**四川师范大学本科毕业设计**

基于硬件抽象层接口场景渲染及拓展

|  |  |
| --- | --- |
| **学生姓名** | **黄家辉** |
| **院系名称** | **影视与传媒学院** |
| **专业名称** | **数字媒体技术** |
| **班级** | **2015级10班** |
| **学号** | **2015321007** |
| **指导教师** | **王聪** |
| **完成时间** | **2019 年3 月31日** |

基于硬件抽象层接口场景渲染及拓展

学生姓名：黄家辉 指导教师：王聪

内容摘要**：**这个作品简单来讲就是就是基于现有渲染层提供的 API 接口通过软件编程的方式将 GPU 中的运算通过代码的方式实现在 CPU 层面。网上相关的各个层面的探究较多但是大多数是以 C++ 编写，这次我通过 js 语言在模拟浏览器上通过 webgl 的 API 实现。这个作品可以称作为web 端简易软件渲染器。

关键词：软件渲染器 webgl GPU CPU

**Interface scene rendering and extending based on hardware abstraction layer**

Abstract：This work is simply said to be based on the API interface provided by the existing rendering layer. Through the way of software programming, the calculation in the GPU is realized at the CPU level through the way of code. There are many explorations in various aspects related to the Internet, but most of them are written in C++. This time, I implemented webgl API in the simulated browser through js language. This work can be called a simple software renderer for the web.

Keywords**：**Software renderer webgl GPU CPU

目录

[一、引言： 4](#_Toc5015334)

[二、项目设计： 4](#_Toc5015335)

[三、最终目标： 5](#_Toc5015336)

[四、功能设计说明 5](#_Toc5015337)

[1. 我的渲染器： 5](#_Toc5015338)

[2. 绘制点： 7](#_Toc5015339)

[3. 绘制线段 8](#_Toc5015340)

[4. 绘制三角形 10](#_Toc5015341)

[5. 绘制多个三角形，仿射变换 14](#_Toc5015342)

[6. 绘制 Cube 15](#_Toc5015343)

[7. 绘制 Texture 19](#_Toc5015344)

[8. 计算 mipmap 21](#_Toc5015345)

[9. 绘制 circle 24](#_Toc5015346)

[10. material 26](#_Toc5015347)

[11. object file 26](#_Toc5015348)

[五、结言 27](#_Toc5015349)

[主要参考文献 27](#_Toc5015350)

**一、引言：**

首先我想来谈一下大学四年从接触编程到学习程序到今天的一些感受，这包含了我制作这个毕设的一些初衷想法。假设我现在已经是以为真正意义上的的程序员，在游戏这个大领域里提供给很多想我一样的人很多选择的方向，比如游戏的程序架构、游戏检测、游戏渲染等方面，越是精致的游戏所涉及的方面就越多。那么在计算机这个领域的就业人员往往只会其中一个方向是不行的，这也是计算机很有意思的地方，单在游戏领域里面你能够选择的方向以及要学习的知识就几乎无穷无尽，我今次选择游戏渲染作为我个人毕设，因为首先是大环境因素的要求，为了更好的发展，这里我不多做叙述，仅是我个人认为。其次是因为渲染的可持续发展性，随着逐步的接触渲染层面的内容，没有接触过的人如何能够想象仅靠几个基础 API 能在程序美工手下呈现出那么多五彩缤纷的效果，这归功于计算机里包含的无穷的数学的思想与方法，这就是渲染的核心之一，不断的计算，这也是不断学习的前提。也许在掌握了足够的基础之后，你还拥有了几项甚至 几十项工具技能，就游戏领域来看一般是游戏引擎，又或者插件工具使用，总之学完你拥有的那些，是否还要继续寻找下个技术点去学习攻克，找什么样的技术点学习攻克都会成为程序在项目或者闲时需要考虑的问题。但是学习渲染就不需要考虑这些问题，因为我保证在游戏渲染领域里面的数学，物理思维是一个人一辈子也学不完的，不仅是因为它日新月异，而且还因为它复杂！

最后，不过是花里胡哨罢了，从一个灯光的效果开始，简单的代码编程是枯燥的，但是在探索学习渲染的过程是充满挑战的，是可以看到效果的，不是单纯的点击事件，而是一个美丽的世界。

**二、项目设计：**

这里我首先讲解一下什么是软件渲染器，从渲染管线开始，opengl，DX，都为我们提供了一套成型的 API，帮助我们渲染事物，但是 GPU 内部还是拥有自己的处理数据方案，当然随着这么长时间的演变下来， GPU 内的运算已经相当的成熟优秀，软件渲染器简单来讲就是通过代码与数学公式将 GPU 中已经封装好的渲染 API 在 CPU 重现。

那么GPU 和 CPU 最大的差距在于处理器的数量，GPU 专门用于处理渲染时的大量数据计算，因此渲染效率远远高于 CPU，在一次性传入所有的渲染数据之后 GPU 将直接计算出并封装好渲染结果呈现给使用者（这也是 webgl 使用时的一个优化处理点），介绍完上述的基本要点，下面谈下我使用 webgl 通过浏览器实现一套软件渲染器的好处与坏处分别是什么？

