

# 毕业设计组会报告

徐海洋

# 目录

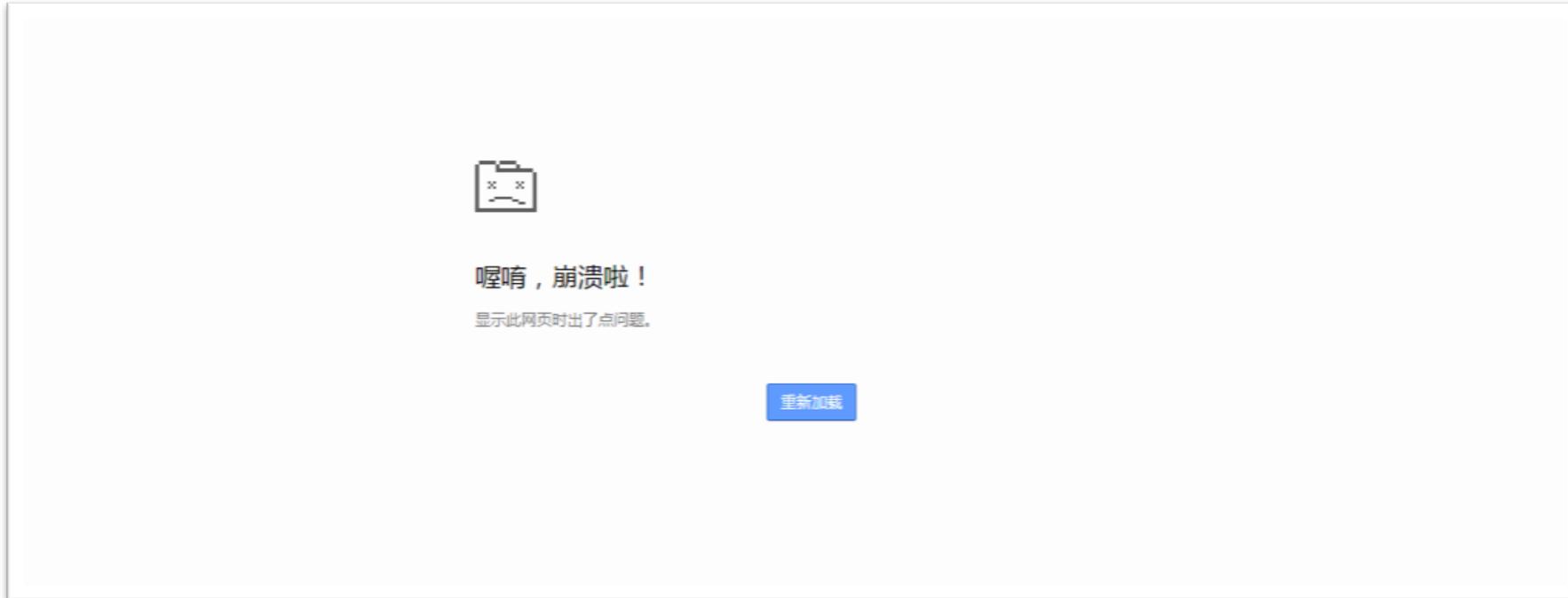
- 项目概述
- 调研的收获
- 一些问题
- 寒假的工作
- 附录

# 项目概述

基于 WebGL 利用 GPU 加速的前端图像处理框架

# 动机

- 在一个医疗图像标注系统的项目中，高分辨率的图像在前端直接使用 CPU 进行像素处理会造成浏览器崩溃。



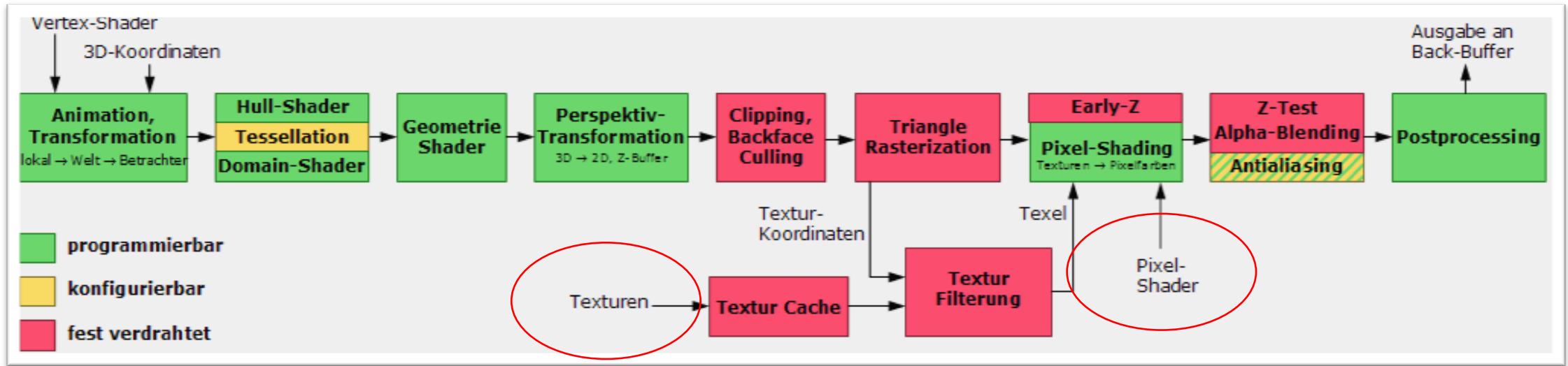
# 目标

- 希望前端工作与所学习的图像处理、**计算机图形学**的知识相融合。
- 实现一个高效、**易用**、可拓展的框架。

# 调研的收获

关于技术原理、框架设计等方面

# 渲染管线



# 着色器

JSHTMLCSSCodepenJSFiddleResultRun

```
36 var fragmentShaderSource = `#version 300 es
37
38 // fragment shaders don't have a default precision so we need
39 // to pick one. mediump is a good default. It means "medium precision"
40 precision mediump float;
41
42 // our texture
43 uniform sampler2D u_image;
44
45 // the texCoords passed in from the vertex shader.
46 in vec2 vTexCoord;
47
48 // we need to declare an output for the fragment shader
49 out vec4 outColor;
50
51 void main() {
52     outColor = texture(u_image, vTexCoord).rgba;
53 }
54 `;
55
56 var image = new Image();
57 image.src = "https://webgl2fundamentals.org/webgl/resources/leaves.jpg"; // MUST BE SAME DOMAIN
58 image.onload = function() {
59     render(image);
60 };
```



WebGL2Fundamentals

# 着色器

JSHTMLCSSCodepenJSFiddleResultRun

WebGL2Fundamentals

```
36 var fragmentShaderSource = `#version 300 es
37
38 // fragment shaders don't have a default precision so we need
39 // to pick one. mediump is a good default. It means "medium precision"
40 precision mediump float;
41
42 // our texture
43 uniform sampler2D u_image;
44
45 // the texCoords passed in from the vertex shader.
46 in vec2 vTexCoord;
47
48 // we need to declare an output for the fragment shader
49 out vec4 outColor;
50
51 void main() {
52     outColor = texture(u_image, vTexCoord).bgra;
53 }
54 ;
55
56 var image = new Image();
57 image.src = "https://webgl2fundamentals.org/webgl/resources/leaves.jpg"; // MUST BE SAME DOMAIN
58 image.onload = function() {
59     render(image);
60 };
```

