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

WebGL

O WebGL é uma API para JavaScript que permite renderizar conteúdos gráficos (3D e 2D) na web por intermédio do recurso do HTML <canvas> nos navegadores (basicamente todos) que suportam HTML5, com a vantagem que não é necessário instalar nenhum plugin. Os programas WebGL consistem em códigos escrito em JavaScript para os navegadores web que transferem a execução gráfica para as GPUs do computador. Os recursos do WebGL podem ser misturados com outros elementos HTML permitindo a criação de páginas com recursos gráficos sofisticados.

Este documento mostra o básico para usar WebGL em uma página. Os exemplos fornecidos servem como referência para o posterior desenvolvimento de aplicações WebGL mais complexos.

1. HTML e Javascript

Para realizar as atividades desse documento você precisa dos conhecimentos mínimos de HTML e JavaScript, que supostamente vocês deveriam ter por ter cursado disciplinas como Co-design de Aplicativos.

O elemento <canvas> do HTML que permitirá você exibir os conteúdos de WebGl nas páginas web. O elemento <canvas> e WebGL não são suportados em alguns navegadores mais antigos, mas são suportados em versões recentes de todos os principais navegadores.

2. Começando com o básico

O WebGL permite que o conteúdo da web use uma API baseada em OpenGL para executar renderizações tanto em 2D, como em 3D, em um navegador HTML.

Os exemplos de código neste tutorial também podem ser encontrados no repositório GitHub https://github.com/mdn/webgl-examples/tree/gh-pages/tutorial.

É importante ressaltar que embora se apresente o WebGL neste documento, há uma série de estruturas disponíveis que encapsulam os recursos básicos do WebGL, tornando mais simples desenvolver aplicativos e jogos 3D, como por exemplo o THREE.js e BABYLON.js.

3. Preparando o arquivo html

A primeira coisa que você precisa para usar o WebGL para renderizar é um elemento <canvas> no seu arquivo html. O código HTML abaixo serve para declarar um canvas no qual será desenhado algo.

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

```
<html>

  <head>
    <meta charset="utf-8">
  </head>

  <body>
    <canvas id="glcanvas" width="640" height="480"></canvas>
  </body>

</html>
```

A parte interessante do HTML é que podemos agora colocar código para fazer alguma coisa na página. Idealmente isso tem de ficar bem organizado e normalmente separados em arquivos de código para as funcionalidades (.js) e arquivos para as questões visuais (.css). Mas para agora vamos deixar tudo mais junto.

Adicione agora um código em JavaScript que executa ao carregar a página.

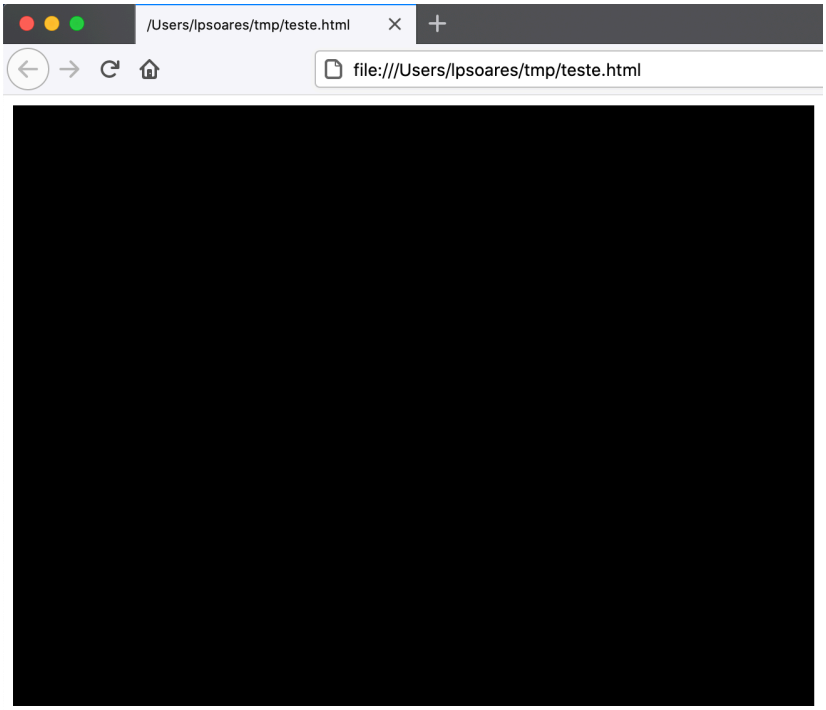
```
<html>

  <head>
    <meta charset="utf-8">
  </head>

  <body>
    <canvas id="glcanvas" width="640" height="480"></canvas>
  </body>

  <script>
    main();

    // Início do Sistema WebGL
    function main() {
      const canvas = document.querySelector('#glcanvas');
      // Initializa o context GL
      const gl = canvas.getContext('webgl');

      // Caso o navegador não suporte WebGL
      if (!gl) {
        alert(' Não foi possível iniciar o WebGL. Seu navegador não deve suportá-lo.');
        return;
      }

      // Defina a cor de limpeza para preto sem transparência
      gl.clearColor(0.0, 0.0, 0.0, 1.0);
      // Limpe o buffer de cores com a cor especificada
      gl.clear(gl.COLOR_BUFFER_BIT);
    }
  </script>
</html>
```

