Um dos principais pontos para um game é a presença de um game loop, que torne as cenas dinâmicas e permita interação com o jogador. Neste artigo, veremos como organizar esse game loop num navegador, e como receber entradas do teclado.

RequestAnimationFrame

Num navegador, não podemos criar nós mesmos um while para conter nosso game loop. Se fizermos isso, isso irá travar a atualização de tela e o navegador avisará que nosso script travou. Isso ocorre porque o próprio navegador possui um ciclo de pintura e nosso javascript já roda dentro desse ciclo.

Para contornar esse problema os programadores utilizavam um objeto de timer do JavaScript. Porém, esse objeto se mostrou impreciso, causando loops instáveis e consumindo processamento mesmo quando a aba estivesse fechada.

A solução final chegou com a criação da função requestAnimationFrame . Essa função permite que forneçamos ao navegador uma função, que será chamada assim que o navegador pintar sua tela. O código nativo do navegador se encarrega de controlar o loop de desenho e tentar garantir a taxa de 60 quadros por segundo.

O navegador também enviará o *timestamp* do momento que a primeira função agendada foi chamada. Assim, podemos calcular facilmente o tempo transcorrido entre cada quadro.

Vamos escrever uma função de game loop que, por sua vez, controlará 3 etapas distintas, definidas pelas seguintes funções:

- init(): Que fará inicialização da cena, antes do primeiro passo do game loop ser chamado
- update(secs): Para processamento da lógica do jogo. O parâmetro secs indica a quantidade de segundos transcorridos entre duas chamadas do update.
- draw(): Para o desenho da cena.

Para que o loop funcione corretamente, precisamos também levar em conta os seguintes aspectos:

- A função update não deve ser chamada no primeiro loop, pois nesse memonto o valor de secs
não existiria;

- Iremos incluir um parâmetro chamado maxUpdateTime que indica qual é o tempo máximo entre dois quadros que pode ter transcorrido para que o update seja chamado. Assim, se o usuário trocar o navegador de aba e retornar a nossa aplicação, o código não entrará no update contendo vários minutos, levando a uma lógica de game inconsistente. Esse parâmetro será opcional e, por padrão, 0.1s

A versão final de nossa função será:

```
glc.animate = function(init, update, draw, maxUpdateTime) {
var start = null;
maxUpdateTime = maxUpdateTime || 0.1;
var updateScene = function(timestamp) {
if (start == null) {
start = timestamp;
} else {
var secs = (timestamp - start) / 1000.0;
start = timestamp;
if (secs <= maxUpdateTime) {
update(secs);
}
}
draw();
requestAnimationFrame(updateScene);
};
init();
requestAnimationFrame(updateScene);
};
```

Observe que a função animate, ao ser chamada irá apenas inicializar a variável start e maxUpdateTime
, chamar a função
init()
e agendar a função interna
updateScene
para ser chamada a primeira vez.

Quando a função updateScene for chamada a primeira vez, ela gravará o timestamp dessa chamada e então chamará a função draw() para desenhar a cena. Em seguida, ela se reagenda, para que seja chamada novamente.

A partir da segunda chamada de updateScene, já poderemos calcular o tempo transcorrido na variável secs. O tempo é convertido para segundos porque, embora seja um pouco mais impreciso, trata-se de uma unidade muito mais fácil de trabalhar. Esse tempo é então

encaminhado para a função update. Em seguida, o draw é chamado e a função torna a ser reagendada, para que o loop ocorra. Note também que, caso o tempo calculado em secs seja superior ao maxUpdateTime, a função update será simplesmente ignorada.

Lendo comandos do teclado

Além da animação, podemos querer ler dados do teclado. O navegador notifica quais teclas foram pressionadas ou liberadas através de listeners, que podem ser registrados na classe Window. Para isso, usamos a função addEventListener que recebe três parâmetros:

- **Tipo do evento**: Indica sobre o que o evento trata. Em nosso caso, queremos escutar os eventos “keyUp” e “keyDown”.
- **Callback**: Função que será disparada quando um evento desse tipo for disparado. A função deve receber como parâmetro um objeto, que conterá detalhes do evento. No caso dos eventos de tecla, a função receberá o código da tecla;
- **Usar captura**: Se verdadeiro, proíbe o evento de ser propagado. Usaremos false. Esse parâmetro é opcional em navegadores mais novos (e, por padrão, seu valor é false).

