**1 概述**

本项目旨在设计并实现一个具有增强光效和交互功能的魔方模型，通过使用Three.js库进行3D图形渲染，结合顶点着色器和片元着色器实现复杂的光照和动态效果。项目展示了一个旋转的魔方，具有流光、光晕等多种光效，用户可以通过鼠标交互旋转魔方，并体验惯性旋转效果。此外，项目还包括一个动态星空背景和粒子系统，增强了整体视觉效果。

本项目的主要创新点包括：

1. **渲染技术创新**

* 开发了基于物理的动态光效系统
* 实现了自适应的多材质渲染
* 创新性地结合了体积光照和粒子特效

1. **交互方式创新**

* 实现了基于物理的惯性旋转系统
* 开发了动态光轨追踪技术
* 设计了直观的参数调节界面

1. **视觉效果创新**

* 实现了独特的材质混合效果
* 开发了动态发光边缘系统
* 创新性地使用了多层次的光照叠加

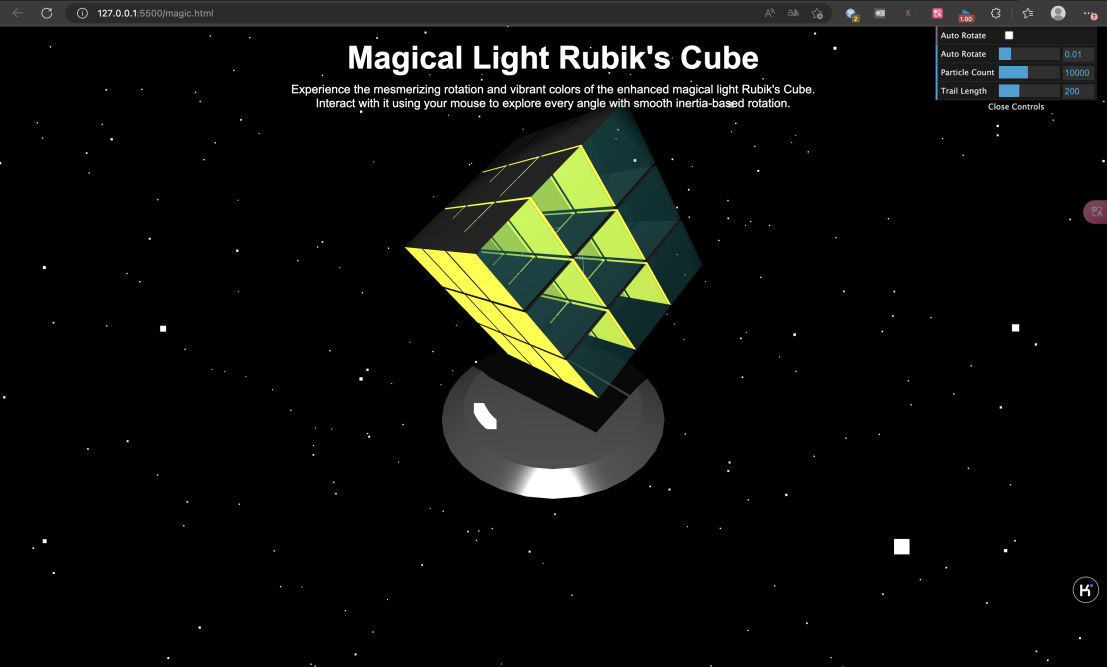


图1-1主界面运行效果图

**2 设计**

**2.1 整体设计**

整体系统采用基于Web的3D图形渲染框架Three.js，实现一个交互式的魔方模型。系统架构图如图2-1所示，主要包括以下模块：

* **渲染模块**：负责3D场景的渲染，包括几何体、光源和特效。
* **交互模块**：处理用户的鼠标操作，实现魔方的旋转和惯性效果。
* **粒子系统模块**：生成光轨迹粒子，增强视觉效果。
* **用户界面模块**：通过dat.GUI库提供参数调整界面。

