计算机图形学

04022405葛浩南

实验一

实验内容

编程的基本模式：

实现并讨论用WebGL编写基本的三维图形显示程序的程序结构和书写规范。 发挥想象力，绘制一只动物（2D）

要求：

1. 窗口背景白色，动物至少有两种颜色；
2. 可利用鼠标选取动物全部或某一部分，并拖动
3. 利用键盘实现前进、后退、跳跃等功能
4. 实现菜单功能

设计思路及实验步骤

本次实验，我负责的是功能2、3、4。

在代码中，这三个功能是通过以下方式实现的：

1. **可利用鼠标选取动物全部或某一部分，并拖动**：
    - 在 pig.html 文件中，通过添加鼠标事件来实现拖动功能。
    - `mousedown` 事件用于检测鼠标按下并判断是否在猪的范围内，如果在范围内则设置 `pig.isDragging` 为 `true`。
    - `mousemove` 事件用于在拖动过程中更新猪的位置。
    - `mouseup` 事件用于在鼠标松开时停止拖动。

```
// filepath: /C:/Users/Johnny/Downloads/pig.html
canvas.addEventListener('mousedown', (e) => {
    const mouseX = e.offsetX;
    const mouseY = e.offsetY;

    if (mouseX >= pig.x && mouseX <= pig.x + pig.width &&
        mouseY >= pig.y && mouseY <= pig.y + pig.height) {
        pig.isDragging = true;
    }
});

canvas.addEventListener('mousemove', (e) => {
    if (pig.isDragging) {
        pig.x = e.offsetX - pig.width / 2;
```

```
        pig.y = e.offsetY - pig.height / 2;
        drawPig();
    }
});

canvas.addEventListener('mouseup', () => {
    pig.isDragging = false;
});
```

2. **利用键盘实现前进、后退、跳跃等功能**:

   - 在 pig.html 文件中，通过添加键盘事件来实现前进、后退、跳跃等功能。
   - `keydown` 事件用于检测按键并调用相应的移动函数。
   - `keyup` 事件用于在拖动过程中更新猪的位置。

```
 // filepath: /C:/Users/Johnny/Downloads/pig.html
window.addEventListener('keydown', (e) => {
switch (e.key) {
    case 'd':
    case 'D':
        moveRight();
        break;
    case 'a':
    case 'A':
        moveLeft();
        break;
    case ' ':
        jump();
        break;
    case 's':
    case 'S':
        pig.isCrouching = true;
        pig.acceleration = 0;
        break;
}
});
window.addEventListener('keyup', (e) => {
    if (e.key === 's' || e.key === 'S') {
    pig.isCrouching = false;
    pig.acceleration = 0.02;
    }
});
```

在代码中，`pig` 的前进、后退、跳跃函数以及加速度是通过以下方式实现的：

1. ***前进函数***:

   - `moveRight` 函数用于使猪向右移动。

- 增加 `pig.velocity` 和 `pig.x` 来更新猪的位置。
- 设置 `pig.isMovingRight` 为 `true`，并重置 `pig.moveDuration`。

```
function moveRight() {
    pig.isMovingRight = true;
    pig.velocity = 1;
    pig.moveDuration = 0;
 }
```

2. ***后退函数***

- `moveLeft` 函数用于使猪向左移动。
  - 增加 `pig.velocity` 和 `pig.x` 来更新猪的位置。
  - 设置 `pig.isMovingLeft` 为 `true`，并重置 `pig.moveDuration`。

```
 function moveLeft() {
     pig.isMovingLeft = true;
     pig.velocity = -1;
     pig.moveDuration = 0;
 }
```

3. ***跳跃函数***

- `jump`函数用于使猪跳跃。
- 设置 `pig.isJumping` 为 `true`，并初始化 `pig.jumpVelocity`。

```
function jump() {
    if (!pig.isJumping) {
        pig.isJumping = true;
        pig.jumpVelocity = -5;
    }
}
```

4. ***加速度***

- 在动画循环中，通过增加或减少`pig.velocity`来实现加速度的效果。
- 当猪在移动时，`pig.velocity`会根据`pig.accelercation`进行更新，实现具有加速的的运动动画。
- 在小猪跳跃时，通过`pig.gravity`来实现小猪跳跃时重力加速度的模拟

```
function animate() {
    // 在循环中更新小猪腿部的动作
    if (pig.isMovingRight || pig.isMovingLeft || pig.isJumping) {
        pig.legOffset += pig.legDirection;
```

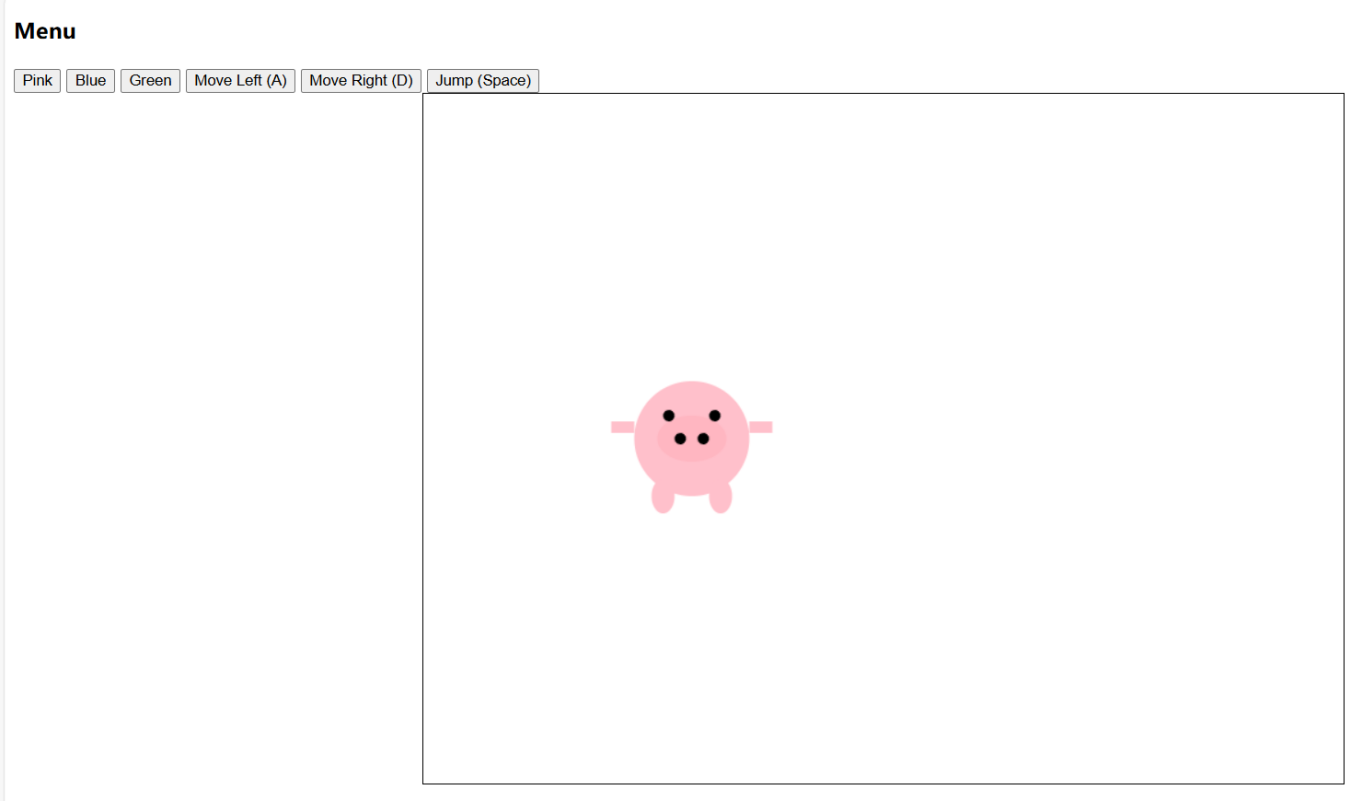
```
            if (pig.legOffset > 5 || pig.legOffset < -5) {
                pig.legDirection *= -1;
            }
        }

        // 当向右运动时更新
        if (pig.isMovingRight) {
            pig.velocity += pig.acceleration;
            pig.x += pig.velocity;
            pig.moveDuration += 1;
            pig.eyeOffest = 6;
            // 停止移动
            if (pig.moveDuration > pig.maxMoveDuration) {
                pig.isMovingRight = false;
            }
        }

        // 当向左运动时更新
        if (pig.isMovingLeft) {
            pig.velocity -= pig.acceleration;
            pig.x += pig.velocity;
            pig.moveDuration += 1;
            pig.eyeOffest = -6;
            // 停止移动
            if (pig.moveDuration > pig.maxMoveDuration) {
                pig.isMovingLeft = false;
            }
        }

        // 跳跃时更新动画
        if (pig.isJumping) {
            pig.y += pig.jumpVelocity;
            pig.jumpVelocity += pig.gravity;
            if (pig.y >= groundLevel) {
                pig.y = groundLevel;
                pig.isJumping = false;
            }
        }

        drawPig();
        requestAnimationFrame(animate);
    }
```