Como o evento só é disparado uma vez, no momento em que a tecla muda de estado, precisamos armazenar o estado de cada tecla num array. Afinal, temos interesse em saber se a tecla se mantém ou não pressionada. Vamos então ao código do objeto que lida com teclado (baixe os fontes ao final do artigo para uma versão da classe Key com todos os códigos de teclas):

```
var Key = { _pressed: {}, ENTER : 13, SHIFT : 16, ALT : 18, PAUSE : 19, ESCAPE : 27, SPACE : 32, LEFT: 37, UP: 38, RIGHT: 39, DOWN: 40, isDown: function(keyCode) { return this._pressed[keyCode]; }, _onKeydown: function(event) { this._pressed[event.keyCode] = true; }, _onKeyup: function(event) { delete this._pressed[event.keyCode]; } };
```

As funções _onKeydown e _onKeyup devem ser registradas no recebimento dos eventos. Utilirazemos _ na primeira letra do nome para indicar que essas funções não devem ser chamadas diretamente pelo programador. Podemos fazer esse registro ao final da função animate
:

```
glc.animate = function(init, update, draw, maxUpdateTime) { var start = null; maxUpdateTime = maxUpdateTime || 0.1; var updateScene = function(timestamp) { if (start == null) { start = timestamp; } else { var secs = (timestamp - start) / 1000.0; start = timestamp; if (secs <= maxUpdateTime) {
```

update(secs); } } draw(); requestAnimationFrame(updateScene); }; window.addEventListener('keyup', function(event) { Key._onKeyup(event); }); window.addEventListener('keydown', function(event) { Key._onKeydown(event); }); init(); requestAnimationFrame(updateScene); };

Utilizando a animação

Vamos então alterar o código do exemplo para utilizar essas novas estruturas?

O primeiro passo é criar a função init(). Ela vai conter basicamente todo o código que possuíamos na função start:

```
function init() {
    //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();
    glc.requestProgram(gl, "shaders/basic").then(function(program) {
        //Inicializa o programa carregado
        initProgram(program);
    }).catch(function(error) {
        alert(error);
    });
}
```

Observe que o comando drawScene foi retirado de dentro do requestProgram. Isso porque agora, ele será chamado automaticamente pelo gameloop.

A função de start fica responsável só por iniciar o contexto gráfico e a animação:

```
scene.start = function() {
    gl = glc.createContext("gameCanvas");
    glc.animate(init, update, drawScene);
}
```

Agora vamos programar a lógica de nossa animação. Para isso, vamos criar uma variável chamada angle, logo no início do módulo e inicializa-la com 0:

```
var angle = 0;
```

Em seguida, vamos atualizar a função de desenho para desenhar o quadrado baseado nesse ângulo, alterando a forma que criamos a matriz model:

```
function drawScene() {
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    if (!shaderProgram) {
        return;
    }
    //Configura a matriz de projeção
    var projection = mat4.perspective(mat4.create(),
        glc.toRadians(45), //Abertura
        gl.width / gl.height, //Aspecto
        0.1, 100.0 //Near e far
    );
    //Configura a matriz da camera
    var view = mat4.lookAt(mat4.create(),
        vec3.fromValues(0.0, 0.0, 5.0), //Onde está
        vec3.fromValues(0.0, 0.0, 0.0), //Para onde olha
        vec3.fromValues(0.0, 1.0, 0.0) //Onde é "para cima"
    );
    var
```

```
model = mat4.create();                mat4.rotateZ(model, model, glc.toRadians(angle));
gl.uniformMatrix4fv(shaderProgram.model, false, model);                    //Atualiza os valores
do shader        gl.uniformMatrix4fv(shaderProgram.projection, false, projection);
gl.uniformMatrix4fv(shaderProgram.view, false, view);                //Associa os buffers que
serao usados no desenho            gl.bindBuffer(gl.ARRAY_BUFFER, vertices);
gl.vertexAttribPointer(shaderProgram.aVertexPosition, vertices.itemSize, gl.FLOAT, false, 0, 0);
                gl.bindBuffer(gl.ARRAY_BUFFER, colors);
gl.vertexAttribPointer(shaderProgram.aVertexColor, colors.itemSize, gl.FLOAT, false, 0, 0);
            gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indices);                    //Comanda o
desenho        gl.drawElements(gl.TRIANGLES, indices.numItems, gl.UNSIGNED_SHORT, 0);
    }
```

