**As Lições**

“Fui criando uma série de aulas como uma forma de ensinar WebGL para mim mesmo. As dez primeiras lições são baseados nos conhecidos tutoriais *NeHe* de *OpenGL,* mas eu tenho variado nos próximos. Se o Inglês não é sua língua fluente, você pode consultar este material nas seguintes linguagens: ”

- Chinês: [本](http://www.hiwebgl.com/?p=42)课程中文版

- Espanhol:  [español](http://www.jlabstudio.com/webgl/tutoriales-webgl/)

- Japones: [日本語でのレッスン](https://sites.google.com/site/hackthewebgl/learning-webglhon-yaku/the-lessons)

\* traduzido do original.

Foi ai que surgiu a ideia de criar a nossa versão em português destas lições, com intuito de enriquecer o material para iniciar os estudos de WebGL em nossa querida língua portuguesa, afinal de contas... todos nós podemos, certo ? Então vamos lá, boa diversão =D

**Lissão 0 – Os primeiros passos com WebGL**

[Este post foi atualizado várias vezes desde que foi escrito pela primeira vez, pois as coisas mudaram desde que foi escrito originalmente em Outubro de 2009, ao meu conhecimento, as informações estão corretas em 17 de Janeiro de 2012.]

O primeiro passo para experimentar o maravilhoso WebGL é conseguir um navegador ~~descente~~ que o suporte. Isso irá permitir visualizar incríveis demos em WebGL ou desenvolver os seus.

**Mantendo isso simples**

Em geral eu recomento você a observar demos legais de WebGL e não ficar realmente preocupado em desenvolver os seus próprios, e não se importe se algum material mais recente não funcionar, então:

- Se você estiver usando no Windows, tenha certeza que o [DirectX](http://www.microsoft.com/downloads/en/details.aspx?FamilyID=2da43d38-db71-4c1b-bc6a-9b6652cd92a3&displaylang=en) esteja instalado (ele pode ser baixado de graça no site da Microsoft).

- Feito isso, garanta que você tenha as versões mais recentes dos drivers da sua placa gráfica.

- Depois, escolha um navegador:

- Firefox: apenas garanta que esteja na versão 4 ou superior.

- Chrome: tudo que você precisa é [instalar](http://www.google.com/chrome) , ou se já estiver usando ele, certifique-se que esteja atualizado para a versão 10 ou superior – isso provavelmente acontece automaticamente (ele foi lançado em Março de 2011), mas não custa nada confirmar no indo em na opção “Sobre o Google Chrome” na barra de menus.

- Safari: nos Macs, o OS X 10.7 possui suporte para WebGl, mas ele está desativado por padrão. Para ativar, habilite o menu de desenvolvedor e marque a opção “Habilitar WebGL (Enable Web GL)”

E é isso, agora você está pronto para essa aventura. Continuando, [clique aqui para ver algumas páginas de WebGL](#exemplos).

**Fazendo da maneira difícil**

Se você está desenvolvendo seus experimentos em WebGL ou precisa testar mais recursos, então use os navegadores com suas versões mais recentes. WebGL é suportando nas versões de desenvolvimento dos navegadores, exceto Internet Explorer, então tudo que você precisa fazer é obter a versão apropriada para sua máquina. A facilidade disso vai depender do tipo de computador que você possui:

- Windows: se você ainda não fez, verifique se o DirectX esteja instalo em sua máquina - [o download é gratuito no site da Microsoft](http://www.microsoft.com/downloads/en/details.aspx?FamilyID=2da43d38-db71-4c1b-bc6a-9b6652cd92a3&displaylang=en). Feito isso, instale o [Firefox](#install-firefox) ou o [Chromium](#install-chromium-nightly), o que você preferir – se não funcionar, consulte o [guia para solução de problemas](#troubleshooting) .

#Podemos atualizar esse trecho do artigo para o cenário dos navegadores atuais

**Alguns exemplos iniciais**

Com o seu browser instalado, você será capaz de visualizar conteúdos em WebGL. Aqui está um primeiro teste; [WebGL Report](http://webglreport.sourceforge.net/), que mostra detalhes das características do WebGL ativas em seu navegador. Se aparecer uma mensagem dizendo que seu navegador não suporta WebGL, verifique nosso [guia para soluções de problemas](#troubleshooting).

Se estiver configurado corretamente, tudo deve funcionar. Infelizmente com as alterações recentes do WebGL, muitos demos foram quebradas, mas aqui estão algumas que foram atualizados e funcionam bem:

- Do criador deste tutorial original: [Cubos Mandelbrot Pulando](http://learningwebgl.com/lessons/example02/)

- Do Google: [Um bule brilhante](https://www.khronos.org/registry/webgl/sdk/demos/google/shiny-teapot/index.html) (clique e arraste para gira-lo), “[San Angeles](https://www.khronos.org/registry/webgl/sdk/demos/google/san-angeles/index.html)”, e para quem tem uma queda por anatomia: [Body Browser](http://www.zygotebody.com/) (o projeto foi feito pelo Google, e vendido para uma empresa com foco em tecnologia médica, por este motivo se chama agora “Zygote Body”).

