Atividades de Laboratório para a Disciplina de Computação Gráfica

LAB1 – Data de Entrega: 23 de agosto de 2017

Esta lista está dividida em duas partes: “Aquecimento” e “Avaliação”. As atividades de aquecimento são para o aluno se situar em relação aos problemas de programação a serem abordados. Por exemplo, para o aluno lidar com configuração de ambiente e entender a estrutura dos programas. Já na parte de avaliação, o aluno resolverá problemas que valem pontos na primeira avaliação parcial.

Aquecimento

Para todas as atividades de aquecimento, quando me referir à “código de exemplo ##”, estou me referindo ao código disponibilizado no diretório acessível pela URL https://drive.google.com/drive/folders/0B42tQIsUw_0uRTQ5aTcyNWVXYTQ?usp=sharing no diretório nomeado ##, onde ## é um valor numérico como, por exemplo, 00, 01, 02, e assim por diante.

01) Em relação ao código de exemplo 01, faça:

a) altere a cor de fundo do *canvas*. Especificamente, você terá que utilizar (ou alterar) o comando *glClearColor* no arquivo de código *gasket1.js;*

b) altere a cor dos objetos desenhados no *canvas*, no caso, do *triângulo de sierpinski*. Para isso, altere o *fragment shader* no arquivo *gasket1.html*. Talvez você tenha que alterar a linha 24 deste arquivo.

02) Veja o *vertex shader* no código de exemplo 02, no arquivo *gasket1.html*, entre as linhas 8 e 16. Na linha 8, declaramos uma variável nomeada de *vPosition*. Neste caso, declaramos o tipo desta variável como *vec4*, que em GLSL significa um vetor numérico com quatro elementos. Nesta declaração, também foi colocado um modificador para esta variável, o modificador *attribute*. Este modificador diz ao *WebGL* que a variável com a qual está associada é um atributo que pode receber valores do código de aplicação. Na prática, para cada vértice a ser processado, uma instância do *script* do tipo *vertex shader* será executada; e a variável *vPosition* assumirá o valor de um vértice enviado pelo código *JavaScript* quando um comando *gl.drawArray* ou *gl.drawElements* é executado. Veja o código do *vertex shader* no arquivo gasket1.html e altere a propriedade *gl_PointSize* para 2 e depois teste o programa. Observe as diferenças para a versão com gl_PointSize igual a 1. Agora altere o valor *gl_PointSize* para 10 e observe as diferenças.

03) Imagine que precisamos gerar uma pequena perturbação nas coordenadas de cada vértice original gerado pelo código da aplicação. Uma pequena perturbação, a grosso modo, é uma pequena alteração nos dados de entrada. Por exemplo, podemos somar às coordenadas do vértice de entrada um pequeno valor gerado aleatoriamente. Podemos enviar um valor aleatório do código de nossa aplicação. Uma forma de fazermos isso é gerando outro *array* numérico a ser enviado para um atributo a ser

processado no *vertex shader*. O exemplo de código 03 mostra este caso. Observe no arquivo *gasket1.html*, temos o seguinte *vertex shader*:

```
7  <script id="vertex-shader" type="x-shader/x-vertex">
8  attribute vec4 vPosition;
9  attribute vec4 noise;
10
11 void
12 main()
13 {
14     gl_PointSize = 1.0;
15     gl_Position = vPosition + noise;
16 }
17 </script>
```

Na linha 9 deste exemplo, declaramos uma variável *noise*. Esta variável foi marcada como *attribute*. Isso significa que guardará valores referentes a um vértice, neste caso, guardará dois valores que serão somados ao valor da variável *vPosition*. Já no arquivo *gasket.js*, adicionamos o array noise e o alimentamos como mostra o seguinte código:

```
37      var noise = [ vec2(Math.random(), Math.random()) ]
38
39      // Compute new points
40      // Each new point is located midway between
41      // last point and a randomly chosen vertex
42
43      for ( var i = 0; points.length < NumPoints; ++i ) {
44          var j = Math.floor(Math.random() * 3);
45          p = add( points[i], vertices[j] );
46          p = scale( 0.5, p );
47          points.push( p );
48          noise.push(vec2(Math.random(), Math.random()))
49      }
```

