No artigo anterior , refatoramos o código para deixá-lo mais elegante. Porém, os shaders que compõe nosso programa ainda estão implementados direto no código, em variáveis string. Essa abordagem é pouco flexível e muito longe do que desejaríamos para uma aplicação web de verdade. Nesse artigo, veremos como utilizar os Promises, do JavaScript 6, em conjunto com o XMLHttpRequest para requisitar os shaders do servidor de maneira elegante.

XMLHttpRequest

O objeto XMLHttpRequest é responsável por realizar uma requisição ao servidor. Inicialmente, esse objeto foi pensado para a troca de XML - e daí também o nome AJAX, de Asynchronous Javascript And XML - mas, felizmente, o objeto pode ser utilizado para requisitar qualquer tipo de arquivo. Hoje, seu uso mais comum é na requisição de arquivos JSON.

A requisição é muito simples e é dividida em quatro passos:

1. Preparar a requisição;
2. Registrar uma função para lidar com o retorno da requisição, quando ocorrer;
3. Registrar uma função para lidar com outros erros de rede;
4. Enviar a requisição.

Veja um exemplo:

```
//1. Preparar a requisição var req = new XMLHttpRequest(); req.open("GET","exemplo.xml");
//2. Registrar uma função para lidar com o retorno req.onload = function() { if (req.status ==
200) { //Trata a resposta que veio em req.response } else { //Houve algum
problema? alert(Error(req.statusText)); } } //3. Registrar uma função para lidar com
erros de rede req.onerror = function() { alert(Error("Erro de rede!")); } //4. Enviar a
requisição req.start();
```

Dentro do if no corpo da função onload temos certeza que a função retornou com sucesso. A abordagem de registrar funções de *callback* como essas é comum no JavaScript até os dias de hoje. Porém, ela é difícil de gerenciar. Imagine que você precisasse esperar pelo retorno de 2 requisições ao invés de uma só. Além disso, note que essa abordagem não é muito padronizada: somente nesse exemplo, temos 2 formas diferentes de tratamento de erro – uma na resposta e outra num callback próprio para error.

Promises

Mas afinal, o que são Promises ? Promise é a interface que representa qualquer processamento que será executado no futuro, provavelmente de maneira assíncrona. A tradução literal da palavra “Promise” é promessa. Ou seja, ao encapsular uma função num promise você estará prometendo que aquele comportamento, descrito pela função, acontecerá em algum momento que não pode ser exatamente previsto pelo programador.

O promise pode ter quatro estados:

- **pending (pendente):** O estado inicial. Não está nem fulfilled nem rejected;
- **fulfilled (satisfeito):** Uma operação de sucesso. A “promessa” foi cumprida;
- **rejected (rejeitado):** A operação falhou.
- **settled (concluído):** Está fulfilled ou rejected, mas não satisfeito.

Um programador pode associar uma função a um promise, que ele gostaria que fosse executada caso ele seja satisfeito (callback then) ou uma função para caso o promise falhe (callback catch). Essas funções executarão quando a promessa estiver concluída. Observe que isso pode ser no futuro, caso o processamento assíncrono ainda não tenha ocorrido, ou mesmo, imediatamente, caso a promessa já tenha sido cumprida. Esse comportamento é extremamente desejável, pois permite que o programador use o promise se preocupando apenas com a ação que será realizada, não com quando.

Requisitando com Promises

Vamos transformar o XMLHttpRequest num promise? Para isso, iremos criar uma função chamada request, que retorna um promise. A idéia é que possamos utiliza-la da seguinte forma:

```
request("exemplo.xml").then(function(response) {      //Faz algo com a response quando o request terminar  }).catch(function(error)) {    alert(error);  };
```

Para tanto, criaremos um objeto do tipo Promise que aceita como parâmetro uma função. Essa será a função que executará o processamento desejado. Essa função, por sua vez, também recebe dois parâmetros, que são duas outras funções. Uma delas chamaremos resolve e será chamada caso o código dê sucesso. E outra será chamada de reject e será chamada caso o código dê errado.

```
var request = function(url) {  return new Promise(function(resolve, reject) {  var req = new XMLHttpRequest();  req.open('GET', url);  req.onload = function() {  if (req.status == 200) {  //Se foi ok, retorna ao resolve o resultado  resolve(req.response);  } else {  //Caso contrário, chama o reject com o erro  reject(Error(req.statusText));  }  };  //Também usar o reject no caso de existirem erros de rede  req.onerror = function() {  reject(Error("Network Error"));  };  //Faz a requisição  req.send();  });  };
```

Observe que agora o tratamento do erro fica uniformizado. Como em todos os casos de erro o reject é chamado, esse erro sempre será encaminhado a função registrada no callback catch.

Encadeando Promises e requisitando shaders

O grande poder dos Promises, entretanto, está no fato de que eles podem ser encadeados, formando promessas ainda maiores. Considere:

```
request("exemplo.xml").then(function(response) {  //Faz qualquer coisa  })
```

O que o método then retorna? A resposta é **outro promise**! Esse novo promise representa todo o processamento, tanto do request, quanto do then. Confuso? Vamos para o exemplo prático que nos interessa. Vamos criar um método que requisita um shader. Para isso, precisamos:

1. Fazer a requisição do shader ao servidor;
2. Compilar o shader recebido de acordo com o seu tipo. Testaremos o tipo verificando se a url termina com “vs” (de vertex shader) ou “fs” (de fragment shader);
3. Retornar o resultado ou rejeitar em caso de erros.

Esse método poderia ser escrito assim:

```
glc.requestShader = function(gl, url) {  return request(url).then(function(str) {  return new Promise(function(resolve, reject) {  var type;  if (endsWith(url, "vs")) {  type = gl.VERTEX_SHADER;  } else if (endsWith(url, "fs")) {  type = gl.FRAGMENT_SHADER;  } else {  reject(Error("Invalid shader type!"));  return;  }  var shader = gl.createShader(type);  //Compila o shader  gl.shaderSource(shader, str);  gl.compileShader(shader);  //Testa se houve erro  if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
```

```
        resolve(shader);                        } else {
reject(Error(gl.getShaderInfoLog(shader)));             }         });      }); }
```

Observe que na função acima, estamos retornando o resultado da função then. E esse resultado é outro promise! Esse promise fará tanto o request ao servidor, quanto o processamento do shader. Note que para o programador, usuário da biblioteca glCommons, que utiliza a função
requestShader
, o fato de serem um ou dois promises encadeados também é completamente irrelevante.

No fundo, só o que ele quer fazer é:

```
   requestShader("basic.vs").then(function(shader) {     //Faz qualquer coisa  })
```

Exatamente como faria com a primeira função request que fizemos. E é exatamente assim que ele utilizará a função!

Finalmente, antes de prosseguir, vou inserir o código da função endsWith, utilizada acima para testar se a url termina com o sufixo “vs” ou “fs”:

```
   var endsWith = function(str, suffix) {      return str.indexOf(suffix, str.length - suffix.length) !==
-1; };
```

Linkando shaders com promises simultâneos

Se você se lembrar da função linkProgram, verá que nela utilizamos não um shader, mas dois. O vertex e o fragment shader:

```
   glc.linkProgram = function(gl, vs, fs) {      var shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vs);      gl.attachShader(shaderProgram, fs);
gl.linkProgram(shaderProgram);            if (!gl.getProgramParameter(shaderProgram,
gl.LINK_STATUS)) {          alert("Could not link shaders!");             }      return shaderProgram;  }
```

Como então utilizaremos os Promises para fazer duas requisições ao invés de uma?

Como esse tipo de situação é muito comum, os criadores do Promise incluíram a função utilitária all, que recebe como parâmetro um array de promises. Essa função retorna outro Promise, representando o fim da execução de todos os promises passados por parâmetro. A esse promisse, podemos associar um método then, que receberá como entrada um array com

todas as respostas, também na mesma ordem dos promises de entrada. Confuso? Veja isso em código:

```
Promise.all([promise0, promise1]).then( function(respostas) { //Faz algo com resposta[0], resposta[1] } );
```

Podemos utilizar esse método para criar uma função chamada requestProgram, que requisitará o vertex e o fragment shader e tentará compilá-los juntos.

A função é descrita da seguinte forma:

```
glc.requestProgram = function(gl, vsUrl, fsUrl) { if (!fsUrl) { fsUrl = vsUrl + ".fs"; vsUrl = vsUrl + ".vs"; } //Carrega os shaders do servidor var vs = glc.requestShader(gl, vsUrl); var fs = glc.requestShader(gl, fsUrl); //Aguarda o carregamento e compila o programa return Promise.all([vs, fs]).then(function(shaders) { return new Promise(function(resolve, reject) { var shaderProgram = gl.createProgram(); gl.attachShader(shaderProgram, vs); gl.attachShader(shaderProgram, fs); gl.linkProgram(shaderProgram); if (gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) { resolve(shaderProgram); } else { reject(Error("Could not link shaders!")); } }); }); };
```

Note que, por comodidade, deixamos o parâmetro da url do fragment shader opcional. Assim, a função pode ser usado para requisitar shaders com nome diferente, como por exemplo request Shader(“basic.vs”, “phong.fs”)
ou, como geralmente é o caso, pares de shaders similares, como por exemplo requestShader(“gourad”)
– carregando os arquivos gourad.vs e gourad.fs.

Eliminando duplicidadades do código

Você deve ter notado que ao implementarmos as funções requestShader e requestProgram duplicamos muito código das funções
loadShader
e
linkProgram
. Se você é um programador purista, deve ter ficado incomodado com esse fato. Há duas formas de resolvê-lo:

1. Eliminar as funções loadShader e linkProgram, já que as versões que criamos as tornam obsoletas;
2. Ajustar o código das funções loadShader e linkProgram para lançar exceções ao invés

de ter alerts, e as usarmos no interior da função loadShader requestProgram;

A alternativa 2 é possível, pois o código no interior de um then não precisa criar outro promise explicitamente, como fizemos acima. Caso a função no interior do
then
retorne normalmente, esse valor automaticamente será encaminhado para o
resolve
. Caso esse código lance exceção, ela será capturada e a encaminhada automaticamente para o
reject
.

Assim, vamos usar a alternativa 2, reescrevendo nossas funções para que lancem exceções, ao invés de dar erro usando alert:

```
    glc.loadShader = function(gl, type, code) {       //Cria o id para o shader de acordo com o tipo
   var shader = gl.createShader(type);             //Compila o shader       gl.shaderSource(shader, code);       gl.compileShader(shader);             //Testa se houve erro       if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {           throw new Error(gl.getShaderInfoLog(shader));               }            return shader;  }
```

E:

```
    glc.linkProgram = function(gl, vs, fs) {       var shaderProgram = gl.createProgram();       gl.attachShader(shaderProgram, vs);       gl.attachShader(shaderProgram, fs);       gl.linkProgram(shaderProgram);            if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {           throw new Error("Could not link shaders!");               }       return shaderProgram;  }
```

E então poderemos reescrever nossas funções de request como:

```
    glc.requestShader = function(gl, url) {       return request(url).then(function(code) {           var type;           if (endsWith(url, "vs")) {               type = gl.VERTEX_SHADER;                        } else if (endsWith(url, "fs")) {               type = gl.FRAGMENT_SHADER;          } else {               throw new Error("Invalid shader type!");                        }                   return glc.loadShader(gl, type, code);       });  }
```

E:

```
    glc.requestProgram = function(gl, vsUrl, fsUrl) {       if (!fsUrl) {           fsUrl = vsUrl + ".fs";       vsUrl = vsUrl + ".vs";       }       //Carrega os shaders do servidor       var vs = glc.requestShader(gl,
```

vsUrl); var fs = glc.requestShader(gl, fsUrl); //Aguarda o carregamento e compila o programa return Promise.all([vs, fs]).then(function(shaders) { return glc.linkProgram(gl, shaders[0], shaders[1]); }); };

Ajustando o código da cena

Para ajustar o código da cena, iremos criar a seguinte estrutura de pastas:

/ |- index.html +- js | |- exemplo.js | |- glcommons.js | |- gl-matrix-min.js +- shaders |- basic.vs |- basic.fs

Os arquivos basic.vs e basic.fs conterão os códigos do vertex e do fragment shader, que hoje encontram-se dentro de strings:

basic.vs

attribute vec3 aVertexPosition; attribute vec4 aVertexColor; uniform mat4 uModel; uniform mat4 uView; uniform mat4 uProjection; varying vec4 vColor; void main(void) { gl_Position = uProjection * uView * uModel * vec4(aVertexPosition, 1.0); vColor = aVertexColor; }

basic.fs

precision mediump float; varying vec4 vColor; void main(void) { gl_FragColor = vColor; }

Em seguida, iremos alterar a função initShaders para se chamar initProgram, e receber o programa já compilado como parâmetro:

function initProgram(program) { shaderProgram = program; gl.useProgram(shaderProgram); shaderProgram.aVertexPosition = gl.getAttribLocation(shaderProgram, "aVertexPosition"); gl.enableVertexAttribArray(shaderProgram.aVertexPosition); shaderProgram.aVertexColor = gl.getAttribLocation(shaderProgram, "aVertexColor"); gl.enableVertexAttribArray(shaderProgram.aVertexColor); shaderProgram.projection = gl.getUniformLocation(shaderProgram, "uProjection"); shaderProgram.view = gl.getUniformLocation(shaderProgram, "uView"); shaderProgram.model = gl.getUniformLocation(shaderProgram, "uModel"); }

Finalmente, alteraremos a função start para chamar initProgram e draw somente após o programa ser carregado:

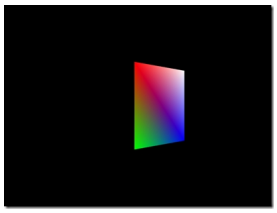
scene.start = function() { gl = glc.createContext("gameCanvas"); //Obtemos o contexto e inicializamos o viewport gl.viewport(0, 0, gl.width, gl.height); //Definimos a

cor de limpeza da tela gl.clearColor(0.0, 0.0, 0.0, 1.0); //Carrega a malha na memória initBuffers(); //Aguarda os shaders e desenha a cena glc.requestProgram(gl, "shaders/basic").then(function(program) { //Inicializa o programa carregado initProgram(program); //Desenhamos a cena drawScene(); }).catch(function(error) { alert(error); }); }

E a partir de agora, estamos carregando shaders do servidor! :)

Concluindo

Obviamente, teremos o mesmo resultado (novamente):



Sei que esse artigo foi longo e complexo. Mas note que mostrei aqui como escrever seus próprios promises e usá-los. Entretanto, para quem usa a biblioteca glCommons, o uso da função linkProgram ficou para lá de trivial. A pessoa não se preocupou em quantos requests assíncronos ou sincronizações foram feitas. Ele só precisou saber que a ***promessa final***, de um shader completamente carregado e linkado será cumprida (ou que o catch será chamado, caso um erro em qualquer etapa acontecer). O fato é que raramente escreveremos nossos próprios promises como fizemos aqui, mas certamente, passaremos a usar várias APIs que os implementam, tornando o código assíncrono bastante fácil.

O fato de agora fazermos requests também tornará obrigatório o uso de um servidor web. No exemplo, adicionei o arquivo moongoose.exe na raíz do projeto. Trata-se de um servidor web simples. Basta dar dois cliques e ele executará rodando o index.html. Caso você esteja usando Mac, você pode inicializar um servidor http através do console. Basta ir até a raiz do seu projeto e digitar:

```
python -m SimpleHTTPServer 8080
```

Como sempre, o código fonte completo do arquivo está disponível para download clicando na

figura abaixo. Recomendo que estude-o cuidadosamente, relendo os trechos do artigo quando necessário para entender cada passo.

Caso queira ler mais um pouco sobre Promises, recomendo a leitura do artigo JavaScript Promises: There and Back Again, do site HTML5 Rocks. Se você se impressionou com os Promises, vai ficar ainda mais animado ao saber que no JavaScript 7 há uma novidade que vai deixar a sintaxe mais fácil e leve, os modificadores async e await. Leia sobre isso no artigo ES7 Async Functions.

No próximo artigo, veremos como incluir um gameloop, animando o quadrado.

Até lá!