Esse código recupera o elemento <canvas> o que permite chegar ao contexto do WebGL.

Assim que recuperarmos o “canvas”, tentaremos obter um WebGLRenderingContext através do método getContext() e passando a string "webgl" como parâmetro. Se o navegador não suportar oWebGL, o getContext() retornará null, e nesse caso exibimos uma mensagem ao usuário e encerramos o janela.

Se o contexto for inicializado com sucesso, a variável gl será usada como a referência para o contexto WebGL. Assim já definimos uma cor para limpar o buffer e já limpamos o contexto com essa cor (redesenhando a tela com a cor de fundo).

Neste ponto, você tem código suficiente para que o contexto WebGL seja inicializado com êxito e você deve conseguir ver uma grande área preta sem mais nada, pronta e esperando para receber os conteúdos gráficos.



4. O quadrado

Depois de criar um contexto WebGL, você pode começar a renderizar nele. Para começar vamos desenhar um quadrado plano sem cor e sem texturiza, e conforme for ir evoluindo o programa para deixá-lo mais interessante.

Este projeto usa a biblioteca glMatrix para realizar suas operações de matriz, portanto, você precisará incluí-la em seu projeto. Você pode carregar uma cópia de um CDN direto no <head> do arquivo HTML.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/gl-matrix/2.8.1/gl-matrix-min.js"
        integrity="sha512-
zhHQR0/H5SEBL3Wn6yYSaTTZej12z0hVZKOv3TwCUXT1z5qeqGcXJLLrbERYRScEDDpYIJhPC1fk31gqR783iQ=="
        crossorigin="anonymous">
</script>
```

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

A proposta deste exemplo é desenhar um quadrado e coloca-lo diretamente na frente da câmera perpendicular à direção da vista. Precisamos criar os *shaders* que definirão a cor de da cena, bem como desenharão o objeto 3D.

Os shaders
Shaders no WebGL são programas escritos usando a OpenGL ES Shading Language (GLSL), que obtém informações sobre os vértices da forma geométrica e gera os dados necessários para renderizar os pixels na tela.

Existem duas funções principais de shaders para desenhar o conteúdo WebGL: o vertex shader e o fragment shader. Ambos são escritos em GLSL que são passados no código WebGL que será compilado para execução na GPU. O conjunto vertex shader e o fragment shader é chamado de shaders program.

Vertex Shader
Cada vez que uma forma geométrica é renderizada, o vertex shader é executado em cada vértice na geometrial, e seu trabalho é transformar o vértice de entrada para o sistema de coordenadas do espaço da tela usado pelo WebGL, no qual cada eixo tem um intervalo de -1,0 a 1,0, independentemente da proporção, tamanho real ou quaisquer outros fatores.

O vertex shader deve realizar as transformações necessárias na posição do vértice, além de quaisquer outros ajustes ou cálculos necessários e, em seguida, retornar o vértice transformado por uma variável especial do GLSL chamada de gl_Position.

O vertex shader pode, conforme necessário, também fazer coisas como determinar as coordenadas de textura, aplicar as normais para o cálculo de iluminação e assim por diante. Essas informações são armazenadas para serem depois usadas no fragmente shader.

O vertex shader apresentado abaixo irá receber os valores da posição de cada vértice em um atributo chamado aVertexPosition. Essa posição é então multiplicada por duas matrizes homogêneas (4x4) de projeção: uProjectionMatrix e de câmera: uModelViewMatrix. A variável gl_Position vai então receber o resultado dessa operação.

Insira o seguinte código na função main():