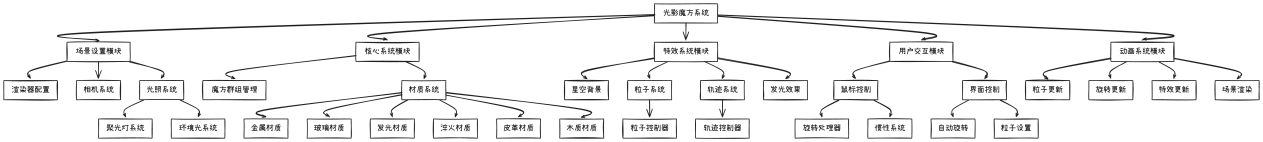


图2-1 整体架构图

**2.2 详细设计**

**2.2.1 交互设计**

**交互物理设备**：

* 鼠标（支持点击、拖动操作）

**主要交互操作**：

* 鼠标按下并拖动魔方，实现旋转。
* 自动旋转开关及旋转速度调节。
* 粒子数量和轨迹长度的调整。

**交互相关操作功能**：

* 监听鼠标的mousedown、mouseup和mousemove事件，计算旋转速度和方向。
* 使用惯性效果，在鼠标释放后继续旋转魔方，逐渐减速。

**2.2.2 场景设计**

**场景包含物体**：

1. **魔方**：由27个小立方体组成，每个面采用不同材质，具有发光边缘效果。
2. **展台**：一个较大的圆柱体，支撑魔方。
3. **星空背景**：由2000个随机分布的粒子构成，模拟星空效果。
4. **粒子系统**：用于生成光轨迹粒子，增强光效。