- Exemplos que são usados no mundo real feito em WebGL: [ChemDoodle 3D](http://web.chemdoodle.com/demos/molgrabber-3d) apresenta moléculas na página web.

- Aqui existem mais demos no [Khronos demo repository](http://www.khronos.org/webgl/wiki/Demo_Repository) e uma sessão com [contribuições de usuários](http://www.khronos.org/webgl/wiki/User_Contributions) na wiki do WebGL. Além disso, no site “Chrome Experiments” do Google, existe uma seção de WebGL, a maioria dos demos também funcionam em outros navegadores que suportam WebGL.

- Finalmente, se você quiser ver os melhores demos de WebGL toda semana, [assine este blog](http://learningwebgl.com/blog/?feed=rss2). Toda quinta-feira será postado um geral de todas as notícias sobre WebGL que forem encontradas nos últimos sete dias, e normalmente haverá várias demos novas.

Este é o primeiro post de iniciação com WebGL. Se você quiser ir mais longe e aprender mais sobre como crias seus próprios experimentos, você pode (e deve) conferir nossa primeira lição.

Lição 1

**Agradecimentos**

Obrigado ao [Vladimir Vukićević](http://blog.vlad1.com/2009/09/18/webgl-in-firefox-nightly-builds/), [Mohamed Mansour](http://www.m0interactive.com/archives/2009/10/26/how_to_enable_webgl_on_google_chrome.html,comments), Ehsun Amanolahi, e [Chris Marrin](http://webkit.org/blog/603/webgl-now-available-in-webkit-nightlies/) pelas informações que estão nesta página!

Adicionar agradecimentos pessoais

**Lição 1 – Um triangulo e um quadrado**

Bem vindo ao nosso primeiro tutorial de WebGL! A primeira lição é baseada na lição [número 2](http://nehe.gamedev.net/tutorial/your_first_polygon/13002/) do tutorial NeHe OpenGL, que é uma forma muito popular de aprender gráficos 3D para desenvolvimento de jogos. Ele irá mostrar a você como desenhar um triângulo e um quadrado em uma página web. Talvez não seja tão excitante, mas é uma ótima forma de introdução aos fundamentos de WebGL; se você entender como isso funciona, o restante será bastante simples...

Aqui está o que está lição irá mostrar quando executado em um navegador com suporte ao WebGL:

Imagem: <http://learningwebgl.com/lessons/lesson01/static.png>

[Clique aqui e você poderá ver uma versão real usando WebGL](http://learningwebgl.com/lessons/lesson01/index.html), se seu navegador não possui suporte, aqui está como conseguir um.

Mais informações sobre como tudo funciona a seguir...

Um aviso rápido: estas lições são recomendadas para pessoas com habilidades razoáveis de conhecimento em programação, mas não é exigido nenhuma experiência real com gráficos 3D;

O objetivo é entregar para você e em funcionamento, com um bom entendimento do que está acontecendo no código, para que você possa começar produzir seus próprios experimentos de páginas Web em 3D o mais rápido possível. Estou escrevendo essas lições da forma como eu aprendi WebGL por conta própria, portanto, pode haver (e provavelmente haverá) erros; use por sua conta em risco =D ([original](http://learningwebgl.com/blog/?page_id=2)). No entanto, estou consertando os bugs e corrigindo os equívocos de correção assim que forem encontrados, então se você encontrar qualquer coisa com problemas, fale sobre isso nos comentários.

Há duas formas que você pode obter o código para este exemplo, a primeira é vendo o código direto no seu navegador (ctrl+u ou inspecionar elemento por exemplo no Google Chrome) quando estiver visualizando a página do exemplo real, ou se você usar o [GitHub](https://github.com/), pode cloná-la (essa, e as lições futuras) do repositório lá. De qualquer maneira, assim que você tiver o código, abra em seu edito de texto preferido e dê uma olhada.

É meio assustador a primeira vista, mesmo se você tiver um conhecimento com OpenGL. Logo no início, estamos definindo um par de shaders, e isso é geralmente considerado como algo relativamente avançado... mas não se desespere, é realmente muito mais simples do que parece.

Como muitos programas, esta página WebGL começa pela definição de um grupo de funçções de baixo nível, que são utilizadas pelo código de alto nível na parte inferior. Para explicar isso, vou começar de baixo para cima, e trabalhar da minha forma, por isso, se você está acompanhando pelo código, pule para a parte inferior.

Você verá o seguinte código HTML:

<body onload="webGLStart();"> ...

</body>

Este é o corpo completo da página – todo o resto é em JavaScript (se você estiver vendo o código através do navegador pode haver algum lixo extra, ignore isso por favor). Obviamente poderíamos colocar mais HTML dentro da tag <body> e construir nossa imagem WebGL em uma página web normal, mas para esta demonstração simples iremos utilizar apenas o necessário para visualizar o conteúdo em WebGL, e a tag <canvas> será a responsável por este feito, é nela onde os gráficos em 3D são carregados. Canvas é uma novidade do HTML5 – é ele que suporta novos gêneros do JavaScript para desenhar elementos em páginas web, tanto em 2D quanto em 3D (através do WebGL). Nós não especicamos nada mais do que as propriedades de layout simples da tela em sua tag, e deixamos todo o código de configuração do WebGL para uma função JavaScript chamada webGLStart, que você pode ver é chamada uma vez que quando a página é carregada.

