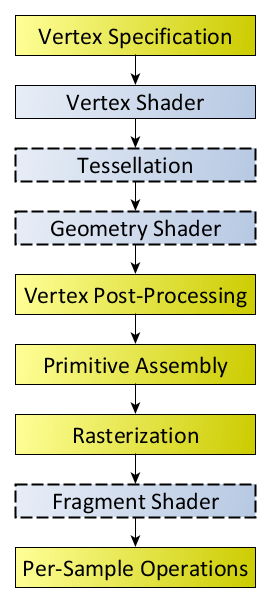
O Que é WebGL – Um Exemplo Prático

WebGL é uma implementação do OpenGL disponível nos navegadores, baseada no profile OpenGL ES, a versão para dispositivos embarcados do OpenGL, que por sua vez é baseada no OpenGL3.2. Com ela é possivel renderização 3d nos navegadores. Muitas coisas interessantes tem sido feitas com WebGL, com bom desempenho.

Pelo o WebGL se basear em ultima instância no OpenGL3.2 ele não tem as funções legadas, como glBegin()/glEnd(), as funções que controlavam as matrizes ou as que controlavam as luzes. Tudo isso tem que ser feito pelo programador usando shaders. Isso aparentemente aumenta o trabalho mas na prática o código para operar as matrizes e enviar as geometrias dos objetos para a placa de vídeo será feito somente uma vez e encapsulado em funções apropriadas. A consequência de não ter as funções consideradas legadas do opengl é que tutoriais excelentes do passado, como os da NeHe (<http://nehe.gamedev.net/>) estão obsoletos.

# 1)Teoria sobre a pipeline gráfica:



O objetivo dessa pipeline é transformar os dados fornecidos pelo usuário, como vértices, texturas, normais, etc, em imagens na tela do computador.

Na imagem ao lado temos as etapas da pipeline gráfica do opengl. As caixas em azul são as etapas programáveis

Especificação de vértices: é onde a aplicação cria uma lista ordenada de vértices e envia-a para a pipeline. Os vértices determinam os limites de uma primitiva. Primitivas são formas básicas como triângulos, linhas e pontos. É nessa etapa que se lida com vertex buffer objects e vertex array objects (no WebGL vertex array objects não são usados). Os dados de vértices são listas de atributos, onde cada atributo é um conjunto de dados que a próxima etapa da pipeline vai usar. A etapa de especificação de vértices ocorre no programa que usa o opengl, quando o programador usa os comandos de passagem de dados de vértices para o opengl. Depois da etapa de especificação, ainda no programa cliente que está usando o opengl, vem o comando de desenho, que começa a execução da pipeline renderizando os vértices como primitivas (ex.: triângulos, triangle strips).

Vertex Shader: vértices vindos como atributos da etapa anterior são processados aqui, com o shader rodando uma vez para cada vértice. É aqui que se transforma geometricamente a posição do vértice usando a matriz MVP (Model View Projection). A definição da MVP, sua passagem para o vertex shader e seu uso nele são definidos pelo usuário, não existindo por padrão (lembre-se que o webgl não tem as funções de gerenciamento de matrizes do opengl legado). Um vertex shader típico recebe como atributos as informações do vértice que ele vai processar, as matrizes que serão usadas no cálculo e retorna uma série de informações para as próximas etapas. Existe uma saída obrigatória do Vertex Shader: gl\_Position. gl\_Position é a posição do vértice após o processamento do vertex shader. Vertex shaders executam para cada vértice e não é possível (até onde sei) comunicação entre as instâncias individuais. A execução em paralelo do vertex shader (e de outros shaders) é a origem do poder de processamento das placas de vídeo: ao realizar dezenas de operações em paralelo e independentes umas das outras o poder do paralelismo é usado ao máximo.

