Engenharia - Computação Gráfica
Prof: Luciano Soares <lpsoares@insper.edu.br>

WebGL : parte 5

Vamos agora tratar iluminação no nosso sistema em WebGL

1. Iluminando o Cubo

O WebGL é uma API de baixo nível, dessa forma precisamos tomar conta dos menores detalhes para que tudo funcione corretamente. O WebGL executa duas funções principais que você fornece (um vertex shader e um fragment shader) e se espera que você escreva funções criativas para obter os resultados desejados. Em outras palavras, se você deseja iluminação, você mesmo deve calculá-la. Assim veja como se pode fazer um tratamento de iluminação simples para o seu modelo.

Nesse exemplo teremos dois tipos básicos de iluminação:

A luz ambiente é a luz que permeia a cena; é não direcional e afeta todos os polígonos na cena igualmente, independentemente da direção em que esteja.

Luz direcional é a luz emitida de uma direção específica. Esta é a luz que vem de tão longe que cada fóton se move paralelamente a todos os outros fótons. A luz solar, por exemplo, é considerada luz direcional.

Assim teremos uma iluminação ambiente mais uma única fonte de luz direcional, voltada para o cubo giratório. Para realizar os cálculos precisamos associar uma normal de superfície a cada vértice. Este é um vetor perpendicular à face nesse vértice.

Precisamos saber a direção em que a luz está vindo, e isso é definido pelo vetor de direção da luz. Assim atualizamos o vexter shader para ajustar a cor de cada vértice, levando em consideração a iluminação ambiente, bem como o efeito da iluminação direcional dado o ângulo em que atinge o face do cubo.

a. Construindo as normais por vértice

A primeira coisa que precisamos fazer é gerar uma lista de normais para todos os vértices que compõem nosso cubo. Como um cubo é um objeto muito simples, isso é fácil de fazer; obviamente, para objetos mais complexos, o cálculo dos normais será mais complicado.

```
        // Defina as normais por vértice para o cálculo da iluminação
        const normalBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);

        const vertexNormals = [
            // Frente
```

```
            0.0,  0.0,  1.0,
            0.0,  0.0,  1.0,
            0.0,  0.0,  1.0,
            0.0,  0.0,  1.0,

            // Traseira
            0.0,  0.0, -1.0,
            0.0,  0.0, -1.0,
            0.0,  0.0, -1.0,
            0.0,  0.0, -1.0,

            // Superior
            0.0,  1.0,  0.0,
            0.0,  1.0,  0.0,
            0.0,  1.0,  0.0,
            0.0,  1.0,  0.0,

            // Inferior
            0.0, -1.0,  0.0,
            0.0, -1.0,  0.0,
            0.0, -1.0,  0.0,
            0.0, -1.0,  0.0,

            // Direita
            1.0,  0.0,  0.0,
            1.0,  0.0,  0.0,
            1.0,  0.0,  0.0,
            1.0,  0.0,  0.0,

            // Esquerda
            -1.0,  0.0,  0.0,
            -1.0,  0.0,  0.0,
            -1.0,  0.0,  0.0,
            -1.0,  0.0,  0.0
        ];

        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexNormals),
                      gl.STATIC_DRAW);

        return {
            position: positionBuffer,
            //color: colorBuffer,
            textureCoord: textureCoordBuffer,
            indices: indexBuffer,
            normal: normalBuffer,
        };
    }
```

Isso deve parecer bastante familiar agora; criamos um novo buffer, associamos a ele o buffer com o qual estamos trabalhando e, em seguida, enviamos a lista de normais por vértice para o buffer chamando bufferData().

Em seguida, adicionaremos o código em drawScene() para vincular a matriz de normais a um atributo do shader para que o código dele possa obter acesso aos dados

```
        // Diga ao WebGL como colocar as normais do buffer de
        // normais no atributo vertexNormal.
        {
            const numComponents = 3;
            const type = gl.FLOAT;
            const normalize = false;
            const stride = 0;
            const offset = 0;
            gl.bindBuffer(gl.ARRAY_BUFFER, buffers.normal);
            gl.vertexAttribPointer(
                programInfo.attribLocations.vertexNormal,
                numComponents,
                type,
                normalize,
                stride,
                offset);
            gl.enableVertexAttribArray(
                programInfo.attribLocations.vertexNormal);
        }

        // Diga ao WebGL quais indices usar para conectar os vértices
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffers.indices);
```

Finalmente, precisamos atualizar o código que constrói as matrizes uniformes para gerar e entregar ao shader uma matriz com as normais nas posições corretas, que é usada assim para tratar as normais baseado na orientação atual do cubo e em relação à fonte de luz.

```
       mat4.rotate(modelViewMatrix,  // matriz de destino
                   modelViewMatrix,  // matriz para rotacionar
                   squareRotation*.7,// quantidade de radianos a se rotacionar * 0.7
                   [0, 1, 0]);       // eixo para ser girar em volta

       const normalMatrix = mat4.create();
       mat4.invert(normalMatrix, modelViewMatrix);
       mat4.transpose(normalMatrix, normalMatrix);

       // Diga ao WebGL como retirar as posições do
       // atributo vertexPosition do buffer
```