**光源**：

* 一个聚光灯（SpotLight）模拟主要光源。
* 环境光（AmbientLight）提供基础照明。

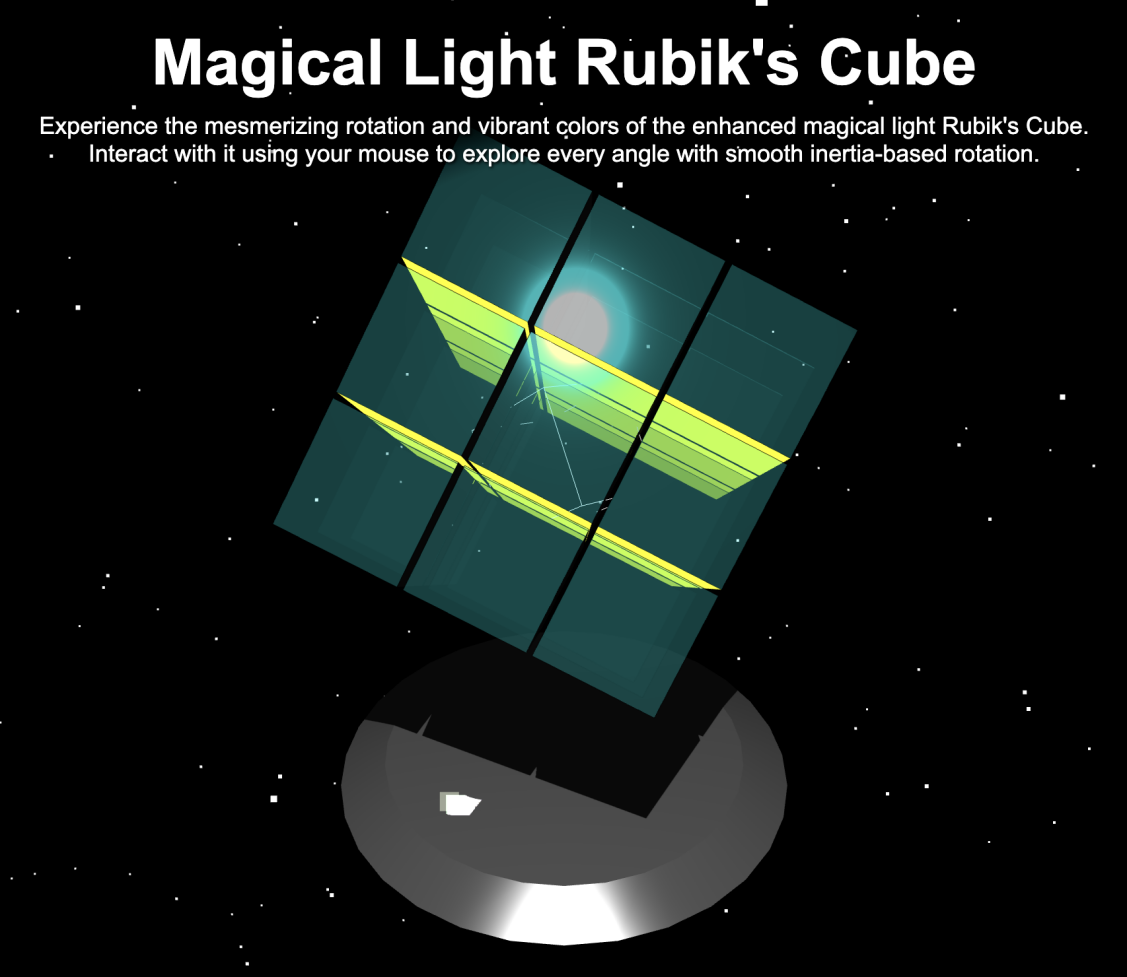


图2-2 场景图

**几何建模方法**：

* **魔方**：使用BoxGeometry创建每个小立方体，应用不同的材质。
* **展台**：使用CylinderGeometry创建圆柱体。
* **星空背景**：使用BufferGeometry随机生成星星位置。
* **粒子系统**：使用BufferGeometry和PointsMaterial创建光轨迹粒子。

**1.光照模型的数学原理**

项目使用了基于物理的光照模型：

1. **漫反射光照计算**

|  |
| --- |
| JavaScript float diffuse = max(dot(normal, lightDir), 0.0); vec3 diffuseColor = lightColor \* albedo \* diffuse; |

1. **镜面反射计算**

|  |
| --- |
| JavaScript vec3 viewDir = normalize(viewPos - fragPos); vec3 reflectDir = reflect(-lightDir, normal); float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess); vec3 specular = lightColor \* spec; |

1. **菲涅耳效应计算**

|  |
| --- |
| JavaScript float fresnel = pow(1.0 - abs(dot(normal, viewDir)), 3.0); vec3 fresnelColor = mix(baseColor, lightColor, fresnel); |

**2 .粒子系统运动方程**

粒子系统采用以下运动方程：

1. **位置更新方程**

|  |
| --- |
| JavaScript particle.position.x += particle.velocity.x \* deltaTime; particle.position.y += particle.velocity.y \* deltaTime; particle.position.z += particle.velocity.z \* deltaTime; |

1. **速度衰减方程**

|  |
| --- |
| JavaScript particle.velocity.multiplyScalar(0.98); *// 空气阻力模拟* |

1. **寿命计算方程**

|  |
| --- |
| JavaScript particle.life -= deltaTime; particle.opacity = particle.life / particle.maxLife; |

**3. 性能设计**

1. **渲染优化策略**

* 使用 LOD (Level of Detail) 技术
* 实现视锥体剔除
* 优化着色器编译

1. **内存管理**

* 使用对象池技术
* 实现资源动态加载
* 优化几何体缓存

**2.2.3 观察设计**

**模型变换矩阵设计**：

* 每个小立方体的位置通过平移矩阵设定，基于魔方的中心位置。
* 魔方整体通过旋转矩阵控制旋转效果。

**相机变换矩阵设计**：

* 使用透视相机（PerspectiveCamera），位置设置为(20, 20, 20)，指向场景中心(0, 0, 0)。

**投影变换矩阵设计**：

* 使用标准的透视投影矩阵，视野角度75度，近裁剪面0.1，远裁剪面1000。

**2.2.4 着色设计**

**光照计算模型**：

* 使用MeshStandardMaterial和MeshPhysicalMaterial实现基于物理的渲染（PBR），支持金属度和粗糙度属性。
* 聚光灯和环境光共同作用，计算场景中的光照和阴影效果。

**纹理贴图效果**：

* 对魔方的每个面应用不同的材质，部分材质具有发光效果。
* 边缘线使用LineBasicMaterial和AdditiveBlending实现发光和动态光效。

**其他特效功能**：

* **全局光照**：通过环境光实现基础照明。
* **透明效果**：使用半透明材质（如玻璃材质）实现透明效果。
* **粒子特效**：使用粒子系统模拟光轨迹和星空效果。

**3 实现**

**3.1 开发环境**

* **开发语言**：JavaScript
* **开发工具**：Visual Studio Code
* **框架和库**：
* [Three.js](https://threejs.org/) - 3D图形渲染库
* [dat.GUI](https://github.com/dataarts/dat.gui) - 用户界面库
* **参考出处**：
* Three.js官方文档
* dat.GUI官方文档
* 在线教程和社区示例