好处：

1. electron 的良好的封装和使用，便于打包成成品

2. js 的提供丰富的工具以及灵活的语法

坏处：

1. 相比于 C++ 或者C 语言编写的软件渲染器，webgl 通过 js 实际上还是通过浏览器最终转换调用的 opengl，那么在渲染效率上如果算法不够精炼或者结构不够完善的话，最终的渲染效率会远远低于直接使用 GPU 渲染。我们可以得到的结论是：渲染效率： webgl < opengl(CPU) < GPU 直接使用现成 API。

**三、最终目标：**

这是在我完成代码编程之后开始写的论文，所以本来的最终目标其实已经偏颇了不少，因为我第一次完整的编写软件渲染器，中间遇到的问题导致我花费的时间不断的增多，当然也因为我有懒癌一天写 4 个小时已经算是不错的成绩了。

|  |  |
| --- | --- |
| 原本的预期目标： | 最终完成 |
| Point | 完成 |
| Circle | 完成 |
| Triangle | 完成 |
| 仿射变换 | 完成 |
| 进入 3D 部分深度检测 | 完成 |
| Cube | 完成 |
| Texture | 完成 |
| PointLight | 半完成 |
| 噪点 Fov等额外效果 | 未完成 |

额外效果其实还有很多基本的效果，但是因为过于庞杂，我对这个部分涉及的不够深入，只能说日后有机会当我更加深入渲染这个领域之后再能查漏补缺。

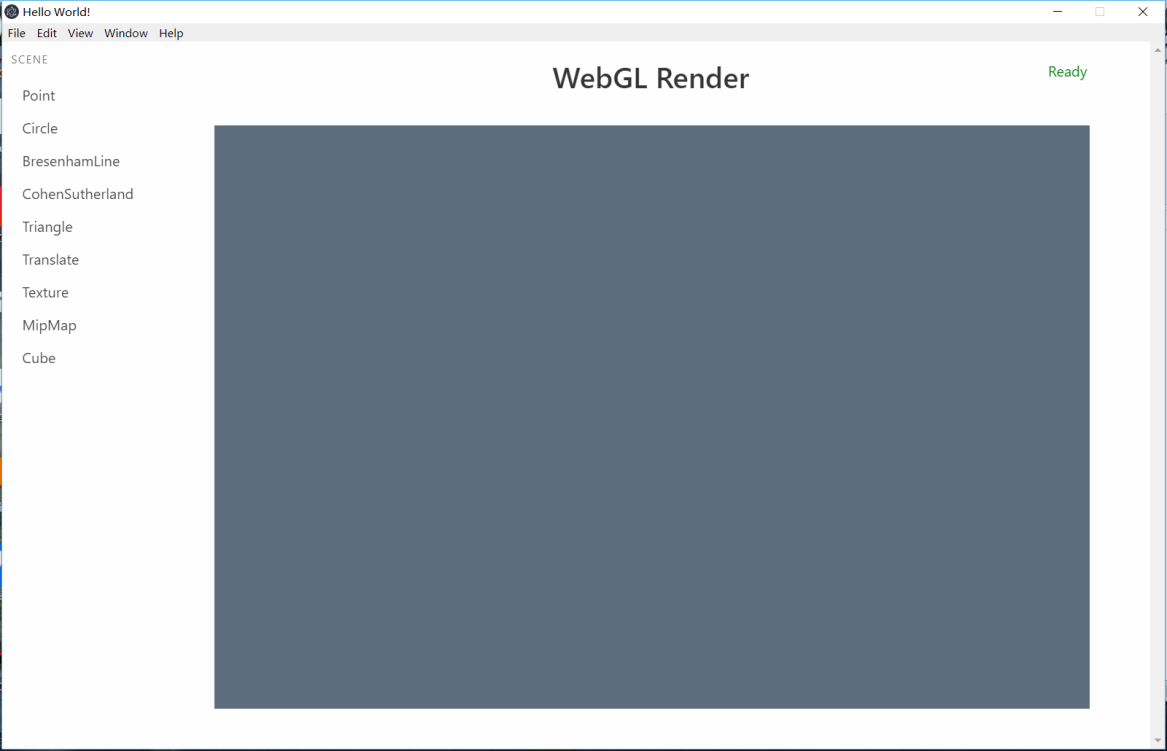
**四、功能设计说明**

下面我会按照优先级的顺序依次讲解这个软件渲染器涉及的功能与知识点，由简入难，里面我会穿插一些效果图片并且我会加入相等部分的反思内容为以后再次完善做准备。

完成这个项目我借鉴了大量晚上的参考博客，文章以及一些参考书，我会在文中加入适量的链接。

1. 我的渲染器：

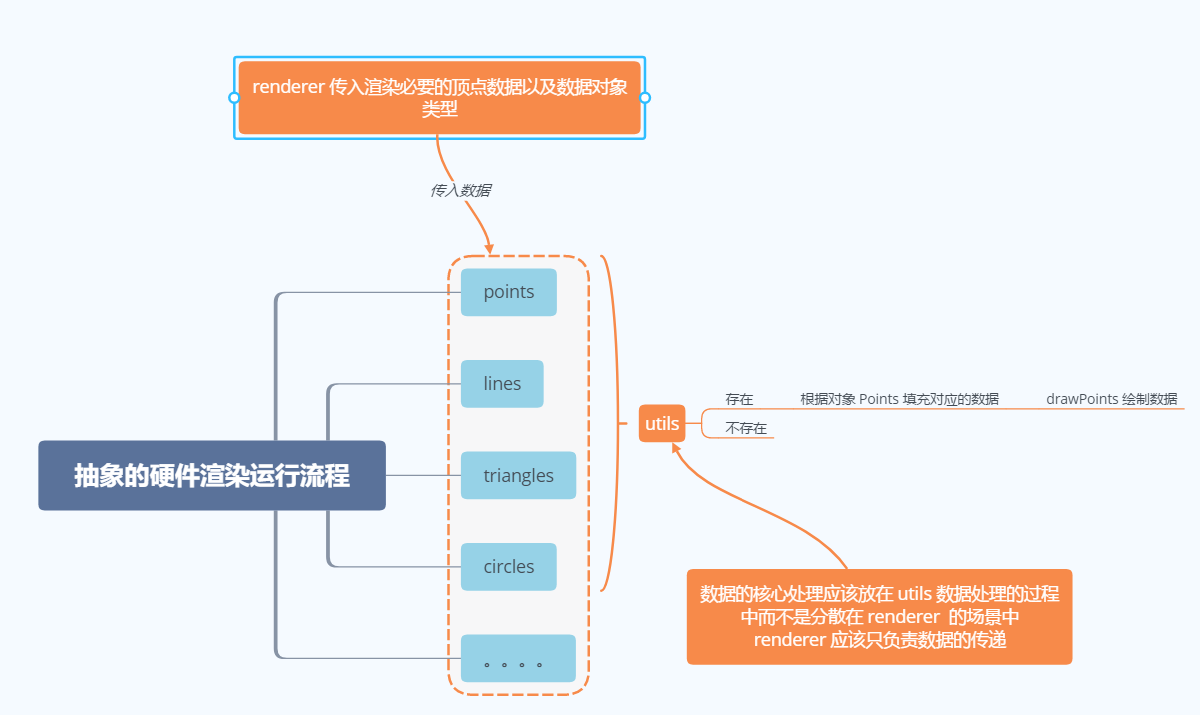