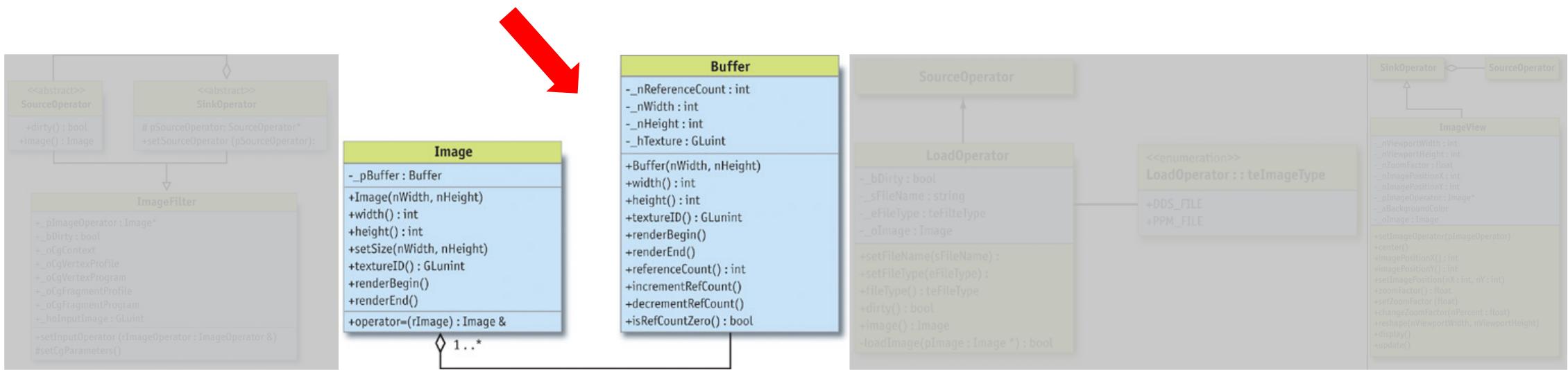


A red arrow points to the line of code `outColor = texture(u\_image, vTexCoord).bgra;` in the JavaScript code editor.

# 纹理

```
1 function initTextures() {
2     cubeTexture = gl.createTexture();
3     cubeImage = new Image();
4     cubeImage.onload = function() { handleTextureLoaded(cubeImage, cubeTexture); }
5     cubeImage.src = "cubetexture.png";
6 }
7
8 function handleTextureLoaded(image, texture) {
9     gl.bindTexture(gl.TEXTURE_2D, texture);
10    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
11    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
12    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_NEAREST);
13    gl.generateMipmap(gl.TEXTURE_2D);
14    gl.bindTexture(gl.TEXTURE_2D, null);
15 }
```

# 框架



# 易用

## Usage

Add a canvas element to your HTML with an id:

```
<canvas id="viewerCanvas" width="500" height="300"></canvas>
```

Then initialize the viewer in javascript:

```
// start a simple image viewer mit zooming and dragging features
var myImageViewer = new ImageViewer('viewerCanvas', 'myImage.jpg');
```

That's it!

## Creating the viewer

```
function ImageViewer(canvasId, imageUrl, options)
```

# 一些問題

在试验性工作中

# 环境

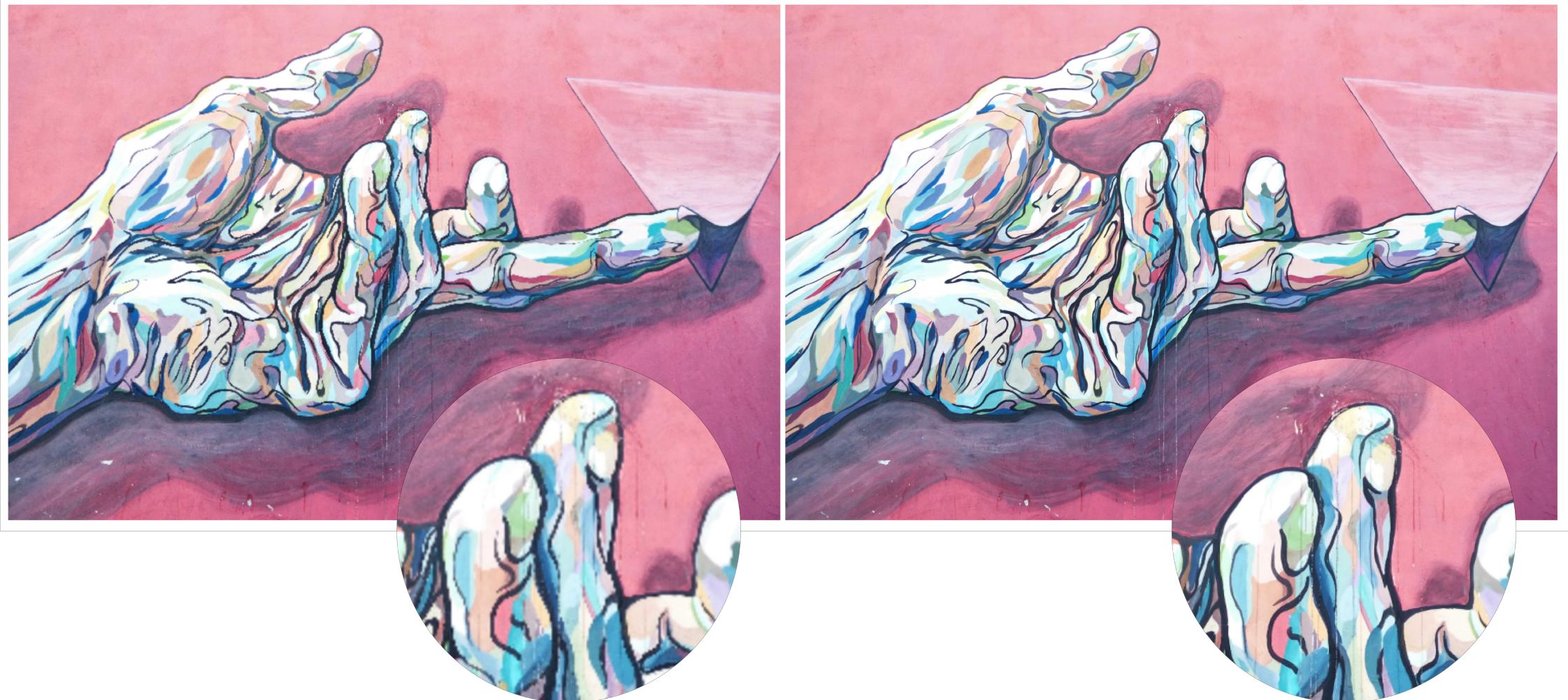
- 图片跨域访问：利用 `webpack-dev-server` 构建本地测试服务器
- 着色器的相关矩阵和向量计算：依赖 `gl-matrix` 的函数库