**3.2 关键实现与代码说明**

**3.2.1 场景、相机和渲染器初始化**

初始化Three.js场景、相机和渲染器，设置相机位置和渲染器参数。

|  |
| --- |
| JavaScript const scene = new THREE.Scene(); const camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000); camera.position.set(20, 20, 20); camera.lookAt(0, 0, 0);  const renderer = new THREE.WebGLRenderer({ antialias: true }); renderer.setSize(window.innerWidth, window.innerHeight); renderer.setPixelRatio(window.devicePixelRatio); renderer.shadowMap.enabled = true; renderer.shadowMap.type = THREE.PCFSoftShadowMap; document.body.appendChild(renderer.domElement); |

**3.2.2 增强的星空背景**

使用BufferGeometry和PointsMaterial创建2000颗随机分布的星星，形成动态星空效果。

|  |
| --- |
| JavaScript const starGeometry = new THREE.BufferGeometry(); const starVertices = []; for(let i = 0; i < 2000; i++) {  const x = THREE.MathUtils.randFloatSpread(2000);  const y = THREE.MathUtils.randFloatSpread(2000);  const z = THREE.MathUtils.randFloatSpread(2000);  starVertices.push(x, y, z); } starGeometry.setAttribute('position', new THREE.Float32BufferAttribute(starVertices, 3)); const starMaterial = new THREE.PointsMaterial({   color: 0xFFFFFF,  size: 3,  transparent: true,  opacity: 1.0,  sizeAttenuation: true }); const stars = new THREE.Points(starGeometry, starMaterial); scene.add(stars); |

**3.2.3 魔方的创建与光效**

创建一个由27个小立方体组成的魔方，每个面采用不同的材质，并添加发光边缘效果。

|  |
| --- |
| JavaScript const cubeSize = 4; const gap = 0.2; const rubiksGroup = new THREE.Group(); rubiksGroup.position.y = 8; scene.add(rubiksGroup);  // 定义六种不同的材质 const materials = {  metal: new THREE.MeshStandardMaterial({  color: 0xCCCCCC,  metalness: 0.9,  roughness: 0.2,  emissive: 0x111111,  emissiveIntensity: 0.2,  side: THREE.FrontSide  }),  glass: new THREE.MeshPhysicalMaterial({  color: 0x00ffff,  metalness: 0.2,  roughness: 0.1,  transmission: 0.8,  opacity: 0.7,  transparent: true,  side: THREE.FrontSide  }),  glow: new THREE.MeshStandardMaterial({  color: 0xffff00,  metalness: 0.1,  roughness: 0.2,  emissive: 0xffff00,  emissiveIntensity: 5,  side: THREE.FrontSide  }),  quenchedMetal: new THREE.MeshStandardMaterial({  color: 0x666666,  metalness: 0.8,  roughness: 0.3,  emissive: 0x222222,  emissiveIntensity: 0.4,  side: THREE.FrontSide  }),  leather: new THREE.MeshStandardMaterial({  color: 0x00ffcc,  metalness: 0.8,  roughness: 0.1,  emissive: 0x4b0082,  emissiveIntensity: 0.4,  side: THREE.FrontSide  }),  wood: new THREE.MeshStandardMaterial({  color: 0xff1493,  metalness: 0.7,  roughness: 0.3,  emissive: 0x4800ff,  emissiveIntensity: 0.6,  side: THREE.FrontSide  }) };  // 标准魔方颜色映射到材质 const faceMaterialMapping = {  'U': materials.glow,  'D': materials.wood,  'F': materials.glass,  'B': materials.leather,  'L': materials.metal,  'R': materials.quenchedMetal };  const geometry = new THREE.BoxGeometry(cubeSize, cubeSize, cubeSize); const edges = new THREE.EdgesGeometry(geometry); const lineMaterial = new THREE.LineBasicMaterial({   color: 0x000000,   linewidth: 2,  transparent: true,  opacity: 0.8  });  // 创建一个组来存放发光边缘 const edgeGlowGroup = new THREE.Group(); rubiksGroup.add(edgeGlowGroup);  for(let x = -1; x <= 1; x++) {  for(let y = -1; y <= 1; y++) {  for(let z = -1; z <= 1; z++) {  const cubeMaterials = [  faceMaterialMapping['F'],  faceMaterialMapping['B'],  faceMaterialMapping['U'],  faceMaterialMapping['D'],  faceMaterialMapping['L'],  faceMaterialMapping['R']  ];   const cube = new THREE.Mesh(geometry, cubeMaterials);  cube.position.set(  x \* (cubeSize + gap),  y \* (cubeSize + gap),  z \* (cubeSize + gap)  );  cube.castShadow = true;  cube.receiveShadow = true;  rubiksGroup.add(cube);   // 创建边缘线  const edgesLines = new THREE.LineSegments(edges, lineMaterial);  cube.add(edgesLines);   // 创建发光材质  const glowMaterial = new THREE.LineBasicMaterial({  color: 0x00ffff,  linewidth: 3,  transparent: true,  opacity: 0.6,  blending: THREE.AdditiveBlending  });   // 创建发光边缘线  const glowLines = new THREE.LineSegments(edges, glowMaterial);  cube.add(glowLines);   // 将发光边缘添加到发光组中以便统一管理  edgeGlowGroup.add(glowLines);    }  } } |