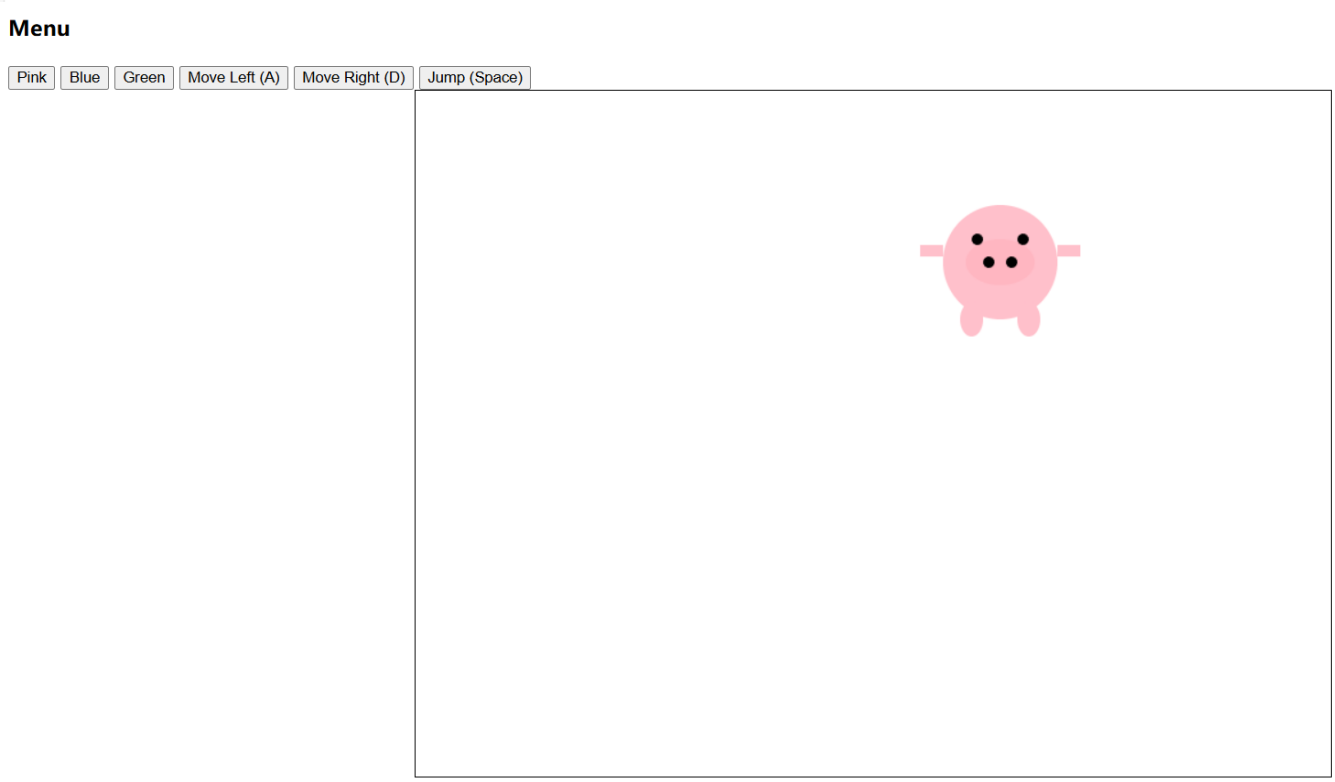
3. **实现菜单功能**

- 在`pig.html` 文件中，通过创建一个菜单 div 元素并添加按钮来实现菜单功能。
- 每个按钮都有一个 `onclick` 事件处理程序，用于执行相应的操作，例如更改颜色或移动猪。

```
const menu = document.createElement('div');
menu.innerHTML = `
    <h3>Menu</h3>
    <button onclick="changeColor('pink')">Pink</button>
    <button onclick="changeColor('blue')">Blue</button>
    <button onclick="changeColor('green')">Green</button>
    <button onclick="moveLeft()">Move Left (A)</button>
    <button onclick="moveRight()">Move Right (D)</button>
    <button onclick="jump()">Jump (Space)</button>
`;
document.body.insertBefore(menu, canvas);

function changeColor(color) {
    pig.color = color;
    drawPig();
}
```

实现效果图





个人总结

通过本次实验,初步了解了webgl中的图形绘制,顶点着色的技术.了解了什么是顶点着色器和片元着色器,以及他们如何用js代码进行数据的传输。了解了javascript的一些编程技巧，学习了它的基础知识。学习了html如何添加事件响应函数，以及如何在chrome浏览器中进行调试程序。我熟悉了 WebGL 的基本编程模式，掌握了使用事件监听和动画循环实现交互式动画的基本方法。