首先来看一下成品吧。（见图 1-1）



**图1-1成品**

在 [electron](https://electronjs.org/) 的结构基础上我们打包出了一个 exe 格式的软件包，结合一套比较成熟时尚的页面模版工具 [Bluma](https://bulma.io/) 指定summary 结构，当然了相当一部分的 electron 附带功能我目前没有使用，比如 menu 之类的设置。我设计了左边框栏，浏览者可以通过点击对应的场景链接跳转场景，右边的 Canvas 会显示当前场景应有的场景效果。通过浏览者也可以通过 View 中的 Toggle Develop Tools 打开后台控制器查看输出输入，当然了为了节省性能我几乎没做什么特殊的数据监控。

下面还要介绍一下我的数据结构，当然了这是我在多次重构之后设定好的版本，前期只完成了 2D 部分的绘制时软件的结构设计出现了偏差，后期考虑到软甲渲染器并非是使用项目而是研究类项目因此我不打算留出特殊接口，并且入口只进行传入数据。（结构见图 1-2）

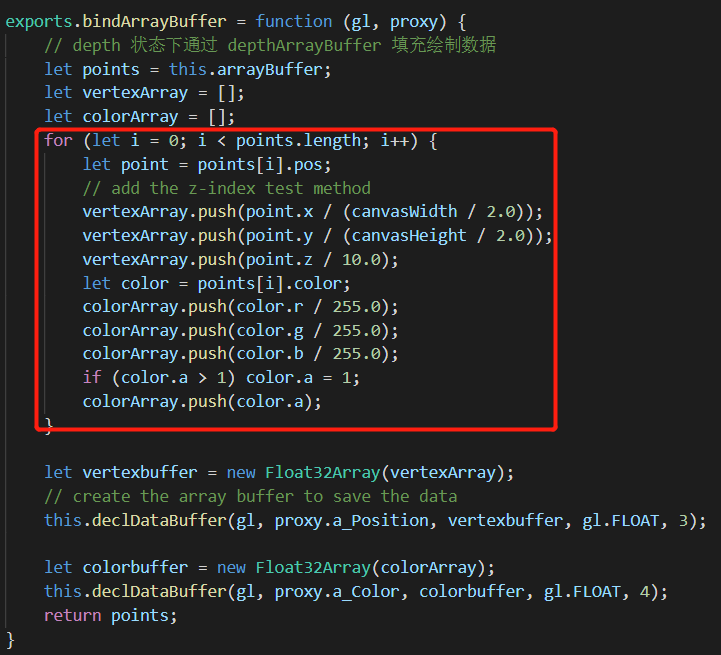


**图 1-2 软件结构设计**

顺便一提我使用的是 xmind 设计的数据结构。

2. 绘制点：

似乎这个部分没有什么特殊的数学计算，但是仍然有一个地方需要着重强调下，毕竟万丈高楼平地起，之后我们的所有功能都将是基于这个绘制点的功能实现的，因此作为所有功能的基石，设计的不好就很容易引起数据的计算混乱尤其是这个中等的小项目，因为我是在完成了所有的功能之后才开始写的论文内容，因此感触比较深的地方在于格式化数据点。Webgl 的点数据使用的是单位数据，以当前的 Canvas 为例，固定 960 \* 640 指的是当前的 Canvas 宽高的像素点数量，这个像素点数量在开始设计的时候我打算作为数据点定位的基础在通过最后传入 webgl 时一起格式化。（相关代码 2-1）