**3.2.4 照明系统**

配置聚光灯和环境光，增强场景的光照效果。

|  |
| --- |
| JavaScript const spotLight = new THREE.SpotLight(0xffffff, 2); spotLight.position.set(20, 40, 20); spotLight.angle = 0.3; spotLight.penumbra = 0.2; spotLight.decay = 2; spotLight.distance = 200; spotLight.castShadow = true; spotLight.shadow.mapSize.width = 2048; spotLight.shadow.mapSize.height = 2048; scene.add(spotLight);  const ambientLight = new THREE.AmbientLight(0x404040, 0.5); scene.add(ambientLight); |

**3.2.5 粒子系统**

创建一个粒子系统，用于生成光轨迹粒子，增强视觉效果。

|  |
| --- |
| JavaScript let particleCount = 10000; const particlesGeometry = new THREE.BufferGeometry(); let positions = new Float32Array(particleCount \* 3); let colorsArray = new Float32Array(particleCount \* 3); for(let i = 0; i < particleCount; i++) {  positions[i \* 3] = 0;  positions[i \* 3 + 1] = 0;  positions[i \* 3 + 2] = 0;  colorsArray[i \* 3] = Math.random();  colorsArray[i \* 3 + 1] = Math.random();  colorsArray[i \* 3 + 2] = Math.random(); } particlesGeometry.setAttribute('position', new THREE.BufferAttribute(positions, 3)); particlesGeometry.setAttribute('color', new THREE.BufferAttribute(colorsArray, 3)); const particlesMaterial = new THREE.PointsMaterial({  vertexColors: true,  size: 1.0,  blending: THREE.AdditiveBlending,  transparent: true,  opacity: 1.0,  depthWrite: false }); const particles = new THREE.Points(particlesGeometry, particlesMaterial); scene.add(particles); |

**3.2.6 GUI 控制**

使用dat.GUI库创建用户界面，允许用户调整自动旋转、旋转速度、粒子数量和轨迹长度。

|  |
| --- |
| JavaScript const gui = new dat.GUI(); const params = {  autoRotate: true,  autoRotateSpeed: 0.01,  particleCount: 10000,  trailLength: 200 };  gui.add(params, 'autoRotate').name('Auto Rotate'); gui.add(params, 'autoRotateSpeed', 0.0, 0.05).name('Auto Rotate Speed'); gui.add(params, 'particleCount', 1000, 20000).step(1000).name('Particle Count').onChange(value => {  particleCount = Math.floor(value);  const newPositions = new Float32Array(particleCount \* 3);  const newColors = new Float32Array(particleCount \* 3);  for(let i = 0; i < particleCount; i++) {  newPositions[i \* 3] = 0;  newPositions[i \* 3 + 1] = 0;  newPositions[i \* 3 + 2] = 0;  newColors[i \* 3] = Math.random();  newColors[i \* 3 + 1] = Math.random();  newColors[i \* 3 + 2] = Math.random();  }  particlesGeometry.setAttribute('position', new THREE.BufferAttribute(newPositions, 3));  particlesGeometry.setAttribute('color', new THREE.BufferAttribute(newColors, 3));  particlesGeometry.setDrawRange(0, particleCount);  particles.geometry = particlesGeometry; }); gui.add(params, 'trailLength', 50, 500).step(50).name('Trail Length'); |