实验二

实验内容

几何对象与变换应用实例：

在实验中体现课堂讲授的几何对象与变换的内容，理解基本变换和复合变换。 将动物拓展为三维物体。自由设计交互方式，利用变换实现动物在三维空间中的运动：

1. 实现基于物体坐标系的运动，即动物头部方向为前进，相反方向为后退，绕动物中轴线的转向、俯仰和翻滚；
2. 实现以屏幕坐标为参考的运动，即上、下、左、右移动，和绕屏幕坐标轴的旋转。
3. 利用虚拟跟踪球技术，实现鼠标交互操作旋转查看对象。

设计思路

这个实验我负责实现除了屏幕坐标系的各种动画效果。

1. 动物头部方向前进、相反方向后退这个功能，实现思路是物体沿自己的对象坐标系的y轴进行移动，而往左右则是沿对象坐标系的x轴进行移动。
   若以 ctm_{old} 为上一次的变换矩阵， ctm_{new} 为新的变换矩阵，t为平移矩阵，则

$$ctm_{new} = ctm_{old} * t$$

2. 动物以屏幕坐标（世界坐标）为参考的上下左右运动则是

$$ctm_{new} = t * ctm_{old}$$

3. 绕动物中轴线的转向、俯仰和翻滚，则是物体绕对象坐标系进行旋转,设 r 为旋转矩阵,则

$$ctm_{new} = ctm_{old} * r$$

4. 绕屏幕坐标系的旋转则是:

$$ctm_{new} = r * ctm_{old}$$

5. 虚拟跟踪球技术，则是记录鼠标的x方向和y方向上的偏移量，以此产生绕x轴的旋转和y轴的旋转矩阵，左乘以cmt，就可以实现虚拟跟踪球技术

实现步骤

1. **基于物体坐标系的运动**：

   - `direct` 向量： `vec4(0.0, 0.0, 1.0, 1.0)` 表示海绵宝宝模型在其局部坐标系中的初始前进方向（即，未旋转时的方向）。最后一个分量 1.0 对于向量与包含平移的矩阵进行乘法运算至关重要。

     - 更新 `direct`： 在 `render` 函数内部，计算 `modelViewMatrix`（包含平移、旋转和缩放）之后，代码会更新 direct 向量：

```
direct = vec4(0.0, 0.0, 1.0, 1.0); // 重置初始方向
direct = multMat4Vec4(m, direct); // m 是旋转矩阵
```

这里，`m` 代表旋转矩阵（Rx * Ry * Rz）。将 `m` 与初始 direct 向量相乘，会根据模型当前的旋转来转换前进方向。这是实现基于物体运动的关键。

- **移动：** “前进”和“后退”按钮使用转换后的 direct 向量来更新平移：

```
document.getElementById("cubeForward").onclick = function() {
    CubeTx += 0.1 * direct[0];
    CubeTy += 0.1 * direct[1];
    CubeTz += 0.1 * direct[2];
};
```

通过将转换后的 `direct` 向量的分量加到或减去 `CubeTx`、`CubeTy` 和 `CubeTz`，海绵宝宝会相对于其当前方向前进或后退。

- 旋转：`MV.js`里的 `CubeRotateAngleX`、`CubeRotateAngleY` 和 `CubeRotateAngleZ` 控制绕物体局部 X、Y 和 Z 轴的旋转。此旋转在平移之前应用，确保移动相对于旋转后的方向。

- **前进/后退：**

```
document.getElementById("cubeForward").onclick = function() {
CubeTx += 0.1 * direct[0];
CubeTy += 0.1 * direct[1];
CubeTz += 0.1 * direct[2];
};
document.getElementById("cubeBack").onclick = function() {
    CubeTx -= 0.1 * direct[0];
    CubeTy -= 0.1 * direct[1];
    CubeTz -= 0.1 * direct[2];
};
```

`direct` 向量表示海绵宝宝的正面方向。每次前进或后退时，将 `direct` 向量乘以一个步长（0.1），然后加到海绵宝宝的平移量 (`CubeTx`, `CubeTy`, `CubeTz`) 上。关键在于 `direct` 向量是**动态更新**的，它始终指向海绵宝宝的当前正面方向。

`direct` 向量的计算在 `render` 函数中的 `updateModelMatrix` 函数中：

```
direct = vec4(0.0, 0.0, 1.0, 1.0); // 初始方向
direct = transform(modelViewMatrix, direct); // 使用模型视图矩阵变换
```

首先初始化 `direct` 为 `(0, 0, 1, 1)`，表示物体坐标系的 Z 轴正方向。然后使用当前的 `modelViewMatrix` 对其进行变换，得到世界坐标系中的方向向量。这样，无论海绵宝宝如何旋转，`direct` 始终指向其正面。

- **绕自身轴旋转（转向、俯仰、翻滚）**

```
var Rx = rotateX(CubeRotateAngleX); // 绕 X 轴旋转（俯仰）
var Ry = rotateY(CubeRotateAngleY); // 绕 Y 轴旋转（转向）
var Rz = rotateZ(CubeRotateAngleZ); // 绕 Z 轴旋转（翻滚）
modelViewMatrix = mult(mult(mult(mult(mult(init, T), Rx), Ry), Rz), S);
```

通过 `rotateX、rotateY` 和 `rotateZ` 函数创建旋转矩阵，并按顺序乘以 `modelViewMatrix`。旋转的顺序很重要，会影响最终效果。在这个代码中，顺序是先平移，然后依次绕 X、Y、Z 轴旋转，最后缩放。键盘事件监听器控制 `CubeRotateAngleX、CubeRotateAngleY` 和 `CubeRotateAngleZ` 的值，从而控制旋转角度。

2. **以屏幕坐标为参考的运动** 这种运动与观察者的视角相关，即屏幕上的上、下、左、右。代码中通过以下方式实现：

- **上/下/左/右移动：**

```
case "ArrowUp":
    CubeTy += 0.1; // 向上移动，增加 Y 坐标
    break;
case "ArrowDown":
    CubeTy -= 0.1; // 向下移动，减少 Y 坐标
    break;
case "ArrowLeft":
    CubeTx -= 0.1; // 向左移动，减少 X 坐标
    break;
case "ArrowRight":
    CubeTx += 0.1; // 向右移动，增加 X 坐标
    break;
```