**图 2-1 utils 格式化代码展示**

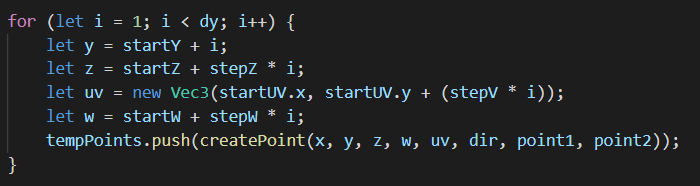
这样设计在最后有好处当然也有坏处。

|  |  |
| --- | --- |
| 设计好处 | 设计坏处 |
| 方便定位数据点（自我认为），意味着在绘制 Triangle 时我们更清楚我们绘制的三角形形状。 | 以 Color.a 为例这是一个比较复杂的数据属性，受到 engine 设计影响（一丢丢）我习惯 rgb 使用 256 而 a 使用 [0-1] 导致很多时候数据不统一问题。 |
| 方便计算步长，因为传入的为 int 数据，减少内存损耗虽然 js 默认存 float 但是小数的计算是会影响到计算时间的。  尤其是在 C++ 之类的语言中，仍然是建议整型计算更加优秀。 | 上面说道的数据不统一问题，在加入投影矩阵之后更加能够体现出来，因为投影矩阵最终计算获得的都是 规范立方体（CVV）下的坐标点数据格式，因此我必须在设计透视投影的 3D 场景中传入 [-1, 1] 的点数据。 |

结论自然不言而喻，需要规划好数据格式化的位置成为了整个 Point 设计的重点。这里需要紧记考虑到 Line 计算插值以及步长的时候使用的应该是 Canvas 整型坐标 （下文称 pixel 坐标）而最后绘制阶段传入 webgl 坐标数据，格式化单独作为一个函数提取出来，并且单独留有反向格式化函数确保数据格式的灵活性。传入数据按照之前的要求若是单纯的以 3D 渲染器来看，顶点传入数据以 [-1, 1] 为单位，若是可以切换数据形式那么 2D 渲染部分还是按照 pixel 坐标数据传参（[-480, 480], [-320, 320]）。

3. 绘制线段

a.绘制线段，作为整个软件结构的第二个大要点，设计相关的内容包含了之后的 Triangle 绘制，因为需要填充像素点，这里我参考了[知乎](https://zhuanlan.zhihu.com/p/43537323)《游戏开发启示录》上的相关软件设计，这里使用 Bresenham 算法进行线段的绘制，算法原理实际上是逐步长比较当前线段的斜率，以 dx < dy 并且 x1 不等于 x2, y1 不等于 y2 为例，我习惯使用从左到右的填充绘制方案（有些参考资料里有人喜欢用从上到下的方案，实际效果类似），判断好 dx, dy 的大小之后，确定两点之间的长度为多少像素，之后开始遍历直到最左边的初始点数值到达右边端点，其中 y 值保持 stepY = 1 or -1 （根据斜率而定），每次遍历都将计算当前的点与初始点的斜率与最终线段斜率的大小用于保证 y 方向的进步。（详细代码见图 3-1）