```
    // Vertex shader
    const vsSource = `
        attribute vec4 aVertexPosition;

        uniform mat4 uModelViewMatrix;
        uniform mat4 uProjectionMatrix;

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

É importante notar que estamos usando um atributo vec4 para a posição do vértice, ou seja, estamos usando coordenadas homogêneas para ele. Neste momento ainda não estamos computando nenhuma iluminação nem texturas que serão tratadas adiante.

Fragment Shader

O fragmente shader é chamado a cada pixel e em cada forma geométrica a ser desenhada, isso ocorre depois que os vértices da geometria forem processados pelo vertex shader. O trabalho do fragment shaders é determinar a cor de cada pixel, usando informações da cor do material, textura, iluminação, etc. A cor calculada é então retornada ao WebGL através da variável gl_FragColor. Essa cor é então desenhada na tela na posição correta do pixel.

Nesse exemplo a seguir, estamos simplesmente retornando o branco todas as vezes que algum pixel gerado pela geometria surgir. No caso estamos apenas desenhando um quadrado branco, sem nenhuma textura, iluminação ou outro recurso no momento.

Insira o seguinte código na função main():

```
    const fsSource = `
        void main() {
            gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
        }
    `;
```

Inicializando os shaders

Agora que definimos os dois shaders, precisamos passá-los para o WebGL, que ira compilá-los e vinculá-los a rotina gráfica que estamos montando. O código a seguir prepara os dois shaders chamando uma função loadShader() que foi criada para simplificar a operação. Nessa função passamos o tipo de shaders e o código fonte do shader. Se a compilação ou vinculação falhar, o código exibirá um alerta.

```
// Initializa os shaders
    function initShaderProgram(gl, vsSource, fsSource) {
        const vertexShader = loadShader(gl, gl.VERTEX_SHADER, vsSource);
        const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER, fsSource);

        // Cria os shaders
        const shaderProgram = gl.createProgram();
        gl.attachShader(shaderProgram, vertexShader);
        gl.attachShader(shaderProgram, fragmentShader);
        gl.linkProgram(shaderProgram);

        // Se a criação dos shaders falhar, alerte o usuário
        if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
            alert('Erro ao inicializar o shader: ' + gl.getProgramInfoLog(shaderProgram));
            return null;
        }
```

```
        return shaderProgram;
    }


    // cria um shader para o código fonte fornecido.
    function loadShader(gl, type, source) {
        const shader = gl.createShader(type);

        // Enviar o código fonte para o objeto do shader
        gl.shaderSource(shader, source);

        // Compila o shader
        gl.compileShader(shader);

        // Verifica se a compilação funcionou
        if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
            alert('Um erro ocorreu ao compilar o shader: ' + gl.getShaderInfoLog(shader));
            gl.deleteShader(shader);
            return null;
        }

        return shader;
    }
```

A função loadShader() recebe como entrada o contexto WebGL, o tipo de shader e o código-fonte e, em seguida, cria e compila o shader da seguinte maneira:

1. Um novo objeto shader é criado chamando gl.createShader().
2. O código-fonte do shader é enviado ao shader pelo método gl.shaderSource().
3. Depois do shader ter o código-fonte, ele é compilado o código usando gl.compileShader().
4. Para verificar se o shader foi compilado com êxito, o parâmetro do shader gl.COMPILE_STATUS é verificado através do gl.getShaderParameter (), se especificando o shader e o nome do parâmetro que queremos verificar (gl.COMPILE_STATUS). Se for falso, sabemos que o shader falhou ao compilar, então um alerta é exibido com as informações de log obtidas do compilador usando gl.getShaderInfoLog(), se isso acontecer o shader é deletado e se retorna null para indicar uma falha ao carregar o shader.
5. Se o shader foi carregado e compilado com êxito, o shader compilado é retornado adequadamente.