Vamos até essa função agora para dar uma olhada:

function webGLStart()

Ela chama outras funções para inicializar o WebGL e os shaders que mencionei anteriormente, passando como parâmetro o elemento canvas sobre o qual queremos desenhar nosso conteúdo em 3D, e então ele inicializa alguns buffers usando a função initBuffers, buffers são coisas que guardam os detalhes do triangulo e do quadrado que vamos desenhar – falaremos mais sobre eles em um outro momento. Em seguida, é feito algumas configurações básicas do WebGL, dizendo que quando limpamos a tela deverá ficar preto, e que devemos fazer o teste de profundidade (para que as coisas desenhadas por trás de outras coisas deve ser escondidas pelas coisas na que estão na frente, entenderão certo?). Essas etapas são implementadas por chamadas a métodos em um objeto gl – vamos ver como que é inicializado mais tarde. Finalmente, chama-se a função drawScene; esta (como já é de se esperar pelo nome) desenha o triangulo e o quadrado, usando os buffers.

Vamos voltar para as funções initGL e initShaders mais tarde, pois elas são importante na compreensão de como a página funciona, mas, primeiro, vamos dar uma olhada nas funções initBuffers e drawScene.

initBuffers primeiro, fazendo passo a passo:

var triangleVertexPositionBuffer;

var squareVertexPositionBuffer;

Declaramos duas variáveis globais para armazenar os buffers. (Em qualquer página de WebGL do mundo real você não teria uma variável global separada para cada objeto na cena, mas estamos usando aqui para manter as coisas simples, pois estamos apenas começando.)

Seguindo:

function initBuffers() {

triangleVertexPositionBuffer = gl.createBuffer();

Criamos um buffer para as posições dos vértices do triangulo. Vértices são os pontos no espaço 3D que definem as formas que estamos desenhando. Para o nosso triangulo, teremos três deles (que vamos definir em um minuto). Este buffer é na realidade um pouco de memória na placa de vídeo; colocando as posições dos vértices na placa em nosso código de inicialização e então, quando chegarmos a desenhar a cena, essencialmente, apenas dizendo ao WebGL para “desenhar aquelas coisas que eu lhe disse mais cedo”, podemos fazer o código muito eficiente, especialmente quando começar a animar a cena e desenhar objetos dezenas de vezes por segundo para fazer o movimento. É claro que, quando é apenas três posições dos vértices como neste caso, não há um custo muito alto para envia-los para a placa de vídeo – mas quando você estiver lidando com modelos grandes, com dezenas de milhares de vértices, pode ser um vantagem real fazer as coisas dessa forma. Continuando:

gl.bindBuffer(gl.ARRAY\_BUFFER, triangleVertexPositionBuffer);

PAREI AQUI

Essa linha diz ao WebGL que quaisquer operações seguintes que atuam em buffers deve usar o que especificamos. Há sempre esse conceito de um “matriz de buffer atual”, e as funções agirem nela em vez de deixar que você especifique qual matriz de buffer que você deseja trabalhar. Estranho, mas eu tenho certeza que há razões para um bom desempenho por trás disso...

var vertices = [

0.0, 1.0, 0.0,

A seguir, definimos nossas posições dos vértices como uma lista do JavaScript. Você pode ver que eles estão nos pontos de um triangulo isósceles, com seu centro em (0,0,0).

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(vertices), gl.STATIC\_DRAW);

Agora, criamos um objeto Float32Array com base em nossa lista de JavaScript, e dizemos para o WebGL usá-lo para preencher o buffer atual, que é naturalmente, o nosso triangleVertexPositionBuffer. Vamos falar mais sobre Float32Arrays em uma lição no futuro, mas por enquanto tudo o que você precisa saber é que eles são uma maneira de transformar uma lista de JavaScript em algo que podemos passar para o WebGL para preencher os seus buffers.

triangleVertexPositionBuffer.itemSize = 3;

triangleVertexPositionBuffer.numItems = 3;

A última coisa que fazemos com o buffer é definir duas novas propriedades sobre ele. Isso não é algo que esta embutido no WebGL, mas será útil mais tarde. Uma coisa legal (alguns diriam ruim) sobre JavaScript é que um objeto não tem que suportar explicitamente uma propriedade particular para que você defina sobre ele. Assim, embora o objeto buffer anteriormente não têm propriedades itemSize e numItens, agora ele tem. Nós estamos usando-as para dizer que este buffer atual de 9 elementos, representa três diferentes posições dos vértices (numItens) , cada um dos quais é composto de três números (itemSize).

Agora nós definimos por completo o buffer para o triângulo, então vamos para o quadrado:

squareVertexPositionBuffer = gl.createBuffer();

gl.bindBuffer(gl.ARRAY\_BUFFER, squareVertexPositionBuffer);

Tudo isso deve ser bastante óbvio – o quadrado tem quadro posições de vértices, em vez de 3, e assim a matriz é maior e o numItens é diferente.