**图 3-1 遍历代码展示**

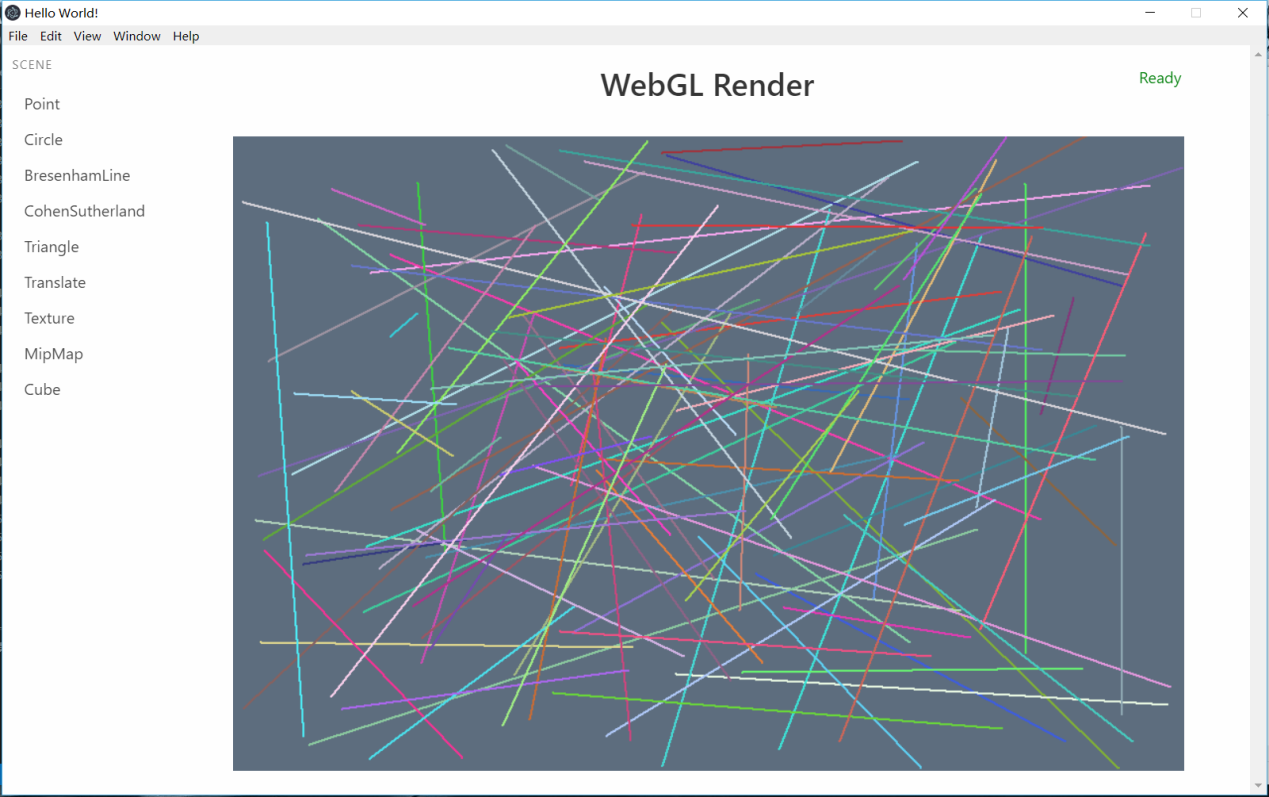
设计初期只是一个简单的线段算法实际上没有什么需要考虑的，唯一要注意的是 四种不同的情况，同 x 或 同 y 当然还有考虑 dx, dy 长度的问题，不同的长度最后遍历的步长依据就会不同。

因为还要设计后期的 triangle 绘制，我们有一个要点需要保证，就是线段绘制信息的同向性，不然在填充数据的时候我们不能判断 y 的上下进步方向，这导致关联的缺陷问题。

无论是 bottom to top 还是 left to right 都会存在线段顺序的问题，这个时候自创的方法就只能消耗性能去完善效果，增加循环数量了但是最后我们还是需要保证传入的点的顺序，因此最后我们只是保证了一个填充条件，线段一定是从由左到右进行绘制的。

在填充 Triangle 的时候，我们采用的方案最后需要判断第二长和第三长的边，必须保证循环边内容是对应正确的，也就是上文中提到的 y 进步的为题，为此还需要考虑一些节省效率的办法，比如直接一个循环内将两条边都遍历一次，保证可以三角形内部可以填充上，当然这都是后话。

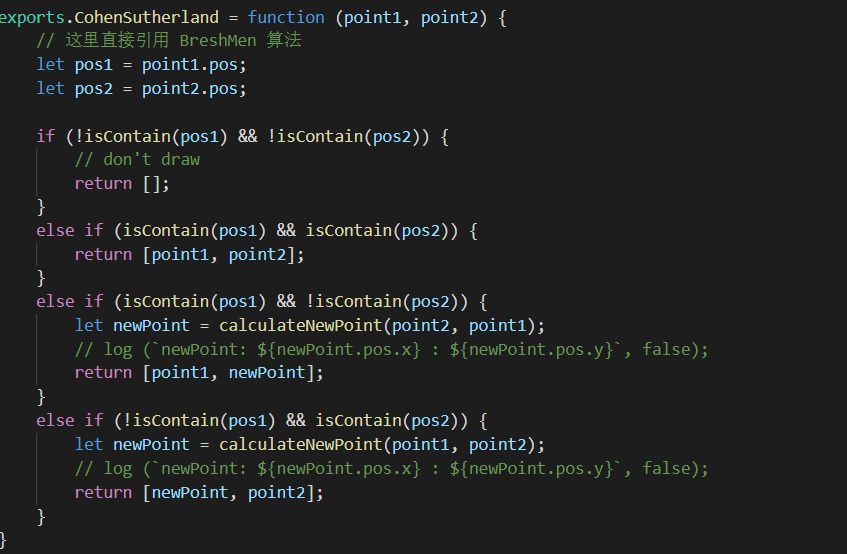
最后我们在场景中绘制了一百条直线（如图 3-2）



**图 3-2 BresenHam 场景展示**

之后考虑第二个场景部分以及第二、三的线段功能。

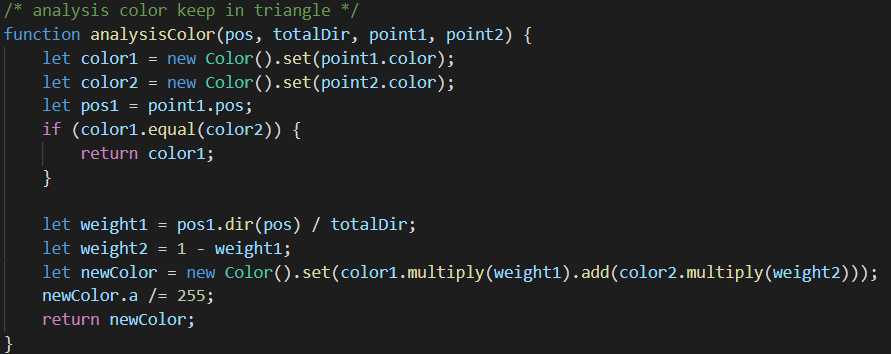
b.CohenSutherLand 是一个剪裁方案，当然了我实现的并不好，因为可以实现这个功能的方案太多了，因此我们只算是借了个名字吧，剪裁方案原理还是相同的，确保我们随机生成的点在显示区域内，不再区域内的点将被剔除，并且提出过后由新的在相同线段上的边界点替代。（代码见图 3-3）



**图 3-3 剪裁方案代码部分展示**

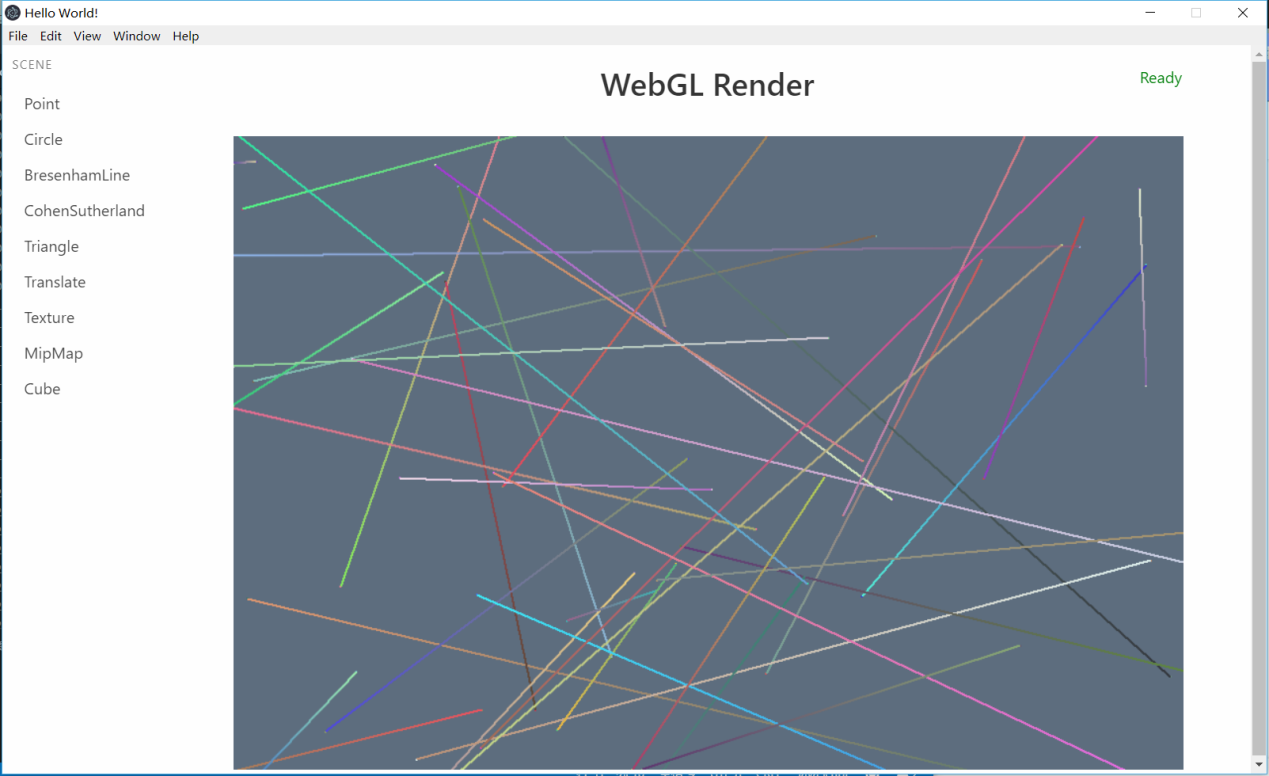
以上，这个方案参考的还是上文中提到的知乎参考，但是实际的实现方案大不相同，原理相似而已，线段的填充方案还是按照 Bresenham 的填充方案，因此这个部分跳过。

c.第三个功能，颜色的渐变，毕竟线段顶点的颜色应该是可以变换的，当然了这个功能并不算重要因为在 Triangle 需要根据三个顶点重新计算颜色值时，需要重新设计计算部分，因此这个部分写出来当作功能效果实验。基本原理是 weight ，根据当前的 point 的位置在整个线段的百分比，获取对应的权重值，使用两个断点分别乘以权重 weight1 以及 1 - weight1 获取对应的显示颜色。（代码如图 3-4）



**图 3-4 线段颜色分析函数**

整合两个功能最终获得 CohenSutherLand 场景。（显示效果如图 3-5）



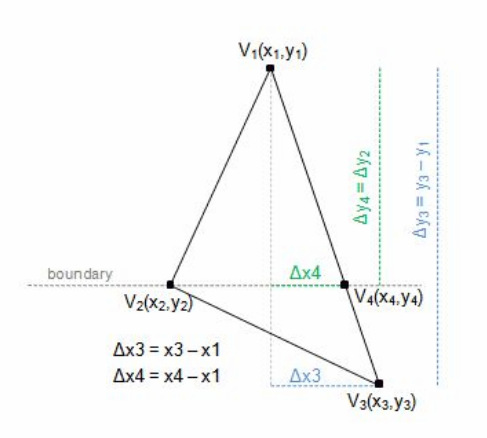
**图 3-5 CohenSutherLand场景显示效果**

4. 绘制三角形

绘制三角形，绘制三角形有两种方案，一种是我设计的当然不够好，一种是参考上文知乎中的设计，分段性设计，我着重会阐释后者的实现原理以及实现效果。

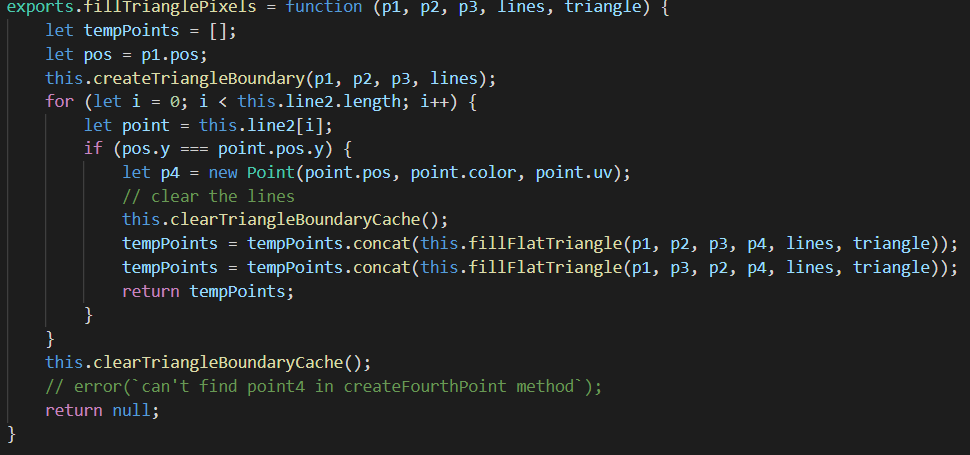
a.myownTriangleFilled 方案，是我自己设计的一个方案，原理是排序三角形的三个顶点顺序后利用最长变进行遍历，第二场以及第三长边步长一定为最长线段步长之和因此我需要通过排序三角形顶点，在计算最长边最终获取可遍历线段。这个方案不算特别好，虽然也能够适应所有三角形，但是前期的顶点排序并不是我们想要的，甚至有些累赘，但是缺少了顶点排序，这个方案又不能够成立，因此这个方案因为过于繁琐被我放弃。

b.normalTriangleFilled 方案相比之下就更加的遍历，我参考了两个较为权威的实现方案原理大致相同，所有的三角形都能够被拆分为两个三角形（如图 4-1）



**图 4-1 三角形剪切方案**

当填充平顶三角形是最轻松的，因为平顶三角形是以最高点为基点的，这样能够帮助我们确认步长，并且所有的三角形都具有以上的特性时，就可以被我们加以利用，分为两个方法，一个用于检查拆分三角形，一个用于填充三角形，因为填充的都是同类型的三角形这就避免了我自己设计方案中提到的顶点排序问题。（拆分三角形代码见图 4-2）