直接修改 `CubeTx` 和 `CubeTy` 的值，分别对应屏幕坐标系的 X 和 Y 轴。这种移动方式不受海绵宝宝自身旋转的影响，始终是相对于屏幕的。

3. **虚拟跟踪球技术** 代码使用虚拟跟踪球技术实现鼠标交互旋转：

- `trackballView(x, y)`： 将屏幕坐标 (x, y) 映射到虚拟球面上。如果 (x, y) 在球内，则计算球面上对应的三维坐标；否则，将其投影到球的边界上。
- `mouseMotion(x, y)`： 当鼠标移动时：
    1. 计算当前鼠标位置在虚拟球面上的坐标 `curPos`。
    2. 计算当前位置与上次位置 `lastPos` 的差值 dx、dy、dz。
    3. 如果差值不为零，则计算旋转角度 `angle` 和旋转轴 `axis`。旋转轴是 `lastPos` 和 `curPos` 的叉积。
    4. 更新 `lastPos` 为 `curPos`。
    5. 调用 `render` 函数重绘场景。
- `startMotion(x, y)`： 当鼠标按下时，记录起始位置，并设置 `trackingMouse` 为 `true`。
- `stopMotion(x, y)`： 当鼠标松开时，设置 `trackingMouse` 为 `false`。
- `render()`函数中的应用

```
if (trackballMove) {
  axis = normalize(axis);
  rotationMatrix = mult(rotationMatrix, rotate(angle, axis));
  gl.uniformMatrix4fv(rotationMatrixLoc, false, flatten(rotationMatrix));
  }
```

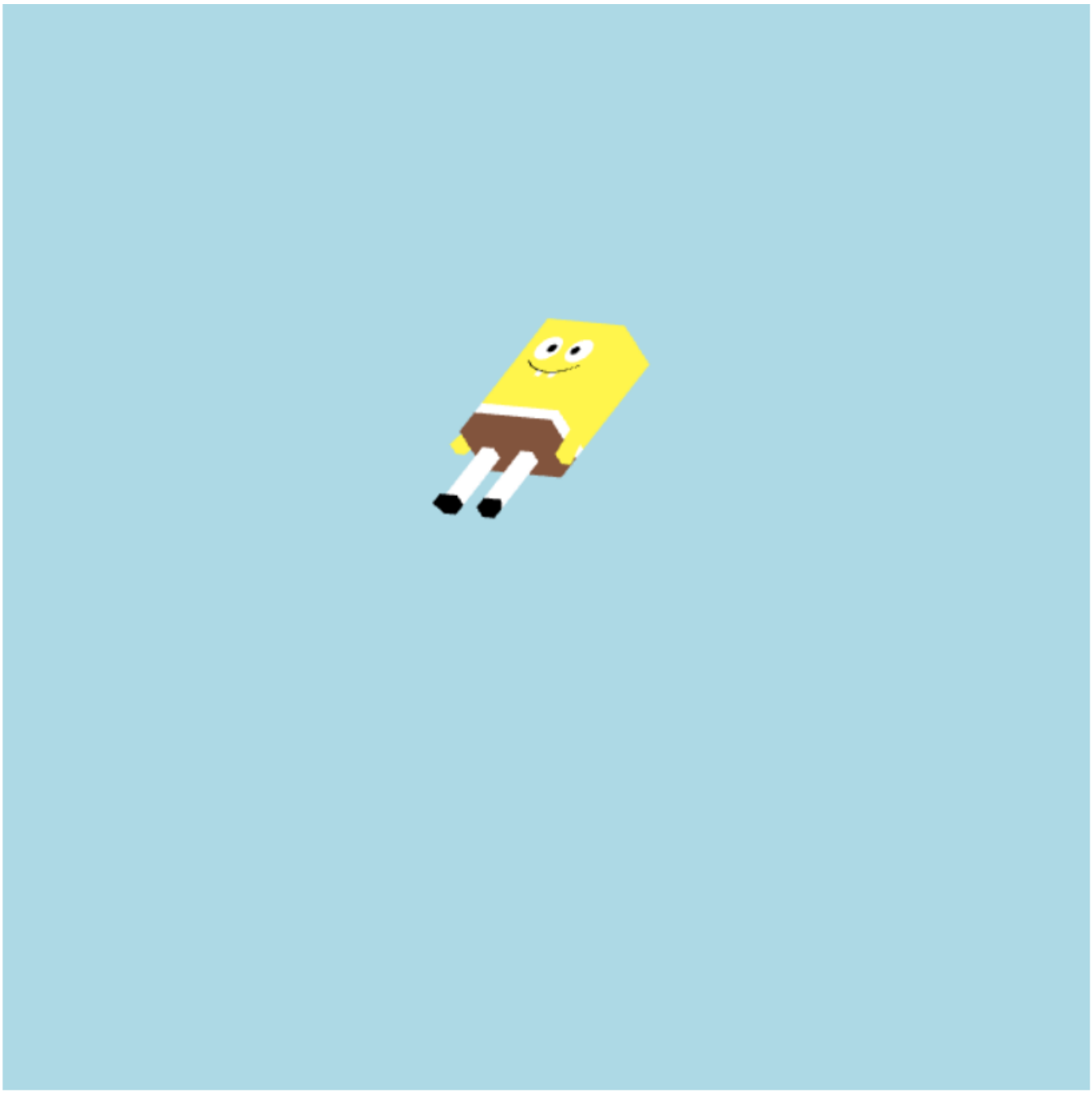
在 `render` 函数中，如果 `trackballMove` 为 `true`，则根据计算出的 `angle` 和 `axis` 更新 `rotationMatrix`，并将其传递给 `shader`。`rotationMatrix` 会影响最终的模型视图矩阵，从而实现旋转。

- 核心代码：

```
function trackballView( x,  y ) {
    var d, a;
    var v = [];

    v[0] = x;
    v[1] = y;

    d = v[0]*v[0] + v[1]*v[1];
    if (d < 1.0)
    v[2] = Math.sqrt(1.0 - d);
    else {
    v[2] = 0.0;
    a = 1.0 /  Math.sqrt(d);
    v[0] *= a;
    v[1] *= a;
    }
    return v;
}

function mouseMotion( x,  y)
{
    var dx, dy, dz;

    var curPos = trackballView(x, y);
    if(trackingMouse) {
    dx = curPos[0] - lastPos[0];
    dy = curPos[1] - lastPos[1];
    dz = curPos[2] - lastPos[2];

    if (dx || dy || dz) {
        angle = -0.1 * Math.sqrt(dx*dx + dy*dy + dz*dz);


        axis[0] = lastPos[1]*curPos[2] - lastPos[2]*curPos[1];
        axis[1] = lastPos[2]*curPos[0] - lastPos[0]*curPos[2];
        axis[2] = lastPos[0]*curPos[1] - lastPos[1]*curPos[0];

        lastPos[0] = curPos[0];
        lastPos[1] = curPos[1];
```