Tessalation Shader: não é relevante para esse tutorial

Geometry Shader: idem

Transform Feedback: idem

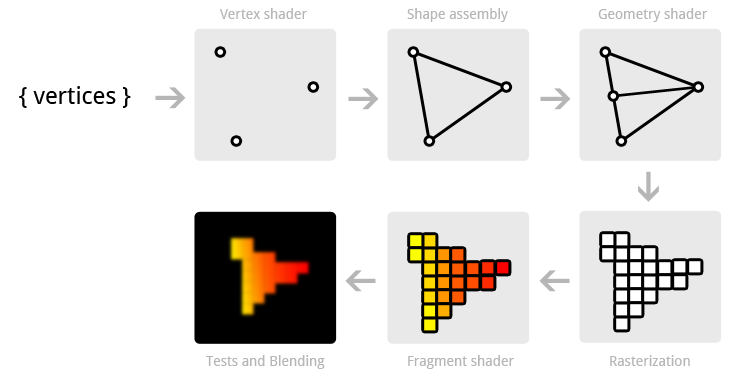
Clipping: Nessa etapa as primitivas que estão no limite da área de visão (a viewport) são cortadas para caber dentro da área, descartando as partes que estão fora.

Primitive Assembly: os dados sobre as primitivas das etapas anteriores é coletado e uma sequência de primitivas é criada, de acordo com o tipo de primitiva escolhida no começo. Aqui ocorre a eliminação de triângulos que não estão com face direcionada para o observador. A face de um triângulo é determinada pela ordem de especificação dos vértices (sentido horário ou sentido anti-horário). Imagino que seja porque é usado produto vetorial para saber o vetor normal à face e no produto vetorial a ordem dos operandos importa, ao determinar o sentido do vetor resultante do produto vetorial.

Rasterization: Nessa etapa as primitivas, como triângulos, são transformados em fragmentos. Fragmentos são conjuntos de dados usados para calcular o dado final de um pixel, eles não são o pixel ainda. Entre os dados de um fragmento temos sua posição na tela e dados que os shaders anteriores tenham emitido como saída (por exemplo, dados que o vertex shader emitiu como saída). Os fragmentos são calculados via interpolação entre os vértices.

Fragment Shader: esse shader é executado para fragmento e determina a cor do fragmento. É aqui, por exemplo, que texturas são aplicadas e iluminação calculada. As saídas desse shader são uma lista de cores para cada buffer de cor sendo escrito (a tela do computador é um buffer de cor e é o buffer padrão, mas outros buffers podem ser criados para técnicas avançadas) e um valor de profundidade. O fragment shader, embora seja normalmente presente, é opcional e sua ausência aparece em técnicas avançadas como testes de Occlusion Query.

Per-Sample Operation: Nessa etapa elimina-se os fragmentos que não aparecerão no buffer de cor. Elimina-se por cobertura por outras janelas, por exemplo, se tem uma janela cobrindo essa parte da tela onde o opengl faria a renderização, por estar fora de uma determinada região da tela (Scissor Test) caso essa região tenha sido definida, devido ao resultado do stencil test, caso um stencil tenha sido definido, ou por teste de profundidade, se um fragmento falha no teste de profundidade definido pelo usuário. É aqui também que ocorrem operações opcionais de blending, caso elas tenham sido definidas.



Para esse artigo será especificado somente o vertex shader e o fragment shader dentre as operações definidas pelo usuário.

# 2)O Servidor Node.js

NodeJS é um runtime de javascript que, entre outras coisas, permite criar servidores web de maneira rápida e leve, quando comparado a servidores java como o tomcat. Ele é capaz de muitas coisas, mas para nós nesse artigo ele será usado para criar um servidor web simples e com poucas linhas. A criação de um servidor simples já foi abordada por mim em <https://domgeronimo.blogspot.com.br/2017/08/meu-primeiro-programa-usando-nodejs.html> e não vou me repetir, inclusive porque a parte do NodeJS será exatamente igual, só vai mudar o conteúdo da pasta “public”.

Caso realmente precise ver os códigos e configurações, o código do server.js está abaixo:

//pega a biblioteca express

var express = require('express');

//Cria o objeto do servidor usando a biblioteca

var server = express();

server.use(express.static(\_\_dirname + '/public'));

server.listen(8080);

Agora é o código do package.json:

{

"name": "a\_cube",

"version": "0.0.0",

"description": "Um cubo usando webgl",

"main": "server.js",

"author": {

"name": "luciano"

},

"dependencies": {

"express": "4.15.3"

}

}

# 3)O programa

Agora que a infraestrutura foi montada, é hora de fazer nosso programa 3d. O programa será um cubo, uma câmera e uma luz. O cubo terá uma textura. Tudo deverá ser controlável pelo usuário. O cubo e outros dados serão carregados de um json. O programa demonstrará como usar a webgl em uma situação real e como carregar os dados de maneira dinâmica.

