dotProduct(normal, viewDir) < 0;
}
```

This is a simple optimization that instantly reduces the rendering workload by up to 50% in closed objects, as approximately half of all triangles face away from the camera at any time.

View Frustum Clipping

The view frustum is the 3D volume visible to the camera. Objects outside this volume should be excluded from rendering to save processing time. Our clipping implementation tests each vertex against the six planes that define the view frustum:

```
function isVertexInFrustum(vertex, frustumPlanes) {
    for (let plane of frustumPlanes) {
        // Calculate signed distance from vertex to plane
        const distance =
            plane.a * vertex.x +
            plane.b * vertex.y +
            plane.c * vertex.z +
            plane.d;

        // If vertex is behind any plane, it's outside the frustum
        if (distance < 0) {
            return false;
        }
    }
```

```
        return true;
    }
```

For more complex objects, we use bounding volumes (spheres or boxes) that encompass the object. Testing these simpler shapes against the frustum is faster than testing every vertex.

Near Plane Handling

The near plane is particularly important in 3D rendering because objects between the camera and this plane would project improperly in perspective projection. Our engine handles near-plane clipping by:

1. Identifying triangles that intersect the near plane
2. Clipping those triangles against the plane to create new triangles
3. Discarding the portions behind the near plane
4. Rendering only the visible portions

This process involves calculating new vertices at the intersection points and interpolating attributes like colors or texture coordinates.

W-Coordinate Significance

In homogeneous coordinates, the w-coordinate plays a critical role in perspective projection. After transformation by the projection matrix, the w-coordinate represents the original vertex's depth information:

```
function perspectiveDivide(vertex) {
    // Store the original w value for depth testing
    const invW = 1.0 / vertex.w;

    // Divide x, y, z by w to get normalized device coordinates
    return {
        x: vertex.x * invW,
        y: vertex.y * invW,
        z: vertex.z * invW,
        w: vertex.w  // Keep original w for later use
    };
}
```

The w-coordinate is used in several important ways:

- Perspective division (dividing x, y, z by w) creates the perspective effect
- The original w value helps with correct attribute interpolation
- W values can be used for depth sorting and visibility determination

Performance Considerations

Optimizations Implemented in Drawing Routines

Our rendering engine includes several optimizations to improve performance:

1. **Integer-only arithmetic**: Where possible, we avoid floating-point operations
2. **Pre-computed tables**: Common calculations are pre-computed and stored
3. **Scan-line coherence**: We reuse calculations across horizontal lines
4. **Early rejection tests**: Quick tests eliminate obviously invisible elements
5. **Batch processing**: Similar operations are grouped to reduce state changes

For example, our line drawing algorithm uses integer arithmetic to avoid the performance cost of floating-point calculations:

```
function drawLineOptimized(x0, y0, x1, y1, color) {
    // Use integer-only arithmetic for core algorithm
    let dx = Math.abs(x1 - x0);
    let dy = Math.abs(y1 - y0);
    let sx = x0 < x1 ? 1 : -1;
    let sy = y0 < y1 ? 1 : -1;
    let err = dx - dy;

    while (true) {
        setPixel(x0, y0, color.r, color.g, color.b);

        if (x0 === x1 && y0 === y1) break;
```

```
        let e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 < dx) { err += dx; y0 += sy; }
    }
}
```

Areas for Potential Improvement

While our current implementation prioritizes clarity and educational value, several areas could be optimized further:

1. **SIMD operations**: Using SIMD.js for parallel processing of multiple vertices
2. **Web Workers**: Offloading computation to background threads
3. **WebGL fallback**: Using hardware acceleration when available
4. **Spatial data structures**: Implementing octrees or BVHs for scene organization
5. **Level of detail**: Dynamically adjusting model complexity based on distance

Trade-offs Between Quality and Speed

Every renderer must balance visual quality against performance. Our engine offers several adjustable parameters:

1. **Resolution**: Higher resolutions provide more detail but require more processing
2. **Draw distance**: Objects beyond a certain distance can be excluded
3. **Shading complexity**: Simple flat shading vs. more complex techniques
4. **Polygon count**: Simplifying models reduces rendering load
5. **Anti-aliasing**: Techniques like MSAA improve quality but impact performance

For example, our engine can switch between different triangle fill methods based on performance requirements:

```
function drawTriangle(v1, v2, v3, color, qualityLevel) {
    switch(qualityLevel) {
        case "low":
            // Simple flat shading - fastest
            drawFlatTriangle(v1, v2, v3, color);
            break;
        case "medium":
            // Gouraud shading - moderate performance
            drawGouraudTriangle(v1, v2, v3, color1, color2, color3);
            break;
        case "high":
            // Phong shading - best quality, slowest
            drawPhongTriangle(v1, v2, v3, normals, material, lights);
            break;
    }
}
```

Rendering Pipeline Walkthrough

Step-by-Step Explanation of a Frame Render

Let's walk through the entire process of rendering a single frame:

1. **Clear buffers**: Reset the color buffer to the background color

```
clearScreen(bgColor.r, bgColor.g, bgColor.b);
```

2. **Update animations**: Calculate new positions and orientations

```
updateAnimations(deltaTime);
```

3. **Transform vertices**: Apply model, view, and projection matrices