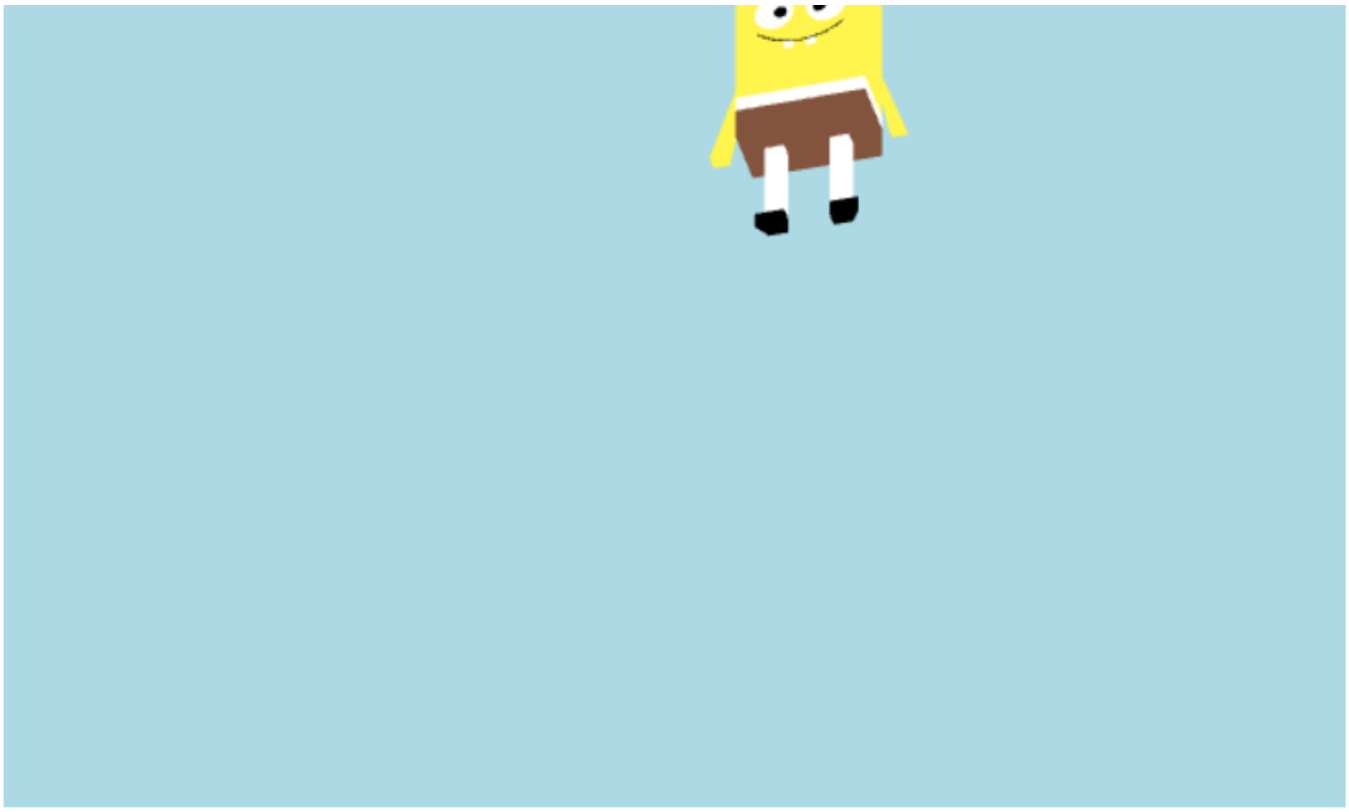
```
            lastPos[2] = curPos[2];
        }
        }
        render();
    }

    function startMotion( x,  y)
    {
        trackingMouse = true;
        startX = x;
        startY = y;
        curx = x;
        cury = y;

        lastPos = trackballView(x, y);
        trackballMove=true;
    }

    function stopMotion( x,  y)
    {
        trackingMouse = false;
        if (startX != x || startY != y) {
        }
        else {
            angle = 0.0;
            trackballMove = false;
        }
    }
```

最后效果截图:



调整视图 重置

海绵宝宝： 前进 后退 逆时针旋转 顺时针旋转 缩小 放大



个人总结

通过本次实验,对仿射空间、Euclid空间的概念更加熟悉，并切了解到齐次坐标系的优越性。这次的实验我主要针对一个对象对其进行变换。也考虑如何交互式地确定变换并把变换平滑地应用到几何对象上。实验过程中，了解了世界坐标系/对象坐标系之间的关系,以及在这两个坐标系下的旋转平移变换的实现，加深了我对变换矩阵和坐标系的理解。通过动态更新 direct 向量，实现了基于物体坐标系的运动，这对于实现更复杂的 3D 动画非常重要。虚拟跟踪球技术的实现让我学习了如何将 2D 鼠标输入转换为 3D 旋转，这是一种常用的 3D 交互技术。

实验三

实验内容

场景漫游与光照：

掌握视点设置和光照的基本原理、实现方式，并讨论实验效果与参数设置的关系。 场景中加入光照

要求实现：

1. 交互式的视点变化，实现场景漫游
2. 光源位置可交互改变，参数可调
3. 场景中至少包含两种以上的材质

设计思路

这个实验中,我实现的是场景漫游,由于上次实验我实现了物体的移动和旋转,不过这次移动的对象不再是物体,而是相机.

本次Project所绘制的物体都是3维立体空间中的，因此，我们所使用的坐标系也是空间中的三维坐标系，有三个相互垂直的坐标轴，分别是X轴，Y轴和Z轴。相应的，模型上的每一个顶点都有三个坐标分量(x, y, z) 。

无论是用人眼看物体，还是摄影，写实油画等，其展现的画面中的物体都是近大远小的。在计算机三维立体绘制中，这样的成像方式我们称之为透视投影，它需要一种特定的相机模型，如下图所示。我们需要四个参数来描述一个透视投影相机远/近成像距离，画面宽高比，垂直张角。

对于3D场景的展现，我们都是在场景中的某一位置，对着某个方向，使用透视投影的方式绘制一幅2D的画面。因此，我们还需要对摄像机的位置，朝向和姿态等信息进行描述。

实现步骤

1. 首先html中,有个投影矩阵`uPMatrix`和模-视变换矩阵`uMVMatrix`,在js里算好这些矩阵,并传送到顶点着色器进行渲染

```
vec4 pos = (uMVMatrix * vec4(aVertexPosition, 1.0));
            gl_Position = uPMatrix * uMVMatrix * vec4
```

2. **相机控制 (`Camera` 类)**

- `Camera.prototype.update()`： 这是核心函数，根据相机的位置 (`position`)、方位角 (`azimuth`) 和仰角 (`elevation`) 构建相机的变换矩阵 (`matrix`)。这个矩阵用于将世界坐标系转换到相机坐标系。

```
Camera.prototype.update = function () {
    this.matrix = mat4();
    this.matrix = mult(this.matrix, translate(this.position)); // 平移到指定位置
    this.matrix = mult(this.matrix, rotateY(this.azimuth)); // 绕 Y 轴旋转（方位
角）
    this.matrix = mult(this.matrix, rotateX(this.elevation)); // 绕 X 轴旋转（仰
角）
    if (this.onChange) {
        this.onChange(); // 调用重绘函数
    }
};
```

