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

WebGL : parte 2

Continuando a atividade de WebGL vamos agora pintar o nosso quadrado pelas pontas, ou seja, vamos definir uma cor para cada vértice e então fazer um degrade das cores para o centro do quadrado. Depois iremos fazer ele ficar girando e assim iremos usar mais um dos recursos de transformações dos objetos.

1. Aplicando Cores por Vértice

Ao criar os polígonos em WebGL é possível definir cores por vértice para eles. Por padrão, as cores dos pixels (bem como os seus outros atributos) são calculados usando alguma estratégia de interpolação, o que no caso das cores acaba criando gradientes suaves automaticamente. Até o momento o vertex shader não tratava nenhuma cor específica para os vértices e o fragmente shader atribuía uma cor fixa branca para cada pixel. No final o quadrado inteiro foi renderizado com uma cor branca sólida.

Vamos agora fazer com que seja renderizado um gradiente de cores em função de uma cor em cada canto do quadrado. Vamos definir que as cores sejam: vermelho, azul, verde e amarelo. A primeira coisa a fazer é estabelecer essas cores para os quatro vértices. Para fazer isso, primeiro precisamos criar uma matriz de cores para os vértice e, em seguida, armazená-la em um buffer WebGL para conseguir se usar. Para isso adicione o seguinte código à nossa função initBuffers():

```
    function initBuffers(gl) {
    ...

        const colors = [
            1.0,  1.0,  0.0,  1.0,    // amarelo
            0.0,  0.0,  1.0,  1.0,    // azul
            1.0,  0.0,  0.0,  1.0,    // vermelho
            0.0,  1.0,  0.0,  1.0,    // verde
        ];

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

        return {
            position: positionBuffer,
            color: colorBuffer,
        };
    }
```

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

Neste código temos a criação de um vetor em JavaScript contendo quatro grupos de 4 valores, um para cada cor de vértice. Em seguida, um novo buffer WebGL é alocado para armazenar essas cores, o array é então convertido em floats e armazenado no buffer.

Para usar essas cores, o vertex shader precisa ser informado adequadamente para extrair a cor apropriada do buffer de cores. Adicione o seguinte código:

```
    const vsSource = `
        attribute vec4 aVertexPosition;
        attribute vec4 aVertexColor;

        uniform mat4 uModelViewMatrix;
        uniform mat4 uProjectionMatrix;

        varying lowp vec4 vColor;

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

Neste código temos o atributo aVertexColor que será passado para o Fragment Shader pela variável vColor. As cores depois no Fragment shader chegarão interpolada justamente porque declaramos a variável como *varying*. Aproveitando, *lowp* significa baixa precisão (mas que para o caso vai servir), se quiser pode usar *mediump* (precisão média) ou mesmo *highp* (alta precisão)

Já no Fragment Shader, modifique o programa para que ele use as cores por vértice na hora de pintar os pixels.

```
    const fsSource = `
        varying lowp vec4 vColor;

        void main() {
            gl_FragColor = vColor;
        }
    `;
```

Cada fragmento em vez de um valor fixo, irá receber a cor interpolada com base em sua posição em relação às posições dos vértices.

Em seguida, é necessário adicionar o código para passar o atributo das cores e configurar esse atributo para o shader:

```
        const programInfo = {
            program: shaderProgram,
            attribLocations: {
```

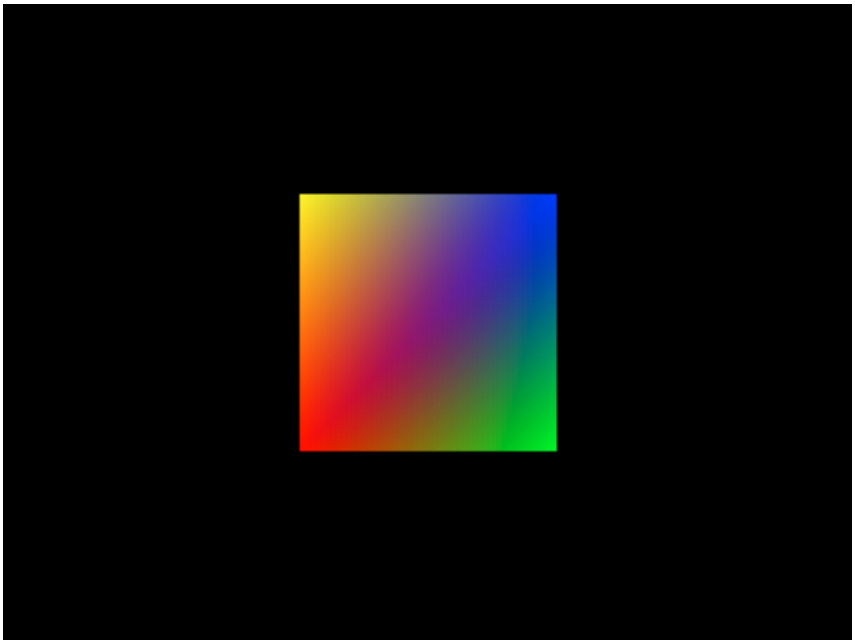
```
            vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),
            vertexColor: gl.getAttribLocation(shaderProgram, 'aVertexColor'),
            },
            uniformLocations: {
            projection
           ...
```

Finalmente, drawScene() pode ter o seguinte código adicionado para use as cores ao desenhar o quadrado:

```
        // 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);
        }
```

Os diversos parâmetros servem para indicar como a memória está organizada. O resultado final deve ser a seguinte imagem:

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



2. Rodando o Quadrado

Provavelmente você tentou mover o objeto na tela. Bem, infelizmente estamos com uma programação bem baixo nível, dessa forma temos de programar tudo, inclusive as interações. Para começar vamos somente rotacionar o objeto.

A primeira coisa de que precisaremos é uma variável para rastrear a rotação atual do quadrado:

```
  <script>

    var squareRotation = 0.0;

    main();
```

Agora precisamos atualizar a função drawScene() para aplicar a rotação no quadrado ao desenhá-lo. Depois de transladar o quadrado para a posição em frente a câmera, aplicamos a rotação:

```
mat4.translate(modelViewMatrix,     // matriz de destino
               modelViewMatrix,     // matriz para transladar
               [0.0, 0.0, -6.0]);  // translação

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

Esse código fara uma modificação na matriz modelViewMatrix pelo valor que esteja na variável squareRotation, em torno do eixo Z.

Para realizar a animação, precisamos adicionar algum código que altere o valor de squareRotation ao longo do tempo. Podemos fazer isso criando uma nova variável para rastrear a hora em que fizemos a última animação. Adicione o código a seguir ao final da função main() para enviar a informação para o drawScene():

```
        const buffers = initBuffers(gl);

        var then = 0;

        // Desenha a cena continuamente
        function render(now) {
            now *= 0.001;  // converte o tempo para segundos
            const deltaTime = now - then;
            then = now;

            drawScene(gl, programInfo, buffers, deltaTime);

            requestAnimationFrame(render);
        }

        requestAnimationFrame(render);

    }
```

Este código usa a função requestAnimationFrame para fazer o navegador chamar a função "render" a cada quadro possível. requestAnimationFrame passa o tempo em milissegundos desde o carregamento da página para a função invocada. Convertemos primeiro o valor em segundos e então subtraímos dele a do valor armazenado da última chamada para calcular deltaTime, que é o número de segundos desde que o último quadro foi renderizado. No final no função drawscene(), adicionamos o código para atualizar squareRotation.

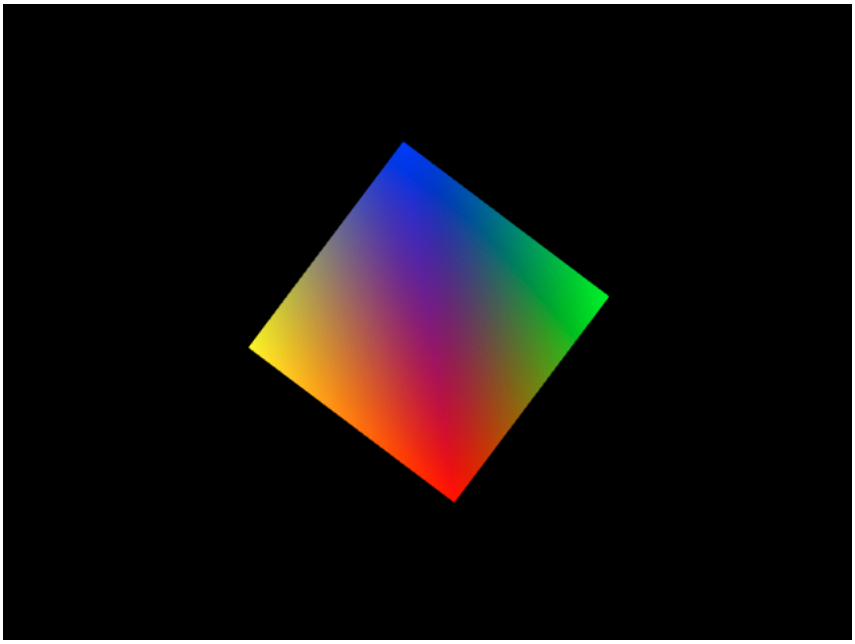
```
function drawScene(gl, programInfo, buffers, deltaTime) {

        // Atualiza o valor da rotação
        squareRotation += deltaTime;
...
```

No final você deve ter um quadrado que fica girando:

Engenharia - Computação Gráfica

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



Referências:

Esse documento foi baseado em:

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