# DEMO

File: a.js

```
1 var f = Fundus(canvasId, imageUrl);
2 var b = document.querySelector('#btn');
3 b.onclick = function () {
4     f.gray();
5 };
```

# 问题

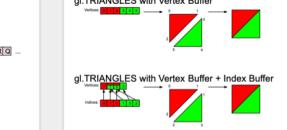
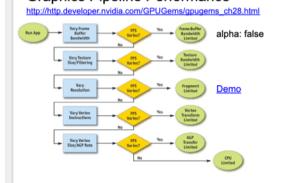
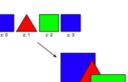
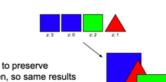


# 探索

- 严重的问题：采用纹理渲染方法后，图像的**分辨率被压缩**
- 可能原因：原图作为纹理直接绑定至较小的画布时，默认的采样方式导致失真（内优化）
- 解决方案：使用顶点着色器缩放（尚未试验）

# 调试与优化

- Ben Vanik & Ken Russell in New Game 2011 SF
- Debugging and Optimizing WebGL Applications

<p><b>WebGL Inspector</b>  <a href="http://benvanik.github.com/WebGL-Inspector/">http://benvanik.github.com/WebGL-Inspector/</a></p> <ul style="list-style-type: none"> <li>• Choose extension for graphical WebGL debugging</li> <li>• Capture entire WebGL frames for inspection</li> <li>• Texture, buffer, and program browser/viewing</li> <li>• Draw call state</li> <li>• Redundant call/error display</li> </ul> <p>Simple embedded demo  <a href="#">WebGL_Aquarium</a></p>	<p><b>Optimizing WebGL</b></p>	<p><b>GPU Optimization Whack-a-Mole</b></p> <ul style="list-style-type: none"> <li>• The best optimizations are often domain-target specific           <ul style="list-style-type: none"> <li>◦ If you presented app preferences with 'User Agent' ...</li> <li>◦ Your Desktop != User Desktop != Mobile != ...</li> </ul> </li> <li>• No one-size fits all solution - not even for different parts of the same app</li> <li>• Redundant often (try to automate)</li> <li>• Micro-optimizations are often not as helpful as real tests           <ul style="list-style-type: none"> <li>◦ GPUs are complex, rarely a single-point bottleneck</li> </ul> </li> <li>• 15 minutes implementing a well-principled optimization can save days hunting extra perf later</li> </ul> <p>Never dismiss an optimization that has no effect when first applied - it may just mean your bottleneck (on a specific machine/browser/etc) is somewhere else... for now</p>	<p><b>General Performance Rules</b></p> <p style="color: red;">Reduce the number of draw calls per frame</p>	<p><b>Vertex Buffer Structure</b></p> <ul style="list-style-type: none"> <li>• Reduce number of vertices           <ul style="list-style-type: none"> <li>◦ Use index buffers</li> <li>◦ Reduce per-vertex data               <ul style="list-style-type: none"> <li>◦ Faster to upload</li> <li>◦ Less data for the GPU to fetch</li> <li>◦ Use fewer components (XYZ, not YZWX)</li> </ul> </li> <li>◦ Keep attributes aligned on native 4-byte boundaries</li> <li>◦ Interleave arrays whenever possible</li> </ul> </li> <li>• On mobile, use smaller data types           <ul style="list-style-type: none"> <li>◦ Prefer &lt; SHORT &lt; FLOAT, etc</li> <li>◦ Beware of performance pitfalls on certain desktop-class GPUs</li> <li>◦ Recommend to only do this if memory bound</li> </ul> </li> </ul>	<p><b>Vertex Buffer Structure</b></p> <p>Pos: XYZW Color: RGBA Tex: STRG            Sort: XYZW XYZW ...  RGBA RGBA ...  STRG STRG ...            Interleaved: XYZW WRGBA STRG DXYZW WRGBA STRG ...            Stride: XYZ WRGBA ST XYZ WRGBA ST XYZ WRGBA ST ...            Alignment: XYZ ST XYZ WRGBA ST XYZ WRGBA ST ...</p> <p><a href="#">iOS Programming Guide on Alignment</a></p>	<p><b>Reusing Vertices with Index Buffers</b></p>  <p>Index buffers enable additional GPU performance features - better caching behavior</p>	<p><b>Dynamic Buffers</b></p> <p>If need to update vertex attributes from the CPU, try to split array buffer based on update frequency</p> <p>e.g., updating only position on sprites:</p> <pre>Pos: XYZW Color: RGBA Tex: STRG Updated every frame: bufferData usage = STREAM_DRAW XYZW XYZW ... Updated infrequently: bufferData usage = STATIC_DRAW RGBEAT ST XYZW ... </pre> <p>Ensure appropriate usage in <code>bufferData</code></p>