- `Camera.prototype.getViewMatrix()`： 获取视图矩阵，它是相机变换矩阵的逆矩阵。视图矩阵用于在着色器中进行坐标变换。

```
Camera.prototype.getViewMatrix = function () {
    return inverse4(this.matrix);
};
```

- `Camera.prototype.changeElevation(el)` 和 `Camera.prototype.changeAzimuth(az)`： 改变相机的仰角和方位角，并调用 `update()` 函数更新相机矩阵。

```
Camera.prototype.changeElevation = function (el) {
    this.elevation += el;
    this.update();
};

Camera.prototype.changeAzimuth = function (az) {
    this.azimuth += az;
    this.update();
};
```

- `Camera.prototype.setLocation(x, y, z)`： 直接设置相机的位置。

```
Camera.prototype.setLocation = function (x, y, z) {
    this.position = vec3(x, y, z);
    this.update();
};
```

3. **鼠标交互 (`CameraInteractor` 类)**

- `CameraInteractor.prototype.rotate(dx, dy)`： 根据鼠标拖动距离计算方位角和仰角的变化量，并调用 `camera.changeAzimuth()` 和 `camera.changeElevation()` 来更新相机。

```
CameraInteractor.prototype.rotate = (dx, dy) => {
    var dElevation = -20.0 / this.canvas.height;
    var dAzimuth = -20.0 / this.canvas.width;

    var nAzimuth = dx * dAzimuth * this.MOTION_FACTOR;
    var nElevation = dy * dElevation * this.MOTION_FACTOR;

    this.camera.changeAzimuth(nAzimuth);
    this.camera.changeElevation(nElevation);
};
```

- 鼠标事件处理函数 (`onMouseDown`、`onMouseUp`、`onMouseMove`)： 检测鼠标事件，并在拖动时调用 `rotate()` 函数。

4. **场景漫游 (`App` 类中的 `animate()` 函数)：**

- `animate()` 函数使用 `now_point_id` 变量控制相机移动的路径，并使用 `camera.setLocation()` 和 `camera.changeAzimuth()` 改变相机的位置和方向

```
function animate() {
    var step = 0.05;
```

```
    if (now_point_id == -1) { // 初始向前移动
        if (camera.position[2] > 5)
            camera.setLocation(vec3(camera.position[0], camera.position[1],
camera.position[2] - step));
        else
            now_point_id = 0;
    } else { // 绕方形路径移动
        switch (now_point_id % 4) {
            case 0: // 向右移动
                if (camera.position[0] < 8) {
                    camera.setLocation(vec3(camera.position[0] + step,
camera.position[1], camera.position[2]));
                } else {
                    changeCameraAngle(); // 转弯
                }
                break;
            // ... 其他方向的移动类似
        }
    }
    if (!stopAnimate)
        requestAnimationFrame(animate); // 动画循环
}

function changeCameraAngle() { // 转弯
    var angle_step = 3;
    if (changeAngle < 90) {
        changeAngle += angle_step;
        camera.changeAzimuth(-angle_step);
    } else {
        now_point_id++;
        changeAngle = 0;
    }
}
```

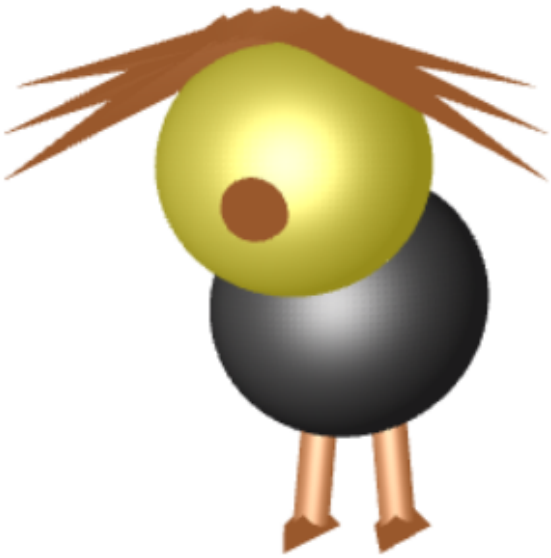
5. **预定义视点切换（按钮事件处理函数）：**这些函数直接调用 `camera.setLocation()` 和 `camera.changeElevation()`/`camera.changeAzimuth()` 来设置相机的特定位置和方向。

```
document.getElementById("down_look").onclick = () => {
    camera.setLocation(vec3(0, 15, 0));
    camera.azimuth = 0;
    camera.elevation = 0;
    camera.changeElevation(90);
    isFirst = false;
    stopAnimate = true;
};
// ... 其他按钮类似
```

总结：

`Camera` 类负责管理相机的状态和变换。 `CameraInteractor` 类处理鼠标交互，更新相机状态。 `animate()` 函数实现场景漫游的动画效果。 按钮事件处理函数提供预定义的视点切换。

实现效果图



鲲鲲

Left Rotate | Right Rotate | Forward | Backwad | Shrink | Expand

篮球

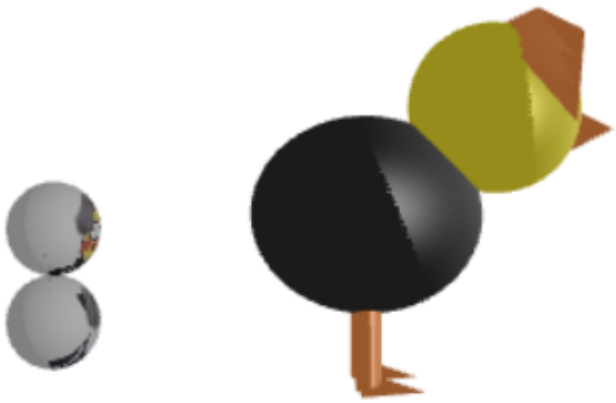
Left Rotate | Right Rotate | Forward | Backwad | Shrink | Expand

相机

Free Walk | Stop Walk

视点

俯视图 | 正视图 | 左视图 | 右视图 | 后视图



鲲鲲

Left Rotate | Right Rotate | Forward | Backwad | Shrink | Expand

篮球

Left Rotate | Right Rotate | Forward | Backwad | Shrink | Expand

相机

Free Walk | Stop Walk

视点

俯视图 | 正视图 | 左视图 | 右视图 | 后视图

个人总结

通过本次实验,了解了观察模型中的各种类型的投影,以及模-视变换矩阵，使用模-视变换矩阵把对象的顶点表示从对象标架变换到眼标架。通过研究经典的观察模型，设计出了场景漫游的算法，并利用现成的设定照相机标