A modificações estão nas linhas 37 e 48 deste código. Já no seguinte pedaço de código, associamos as variáveis do tipo atributo em nosso *vertex shader* com os dados em nosso array. Primeiramente, criamos um buffer em WebGL para depois enviarmos os dados do array para este buffer criado.

```
65      var noiseBufferId = gl.createBuffer();
66      gl.bindBuffer( gl.ARRAY_BUFFER, noiseBufferId );
67      gl.bufferData( gl.ARRAY_BUFFER, flatten(noise), gl.STATIC_DRAW );
68      // Associate out shader variables with our data buffer
```

Mas ainda precisamos dizer que o *buffer* indicado por *noiseBufferId* está associado com o atributo *noise* em nosso *vertexshader*, o que é feito do seguinte modo, ainda com o buffer com ID *noiseBufferId* ligado:

```
var noiseAttrID = gl.getAttribLocation( program, "vPosition" ); //aqui pegamos o ID do atributo
gl.vertexAttribPointer( noiseAttrID, 2, gl.FLOAT, false, 0, 0 ); //aqui dizemos como lê os dados do
//buffer e coloca-los no atributo
gl.enableVertexAttribArray( noiseAttrID ); //aqui dizemos que o pipeline tem que efetivamente carregar
//este atributo
```

Observe que *Math.random* gera valores aleatórios uniformemente distribuídos no intervalo [0, 1). Altere este programa para que os valores gerados fiquem no intervalo [0, 0.2].

4) Uma forma de transferir dados entre o *vertex shader* e o *fragment shader* é por meio de variáveis marcadas com ***varying***. Uma variável marcada como ***varying*** no *vertex shader* deve ter uma declaração equivalente, do mesmo tipo e do mesmo nome, no *fragement shader*. O exemplo de código 04 associa a cada vértice uma variável de cor que é definida do seguinte modo: a coordenada x do vértice define o valor da cor vermelha e a coordenada y define a cor verde. A cor azul é sempre definida como 1.0, como mostra o seguinte código.

```
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
varying vec4 color;
void
main()
{
    color = vec4(0.0, 1.0, 1.0, 1.0);
    color[0] = vPosition[0];
    color[1] = vPosition[1];
    gl_PointSize = 1.0;
    gl_Position = vPosition;
}
</script>
```

O erro neste código é que as coordenadas podem assumir valores no intervalo [-1, 1], enquanto os valores de cor devem permanecer no intervalo [0, 1]. Faça as correções necessárias para que os valores de cor permaneçam no intervalo correto. Além disso, tente entender o *fragment shader* em *gasket1.html*, que foi definido do seguinte modo:

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
varying vec4 color;
void
main()
{
    gl_FragColor = color;
}
</script>
```

Notas de Aula

Nesta nota, tome como referência o código de exemplo 01. Quando realizamos o comando *drawArrays*, estamos dizendo que a pipeline do WebGL processe todos os buffers de *array* declarados e os envie para serem tratados pelos *shaders* de vértice. Assim, quando executamos o comando *gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0)*, estamos dizendo que, ao transferir dados do buffer ativo atual para o atributo *vPosition*, queremos os valores devem ser lidos do *buffer* ativo aos pares; dessa forma, para cada dois valores lidos, o atributo *vPosition* armazene esses dois valores em suas duas primeiras posições. Observe *vPosition* é um *vec4*, portanto, tem quatro valores reais. O valor padrão de um vec4 é (0.0, 0.0, 0.0, 1.0).

O comando g*l.vertexAttribPointer* foi executado depois de *gl.bindBuffer(gl.ARRAY_BUFFER, bufferId)*. O comando *gl.bindBuffer* serve justamente para dizermos qual buffer queremos ativar. Neste caso, ativamos o buffer identificado por *bufferId*, ou seja, o buffer onde armazenamos os dados da variável points.

Dada essa retrospectiva, a sequência de passos necessários para enviarmos dados do *JavaScript* para serem processamos por nossos *shaders* é a seguinte:

1) Gerar os dados em JavaScript e armazená-los em um *array* de números

2) Converter o *array* de números em um *array* que o WebGL entenda (feito, neste exemplo, com o comando *flatten* – abra o aquivo MV.js, na linha 664, e veja o que este método faz).

3) Criar um *buffer* no *framebuffer* onde os valores do *array* de números convertido possam ser armazenados. Para isso, utilize o seguinte comando: *gl.createBuffer*. Este comando retorna o identificador (ID) do buffer criado; é por meio do ID que ativamos o buffer e enviamos dados para ele.

4) Ativamos o *buffer*, agora dizemos que este buffer é o buffer ativo do momento. Quando um *buffer* está ativo, podemos enviar dados para ele ou associá-lo com algum atributo (declaro no *vertex shader*). Ativamos um *buffer* por meio do comando *gl.bindBuffer* passando como argumento o tipo de *buffer* (no caso, ARRAY_BUFFER) e o ID do *buffer* a ser ativado.

5) Com o buffer ativo, o alimentamos com dados; enviamos dados para um *buffer* por meio do comando *gl.bufferData*, em que especificamos o tipo de *buffer* (ARRAY_BUFFER ou ELEMENT_ARRAY_BUFFER), os dados em si e como estes dados podem ser tratados (por exemplo, gl.STATIC_DRAW). Observe que não precisamos informar o ID do *buffer* para o qual enviamos os dados, pois os dados serão enviados apenas para o buffer ativo no momento.

6) Finalmente, precisamos associar o *buffer* com algum atributo declarado no *vertex shader*. Para isso, precisamos obter o ID do atributo ao qual queremos associar um buffer. O que pode ser feito com o comando *gl.getAttribLocation(nome_atributo)*, onde *nome_atributo* é o nome de uma variável marcada como attribute em nosso código *vertex shader.*

7) Com o ID do atributo, podemos associá-lo ao buffer ativo no momento, isso pode ser feito com o comando *gl.vertexAttribPointer (pesquise em https://developer.mozilla.org/en-*

US/docs/Web/API/WebGLRenderingContext/vertexAttribPointer) para ver como este comando funciona.

8) Então ativamos o atributo, de modo que a pipeline do WebGL associe um endereço específico ao atributo genético declarado no *vertex shader*. Isso é feito com o comando *gl.enableVertexAttribArray(…)*.

9) Finalmente, inicie o processo de renderização com o comando *gl.drawArrays* ou *gl.drawElements*. Para saber mais sobre estes métodos, visite: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArrays.xhtml e https://www.khronos.org/registry/OpenGL-Refpages/es2.0/xhtml/glDrawElements.xml.

Avaliação

1) Altere o código de exemplo 01 de modo que um único triângulo seja desenhado e que os vértices deste triângulo sejam coloridos com as cores vermelho, verde e azul.

2) Altere o código de exemplo 01 de modo que seja desenhado um pixel por vez na tela toda vez que o usuário clicar com o mouse. O pixel deve ser colocado na posição em que o usuário clicou.

3) Um sistema de gráficos de tartaruga é um sistema de posicionamento alternativo que é baseado no conceito de uma tartaruga se movendo em uma tela com uma caneta amarrada por embaixo sua casca. A posição da tartaruga é definida por uma tripla (x, y, θ), dando a localização do centro e a orientação da tartaruga. Um sistema típico desses inclui as seguintes funções:

```
init(x,y,theta); // initialize position and orientation of turtle
forward(distance);
right(angle);
left(angle);
pen(up_down);
```

Implemente estas funções de tartaruga utilizando WebGL (2.0). As funções devem funcionar o seguinte modo:

init(x, y, theta): define a posição inicial da tartaruga na tela e limpa a tela.

forward(distance): movimenta a tartaruga *distance* unidades na direção em que se encontra. Assim, *distance* deve ser um número real. Se a caneta estiver levantada (*up*), um linha é desenhada da posição anterior à nova posição da tartaruga.

***right(angle)*:** gira a tartaruga *angle* graus no sentido anti-horário.

left(angle): gira a tartaruga *angle* graus no sentido horário.

pen(up_down): se *up_down* for *true*, a canela passar ao estado baixada (*down*); caso contrário, fica no estado levantada (*up*).

Tabela de pontuações

Questão 01	Questão 02	Questão 03
0.5	0.5	2.0

O trabalho deve ser enviado com um relatório explicando como cada implementação foi feita. O relatório é simplesmente um texto estruturado de forma livre explicando como o usuário combinou os comandos básicos usados em WebGL para atingir o objetivo de cada atividade. O arquivo de texto deve ser convertido em pdf e compactado junto com o código de cada atividade. O arquivo compactado deve ser enviado para o email gilzamir@gmail.com até à data máxima estabelecida para entrega destas atividades.