Ok, então isso era o que precisávamos para enviar as posições dos vértices dos nossos dois objetos até a placa gráfica. Agora vamos olhara para a função drawScene, que é onde nós usamos aqueles buffers para realmente desenhar a imagem que estamos vendo. Fazendo passo-a-passo:

function drawScene() {

gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);

O primeiro passo é informar ao WebGL o tamanho da tela usando a função viewport, nós vamos voltar a falar sobre isso mais tarde pois é uma importante lição (e muito!) , por enquanto, você só precisa saber que a função precisa ser chamada com o tamanho da tela antes de começar a desenhar. Em sequida, limpamos a tela para preparar para desenhar sobre ela:

gl.clear(gl.COLOR\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT);

... e então:

mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);

Aqui nós estamos criando a perspectiva com a qual queremos ver a cena. Por padrão, o WebGL vai chamar as coisas que estão por perto do mesmo tamanho que as coisas que estão longe (um estilo 3D conhecido como projeção ortográfica). A fim de fazer as coisas que estão mais distantes parecerem menores, precisamos contar um pouco sobre a perspectiva que estamos usando. Para esta cena, estamos dizendo que o nosso campo (vertical) de vista é de 45º, estamos dizendo a ela sobre a relação entre largura e altura da nossa tela, e dizendo que nós não queremos ver as coisas que são mais perto do que 0.1 unidades para o nosso ponto de vista, e que nós não queremos ver as coisas que estão mais distantes do que 100 unidades.

Como você pode ver, para essas definições de perspectiva estamos usando uma função de um módulo chamado mat4, e envolve uma intrigante variável chamada pMatrix. Mais falaremos sobre isso mais tarde, espero que por agora esteja claro como usá-los sem a necessidade de conhecer seus detalhes.

Agora que temos a nossa perspectiva criada, podemos começar desenhar algumas coisas:

mat4.identity(mvMatrix);

O primeiro passo é o de “mover” para o centro da cena 3D. Em OpenGL, quando voe está desenhando uma cena, você diz a ele para desenhar cada coisa em uma posição “atual” com uma rotação “atual” – assim por exemplo, você diz “mover 20 unidades para frente, girar 32 graus, depois desenhar o robô”, o ultimo bit será um conjunto complexo de “mover muito, rotacionar um pouco, desenhar isso” instruções em si. Isso é útil porque você pode encapsular o código para “desenhar o robô” em uma função, e então mover facilmente o robô apenas mudando o mover/rotacionar que você fez antes de chamar essa função.

A posição atual e rotação atual são ambas guardadas em uma matriz, como você provavelmente aprendeu na escola, as matrizes podem representar translações (mova-se de um lugar para o outro) , rotações e outras transformações geométricas. Por razões que eu não vou entrar agora, você pode usar uma única matriz 4x4 (não 3x3) para representar qualquer número de transformações no espaço 3D, você começa com a matriz de identidade – ou seja, a matriz que representa uma transformação que não faz nada – então multiplica-la pela matriz que representa a sua primeira transformação, em seguida, por aquele que representa a sua segunda transformação, e assim por diante. A matriz combinada representa todas as suas transformações em uma. A matriz que usamos para representar o atual estado mover/rotacionar é chamada de matriz modelo de visão (Model View Matrix), e agora que você provavelmente percebeu que a variável mvMatrix matem nossa matriz modelo de visão, e a função mat4.identity que acabamos de chamar define como a matriz de identidade, de modo que nós estamos prontos para começar multiplicar translações e rotações nela. Ou em outras palavras, é transferida para um ponto de origem a partir do qual podemos começar a desenhar o nosso mundo 3D.

Leitores com olhos aguçados devem ter notado que, no início dessa discução de matrizes eu disse “em OpenGL” não em “WebGL”. Isto porque WebGL não tem isso construído para a biblioteca de gráficos. Em vez disso usamos uma biblioteca de matriz feita por terceiros – a excelente [glMatrix](http://code.google.com/p/glmatrix/) feita por [Brandon Jones](http://blog.tojicode.com/) – além de alguns truques elegantes de WebGL para obter o mesmo efeito, Mais sobre esses truques elegantes mais tarde.

mat4.translate(mvMatrix, [-1.5, 0.0, -7.0]);

Tendo movido para o centro do espaço 3D com a nossa mvMatriz definida para a matriz identidade, partimos do triangulo, movendo 1.5 unidades para a esquerda (isto é, no sentido negativo ao longo do eixo X), e sete unidades dentro da cena (isto é, para longe do espectador, o sentido negativo ao longo do eixo Z). (mat4.translate, como você pode imaginar, significa “multiplicar a matriz dada por uma matriz de translação com os seguintes parametros”.)