架的API，利用原声webgl实现了这一算法。

实验四

实验内容

纹理映射：

掌握几种映射方法及WebGL中的纹理映射的设置方法。 实现背景和物体表面的贴图，实现下面几种效果：

1. 贴图固定在物体表面，随物体变动
2. 纹理场效果
3. 环境贴图效果

设计思路

本次实验我负责的是环境贴图效果。
环境贴图的效果关键技术,是物理学的镜面反射原理

上图,首先需要计算视线向量,然后以物体表面法线为轴,计算反射向量,着色器语言(GLSL ES)内置了`reflect`函数来计算反射向量,注意这里的视线向量 `eye`的方向是反的,所以传到 `reflect`函数中的时候是 `-eye` .

```
// 计算视线向量eye
        vec3 eye = normalize( u_eyePosition - v_position );
// 计算反射向量，即纹理坐标
        vec3 texCoord = reflect( -eye, v_normal );
        gl_FragColor = textureCube( u_sampler, texCoord );
```

环境反射的基本思想是模拟物体表面像镜子一样反射周围环境。为了实现这一点，我们使用立方体贴图。立方体贴图由六张正方形纹理组成，分别代表一个立方体的六个面（+X, -X, +Y, -Y, +Z, -Z）。这些纹理共同构成了一个“天空盒”，它包围着场景，并被用来模拟环境。

当渲染一个物体时，对于每个片段（像素），我们计算从摄像机到该片段的向量（`eyeToSurfaceDir`），以及该片段的法线向量（`worldNormal`）。然后，我们使用 `reflect` 函数计算反射向量。这个反射向量指向环境中的某个方向。我们使用这个反射向量作为立方体贴图的纹理坐标，从立方体贴图中采样颜色。这个采样得到的颜色就是该片段的环境反射颜色。

实现步骤

1. **顶点着色器：**

- 接收顶点的位置（`vPosition`）和法线（`vNormal`）属性。
- 接收投影矩阵（`u_projection`）、视图矩阵（`u_view`）和世界矩阵（`u_world`）uniform 变量。
- 计算顶点在世界空间中的位置（`v_worldPosition`）和法线（`v_worldNormal`），并将它们传递给片段着色器。

```
attribute vec4 vPosition;
attribute vec3 vNormal;

uniform mat4 u_projection;
uniform mat4 u_view;
uniform mat4 u_world;

varying vec3 v_worldPosition;
varying vec3 v_worldNormal;

void main() {
    gl_Position = u_projection * u_view * u_world * vPosition;
    v_worldPosition = (u_world * vPosition).xyz;
    v_worldNormal = mat3(u_world) * vNormal;
}
```

2. **片段着色器：**

- 接收来自顶点着色器的世界坐标（`v_worldPosition`）和世界法线（`v_worldNormal`）。
- 接收立方体贴图采样器（`u_texture`）和相机在世界坐标系中的位置（`u_worldCameraPosition`）uniform 变量。
- 计算反射向量（`direction`）。

- 使用 `textureCube` 函数使用反射向量从立方体贴图中采样颜色，并将结果赋给片段颜色（`gl_FragColor`）。

```
precision mediump float;

varying vec3 v_worldPosition;
varying vec3 v_worldNormal;

uniform samplerCube u_texture;
uniform vec3 u_worldCameraPosition;

void main() {
    vec3 worldNormal = normalize(v_worldNormal);
    vec3 eyeToSurfaceDir = normalize(v_worldPosition -
u_worldCameraPosition);
    vec3 direction = reflect(eyeToSurfaceDir, worldNormal);
    gl_FragColor = textureCube(u_texture, direction);
}
```

3. **JavaScript代码**

- `configureCubeMap` 函数加载六张图像并创建立方体贴图。重要的是要设置 `image.crossOrigin = "anonymous";` 以允许跨域加载图像。并且在图片加载完成后使用 `gl.generateMipmap(gl.TEXTURE_CUBE_MAP)`生成mipmap，并设置 `gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);`来使用mipmap进行纹理过滤，以减少远处物体的锯齿感。
- `render` 函数计算投影矩阵、视图矩阵和世界矩阵，并将它们传递给着色器。它还设置了立方体贴图uniform变量。
- `generateNormals` 函数为立方体的每个面生成法线向量。这里使用的是面法线，因为代码没有使用顶点法线进行平滑处理。
- 通过按钮控制旋转。
- 1. **配置立方体贴图 (configureCubeMap 函数):**

  这是设置环境反射的核心部分。它负责加载 6 张图像并创建立方体贴图。

```
function configureCubeMap(gl) {
var texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_CUBE_MAP, texture);

const faceInfos = [
    { target: gl.TEXTURE_CUBE_MAP_POSITIVE_X, url: 'pos-x.jpg' },
    { target: gl.TEXTURE_CUBE_MAP_NEGATIVE_X, url: 'neg-x.jpg' },
    { target: gl.TEXTURE_CUBE_MAP_POSITIVE_Y, url: 'pos-y.jpg' },
    { target: gl.TEXTURE_CUBE_MAP_NEGATIVE_Y, url: 'neg-y.jpg' },
    { target: gl.TEXTURE_CUBE_MAP_POSITIVE_Z, url: 'pos-z.jpg' },
    { target: gl.TEXTURE_CUBE_MAP_NEGATIVE_Z, url: 'neg-z.jpg' }
```

```
    ];

    faceInfos.forEach((faceInfo) => {
        const { target, url } = faceInfo;

        const level = 0;
        const internalFormat = gl.RGBA;
        const width = 1024;
        const height = 1024;
        const format = gl.RGBA;
        const type = gl.UNSIGNED_BYTE;

        gl.texImage2D(target, level, internalFormat, width, height, 0, format,
    type, null);

        const image = new Image();
        image.src = url;
        image.crossOrigin = "anonymous"; // 允许跨域加载图像，非常重要!
        image.addEventListener('load', function() {
            gl.bindTexture(gl.TEXTURE_CUBE_MAP, texture);
            gl.texImage2D(target, level, internalFormat, format, type, image);
            gl.generateMipmap(gl.TEXTURE_CUBE_MAP); // 加载完成后生成mipmap
        });
    });

    gl.generateMipmap(gl.TEXTURE_CUBE_MAP); // 初始生成mipmap，也可能在图片加载前执
    行
    gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_MIN_FILTER,
    gl.LINEAR_MIPMAP_LINEAR); // 设置mipmap过滤

    return texture;
    }
```

- `gl.createTexture()`: 创建一个纹理对象。
- `gl.bindTexture(gl.TEXTURE_CUBE_MAP, texture)`: 将纹理绑定到立方体贴图目标。
- `faceInfos` 数组: 存储立方体每个面的目标和图像 URL。
- `image.crossOrigin = "anonymous"`;: 允许跨域加载图像。如果图像和网页不在同一个域名下，**这一行是必需的，否则纹理将无法加载。**
- `gl.texImage2D(...)`: 将图像数据加载到立方体贴图的每个面上。初始调用使用 null 占位，在图像加载完成后再次调用以填充实际图像数据。
- `gl.generateMipmap(gl.TEXTURE_CUBE_MAP)`: 生成 Mipmap，用于在物体距离较远时提供更平滑的纹理效果。在图片加载完成后再次调用确保mipmap正确生成。
- `gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR)`;: 设置纹理过滤方式为使用mipmap进行线性插值，以减少远处物体的锯齿感。