A carga é uma parte importante: os dados estão no mesmo servidor que a página e poderiam estar sendo carregados diretamente, mas não o farei porque há muitas situações onde o dado pode não estar no servidor, seja por estar sendo gerado dinamicamente, seja por estar em outro servidor e nesse caso é preciso carrega-lo. Cargas de arquivo, como todas as operações de IO no javascript são assíncronas e é melhor lidar com assincronicidade do javascript em um exemplo pequeno como esse programa do que em um programa mais complexo.

Começando a fazer a tela:

A tela terá de um lado uma canvas do opengl e do outro os controles dela. Estou usando o material design lite na página: Nesse momento o html está assim:

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<!--Começo do Material Design Lite-->

<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">

<link rel="stylesheet" href="https://code.getmdl.io/1.3.0/material.teal-red.min.css">

<script defer src="https://code.getmdl.io/1.3.0/material.min.js"></script>

<!--Fim do Material Design Lite-->

<head>

<meta charset="utf-8"/>

<title></title>

</head>

<body>

<div class="content-grid mdl-grid mdl-layout mdl-shadow--2dp ">

<div class="mdl-shadow--2dp mdl-cell--top mdl-cell--12-col">

<strong>WebGL - Um exemplo prático</strong>

</div>

<div class="mdl-cell mdl-shadow--2dp mdl-cell--6-col">

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp mdl-cell--top">

<strong>Cena</strong>

</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp mdl-cell--stretch">

<canvas width="400" height="400"></canvas>

</div>

</div>

<div class="mdl-cell mdl-shadow--2dp mdl-cell--6-col">

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp mdl-cell--top">

<strong>Controles</strong>

</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Posição da câmera</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Foco da câmera</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">View Up da câmera</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Tipo de Projeção</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Translação do objeto</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Rotação do objeto</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Pivot do objeto</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Posição da luz</div>

<div class="mdl-cell mdl-cell--12-col mdl-shadow--2dp">Intensidade da luz</div>

</div>

<div class="mdl-shadow--2dp mdl-cell--bottom mdl-cell--12-col">

FPS:

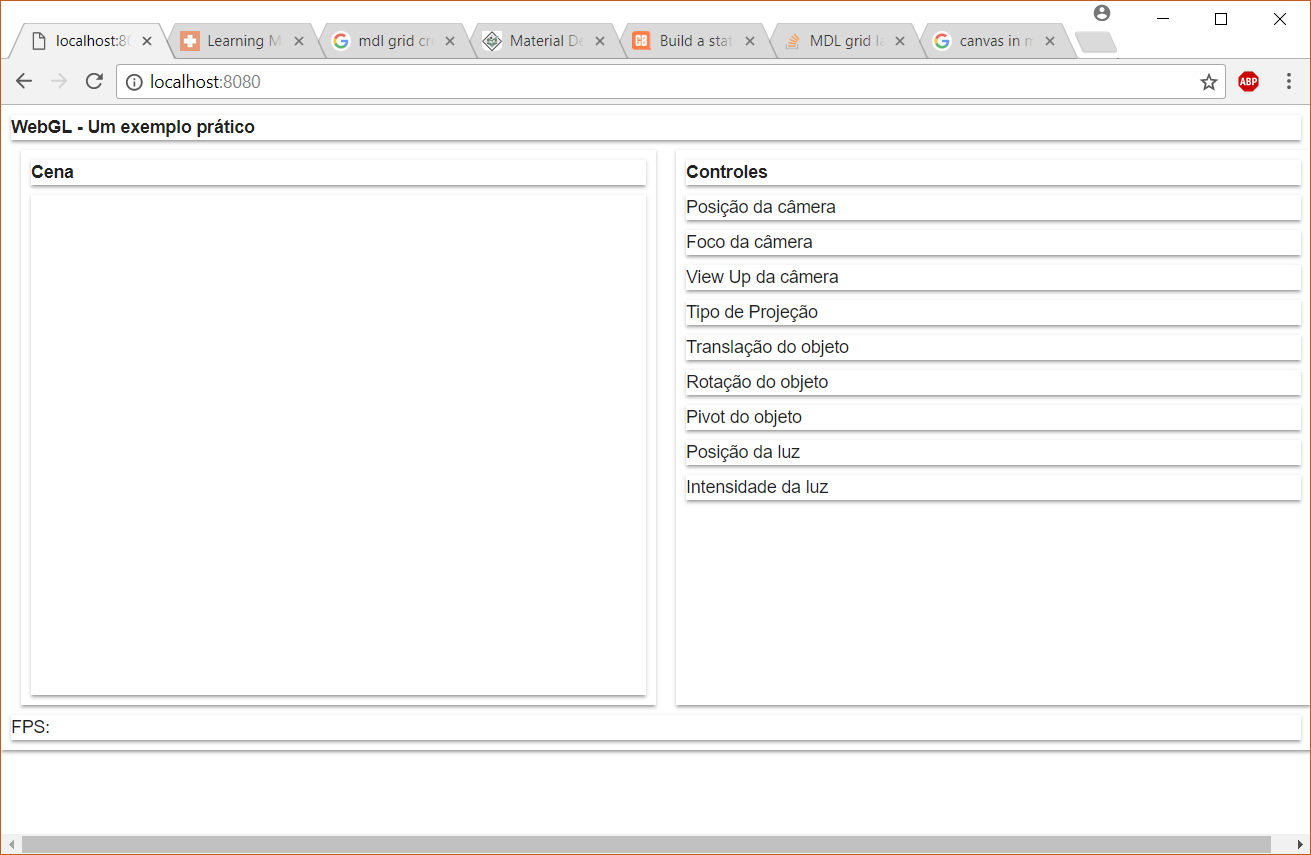
</div>

</div>

</body>

</html>

E sua imagem na tela é:



Nesse momento só temos o layout, sem programação. A partir daqui faremos o opengl funcionar, passando o máximo de funcionalidade para um arquivo de javascript, o openg.js.

Criando o contexto e testando

Para criar o contexto nós definiremos duas funções: initWebGL e initViewport. Uma é responsável por inicializar o contexto do webgl a partir de uma canvas, a outra por determinar o tamanho da viewport, a região onde desenho ocorre.

Em opengl.js temos:

//Cria o contexto do webgl

function initWebGL(canvas) {

var gl;

try {

gl = canvas.getContext("experimental-webgl");

}

catch (e) {

var msg = "Erro na criação do contexto: " + e.toString();

alert(msg);

throw Error(msg);

}

return gl;

}

//Cria a viewport

function initViewport(gl, canvas) {

gl.viewport(0, 0, canvas.width, canvas.height);

}

Para usar essas funções eu criei uma função que trata o onload da tela:

<script type="text/javascript">

var gl = undefined;

window.onload = function () {

let canvas = document.getElementById("webglcanvas");

//Inicialização do webgl

gl = initWebGL(canvas);

initViewport(gl, canvas.width, canvas.height);

}

</script>

Para terminar de ver a renderização falta algo que faça a tela ser renderizada. Isso é feito pela função, draw, invocada regularmente.

A função drawFrame():

//Onde desenho os quadros

function drawFrame() {

const t0 = (new Date()).getTime();

//....

const t = (new Date()).getTime();

showFPS(t0, t, $("#saida\_fps"));

requestAnimationFrame(drawFrame);

}

Ela faz uso de uma função para calcular e exibir o fps em uma div.

//Calcula o FPS e exibe na div dada.

function showFPS(t0, t, targetDiv) {

const fps = 1000.0 / ((t - t0) === 0 ? 1 : (t - t0));

targetDiv.text("FPS = " + fps);

}

Nós chamamos a drawFrame usando a função do javascript requestAnimationFrame, que é uma versão melhorada do método antigo de se fazer animação, que usava setInterval. Então o onload está assim:

//Coisas a fazer no callback da carga

window.onload = function () {

let canvas = document.getElementById("webglcanvas");

//Inicialização do webgl

gl = initWebGL(canvas);

initViewport(gl, canvas.width, canvas.height);

//...

requestAnimationFrame(drawFrame);

}

O resultado é que a função de desenho de frame é invocada constantemente. Agora, só falta efetivamente usar o opengl para desenhar algo e ver se está funcionando. Para isso faremos com que a cor da canvas seja mudada de maneira aleatória pelo opengl usando glClearColor e glClear:

//Onde desenho os quadros

function drawFrame() {

const t0 = (new Date()).getTime();

gl.clearColor(Math.random(), Math.random(), Math.random(), 1.0);

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

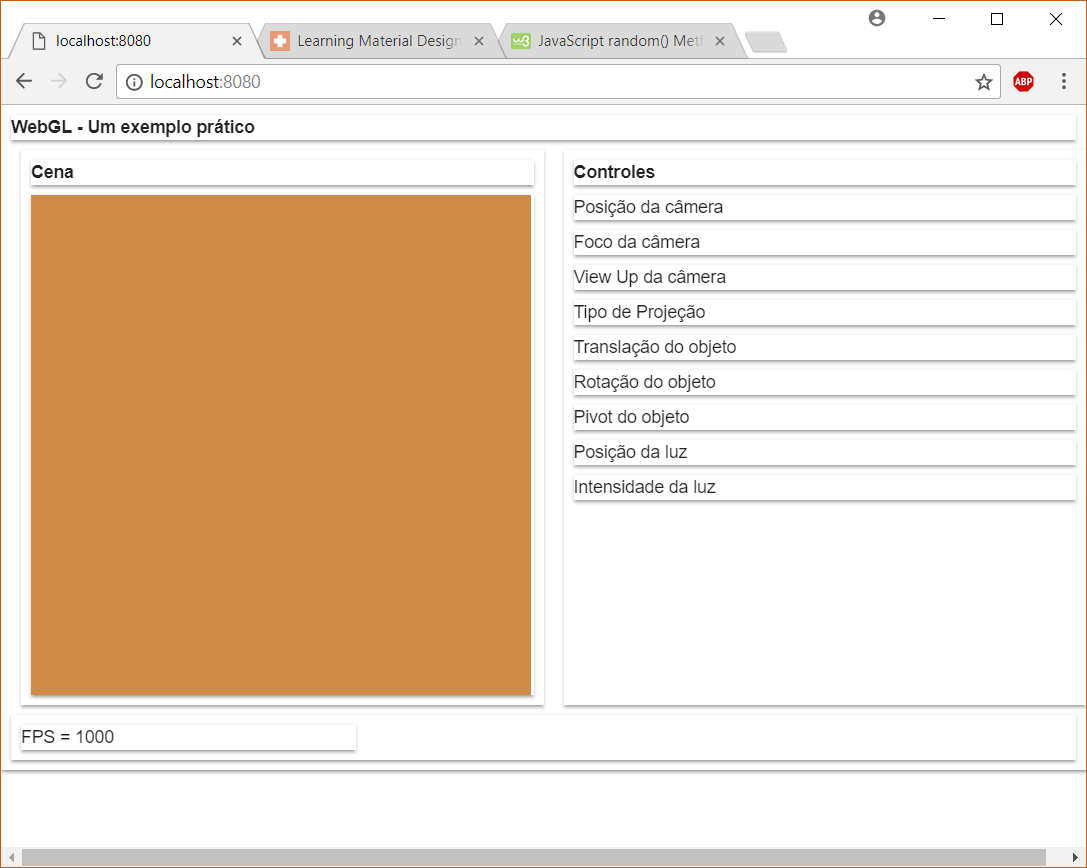
const t = (new Date()).getTime();

showFPS(t0, t, $("#saida\_fps"));

requestAnimationFrame(drawFrame);

}

O resultado é a canvas piscando de maneira psicodélica, mostrando que o contexo do opengl está funcionando:



Criação da geometria

Se o contexo do opengl está funcionando podemos desenhar algo mais complexo do que um fundo com cor aleatória. Criaremos um cubo, definido pelas seguintes informações: vértices, coordenadas de textura e normais. Os vértices determinam as faces do cubo, as coordenadas de textura determinam como mapear as texturas nas superfícies e a normal é uma informação necessária para o cálculo de iluminação.

Para passarmos as informações que o opengl precisa nós criaremos vertex buffers objects, um para cada bloco de informação. Seria possível que todas as informações estivessem em um mesmo bloco, organizadas por stride e offset, mas não o farei nesse exemplo sendo criado.

Criar uma geometria pra testar a exibição

Puxar a geometria de um arquivo

Texturas

Luzes

Controle da cena