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

WebGL : parte 4

Texturas, vamos colocar texturas no cubo.

1. Texturizando o Cubo

Até agora usamos só cores e degrade de cores para colorir os objetos. Vamos agora usar texturas sobre os objetos, e no caso vamos fazer isso no cubo que criamos. Assim a primeira coisa a fazer é carregar as texturas dos arquivos de imagens. No caso, usaremos uma única textura, mapeada em todos os seis lados de nosso cubo, mas a mesma técnica pode ser usada para qualquer número de texturas.

Observação: o carregamento de texturas segue regras de domínio cruzado, ou seja, você só pode carregar texturas de sites para os quais seu conteúdo tem aprovação do CORS. Assim, você tem algumas opções para visualizar a textura do seu projeto. Uma é habilitar o Firefox para fazer isso, para isso acesse o about:config e troque o privacy.file_unique_origin para false, no Chrome: https://chrome.google.com/webstore/detail/allow-cors-access-control/lhobafahddgcelffkeicbaginigeejlf?hl=en-US
outra opção é seguir este tutorial: http://hacks.mozilla.org/2011/11/using-cors-to-load-webgl-textures-from-cross-domain-images/, você pode criar um servidor web local (Flask por exemplo) ou finalmente hospedar em algum lugar, por exemplo o GitHub pages.

Comece inserindo o seguinte código para ter as funções para ler a textura e verificar se as dimensões dela são potência de dois.

```
    // Carrega a textura para o contexto WebGL
    function loadTexture(gl, url) {
        const texture = gl.createTexture();
        gl.bindTexture(gl.TEXTURE_2D, texture);

        // Como as imagens precisam ser baixadas da internet,
        // elas podem demorar um pouco até estarem prontas.
        // Assim coloque um único pixel na textura para que
        // possamos usá-la imediatamente. Quando a imagem terminar
        // de carregar, vamos atualizar a textura com o conteúdo dela.
        const level = 0;
        const internalFormat = gl.RGBA;
        const width = 1;
        const height = 1;
        const border = 0;
        const srcFormat = gl.RGBA;
        const srcType = gl.UNSIGNED_BYTE;
        const pixel = new Uint8Array([0, 0, 255, 255]);  // opaque blue
```

```
        gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
                      width, height, border, srcFormat, srcType,
                      pixel);

        const image = new Image();
        image.onload = function() {
            gl.bindTexture(gl.TEXTURE_2D, texture);
            gl.texImage2D(gl.TEXTURE_2D, level, internalFormat,
                          srcFormat, srcType, image);

            // WebGL1 tem requisitos diferentes para potência de 2 em imagens
            // versus não potência de 2 nas imagens, portanto, verifique se
            // a imagem é potência de 2 em ambas as dimensões.
            if (isPowerOf2(image.width) && isPowerOf2(image.height)) {
                gl.generateMipmap(gl.TEXTURE_2D);  // Potência de 2, gere os mipmaps
            } else {
                // Não é potência de 2. Desligar mipmaps e wrapping para clamp to edge
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
            }
        };

        image.src = url;

        return texture;
    }

    function isPowerOf2(value) {
        return (value & (value - 1)) == 0;
    }
```

A rotina loadTexture() começa criando uma textura WebGL chamando a função createTexture() do próprio WebGL. Em seguida, ela carrega um único pixel azul usando texImage2D(). Isso torna a textura imediatamente utilizável como uma cor azul sólida, embora possa levar algum tempo para o download da imagem, já será possível ver o objeto na cena (mesmo que azul).

Para carregar a textura do arquivo de imagem, se deve criar um objeto Image e atribui o valor do campo src para a url da imagem que desejamos usar como textura. A função image.onload() será chamada assim que o download da imagem for concluído. Nesse ponto, chamamos novamente texImage2D(), desta vez usando a imagem como fonte para a textura. Depois disso, configuramos a filtragem e o empacotamento da textura com base no fato da imagem que baixamos ter ou não uma potência de 2 em ambas as dimensões.

No WebGL 1 só se pode usar texturas sem potência de 2 com filtro definido para NEAREST ou LINEAR e não se pode gerar um mipmap para elas. Seu modo também deve ser definido como CLAMP_TO_EDGE. Por outro lado, se a textura for uma potência de 2 em ambas as dimensões,

o WebGL pode fazer uma filtragem de qualidade superior, se pode usar mipmap e pode definir o modo de empacotamento para REPEAT ou MIRRORED_REPEAT.

Um exemplo de textura que se repete é colocar lado a lado uma imagem para cobrir uma parede com se fossem de tijolos.

Mipmapping e repetição de UV podem ser desabilitados com texParameteri(). Isso permitirá texturas sem potência de dois (NPOT) às custas de mipmapping, UV wrapping, UV tiling, e seu controle sobre como o dispositivo tratará sua textura.

Novamente, com esses parâmetros, dispositivos WebGL compatíveis aceitarão automaticamente qualquer resolução para aquela textura (até suas dimensões máximas). Sem realizar a configuração acima, o WebGL fará com que todas as amostras de texturas que não seja potência de 2 falhem, retornando um preto transparente: rgba (0,0,0,0).

Para carregar a imagem, adicione uma chamada à função loadTexture() dentro da função main(). Isso pode ser adicionado após a chamada initBuffers(gl).

```
        const buffers = initBuffers(gl);

        const texture = loadTexture(gl, 'logo.png');

        var then = 0;
```

a. Mapeando Texturas nas Faces do Cubo

Neste ponto, a textura está carregada e pronta para uso. Mas antes de podermos usá-lo, precisamos estabelecer o mapeamento das coordenadas de textura para os vértices das faces do cubo. Para isso você irá substitui todo o código existente anteriormente para configurar cores para cada uma das faces do cubo em initBuffers().

```
// // Definindo uma cor para cada face do cubo
// const faceColors = [
//   [1.0,  1.0,  1.0,  1.0],    // Face frontal: branca
//   [1.0,  0.0,  0.0,  1.0],    // Face traseira: vermelha
//   [0.0,  1.0,  0.0,  1.0],    // Face superior: verde
//   [0.0,  0.0,  1.0,  1.0],    // Face inferior: azul
//   [1.0,  1.0,  0.0,  1.0],    // Face direita: amarela
//   [1.0,  0.0,  1.0,  1.0],    // Face esquerda: violeta
// ];

// // Convertendo o vetor de cores para uma matriz para todos os 24 vértices
// var colors = [];
```

Prof: Luciano Soares <lpsoares@insper.edu.br>

```
// for (var j = 0; j < faceColors.length; ++j) {
//   const c = faceColors[j];

//   // Repita cada cor 4 vezes para os 4 vértices das faces do cubo
//   colors = colors.concat(c, c, c, c);
// }

// const colorBuffer = gl.createBuffer();
// gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
// gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);

        // Definindo as coordenadas de textura
        const textureCoordBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordBuffer);

        const textureCoordinates = [
            // Frente
            1.0,  0.0,
            0.0,  0.0,
            0.0,  1.0,
            1.0,  1.0,
            // Traseira
            1.0,  0.0,
            0.0,  0.0,
            0.0,  1.0,
            1.0,  1.0,
            // Superio
            1.0,  0.0,
            0.0,  0.0,
            0.0,  1.0,
            1.0,  1.0,
            // Inferior
            1.0,  0.0,
            0.0,  0.0,
            0.0,  1.0,
            1.0,  1.0,
            // Direita
            1.0,  0.0,
            0.0,  0.0,
            0.0,  1.0,
            1.0,  1.0,
            // Esquerda
            1.0,  0.0,
            0.0,  0.0,
            0.0,  1.0,
            1.0,  1.0,
        ];
```

```
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoordinates),
                      gl.STATIC_DRAW);
```

Ajuste também a rotina para retornar o novo buffer e não mais retornar o buffer de cores.

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

Esse código acima cria um buffer WebGL no qual armazenaremos as coordenadas de textura para cada face, depois vinculamos esse buffer como o vetor no qual iremos colocar os valores.

A matriz textureCoordinates define as coordenadas de textura correspondentes de cada vértice de cada face do cubo. Observe que as coordenadas da textura variam de 0.0 a 1.0; as dimensões das texturas são normalizadas para um intervalo de 0.0 a 1.0, independentemente de seu tamanho real, para fins de mapeamento de textura.

Depois de configurar o vetor de mapeamento de textura, passaremos o vetor para o buffer, para que o WebGL tenha esses dados prontos para uso.

Precisamos agora atualizar os Shaders para usar de fato as coordenadas de textura e a textura. Comece pelo Vertex Shader.

```
        // Vertex shader
        const vsSource = `
            attribute vec4 aVertexPosition;
            //attribute vec4 aVertexColor;
            attribute vec2 aTextureCoord;

            uniform mat4 uModelViewMatrix;
            uniform mat4 uProjectionMatrix;

            //varying lowp vec4 vColor;
            varying highp vec2 vTextureCoord;

            void main() {
                gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
                //vColor = aVertexColor;
                vTextureCoord = aTextureCoord;
            }
        `;
```

A principal mudança aqui é que em vez de buscar a cor do vértice, buscamos as coordenadas de textura e as passamos para o vexter shader; isso indicará a localização dentro da textura para os vértices do objeto.

Da mesma forma o Fragment Shader precisa ser atualizado.

```
        const fsSource = `
            //varying lowp vec4 vColor;
            varying highp vec2 vTextureCoord;

            uniform sampler2D uSampler;

            void main(void) {
                gl_FragColor = texture2D(uSampler, vTextureCoord);
            }
        `;
```

Em vez de atribuir um valor de cor ao fragmento shader, a cor do pixel é calculada buscando o texel (ou seja, o pixel dentro da textura) com base no valor de vTextureCoord que, como as cores, é interpolado entre os vértices.

Como alteramos um atributo e adicionamos um uniform, precisamos ajustar suas localizações no código:

```
        const programInfo = {
            program: shaderProgram,
            attribLocations: {
                vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),
                //vertexColor: gl.getAttribLocation(shaderProgram, 'aVertexColor'),
                textureCoord: gl.getAttribLocation(shaderProgram, 'aTextureCoord'),
            },
            uniformLocations: {
                projectionMatrix: gl.getUniformLocation(shaderProgram, 'uProjectionMatrix'),
                modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),
                uSampler: gl.getUniformLocation(shaderProgram, 'uSampler'),
            },
        };
```

As mudanças na função drawScene() são simples.

Primeiro, o código para especificar o buffer de cores foi removido, substituído por este:

```
        //   // Diga ao WebGL como retirar as cores do buffer de cores
        //   // para colocar no atributo vertexColor.
        //   {
        //     const numComponents = 4;
        //     const type = gl.FLOAT;
```

```
        //     const normalize = false;
        //     const stride = 0;
        //     const offset = 0;
        //     gl.bindBuffer(gl.ARRAY_BUFFER, buffers.color);
        //     gl.vertexAttribPointer(
        //         programInfo.attribLocations.vertexColor,
        //         numComponents,
        //         type,
        //         normalize,
        //         stride,
        //         offset);
        //     gl.enableVertexAttribArray(
        //         programInfo.attribLocations.vertexColor);
        // }

        // Diga ao WebGL como retirar as coordenadas de textura do
        // buffer de coordenadas de textura para colocar no atributo textureCoord
        {
            const numComponents = 2;
            const type = gl.FLOAT;
            const normalize = false;
            const stride = 0;
            const offset = 0;
            gl.bindBuffer(gl.ARRAY_BUFFER, buffers.textureCoord);
            gl.vertexAttribPointer(
                programInfo.attribLocations.textureCoord,
                numComponents,
                type,
                normalize,
                stride,
                offset);
            gl.enableVertexAttribArray(
                programInfo.attribLocations.textureCoord);
        }
```

Em seguida, adicione o código para especificar a textura a ser mapeada nas faces do cubo, logo antes da rotina para desenhar os objetos:

```
        // Especifica o mapeamento de texturas

        // Informe o WebGL que se deseja trabalhar com a unidade de textura 0
        gl.activeTexture(gl.TEXTURE0);

        // Vincule (bind) a textura com a unidade 0
        gl.bindTexture(gl.TEXTURE_2D, texture);

        // Diga ao shader que limitamos a textura à unidade de textura 0
```

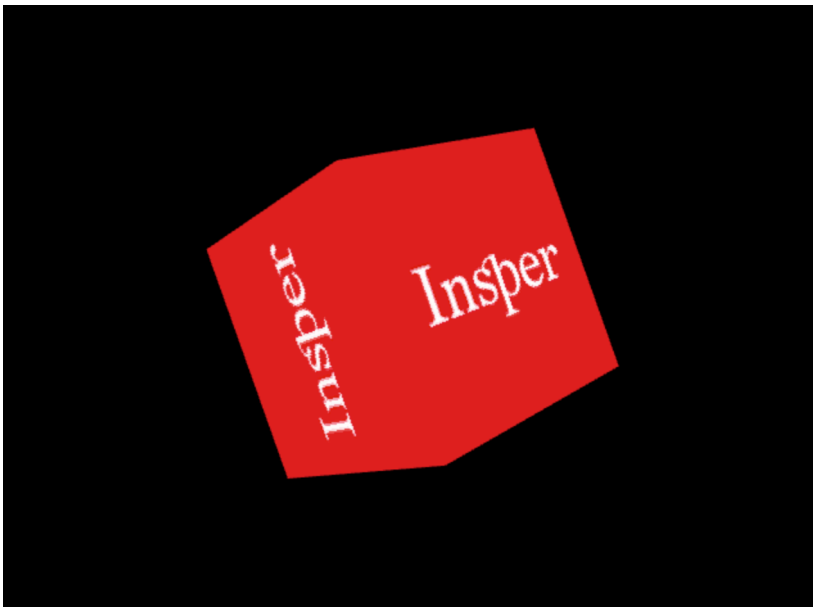
```
        gl.uniform1i(programInfo.uniformLocations.uSampler, 0);

        {
            const offset = 0;
            const vertexCount = 36;
            const type = gl.UNSIGNED_SHORT;
            gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);
        }
```

O WebGL fornece um mínimo de 8 unidades de textura; a primeira delas é gl.TEXTURE0. Dizemos ao WebGL que queremos afetar a unidade 0. Em seguida, chamamos bindTexture() que liga a textura a unidade de textura 0. Em seguida, dizemos ao shader que, para o uSampler, use a unidade de textura 0.

Por último, adicione textura como parâmetro à função drawScene(), tanto onde é definida quanto onde é chamada.

```
drawScene(gl, programInfo, buffers, texture, deltaTime);
...
function drawScene(gl, programInfo, buffers, texture, deltaTime) {
```

Se tudo funcionou devemos conseguir ver o cubo com as texturas nas faces:



Referências:

Esse documento foi baseado em:

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