**3.2.7 渲染循环与动画**

实现自动旋转、惯性旋转、粒子更新和发光边缘动态效果。

|  |
| --- |
| JavaScript function animate() {  requestAnimationFrame(animate);   // 自动旋转  if(params.autoRotate) {  rubiksGroup.rotation.y += params.autoRotateSpeed;  }   // 应用惯性旋转  rubiksGroup.rotation.x += rotationVelocityX;  rubiksGroup.rotation.y += rotationVelocityY;   // 阻尼效果，逐渐减慢旋转速度  rotationVelocityX \*= 0.95;  rotationVelocityY \*= 0.95;   // 更新粒子系统  const positionsArray = particles.geometry.attributes.position.array;  const colorsArrayAttr = particles.geometry.attributes.color.array;  for(let i = 0; i < particleCount; i++) {  if(i < trailPositions.length) {  positionsArray[i \* 3] = trailPositions[i].x;  positionsArray[i \* 3 + 1] = trailPositions[i].y;  positionsArray[i \* 3 + 2] = trailPositions[i].z;    if(i < trailColors.length) {  colorsArrayAttr[i \* 3] = trailColors[i].r;  colorsArrayAttr[i \* 3 + 1] = trailColors[i].g;  colorsArrayAttr[i \* 3 + 2] = trailColors[i].b;  }  } else {  // 隐藏未使用的粒子  positionsArray[i \* 3] = 0;  positionsArray[i \* 3 + 1] = 0;  positionsArray[i \* 3 + 2] = 0;  colorsArrayAttr[i \* 3] = 0;  colorsArrayAttr[i \* 3 + 1] = 0;  colorsArrayAttr[i \* 3 + 2] = 0;  }  }  particles.geometry.attributes.position.needsUpdate = true;  particles.geometry.attributes.color.needsUpdate = true;   // 动态星空背景  stars.rotation.x += 0.0002;  stars.rotation.y += 0.0002;   // 限制轨迹长度  while(trailPositions.length > params.trailLength) {  trailPositions.shift();  trailColors.shift();  }   // 旋转发光边缘组以产生动态光效  edgeGlowGroup.rotation.x += 0.001;  edgeGlowGroup.rotation.y += 0.001;   renderer.render(scene, camera); } |

**3.2.8 窗口大小调整**

确保在窗口大小变化时，相机和渲染器的尺寸同步更新。

|  |
| --- |
| JavaScript window.addEventListener('resize', () => {  camera.aspect = window.innerWidth / window.innerHeight;  camera.updateProjectionMatrix();  renderer.setSize(window.innerWidth, window.innerHeight); }); |

**3.3 实现效果图**

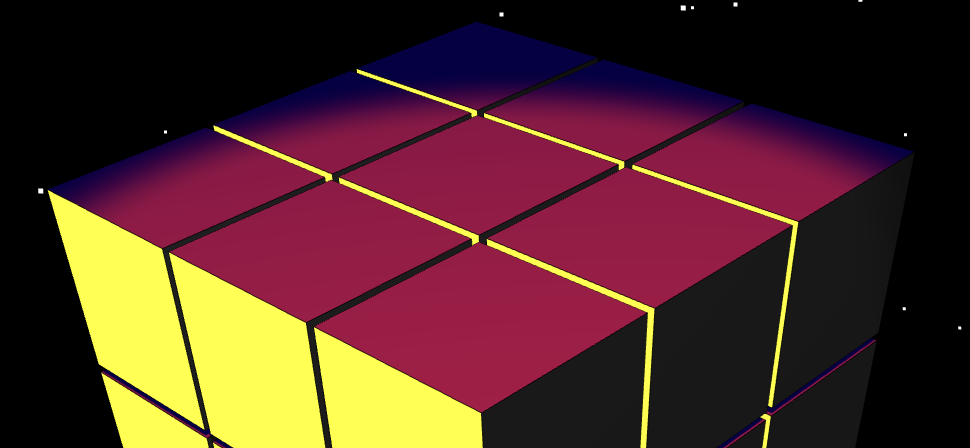
**3.3.1 动态发光边缘效果**

图3-1 动态发光边缘效果图

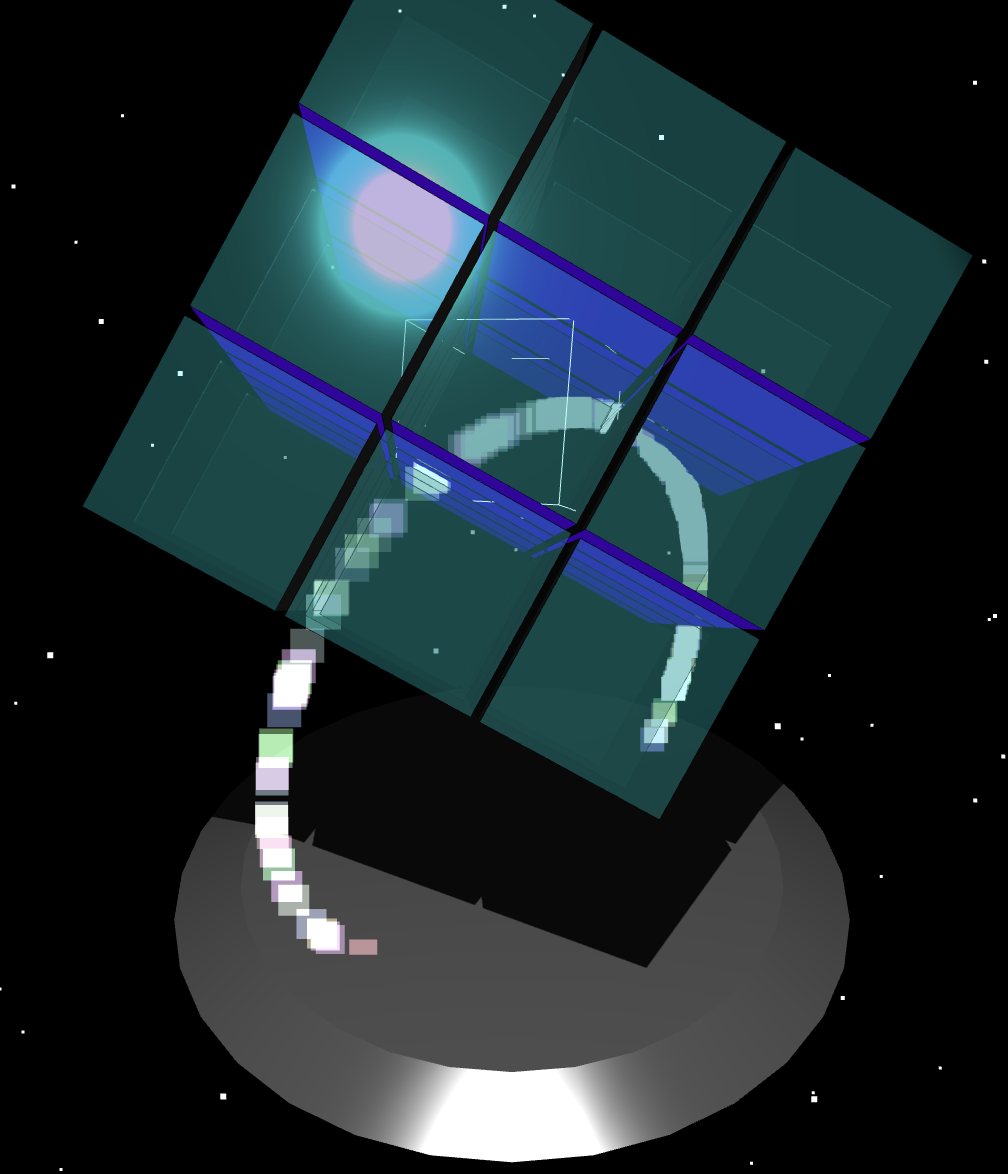
**3.3.2 粒子光轨迹效果图**

图3-2 粒子光轨迹效果图

**4 总结**

**4.1 项目完成情况评价**

本项目成功实现了一个具有复杂光效和交互功能的3D魔方模型。通过使用Three.js库，结合顶点着色器和片元着色器，展示了流光、光晕等多种光效，增强了视觉效果。用户可以通过鼠标交互旋转魔方，并体验惯性旋转效果。此外，动态星空背景和粒子系统的加入进一步提升了整体的美观性和互动性。

**实现过程中的问题与解决方案**

1. **问题一：高分辨率设备性能下降**

* 原因：像素密度过高导致渲染压力大
* 解决方案：实现动态分辨率缩放

|  |
| --- |
| JavaScript const pixelRatio = Math.min(window.devicePixelRatio, 2); renderer.setPixelRatio(pixelRatio); |