Para executar este código, você pode usar a seguinte chamada dentro da função main():

```
const shaderProgram = initShaderProgram(gl, vsSource, fsSource);
```

Depois de criar os objetos dos shaders será preciso relacionar os locais que o WebGL atribuiu às entradas nos códigos. Neste caso temos um atributo e dois uniforms. Atributos recebem valores dos buffers. Cada iteração do vertex shader recebe o próximo valor do buffer atribuído a ele. Os *uniforms* são semelhantes às variáveis globais do JavaScript. Eles permanecem com o mesmo valor para todas as iterações de um shader. Uma vez que os locais dos atributos e *uniforms* são específicos para um único shader.

Insira o seguinte código na função main():

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

Criando o quadrado

Antes de renderizar o quadrado, precisamos criar alguma estrutura de dados (um buffer) que irá conter as posições dos vértices e então iremos colocar as posições dos vértices nele. Faremos isso usando uma função que chamamos de initBuffers().

```
    function initBuffers(gl) {

        // Cria um buffer para as posições dos vértices do quadrado.
        const positionBuffer = gl.createBuffer();

        // Selecione o positionBuffer para aplicar as operações de buffer.
        gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

        // Posições dos vértices do quadrado.
        const positions = [
            -1.0,  1.0,
            1.0,  1.0,
            -1.0, -1.0,
            1.0, -1.0,
        ];

        // Passe a lista de posições para o WebGL
        gl.bufferData(gl.ARRAY_BUFFER,
                      new Float32Array(positions),
                      gl.STATIC_DRAW);
```

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

Essa rotina começa chamando o método createBuffer() do objeto gl para obter um buffer no qual será possivel armazenar as posições dos vértices. Isso é então vinculado ao contexto chamando o método bindBuffer(). Após isso é criado um array JavaScript contendo a posição de cada vértice do plano quadrado. Isso é então convertido em um array de floats e passado para o método bufferData() do objeto gl para estabelecer as posições dos vértices para o objeto.

Renderizando a cena

Depois que dos shaders estabelecidos, as posições dos vértices do quadrado são colocadas em um buffer apropriado, onde podemos finalmente renderizar a cena. Como não estamos animando nada neste exemplo, a nossa função drawScene() é muito simples.

```
    function drawScene(gl, programInfo, buffers) {
        gl.clearColor(0.0, 0.0, 0.0, 1.0);  // define cor para pintar de preto sem
transparência
        gl.clearDepth(1.0);                 // Limpa o buffer de profundidade
        gl.enable(gl.DEPTH_TEST);           // Liga o teste de profundidade (Z-Buffer)

        // Pinta todo o canvas com a cor padrão (preto)
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

        // Cria uma matriz de perspectiva com um campo de visão de 45 graus,
        // com a proporção de largura/altura correspondente ao tamanho de
        // exibição da tela, com objetos visiveis entre 0.1 e 100 unidades
        // de distância da câmera.
        const fieldOfView = 45 * Math.PI / 180;
        const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
        const zNear = 0.1;
        const zFar = 100.0;
        const projectionMatrix = mat4.create();

        mat4.perspective(projectionMatrix,
                         fieldOfView,
                         aspect,
                         zNear,
                         zFar);

        // Define a posição do desenho para a identidade, que é
        // o centro da cena.
        const modelViewMatrix = mat4.create();

        // Move a posição do desenho para onde queremos
```

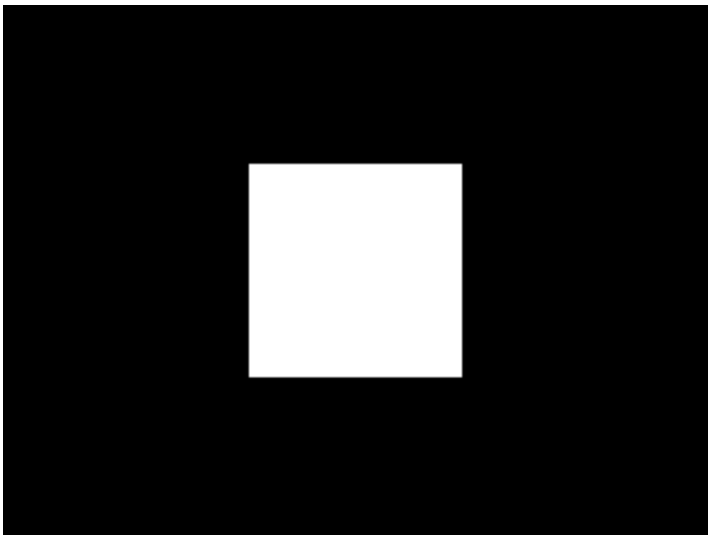
```
        // desenhar o quadrado.
        mat4.translate(modelViewMatrix,     // destino
                       modelViewMatrix,     // matriz para transladar
                       [0.0, 0.0, -6.0]);  // translação

        // Diga ao WebGL como retirar as posições do
        // atributo vertexPosition do buffer
        {
            const numComponents = 2;  // pega 2 valores por iteração
            const type = gl.FLOAT;    // the data in the buffer is 32bit floats
            const normalize = false;  // don't normalize
            const stride = 0;         // quantos bytes para um conjunto de valores
                                      // 0 = use type and numComponents above
            const offset = 0;         // how many bytes inside the buffer to start from
            gl.bindBuffer(gl.ARRAY_BUFFER, buffers.position);
            gl.vertexAttribPointer(
                programInfo.attribLocations.vertexPosition,
                numComponents,
                type,
                normalize,
                stride,
                offset);
            gl.enableVertexAttribArray(
                programInfo.attribLocations.vertexPosition);
        }

        // Diga ao WebGL para usar nosso programa ao desenhar
        gl.useProgram(programInfo.program);

        // Defina os uniforms dos shaders
        gl.uniformMatrix4fv(
            programInfo.uniformLocations.projectionMatrix,
            false,
            projectionMatrix);
        gl.uniformMatrix4fv(
            programInfo.uniformLocations.modelViewMatrix,
            false,
            modelViewMatrix);

        {
            const offset = 0;
            const vertexCount = 4;
            gl.drawArrays(gl.TRIANGLE_STRIP, offset, vertexCount);
        }
    }
```

A primeira etapa agora é limpar a tela com a cor de fundo; então estabelecemos a matriz de perspectiva. Definimos um campo de visão de 45°, com uma proporção entre largura e altura que corresponde às dimensões de exibição de nossa tela. Também especificamos que queremos apenas objetos entre 0,1 e 100 unidades da câmera a serem renderizados.

Em seguida, estabelecemos a posição do quadrado carregando a posição de identidade e afastando-a da câmera em 6 unidades. Depois disso, vinculamos o buffer de vértice do quadrado ao atributo que o shader está usando para a VertexPosition e informamos ao WebGL como extrair os dados dele. Por fim, desenhamos o objeto chamando o método drawArrays(). Para isso inclua na função main():

```
        // Rotina para contruir todos os buffers
        const buffers = initBuffers(gl);

        // Desenhando a cena
        drawScene(gl, programInfo, buffers);
```

E lá está o seu quadrado sendo desenhado:



Referências:

Esse documento foi baseado em:

- WebGL tutorial : https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial
- An Introduction to WebGL: https://dev.opera.com/articles/introduction-to-webgl-part-1/
- WebGL Fundamentals: https://webglfundamentals.org/
- An intro to modern OpenGL: http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Table-of-Contents.html
-