O próximo passo é realmente começar a desenhar alguma coisa! UFFA.

gl.bindBuffer(gl.ARRAY\_BUFFER, triangleVertexPositionBuffer);

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute

Então, você se lembra que, para usar um dos nosso buffer, chamamos gl.bindBuffer para especificar um buffer atual, e em seguida, chamar o código que opera sobre ele. Aqui estamos selecionando nosso triangleVertexPositionBuffer, em seguida, dizendo ao WebGL que os valores que deve ser usado para as posições dos vértices. Vou explicar um pouco mais sobre como isso funciona depois, por enquanto, você pode ver que nós estamos usando a propriedade itemSize que definimos no buffer para dizer ao WebGL que cada item no buffer é de três dígitos. A seguir, temos:

setMatrixUniforms();

Isso diz ao WebGL para tomar conta da nossa matriz modelo de visualização atual (e também a matriz de projeção, sobre o qual falaremos mais tarde). Isso é necessário porque todas essas coisas de matriz não é construído para WebGL. A maneira de olhar para ele é que você pode fazer tudo o que se mover, mudando a variável mvMatrix se quise, mas tudo isso acontece no espaço privado do JavaScript. setMatrixUniforms, uma função que é definida mais acima neste arquivo, move para a placa gráfica.

Uma vez feito isso, o WebGL tem uma série de números que ele conhece para ser tratado como as posições dos vértices, e ele sabe sobre as nossas matrizes. O próximo passo lhe diz o que fazer com eles:

gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.numItems);

Ou, dito de outra forma, “desenhar a matriz de vértices que eu te dei anteriormente como triângulos, começando com o item 0 no array e indo até o elemento numItems”.

Uma vez feito isso, o WebGL vai ter desenhado nosso triangulo.

Próximo passo, desenhar o quadrado:

mat4.translate(mvMatrix, [3.0, 0.0, 0.0]);

Começamos movendo nossa matriz modelo de visão três unidades para a direita. Lembre-se, estamos atualmente já 1.5 para a esquerda e 7 de distância da tela, então isso nos deixa 1.5 para a direita e 7 de distância. Continuando:

gl.bindBuffer(gl.ARRAY\_BUFFER, squareVertexPositionBuffer);

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute

Então, dizemos ao WebGL para usar o nosso buffer do quadrado para suas posições de vértice...

setMatrixUniforms();

... nós enviamos sobre as matrizes de modelo de visão e de projeção novamente (para levar em conta o ultimo mvTranslate), o que significa que nós podemos finalmente:

gl.drawArrays(gl.TRIANGLE\_STRIP, 0, squareVertexPositionBuffer.numItems);

Desenhar os pontos. O que você pode perguntar é: o que uma tira de triangulo (TRIANGLE\_STRIP)? Bem, é uma tria de triângulos =D mais necessariamente, que é uma tira de triângulos, onde os primeiros três vértices que vão especificar o primeiro triangulo, em seguida, os dois últimos desses vértices mais a próxima especifica o próximo triangulo e assim por diante. Neste caso, é uma maneira rápida e suja de especificar um quadrado. Em casos mais complexos, pode ser uma forma muito útil de especificação de uma superfície complexa em termos dos triângulos que aproximam isso.

De qualquer forma, uma vez feito isso, nós terminamos a função drawScene.

}

Se você chegou até aqui, você está definitivamente pronto para começar a experimentar. Copie o código em um arquivo local, pegando do GitHub ou diretamente do exemplo, se você optou pelo último, você precisa de um índex.html e do glMatrix-0.9.5.min.js. Execute local para se certificar que funciona, então tente mudar algumas das posições dos vértices acima, e em especial, a cena agora é bastante plana, tente alterar os valores de Z para o quadrado para 2, ou -3, e poderá ver ele chagando mais perto e maior, ou menor na medida que se move para trás ou para frente. Ou então, tente mudar apenas um ou dois deles, e ver distorcer em perspectiva. Faça loucuras e não se importe comigo. Vou esperar.

...

Bom, agora que você voltou e já se divertiu, vamos dar uma olhada nas funções de apoio que fizeram com que todo o código que fizemos funcionasse. Como eu disse antes, se você está feliz em ignorar os detalhes e apenas copiar e colar as funções de apoio que vêm acima de initBuffers na página, você pode provavelmente começar longe dele e contruir páginas interessantes em WebGL (embora em preto e branco – as cores ficam para a próxima lição). Mas nenhuns dos detalhes são difíceis de entender, e entendendo como eles funcionam você provavelmente irá escrever um código melhor em WebGL mais tarde.

Continua comigo? Obrigado =D Então vamos a mais chatas das funções fora do primeiro caminho, a primeiro chamada por webGLStart, que é InitGL. É perto do topo da página, e aqui está uma cópia para referência:

var gl;

function initGL(canvas) {

try {

Isso é muito simples. Como você notou, as funções initBuffers e drawScene frequentemente se referenciam a um objeto chamado gl, o que claramente se refere a algum tipo de “coisa” como um núcleo do WebGL. Esta função pega aquela “coisa”, que é chamado de um contexto WebGL, e faz isso por pedir o canvas dando o centexto, usando um nome de contexto padrão. (Como você pode imaginar, em algum momento o nome do contexto irá mudar de “experimental-WebGL” para “WebGL”), iremos informar quando isso acontecer. Uma vez que temos o contexto, votamos a usar a vontade do JavaScript para nos permitr definir qualquer propriedade que gostamos em qualquer objeto para armazenar nele a largura e altura do canvas a que se refere, isso é, para que possamos usá-lo no código que estabelece o viewport e a perspectiva no início da drawScene. Uma vez feito isso, o nosso contexto GL esta configurado.