<p><b>General Performance Rules</b></p> <p>Use <code>requestAnimationFrame</code></p> <ul style="list-style-type: none"> <li>• More robust framerate (vs. <code>setInterval/setTimeout</code>)</li> <li>• Browser can stop rendering when hidden</li> <li>• Browser can throttle if many tabs rendering</li> </ul> <p>Because callbacks can be delayed (if hidden), put networking/etc on alternate timing mechanism</p> <p>Always use if available (for 2D canvas too) - fallback with <code>setTimeout</code></p>	<p><b>General Performance Rules</b></p> <p>Avoid get/read* calls</p> <ul style="list-style-type: none"> <li>• Cause full flushes/blocks the GPU</li> <li>• Often incur expensive copies/allocations</li> <li>• Takeaway: cache state yourself in Javascript</li> <li>• Takeaway: readback only what is required and very carefully</li> </ul> <p><code>getError</code></p> <ul style="list-style-type: none"> <li>• Not a call in production!</li> <li>• Not in anywhere</li> <li>• Multi-process renderers like Chrome can suffer greatly           <ul style="list-style-type: none"> <li>◦ getError blocks!</li> </ul> </li> <li>• Takeaway: don't use webgl-debug.js outside of development</li> </ul>	<p><b>General Performance Rules</b></p> <p>Avoid redundant calls</p> <ul style="list-style-type: none"> <li>• Best case: extra Javascript overhead</li> <li>• Worst case: will cause GPU to block (changing state/etc)</li> </ul> <p>Use WebGL Inspector to find redundant calls and identify where batching can be employed</p>  <p>Demo</p>	<p><b>General Performance Rules</b></p> <p>Disable unused GL features</p> <ul style="list-style-type: none"> <li>• Blending, alpha testing, etc are not always free</li> <li>• ...but don't change state excessively</li> </ul> <p>Link programs infrequently</p> <ul style="list-style-type: none"> <li>• Shader verification/translation can take a long time           <ul style="list-style-type: none"> <li>◦ Worse on Windows with ANGLE</li> </ul> </li> <li>• Create/link programs as early as possible/on load</li> <li>• Balance program complexity vs. number of programs</li> </ul> <p>Don't change Renderbuffers, change Framebuffers</p> <ul style="list-style-type: none"> <li>• Attaching Renderbuffers requires a lot of validation (note this is counter to iOS perf guidelines)</li> </ul>	<p><b>Dynamic Buffers</b></p> <p>WebGL (currently) mandates that implementations validate indices during <code>drawElements</code> calls</p> <p>Caches of index validation results are cleared if indices are modified</p> <p>Avoid updating index buffers if at all possible</p>	<p><b>Packing</b></p> <ul style="list-style-type: none"> <li>• If the range of the value is constrained, pack multiple values into single components           <ul style="list-style-type: none"> <li>◦ Pack RGB &gt; single float, unpack in vertex shader</li> <li>◦ Reduces upload/stream bandwidth at the cost of extra arithmetic</li> <li>◦ Can do processing offline when building models</li> <li>◦ Google "your float packing" - lots of clever math tricks out there</li> </ul> </li> </ul> <p>Can also be used to output complex values from fragment shaders into RGBA 32-bit textures for readback</p>	<p><b>Optimizing Shaders</b></p> <p>Always ask yourself...</p> <p>Can it be constant(sh)?</p> <p>(...the answer may surprise you)</p>	<p><b>Compute Infrequently</b></p>