```
for (let object of scene.objects) {
    // Apply world transformation (position, rotation, scale)
    const worldMatrix = calculateWorldMatrix(object);

    // Apply camera transformation
    const viewMatrix = calculateViewMatrix(camera);
```

```
    // Apply projection transformation
    const projMatrix = calculateProjectionMatrix(camera);

    // Combined transformation matrix
    const mvpMatrix = multiplyMatrices(
        projMatrix,
        multiplyMatrices(viewMatrix, worldMatrix)
    );

    // Transform each vertex
    for (let vertex of object.vertices) {
        const transformedVertex = multiplyMatrixVector(mvpMatrix, vertex);
        // Store for rasterization
        object.transformedVertices.push(transformedVertex);
    }
}
```

4. **Cull invisible triangles**: Remove backfaces and out-of-view triangles

```
const visibleTriangles = [];
for (let triangle of object.triangles) {
    if (!isBackface(triangle, camera.position) &&
        isInViewFrustum(triangle, viewFrustum)) {
        visibleTriangles.push(triangle);
    }
}
```

5. **Clip triangles**: Handle triangles that cross the view frustum

```
const clippedTriangles = [];
for (let triangle of visibleTriangles) {
    const clipped = clipTriangleAgainstFrustum(triangle, viewFrustum);
    clippedTriangles.push( ... clipped);
}
```

6. **Sort triangles**: Order by depth for correct transparent rendering

```
clippedTriangles.sort((a, b) => {
    // Calculate average Z depth of each triangle
    const zAvgA = (a.vertices[0].z + a.vertices[1].z + a.vertices[2].z) / 3;
    const zAvgB = (b.vertices[0].z + b.vertices[1].z + b.vertices[2].z) / 3;

    // Sort back-to-front for transparency
    return zAvgB - zAvgA;
});
```

7. **Perspective division**: Convert to normalized device coordinates

```
for (let triangle of clippedTriangles) {
    for (let i = 0; i < 3; i++) {
        const v = triangle.vertices[i];
        v.x /= v.w;
        v.y /= v.w;
        v.z /= v.w;
    }
}
```

8. **Viewport transformation**: Convert to screen coordinates

```
function viewportTransform(vertex, width, height) {
    return {
        x: Math.floor((vertex.x + 1) * width / 2),
        y: Math.floor((1 - vertex.y) * height / 2),
        z: vertex.z  // Keep for depth testing
    };
}
```

9. **Rasterize triangles**: Convert to pixels with appropriate shading

```
for (let triangle of clippedTriangles) {
    // Convert vertices to screen space
    const screenVerts = triangle.vertices.map(v =>
        viewportTransform(v, canvas.width, canvas.height)
    );

    // Draw the triangle
    drawTriangle(
        screenVerts[0],
        screenVerts[1],
        screenVerts[2],
        triangle.color
    );
}
```

10. **Display the frame**: Transfer the framebuffer to the canvas

```
ctx.putImageData(frameBuffer, 0, 0);
```

Data Transformation from 3D to Screen

The journey from 3D model data to screen pixels involves several coordinate transformations:

1. **Model Space → World Space**: Places the model in the 3D world
   - Applied through the model matrix (translation, rotation, scale)
   - Converts from object-local coordinates to world coordinates
2. **World Space → View Space**: Positions everything relative to the camera
   - Applied through the view matrix
   - Makes the camera the origin of the coordinate system
   - Forward direction becomes -Z, up becomes +Y, right becomes +X
3. **View Space → Clip Space**: Applies perspective and defines visible volume
   - Applied through the projection matrix
   - Scales visible area based on field of view and aspect ratio
   - Maps visible volume to a canonical cube (from -1 to 1 in all dimensions)
4. **Clip Space → NDC**: Normalizes coordinates after clipping
   - Performed by dividing X, Y, Z by W (perspective division)
   - Results in normalized device coordinates from -1 to 1
5. **NDC → Screen Space**: Maps to actual screen pixels
   - Applied through the viewport transformation
   - Scales coordinates to screen dimensions
   - Flips Y axis (screen coordinates increase downward)

Transformation Sequence

The complete transformation sequence combines all these steps:

```
// Combined matrix calculation
const modelViewProjection = multiplyMatrices(
    projectionMatrix,
    multiplyMatrices(viewMatrix, modelMatrix)
);

// Apply to vertex
let clipSpaceVertex = multiplyMatrixVector(modelViewProjection, modelVertex);

// Perspective division to get NDC
let ndcVertex = {
    x: clipSpaceVertex.x / clipSpaceVertex.w,
    y: clipSpaceVertex.y / clipSpaceVertex.w,
    z: clipSpaceVertex.z / clipSpaceVertex.w
};

// Viewport transform to get screen coordinates
let screenVertex = {
    x: Math.floor((ndcVertex.x + 1) * canvas.width / 2),
    y: Math.floor((1 - ndcVertex.y) * canvas.height / 2),
```

```
    z: ndcVertex.z  // Keep for depth testing
};
```

This sequence transforms our 3D model data into the exact pixel locations needed for rendering on screen. Understanding this transformation pipeline is fundamental to working with any 3D graphics system, whether a simple software renderer like ours or a sophisticated GPU-accelerated engine.