- 2. **渲染循环** 在渲染循环中，计算视图矩阵、世界矩阵，并将它们以及相机位置传递给着色器。

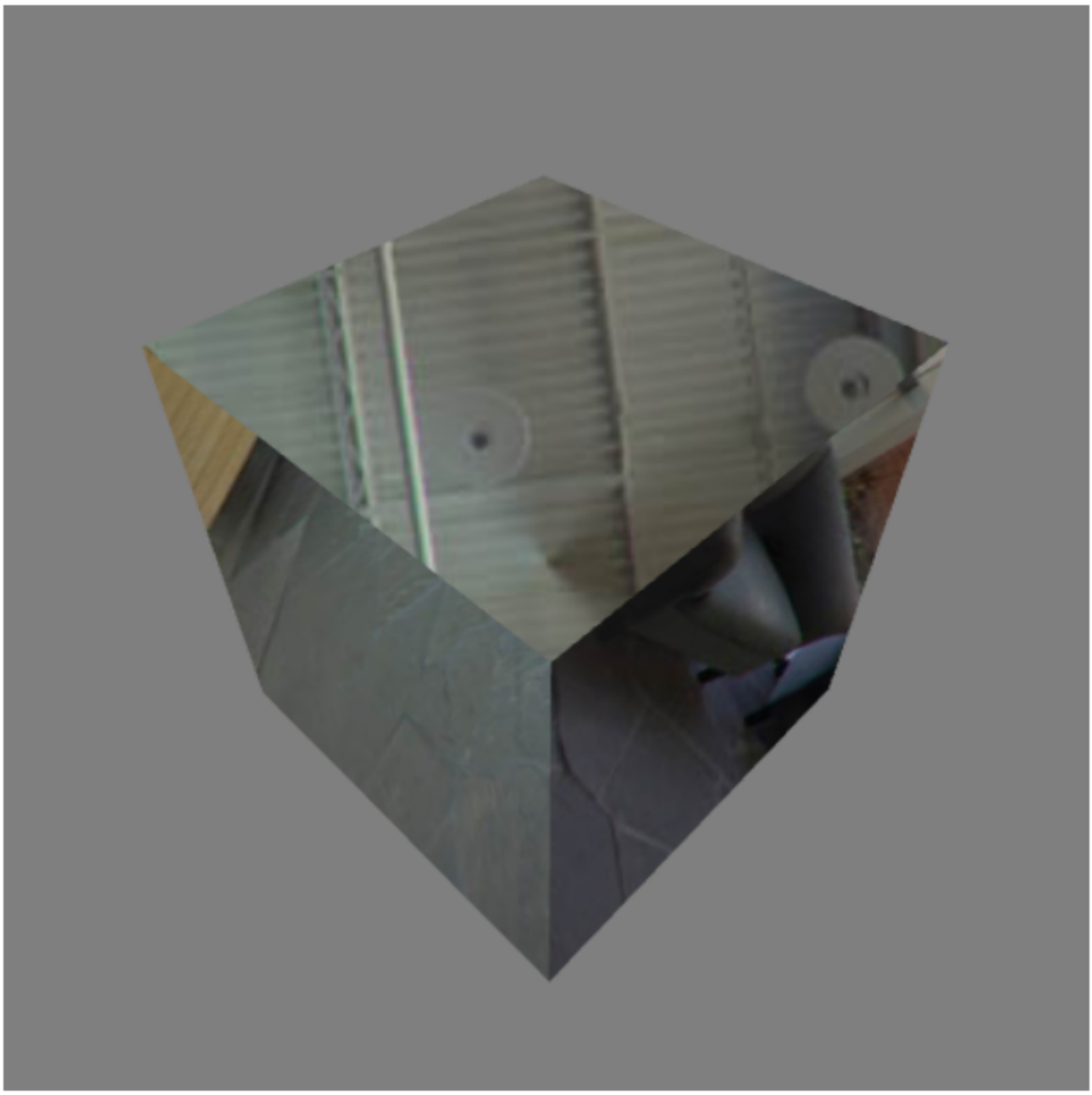
```
function render(time) {
// ... (时间计算等)

// 设置相机
var cameraPosition = vec3(0, 0, 2);
var target = vec3(0, 0, 0);
var up = vec3(0, 1, 0);
var viewMatrix = lookAt(cameraPosition, target, up);
gl.uniformMatrix4fv(viewLocation, false, flatten(viewMatrix));

// 更新世界矩阵
if (isRotating) { // 根据状态旋转
    modelYRotationRadians += -0.4 * deltaTime;
    modelXRotationRadians += -0.4 * deltaTime;
}

var worldMatrix = mult(rotateX(radToDeg(modelXRotationRadians)),
                       rotateY(radToDeg(modelYRotationRadians)));
gl.uniformMatrix4fv(worldLocation, false, flatten(worldMatrix));

// 传递相机位置和纹理
gl.uniform3fv(worldCameraPositionLocation, cameraPosition);
gl.uniform1i(textureLocation, 0); // 激活纹理单元 0

gl.drawArrays(gl.TRIANGLES, 0, numVertices);

requestAnimFrame(render);
}
```

- `lookAt(cameraPosition, target, up)`: 创建视图矩阵，定义了摄像机的位置、观察目标和“向上”方向。
- `rotateX/rotateY`: 创建旋转矩阵，用于旋转模型。
- `gl.uniform3fv(worldCameraPositionLocation, cameraPosition)`: 将摄像机在世界坐标系中的位置传递给片段着色器，这是计算反射向量所必需的。
- `gl.uniform1i(textureLocation, 0)`: 将立方体贴图绑定到纹理单元 0。
- 3. **生成法线 (generateNormals 函数):**

为立方体的每个面生成法线。

```
  function generateNormals() {
    // ... (为每个面定义法线)
    return normals;
}
```

效果图



个人总结

通过本次实验，我成功实现了环境贴图效果。物体表面能够正确地反射周围环境，呈现出逼真的镜面反射效果。通过调整立方体贴图的图像，可以改变反射的环境。本次实验加深了我对纹理映射的理解，特别是环境贴图的实现原理和步骤。我学习了如何使用立方体贴图和反射向量来模拟环境反射，以及如何在 WebGL 中配置和使用立方体贴图。image.crossOrigin = "anonymous"; 的重要性再次得到强调，跨域资源共享（CORS）是 WebGL 开发中需要注意的一个重要问题。Mipmap的使用有效的提升了渲染质量。