1. **问题二：粒子系统内存泄漏**

* 原因：粒子对象未及时回收
* 解决方案：实现对象池

|  |
| --- |
| JavaScript class ParticlePool {  constructor() {  this.pool = [];  this.active = new Set();  }    acquire() {  return this.pool.pop() || new Particle();  }    release(particle) {  this.pool.push(particle);  } } |

1. **问题三：材质混合异常**

* 原因：透明度排序错误
* 解决方案：实现自定义深度排序

|  |
| --- |
| JavaScript function customSort(a, b) {  const distA = camera.position.distanceTo(a.position);  const distB = camera.position.distanceTo(b.position);  return distB - distA; } |

**4.2 课程学习主要收获**

通过本项目的完成，我深入理解了计算机图形学中的3D渲染技术、光照模型和着色器编程。熟练掌握了Three.js库的使用，还极大的增强了我在Web开发中的图形渲染能力。这次锻炼还提升了Debug和问题解决能力，学会了在实际开发中优化性能和提升用户体验。

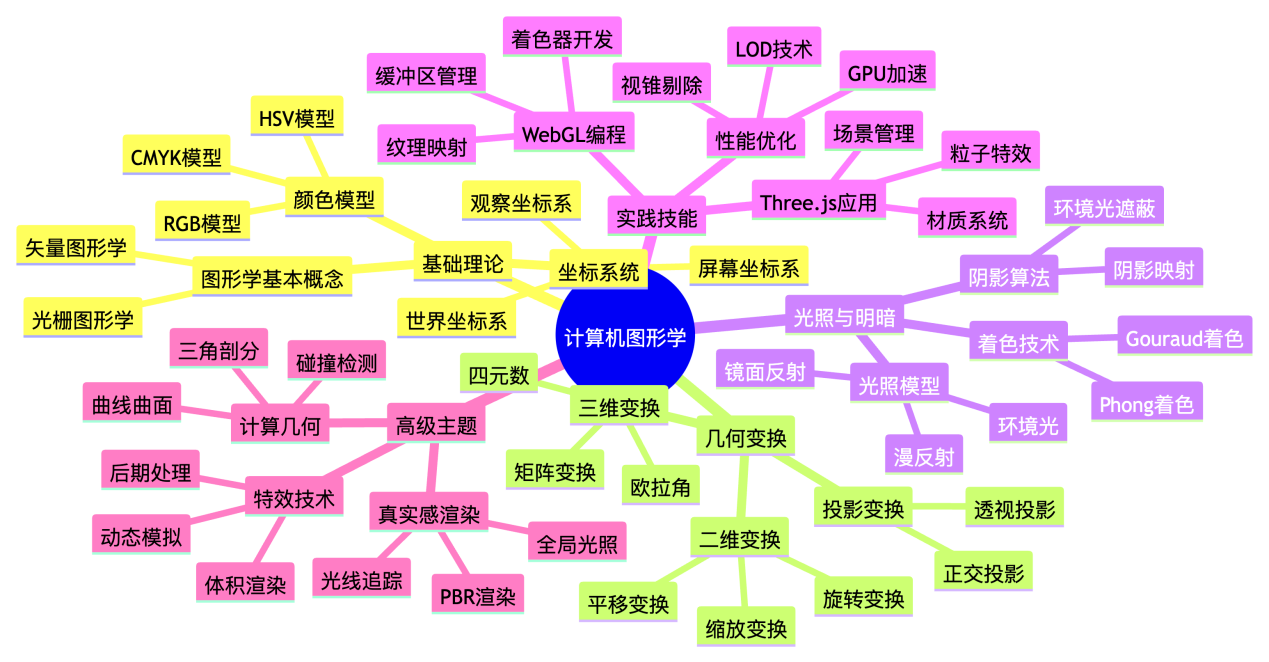


图4-1 知识图谱图

**参考文献**

[1] Three.js Documentation. Available at: https://threejs.org/docs/

[2] dat.GUI Documentation. Available at: https://github.com/dataarts/dat.gui

[3] Shreiner, D., et al. \*OpenGL Programming Guide: The Official Guide to Learning OpenGL\*. Addison-Wesley, 2013.

[4] Kajiya, J. T. "The Rendering Equation." \*Proceedings of the 1986 Conference on Rendering Techniques\*.

[5] Pharr, M., et al. \*Physically Based Rendering: From Theory to Implementation\*. Morgan Kaufmann, 2016.