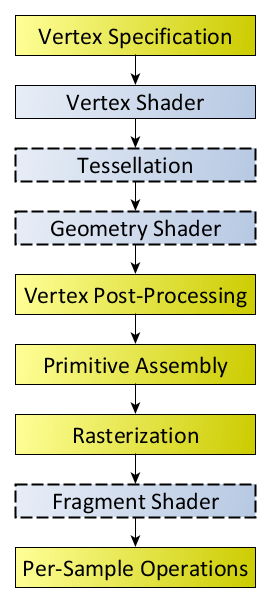
O Que é WebGL

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.

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

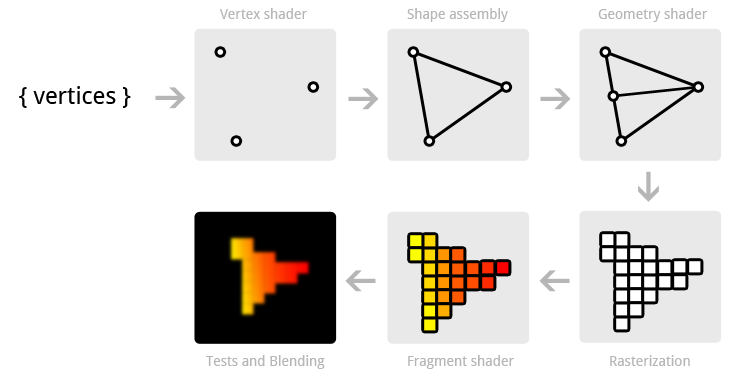
Clipping: Nessa etapa as primitivas que estão no limite da área de visão (a viewport) são cortadas para caber dentro da área, descartando as partes que estão fora.

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.