<p><b>'Graphics Pipeline Performance'</b>  <a href="http://http://developer.nvidia.com/GPUGems/gpugems_ch28.html">http://http://developer.nvidia.com/GPUGems/gpugems_ch28.html</a></p> 	<p><b>Optimizing Drawing</b></p>	<p><b>Drawing scenes in Canvas</b>            (aka Painters Algorithm)</p> <p>sort objects by z-index for each object:            draw object</p> 	<p><b>Drawing scenes in WebGL</b></p> <p>sort objects by state, then depth for each state:            for each object:            draw object</p>  <p>Depth buffer used to preserve order on the screen, so same results as Painter's, with batched states</p>	<p><b>Compute Early</b></p>  <p>A <math>\text{world} \times \text{viewProj}</math> matrix multiply in a vertex shader will limit your geometry stage, vs. a uniform <math>\text{worldViewProj}</math> matrix</p> <p>A <math>\text{viewProj} \times \text{normal}</math> vector multiply in a fragment shader will limit your shading stage, vs. a varying passed from the vertex shader</p> <p>&gt; One javascript matrix multiply is better than 40k vertex shader multiplies (or 40k vertex vs. 2m fragment!)</p>	<p><b>Compute Inexactly</b></p> <ul style="list-style-type: none"> <li>• Use the lowest precision possible in a shader           <ul style="list-style-type: none"> <li>◦ Lower default precision</li> <li>◦ Lower varying precision</li> <li>◦ Go as low as compiler/allowed</li> <li>◦ Dithered planks may ignore precision, be careful</li> <li>◦ <code>highp</code> is optional in fragment shaders in OpenGL ES 2</li> </ul> </li> <li>• Use multiple programs to provide LOD if shading bound           <ul style="list-style-type: none"> <li>◦ Fewer/no texture samples (constant color for distant objects)</li> <li>◦ No lighting, skinning, etc</li> </ul> </li> <li>• Prefer math that works, not math that is 'correct'</li> </ul>	<p><b>Optimize Texture Sampling</b></p> <ul style="list-style-type: none"> <li>• Use mipmaping           <ul style="list-style-type: none"> <li>◦ Gains quality and performance</li> <li>◦ Lower varying precision</li> <li>◦ Marginal memory hit (33% of total, at most)</li> <li>◦ Longer compile times, yet you get better screen size, etc</li> </ul> </li> <li>• Sample predictably in indirect lookups           <ul style="list-style-type: none"> <li>◦ Exploit the (often small) GPU sampling cache</li> <li>◦ Reorganize texture layout to match sampling pattern</li> <li>◦ Tightly pack data in textures</li> </ul> </li> <li>• Use the proposed compressed texture extensions, when available</li> </ul>	<p><b>Dependent Reads/Instructions</b></p> <p>GPUs are good at parallelizing fragment shaders... unless you prevent them from doing so!</p> <p><b>Dependent read:</b></p> <pre>void main() {     vec2 value = texture2D(s.lookupSampler, uv).st;     // GPU stalled waiting for value...     gl_FragColor = texture2D(s.textureSampler, value); }</pre> <p>• Try to insert expensive math in between samples so the GPU is not idle</p> <p>• Move first sample to vertex shader if VTF supported</p> <p>• Avoid dependent reads altogether if possible</p>

# 寒假的工作

目标及安排

# 目标

- 高效：
  - 工程细节优化，非图像处理算法部分
  - 探索渲染效率评价标准或方式
- 易用：
  - 继续完善基本的框架模型
  - 添加对图像的基本功能，从显示到滤镜、缩放

# 可拓展性

- 如何让库真正成为框架？
- 如何提供一种更好的着色器编写或引用的方式？

# 附录

相关资料

# 清单

- [Graphics pipeline - Wikipedia](#)
- [WebGL2 Image Processing – WebGL2Fundamentals](#)
- [Using textures in WebGL – MDN](#)
- [GPU Gems - NVIDIA Developer](#)
- [HTML5/Canvas Image Viewer - Github](#)
- [DevServer – webpack](#)
- [glMatrix - Github](#)

# 参考项目

[evanw / webgl-filter](#)  
An image editor in WebGL  
 JavaScript   755   107   Updated on 14 Oct 2016

---

[phoboslab / WebGLImageFilter](#)  
Fast image filters for Browsers with WebGL support  
 JavaScript   190   41   Updated on 4 Nov 2015

---

[evanw / glfx.js](#)  
An image effects library for JavaScript using WebGL  
 JavaScript   2,111   293   Updated on 21 Mar 2016

---

[gre / gl-react](#)  
gl-react – React library to write and compose WebGL shaders  
 JavaScript   1,713   93   Updated on 7 Nov

---

[gpujs / gpu.js](#)  
GPU Accelerated JavaScript  
 JavaScript   6,639   315   Updated 13 days ago

[npm v2.2.8](#) [downloads 7k/m](#) [license MIT](#) [build error](#) [coverage 66%](#)

## Cornerstone Core

[Greenkeeper enabled](#)

Cornerstone is an open source project with a goal to deliver a complete web based medical imaging platform. This repository contains the Cornerstone Core component which is a lightweight JavaScript library for displaying medical images in modern web browsers that support the HTML5 canvas element. Cornerstone Core is not meant to be a complete application itself, but instead a component that can be used as part of larger more complex applications. See the [OHIF Viewer](#) for an example of using the various Cornerstone libraries to build a simple study viewer.

Cornerstone Core is agnostic to the actual container used to store image pixels as well as the transport mechanism used to get the image data. In fact, Cornerstone Core itself has no ability to read/parse or load images and instead depends on one or more [ImageLoaders](#) to function.

The goal here is to avoid constraining developers to work within a single container and transport (e.g. DICOM) since images are stored in a variety of formats (including proprietary). By providing flexibility with respect to the container and transport, the highest performance image display may be obtained as no conversion to an alternate container or transport is required. It is hoped that developers feel empowered to load images from any type of image container using any kind of transport. See the [CornerstoneWADOImageLoader](#) project for an example of a DICOM WADO based Image Loader.

Cornerstone Core is agnostic to the exact interaction paradigm being used. It does not include any mouse, touch or keyboard bindings to manipulate the various image properties such as scale, translation or ww/wc. The goal here is to avoid constraining developers using this library to fit into a given ui paradigm. It is hoped that developers are empowered to create new paradigms possibly using new input mechanisms to interact with medical images (e.g. [Kinect](#) or [Accelerometer](#)). Cornerstone does provide a set of API's allowing manipulation of the image properties via javascript. See the [CornerstoneTools](#) library for an example of common tools built on top of Cornerstone.