Depois de chamar a initGL, webGLStart chama initShaders. Isso, é claro, inicializa os shaders (¬¬). Nós vamos voltar para falar sobre isso depois, porque primeiro, devemos dar uma olhada em nossa matriz modelo de visão, e a matriz de projeção que eu mencionei antes. Aqui está o código:

var mvMatrix = mat4.create();

var pMatrix = mat4.create();

Assim declaramos uma variável chamada mvMatrix para manter a matriz modelo de visão e uma outra chamada pMatrix para a matriz de projeção, e depois defini-las como vazio (tudo zero) para começar. Vale a pena dizer um pouco mais sobre a matriz de projeção. Como você vai se lembrar, foi aplicada a função mat4.perspective a variável glMatrix para configurar a perspectiva, bem no inicio da função drawScene. Isso aconteceu porque o WebGL não suporta nativamente perspectiva, assim como ele também não suporta nativamente uma matriz modelo de visão. Mas, assim como o processo de mover e girar as coisas que é encapsulado na matriz modelo de visão, o processo de fazer as coisas que estão longe parecer proporcionalmente menor do que as coisas que estão perto é o tipo de coisa que as matrizes são realmente boas para representar. E, como você já adivinhou até agora, sem dúvida, a matriz de projeção faz exatamente isso. A função mat4.perspective, com sua proporção e campo de visão, preenche a matriz com os valores que devem fazer o tipo de perspectiva que queríamos.

Certo, agora nós já passamos por tudo além da função setMatrixUNiforms, que, como disse anteriormente, move o modelo de visão e projeção de matrizes a partir do JavaScript para o WebGL, e as coisas assustadoras relacionadas aos shaders. Eles estão inter-relacionas, por isso vamos começar com algumas informações.

Agora, você pode estar se perguntando: o que é um shader? Bem, em algum momento na história dos gráficos 3D eles podem ter sido como o que soam como eles podem ser – pedaços de código que indicam ao sistema como a sombra, ou cor, partes de uma cena antes de ser desenhada. No entanto, ao longo do tempo têm crescido em escopo, na medida em que eles podem agora ser melhor definidos como bits de código que podem fazer absolutamente tudo o que deseja bits antes da cena ser desenhada. E isso é realmente muito útil, porque: a) eles rodam na placa gráfica, então eles fazem o que tem que fazer muito rápido e b) o tipo de transformações que podem fazer pode ser muito conveniente, mesmo em exemplos simples como este.

A razão pela qual estamos introduzindo shaders no que pretende ser um exemplo simples de WebGL (eles são, no mínimo nível “intermediário” em tutoriais de OpenGL) é que nós iremos usá-los para obter que o sistema WebGL, esperando que esteja rodando na placa de vídeo, para aplicar a nossa matriz modelo de visão e nossa matriz de projeção da nossa cena sem ter que deslocar em trono de cada ponto e cada vértice no (relativamente) lento JavaScript. Isso é incrivelmente útil, e vale a pena o esforço.

Então, aqui está como eles são configurados. Como você vai se lembrar, webGLStart chama initShaders, por isso vamos passar por esse passo-a-passo:

var shaderProgram;

function initShaders() {

var fragmentShader = getShader(gl, "shader-fs");

var vertexShader = getShader(gl, "shader-vs");

Como você pode ver, ele usa uma função chamada getShader para obter duas coisas: um “fragmento de shader” e um “vertex shader”, e em seguida, atribui a ambos uma coisa chamada no WebGL de “programa”. Um programa é um pouco de código que fica no lado WebGL do sistema, você pode ver isso como uma maneira de especificar algo que pode funcionar com a placa gráfica. Como era de se esperar, você pode associar a ele um número de shaders, cada um dos quais você pode ver como um trecho de código dentro desse programa, especificamente, cada programa pode conter um fragmento e um vertex shader. Nós iremos olhar para eles em breve.

shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aVertexPosition");

gl.enableVertexAttr

Uma vez que a função criou o programa e anexou os shaders, torna-se uma referência a um “atributo”, que ele armazena em um novo campo no objeto programa chamado vertexPositionAtribute. Mais uma vez estamos aproveitando a vantagem do JavaScript para definir qualquer campo em qualquer objeto; objetos de programa não tem um campo vertexPositionAttribute por padrão, mas é conveniente para nós manter os dois valores, por isso, apenas fazer do atributo um novo campo do programa.

Então, para que o vertexPositionAttribute serve? Como você pode lembrar, foi utilizado em drawScene, se você olhar de volta para o código que define as posições dos vértices do triangulo para seu buffer apropriado, você vera que as coisas que fizemos associam o buffer com esse atributo. Você vai ver o que isso significa em um outro momento, agora, vamos apenas observar que também usamos gl.enableVertexAtttribArray para dizer ao WebGL que nós queremos fornecer valores para o atributo usando um array.

shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");

shaderProgram.mvMatrixUniform

A última coisa que o initShaders faz é obter mais dois valores do programa, os locais das duas coisas chamadas variáveis uniformes. Nós vamos encontra-los em breve, por enquanto, você deve apenas ver que, como o atributo, nós armazenamos no objeto do programa por ser conveniente.