Observe na função drawScene acima, que agora colocamos um if, para testar se o shaderProgram existe. Esse if é importante, pois como a função será disparada no gameLoop, isso pode ocorrer antes do Promise que carrega os shaders retornar. **Sempre teste se seus recursos já estão carregados nas funções que rodam em loop antes de utilizá-los!**

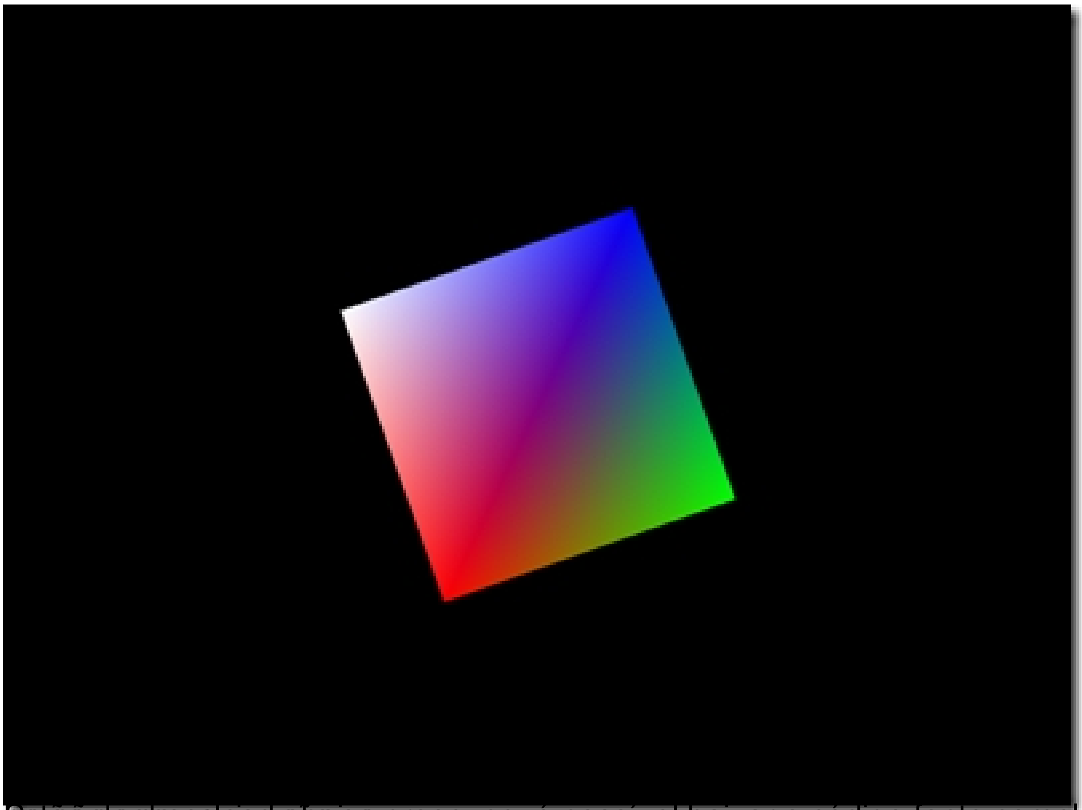
Por fim, basta criar a lógica que atualiza o ângulo na função update. Vamos fazer uma lógica que faz o seguinte gira o quadrado baseado nas setas para esquerda e para direita a uma velocidade angular de 72 graus por segundo (1 volta a cada 5 segundos). Caso o jogador pressione shift, a velocidade se multiplicará por 5 (1 volta por segundo).

```
    function update(secs) {      var speed = 72 * secs;      if (Key.isDown(Key.SHIFT)) {
speed *= 5;      }          if (Key.isDown(Key.LEFT)) {          angle += speed;      } else if
(Key.isDown(Key.RIGHT)) {          angle -= speed;      }    }
```

Caso você esteja confuso sobre o motivo pelo qual multiplicamos a velocidade por secs, leia o artigo Animação baseada em tempo , para maiores explicações.

Concluindo

Se você seguiu os passos desse tutorial corretamente, deve agora ter um quadradinho que gira na tela sempre que as setas são pressionadas!



Se você não conseguiu, não se preocupe, é possível baixar o código fonte complexo clicando no botão de download abaixo:

Ou veja o exemplo ao vivo clicando aqui. No próximo artigo, resolveremos o desafio proposto no artigo "Um quadrado com IndexBuffer" ... possibilidade para diversos efeitos gráficos, que