Crie a entrada para a matriz uniforme para as normais.

```
        gl.uniformMatrix4fv(
            programInfo.uniformLocations.modelViewMatrix,
            false,
            modelViewMatrix);
        gl.uniformMatrix4fv(
            programInfo.uniformLocations.normalMatrix,
            false,
            normalMatrix);
```

b. Atualize os Shaders

A primeira coisa a fazer é atualizar o vertex shader para que ele gere um valor no shader para cada vértice com base na iluminação ambiente e também na iluminação direcional:

```
const vsSource = `
    attribute vec4 aVertexPosition;
    attribute vec3 aVertexNormal;
    attribute vec2 aTextureCoord;

    uniform mat4 uNormalMatrix;
    uniform mat4 uModelViewMatrix;
    uniform mat4 uProjectionMatrix;

    varying highp vec2 vTextureCoord;
    varying highp vec3 vLighting;

    void main(void) {
      gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
      vTextureCoord = aTextureCoord;

      // Apply lighting effect

      highp vec3 ambientLight = vec3(0.3, 0.3, 0.3);
      highp vec3 directionalLightColor = vec3(1, 1, 1);
      highp vec3 directionalVector = normalize(vec3(0.85, 0.8, 0.75));

      highp vec4 transformedNormal = uNormalMatrix * vec4(aVertexNormal, 1.0);

      highp float directional = max(dot(transformedNormal.xyz, directionalVector), 0.0);
      vLighting = ambientLight + (directionalLightColor * directional);
    }
  `;
```

Uma vez que a posição do vértice é calculada e passamos as coordenadas do texel correspondentes do vértice para o fragmente shader, podemos trabalhar no cálculo do vertex shader.

A primeira coisa que fazemos é transformar a normal com base na orientação atual do cubo, multiplicando a normal do vértice pela matriz normal. Podemos então calcular a quantidade de iluminação direcional que precisa ser aplicada ao vértice, calculando o produto escalar da normal transformada e do vetor direcional (ou seja, a direção de onde a luz está vindo). Se esse valor for menor que zero, fixamos o valor em zero, já que você não pode ter menos que luz zero.

Uma vez que a quantidade de iluminação direcional é calculada, podemos gerar o valor da iluminação pegando a luz ambiente e adicionando o produto da cor da luz direcional e a quantidade de iluminação direcional a ser fornecida. Como resultado, agora temos um valor RGB que será usado pelo fragmente shader para ajustar a cor de cada pixel que renderizamos.

O fragmente shader agora precisa ser atualizado para levar em consideração o valor de iluminação calculado pelo vertex shader.

```
  const fsSource = `
    varying highp vec2 vTextureCoord;
    varying highp vec3 vLighting;

    uniform sampler2D uSampler;

    void main(void) {
      highp vec4 texelColor = texture2D(uSampler, vTextureCoord);

      gl_FragColor = vec4(texelColor.rgb * vLighting, texelColor.a);
    }
  `;
```

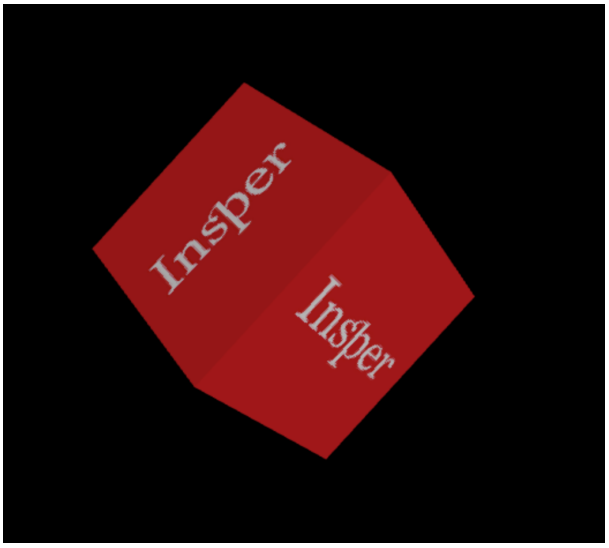
Aqui, buscamos a cor do texel, assim como fizemos no exemplo anterior, mas antes de definir a cor do fragmento, multiplicamos a cor do texel pelo valor de iluminação para ajustar a cor do texel para levar em conta o efeito de nossa luz fontes.

A única coisa que resta é procurar a localização do atributo aVertexNormal e do uniform uNormalMatrix.

```
        const programInfo = {
            program: shaderProgram,
            attribLocations: {
                vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),
                vertexNormal: gl.getAttribLocation(shaderProgram, 'aVertexNormal'),
                //vertexColor: gl.getAttribLocation(shaderProgram, 'aVertexColor'),
                textureCoord: gl.getAttribLocation(shaderProgram, 'aTextureCoord'),
            },
            uniformLocations: {
```

```
                projectionMatrix: gl.getUniformLocation(shaderProgram, 'uProjectionMatrix'),
                modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),
                normalMatrix: gl.getUniformLocation(shaderProgram, 'uNormalMatrix'),
                uSampler: gl.getUniformLocation(shaderProgram, 'uSampler'),
            },
        };
```

E é isso, vejo como seu cubo parece mais natural agora:



Referências:

Esse documento foi baseado em:

- WebGL tutorial : https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial