No artigo passado , criamos um código capaz de desenhar um quadrado colorido na tela. Apesar de funcional, o código carece de boas práticas. Há trechos de lógica repetida e há a mistura de trechos que lidam com a OpenGL com trechos que lidam com o desenho em si. Nesse artigo, iremos começar uma série de refatorações para tratar desses problemas.

Esse artigo é dividido em duas partes. Na primeira, damos uma breve introdução sobre o funcionamento do conceito de módulos em JavaScript, revisando conceitos como escopo, escopo léxico, funções como valor e objetos dinâmicos. Caso você já saiba desses conceitos, pode pular essa parte e ir direto para o tópico “Refatorando o código do artigo anterior”.

Módulos em JavaScript

Em JavaScript, não existe de maneira formal módulos ou pacotes. Entretanto, a linguagem oferece uma série de mecanismos flexíveis, que permitem ao programador simular esse comportamento.

Para entendermos como modularizar um programa em JavaScript, é importante que entendamos alguns conceitos da linguagem:

- Funções como dados de primeira ordem
- Como o JavaScript lida com escopos
- Como funcionam objetos em JavaScript

Em JavaScript, os módulos combinam esses três conceitos.

Funções como dados de primeira ordem

Em JavaScript todas as funções são anônimas. Você pode estar estranhando essa informação, já que no artigo passado criamos várias funções, e as chamamos pelo nome mas, diferentemente do que ocorre em C++ ou em Java, a função em si é apenas um valor, armazenado numa variável. O nome que utilizamos é o nome dessa variável, e não da função. Assim, do ponto de vista semântico, essas duas declarações são equivalentes:

```
function soma(a, b) {      return a + b;  }    var soma = function(a, b) {      return a + b;  }
```

O fato de funções se comportarem como dados, permite-nos fazer diversas coisas, tais como renomear funções:

```
var seno = Math.sin;  alert(seno(1.12));
```

Passar funções como parâmetro:

```
function comGraus(trigFunc, graus) {      var rad = graus * 180 / Math.PI;      return trigFunc(rad);  }    alert(comGraus(Math.sin, 30)));
```

Ou mesmo, retornar funções:

```
function soma(a, b) {     return a + b;  }    function subtracao(a, b) {     return a - b;  }    function getOperacao(nome) {     if (nome === "mais") {         return soma;     } else if (nome === "menos") {         return subtracao;     }     return undefined;  }    var func = getOperacao("mais");  alert(func(25, 10)); //Imprime 35  func = getOperacao("menos");  alert(func(25, 10)); //Imprime 15
```

Escopos em javascript

Em Javascript, apenas a funções são capazes de delimitar escopos. Isso é diferente do Java, C++ ou C#, onde quem delimita novos escopos é o uso de chaves. Considere, por exemplo, o código abaixo:

```
function exemplo(a, b) {      if (a
```

Este código, mesmo em “strict mode” é válido. Apesar de criada dentro do if, a variável continua existindo no momento do return. Na verdade, o escopo de x é até maior do que isso. Como quem delimita é a função, o JavaScript moverá a definição da variável para o início do escopo. Ou seja, esse código se comporta como se tivesse sido escrito assim:

```
function exemplo(a, b) {      var x;      if (a
```

Como, então, isolamos o escopo do if? Em Javascript, podemos criar **funções internas**. Ou seja, funções dentro de funções, com escopo próprio. Por exemplo:

```
function exemplo(a, b) {      if (a
```

O javascript também implementa um conceito chamado **escopo léxico**. Isso significa que, ao criar uma função interna, todas as variáveis que estão visíveis no escopo onde a função foi criada, serão também visíveis no interior da função. No exemplo anterior, não precisaríamos passar a e b como parâmetro para imprimirSoma. Como a e b existem no escopo onde imprimirSoma foi criado, essas variáveis existem dentro da função imprimirSoma automaticamente. Veja:

```
function exemplo(a, b) {      if (a
```

De fato, podemos até deixar anônima a função de imprimir soma, e chama-la logo após usa criação:

```
function exemplo(a, b) {      if (a
```

Esse tipo de bloco de código ficou tão comum em JavaScript que ganhou o nome de IFFE (Immediately-invoked function expression, numa tradução direta, expressão funcional imediatamente invocada). Alguns

programadores usam para delimitar escopos pequenos, como o do if do exemplo. Outros, preferem usa-la em escopos maiores e ter mais critério ao usa-la em escopos menores, para evitar comprometer a legibilidade do código. Como nossos artigos tem o intuito de serem tutoriais e fáceis de ler, seguiremos a segunda alternativa.

Uma característica muito poderosa do escopo léxico é que podemos retornar uma função interna que esteja utilizando variáveis da função onde foi declada. Nesse caso, as variáveis criadas no momento da execução da função principal serão mantidas enquanto for necessário. Veja um exemplo:

```
function criarContador() {  int contador = 0;  return function() {  contador ++;
return contador;  } }  var cont1 = criarContador();  alert(cont1()); //Imprime 1  alert(cont1());
//Imprime 2  var cont2 = criarContador();  alert(cont2()); //Imprime 1  alert(cont1()); //Imprime 3
alert(cont2()); //Imprime 2
```

Observe que a função criarContador retorna uma função anônima interna. Quando essa função é chamada, ela soma 1 a variável contadora declarada em criarContador e, logo em seguida, retorna seu valor. No exemplo, chamamos criarContador duas vezes. Como a variável contador é local, ela foi criada duas vezes, uma para cada chamada. É por isso que os contadores de cont1() e cont2() não se misturam. Tente memorizar esse comportamento, ele será fundamental para criarmos variáveis privadas.

Objetos em javascript

Em JavaScript, objetos são criados de maneira totalmente dinâmica. Não existe formalmente o conceito de classes e, devido a isso, não se pode considerar Javascript exatamente uma linguagem orientada a objetos (é possível simular um comportamento bastante próximo ao da OO tradicional através de um conceito chamado de Prototype).

Para criar um objeto, usa-se a declaração de chaves.

```
var objeto1 = {};  var objeto2 = {  nome : "Vinícius",  idade : 34  };
```

No exemplo acima, criamos dois objetos. Um deles sem conter nenhum atributo, e o segundo, com dois atributos.

Objetos são 100% dinâmicos, e podem receber propriedades e métodos a qualquer momento:

```
var objeto1 = {};  objeto1.nome = "Camila";  objeto1.idade = 31;  objeto1.dizerNome = function() {      alert("Eu sou a " + this.nome);  }
```

Se você voltar aos artigos anteriores, verá que nós já usamos essa característica ao adicionar duas propriedades chamadas itemSize e numItems ao objeto retornado pela função gl.createBuffer() .

Repare que no último exemplo, utilizamos a palavra chave this. Essa palavra chave representa o objeto que é “dono” da função no momento que ela foi invocada. Embora esse conceito muitas vezes represente o objeto onde a função foi declarada, esse nem sempre é o caso pois, em JavaScript, funções podem ser passadas por parâmetro. Para uma explicação detalhada sobre o this, considere a leitura
desse artigo
.

Combinando as duas coisas

Finalmente, poderemos utilizar todos esses conceitos para entender como módulos são implementados em Javascript! Definimos um módulo através de:

- A criação de um objeto, que conterá o “nome” do módulo;
- A criação de uma função anônima auto-invocada, para poder conter o escopo do módulo e suas variáveis “privadas”;

Por exemplo, iremos criar o módulo glcommons, que conterá as funções que comumente utilizamos em todas as aplicações WebGL. Iremos usar o objeto glc.

```
//Declaração do “nome” do módulo  var glc = {} || glc;    (function() {      //Corpo do módulo })();
```

A sintaxe de declaração com o operador de OR funciona da seguinte forma. Se já existir uma variável glc declarada, o JavaScript irá utiliza-la. Caso não exista, ele irá declarar um objeto em branco. Isso permite que um módulo seja dividido em vários arquivos, caso necessário.

Podemos incluir funções e variáveis na área do corpo do módulo. Elas serão consideradas privadas, a menos que sejam inseridas como atributos do objeto glc. O exemplo abaixo mostra 2 funções, uma privada e uma publica:

```
//Declaração do “nome” do módulo var glc = {} || glc; (function() { var privada = function() { alert("privada!"); } glc.publica = function() { alert("publica!"); } })();
```

Note que graças ao escopo lexico, o objeto glc, externo, será visível. Da mesma forma, todas as variáveis internas à função anônima serão mantidas, como esperaríamos de variáveis privadas no módulo.

Refatorando o código do artigo anterior

Vamos iniciar criando o arquivo glcommons.js e nele, incluiremos o protótipo do módulo vazio, descrito anteriormente:

```
var glc = {} || glc; (function() { })();
```

No código do exemplo anterior, vemos dois tipos de construção:

- Funções que utilitárias e lidam com a WebGL em si: createContext, loadShader, linkProgram e toRadians. Essas são as funções ideais para a biblioteca glCommons.
- Funções que realmente estão preocupadas em organizar o desenho e pintar essa cena específica: startWebGL, initBuffers e drawScene. Essas funções permanacem no arquivo exemplo.js, pois ele representará a cena.

Para separá-las, iremos partir do pressuposto que quem controla o contexto gráfico (variável gl) é cliente, e não a biblioteca glcommons. Esse seria o caso em engines mais avançadas como a Unity ou a Three.js mas, aqui, estamos só interessados numa lib tênue que nos permite manter o estudo organizado. Por isso, não iremos implementar todo o código necessário para encapsular esse contexto.

Dessa forma, funções que dependem da variável gl devem recebê-la como parâmetro. Veja por exemplo, a migração da função loadShader:

```
var glc = {} || glc;  (function() {  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)) {  alert(gl.getShaderInfoLog(shader));  return null;  }  return shader;  }  })();
```

Outro ponto que merece nossa atenção está na criação dos buffers. Há ali um código repetido, que também é mais relacionado a WebGL do que ao conteúdo do buffer em si. Veja:

```
//Criação do buffer de cores  var vertexColors = [  1.00, 0.00, 0.00, 1.0,  1.00, 1.00, 1.00, 1.0,  0.00, 1.00, 0.00, 1.0,  0.00, 0.00, 1.00, 1.0,  ];  colors = gl.createBuffer();  gl.bindBuffer(gl.ARRAY_BUFFER, colors);  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexColors), gl.STATIC_DRAW);  colors.itemSize = 4;  colors.numItems = 4;
```

Podemos eliminar essa duplicação criando na biblioteca glc uma função chamada createBuffer, escrita da seguinte maneira:

```
glc.createBuffer = function(gl, type, itemSize, data) {  var buffer = gl.createBuffer();  gl.bindBuffer(type, buffer);  gl.bufferData(type, type == gl.ARRAY_BUFFER ? new Float32Array(data) : new Uint16Array(data), gl.STATIC_DRAW);  buffer.itemSize = itemSize;  buffer.numItems = data.length / itemSize;  return buffer;  };
```

Isso permite simplificar o código anterior para:

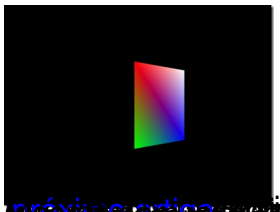
```
//Criação do buffer de cores  var vertexColors = [  1.00, 0.00, 0.00, 1.0,  1.00, 1.00, 1.00, 1.0,  0.00, 1.00, 0.00, 1.0,  0.00, 0.00, 1.00, 1.0,  ];  colors = glc.createBuffer(gl, gl.ARRAY_BUFFER, 4, vertexColors);
```

Por fim, resta apenas criarmos também um módulo para nossa cena. Iremos criar um módulo chamado scene, e deixar apenas o método startWebGL público. O método será renomeado para start(), por questões de breviedade.

Pronto! Nossa refatoração está completa!

Concluindo

O resultado final, evidentemente, é exatamente o mesmo que tínhamos antes:



Download

Para ver o código final refatorado, clique no link de download abaixo. Certifique-se de estudá-lo ao rever os tópicos desse artigo.