Agora, vamos dar uma olhada na função getShader:

function getShader(gl, id) {

var shaderScript = document.getElementById(id);

if (!shaderScript) {

return null;

}

Esta é mais uma daquelas funções que é muito mais simples do que parece. Tudo o que estamos fazendo é procurar um elemento em nossa página HTML que tem uma identificação que corresponde a um parâmetro passado, retirando o seu conteúdo, criando tanto um fragmento ou um vertex shader baseado nos seu tipo (mais sobre a diferença deles em uma lição futura) e em seguida, passando-o para o WebGL para ser compilado em um fromulário que pode ser executado na placa de vídeo. O código trata os erros, e ponto! Claro, nós poderíamos apenas fdefinir os shaders como strings dentro de nosso código JavaScript e não mexer extraindo do HTML – mas fazendo dessa forma, tornamos muito mais fácil de ler, por que eles são definidos como os scripts da página web, como se fossem JavaScript em si.

Tendo visto isso, deve dar uma olhada no código dos shaders:

<script id="shader-fs" type="x-shader/x-fragment">

precision mediump float;

void main(void) {

gl\_FragColor = vec4(1.0, 1.0, 1.0, 1.0);

}

</script>

A primeira coisa a lembrar sobre eles é que não são escrito em JavaScript, mesmo que a sintaxe da linguagem seja similar. Na verdade, eles são escritos em uma linguagem de shader especial – chamada de GLSL – que é muito baseado no C (como também, claro, é o JavaScript).

O primeiro dos dois, o shader de fragmento, não faz nada muito bonito, tem um pouco de código clichê, obrigatório para dizer a placa de vídeo como preciso que quer que seja com número de ponto flutuante (precisão média é bom, pois é necessário para ser suportado por todos os dispositivos WebGL – alto para alta precisão não funciona em todos os dispositivos móveis), então simplesmente especifica que tudo o que é desenhado será desenhado em branco. (Como fazer coisas coloridas é o tema da próxima lição).

O segundo shader é um pouco mais interessante. É um shader de vértice – que, você vai se lembrar, significa que é um pouco de código para a placa gráfica fazer praticamente tudo que quiser com um vértice. Associado a isso, tem duas variáveis uniformes chamadas uMVMatrix e uPMatrix. Variáveis uniformes são uteis porque elas podem ser acessadas de fora do shader – na verdade, de fora do seu programa , como você provavelmente se lembra de quando nós extraímos sua localização no initShaders, e do código nós veremos depois, onde (como eu tenho certeza que você percebeu) que configuramos os valores do modelo de visão e da matriz de projeção. Você pode pensar em programa do shader como um objeto (no sentido de orientação a objetos) e as variáveis uniformes como campos.

Agora, o shader é chamado para cada vértice, e o vértice é passado para o código shader como aVertexPosition, graças à utilização do vertexPositionAttribute na drawScene, quando associado um atributo com o buffer. O pequeno pedaço de código na rotina principal do shader apenas multiplica a posição do vértice pelo modelo de visão e as matrizes de projeção, e empurra para fora o resultado como a posição final do vértice.

Então, webGLStart chama initShaders, que usou getShader para carregar os shaders de fragmento e vértices a partir de scripts na página web, para que eles pudessem ser compilado e passado para o WebGL e usado mais tarde ao renderizar nossa cena 3D.

Depois de tudo isso, o único código restante não explicado é o setMatrixUniforms, o que é fácil de entender quando você já sabe tudo acima =D

function setMatrixUniforms() {

gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);

gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);

}

Assim, usando as referencia aos uniformes que representam a nossa matriz de projeção e nossa matriz de visualização de modelo, que temos de volta em initShaders, enviamos ao WebGL os valores de nossas matrizes feitas no “estilo JavaScript” .

Ufa! Isso era muito pra uma primeira lição, mas espero que agora você (e eu) compreenda todas as bases que nós vamos precisar para começar a contruir algo mais interessante – colorido, em movimento, devidamente modelos WebGL tridimencionais. Para saber mais, leia a lição 2.

Licao 0

Licao 2

Agradecimentos: obviamente estou em profunda divida ao NeHe por seu tutorial de OpenGL que serve de script para esta lição, mas eu também gostaria de agradecer ao Benjamin DeLillo e Vladimir Vukićević por seu código de exemplo WebGL, que eu tenho investigado, analisado, provavelmente mal, e eventualmente atrapalhado juntos no código em que eu me baseei neste post =D. Obrigado também a Brandon Jones pela glMatrix. Finalmente, graças a James Coglan, que escreve de propósito geral a biblioteca de matrix; as primeiras versões desta lição foi usado bem mais do WebGL em vez do glMatrix, então assim a versão atual talvez nunca existiria sem isso.

**Lição 2**

Bem vindos a segunda lição de WebGL! Desta vez iremos dar uma olhada em como obter cor para a cena. É baseado no número 3 dos tutorial NeHe de OpenGL.

Aqui está o que a lição deve parecer quando executada em um navegador com suporte a WebGL:

Imagem: <http://learningwebgl.com/lessons/lesson02/static.png>

Clique aqui e você poderá ver uma versão real usando WebGL, se seu navegador não possui suporte, [aqui está como conseguir um](file:///C:\Users\Vagner\Dropbox\Web\licao0).

Mais informações sobre como tudo funciona a seguir...

Um aviso rápido: estas lições são recomendadas para pessoas com habilidades razoáveis de conhecimento em programação, mas não é exigido nenhuma experiência real com gráficos 3D;

O objetivo é entregar para você e em funcionamento, com um bom entendimento do que está acontecendo no código, para que você possa começar produzir seus próprios experimentos de páginas Web em 3D o mais rápido possível. Se você ainda não leu o primeiro tutorial, você deve fazer isso antes de ler este – aqui vou apenas explicar as diferenças entre aquele código e este novo código.

Assim como antes, pode haver erros e equívocos neste tutorial, Se você encontrar alguma coisa errada, diga nos comentários que irei corrigir o mais rápido possível.

Há duas formas que você pode obter o código para este exemplo, a primeira é vendo o código direto no seu navegador (ctrl+u ou inspecionar elemento por exemplo no Google Chrome) quando estiver visualizando a página do exemplo real, ou se você usar o [GitHub](https://github.com/), pode cloná-la (essa, e as lições futuras) do repositório lá. De qualquer maneira, assim que você tiver o código, abra em seu edito de texto preferido e dê uma olhada.

A maior parte dele deve parecer muito com o primeiro tutorial. Olhando rápido de cima para baixo, nós:

- Definimos vertex e fragment shaders, usando a tag HTML <script> com tipos “x-shader/x-vertex” e “x-shader/x-fragment”

- Inicializamos um contexto WebGl em initGL

- Carregamos os shaders em um objeto programa WebGL usando getShader e initShaders

- Definimos a matriz modelo de visão mvMatrix e a matriz de projeção pMatrix, junto com a função setMatrixUniforms para envia-las para o JavaScript/WebGL divididos de modo que os shaders pudessem vê-las.

- Carregamos os buffers contendo as informações sobre os objetos na cena usando initBuffers

* Desenhamos a cena propriamente dita, com a função de nome apropriado drawScene.
* Finalmente, fornecemos o minimo de HTML necessário para exibir tudo isso.

As únicas coisas que mudaram neste código a partir da primeira aula são os shaders, initBuffers e a função DrawScene. A fim de explicar como as mudanças funcionam, você precisa saber um pouco sobre o pipeline de processamento WebGL. Aqui está um diagrama:

img: <http://learningwebgl.com/lessons/lesson02/simple-rendering-pipeline.png>

O diagrama mostra, de uma forma muito simplificada, a forma como os dados passados para as funções JavaScript em drawScene é transformado em pixels no canvas com WebGL exibidos na tela. Ele apenas mostra os passos necessários para explicar esta lição, nós vamos olhar para versões mais detalhadas em aulas futuras.

No mais alto nível, o processo fuciona assim: cada vez que você chamar uma função como drawArrays, o WebGL processa os dados que você já forneceu na forma de atributos (como os buffers que usamos para os vértices na lição 1) e variáveis uniformes (o que foi usado para a matriz de projeção e modelo de visão) e passa isso ao vertex shader.

Ele faz isso chamando o vertex shader de uma vez para cada vértice, cada vez com os atributos configurados adequadamente para o vértice, as variáveis uniformes também são passada, mas como seu nome sugere, elas não mudam de chamada para chamada. O vertex shader faz o que precisa com esses dados – na lição 1, aplicou as matriz de projeção e modelo de visão para que os vértices ficassem todos em perspectiva e movimentados de acordo com o estado atual do nosso modelo de visão – e coloca os seus resultados em coisas chamadas varying variables; uma detalhe que é obrigatório, gl\_Position, que contém as coordenadas do vértice uma vez que o shader tenha terminado de brincar com ele.

Uma vez que o vertex shader é feito, o WebGL faz a mágica necessária para transformar a imagem em 3D a partir dessas varying variables em uma imagem 2D, e depois chama o fragment shader uma vez para cada pixel na imagem. (Em alguns sistemas 3D você vai ouvir fragment shaders ser chamado de pixel shaders para isso.) Claro, isso significa que ele está chamando o fragment shader para os pixels que não tem vértices neles – isto é, os de entre os pixels que os vértices terminam. Para estes, ele preenche pontos para as posições entre os vértices, através de um processo chamado interpolação linear – para as posições dos vértices que compõem nosso triângulo, este processo “preenche” o espaço delimitado pelos vértices com pontos para fazer o triângulo visível. O objetivo do frament shader é retornar a cor de cada um destes pontos interpolados, e ele faz isso com uma varying variable chamada gl\_FragColor.

* Uma vez que o fragment shader é feito, seus resultados são bagunçados ao redor com um pouco mais de WebGL (novamente, nós iremos voltar nisso em uma futura lição) e eles são colocados no frame buffer, que é finalmente mostrado em tela.
* Esperamos que até agora, fique claro que o truque mais importante que esta lição ensina é como pegar a cor para os vértices do código JavaScript enviando para o fragment shader, uma vez que não temos acesso direto de um para o outro.