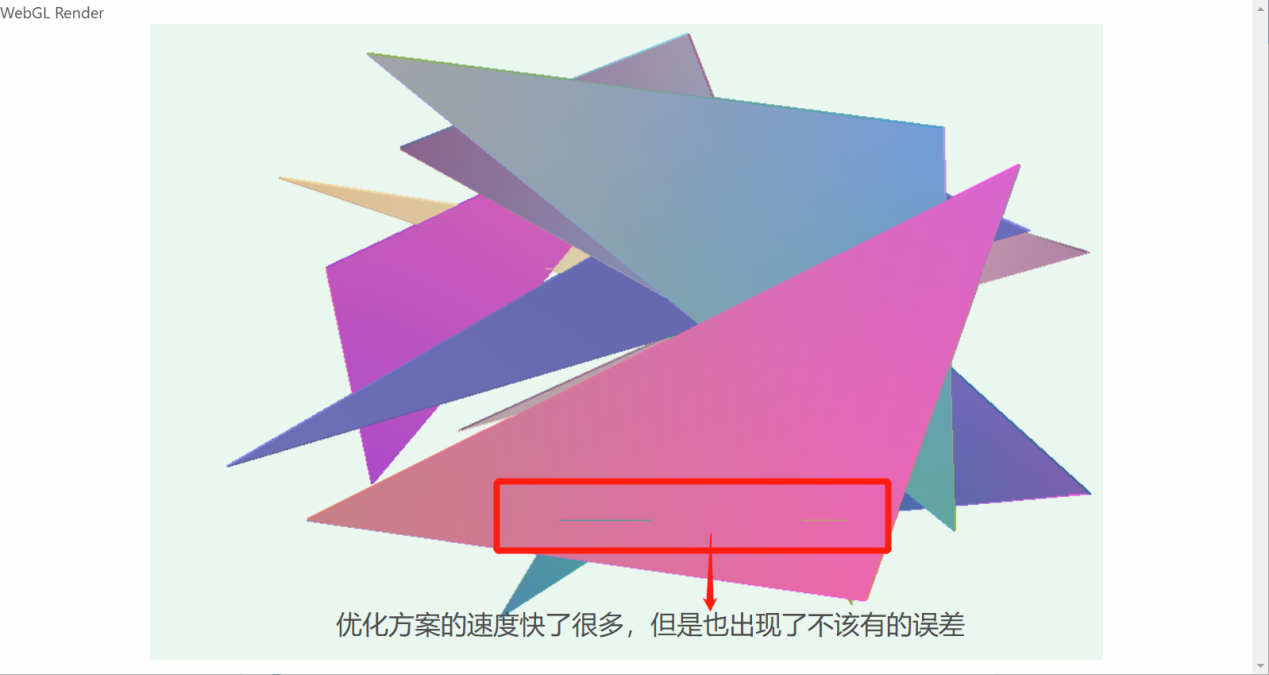


**图 4-2 三角形拆分代码部分展示**

当然在拆分三角形之前我们还是需要知道中间点，有些方案人喜欢使用 x 判断，拆分出左右两个三角形，但是同样要获取中间顶点，确定拆分的对应 y 或 x。

在填充平顶三角形之前，我们还需要获取三个边界数据，这样我们才能够进行正确的遍历，这里还有一个要点需要注意，和上方的确定中间值一样，我们仍然需要确定在填充时三角形的顶点，这里有两个方案，一是分别写出上顶点平底三角形方法和下顶点平顶三角形方法，这里我为了节省码量从上文的 fillFlatTriangle 方案中可以看出我特意调整了传点顺序，这是必要的处理。

最后我们填充三角形函数，查看最后效果。（如图 4-4）

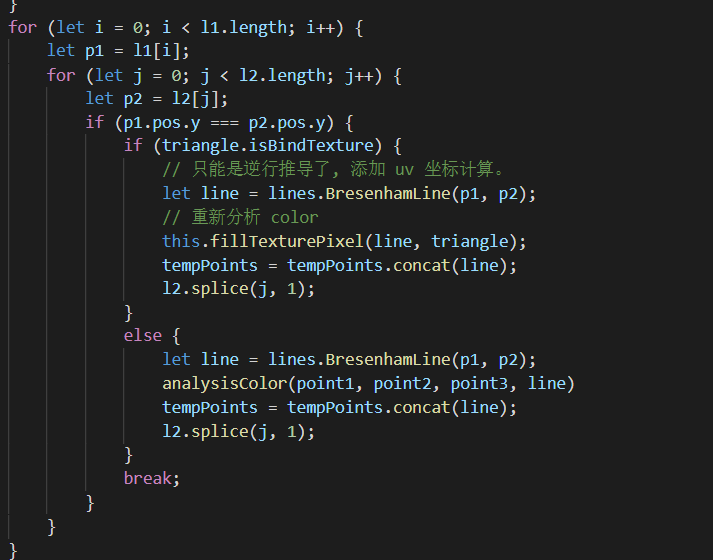


**图 4-4 初期的三角形绘制场景显示**

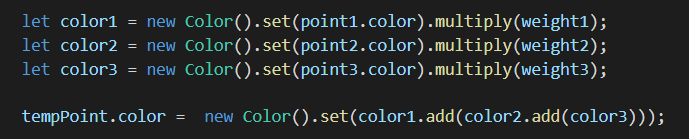
如图中我们似乎遇到了 bug，当然后期的我已经解决了当时我在处理 depth 检测的数据方案，优化结果之后发现出现了数据丢失的情况，起初以为是三角形绘制的时候遍历误差，但是我是整型遍历，不会出现这个问题，之后我甩锅给了 js 自带的误差数据，比如 0.1 + 0.2 = 0.3000000…4 的问题，这归结与 js 的全 float 转换为二进制的运算方式原本是为了加快效率的方案，最后导致 js 天生自带的 bug （这个知识点作为参考因素简单提及）。但是我错了，实际上很明显的可以判断出，这是边界数据丢失的问题，但是起初查看场景发现只有部分线段的丢失，导致我没有往这个方面思考。后期修改 texture 错误时暴露了这个问题。

这也是软件渲染器比较难写的原因之一吧，在只有少量的数据发生错误时，你已经存有了大量的数据信息，你无法通过断点直接判断原因，并且因为计算和遍历数据过大的原因，采用 console.log 的方案会导致 electron 的编辑器直接卡死，这个问题到软件编写结束我也没有特别好的解决方案，因此只有在最后于到特殊情况进行关联时才能够发现一些问题。

展示一下三角形像素点填充算法（见图 4-5），代码涉及到后期的 texture 填充，因为是我后期截的图因此 texture 的部分下文会再做讲解，这里还涉及到上文线段绘制时的一个相同的功能颜色渐变，实现原理还是如出一辙，方位权重计算各个顶点的颜色。（详细代码见图 4-6）



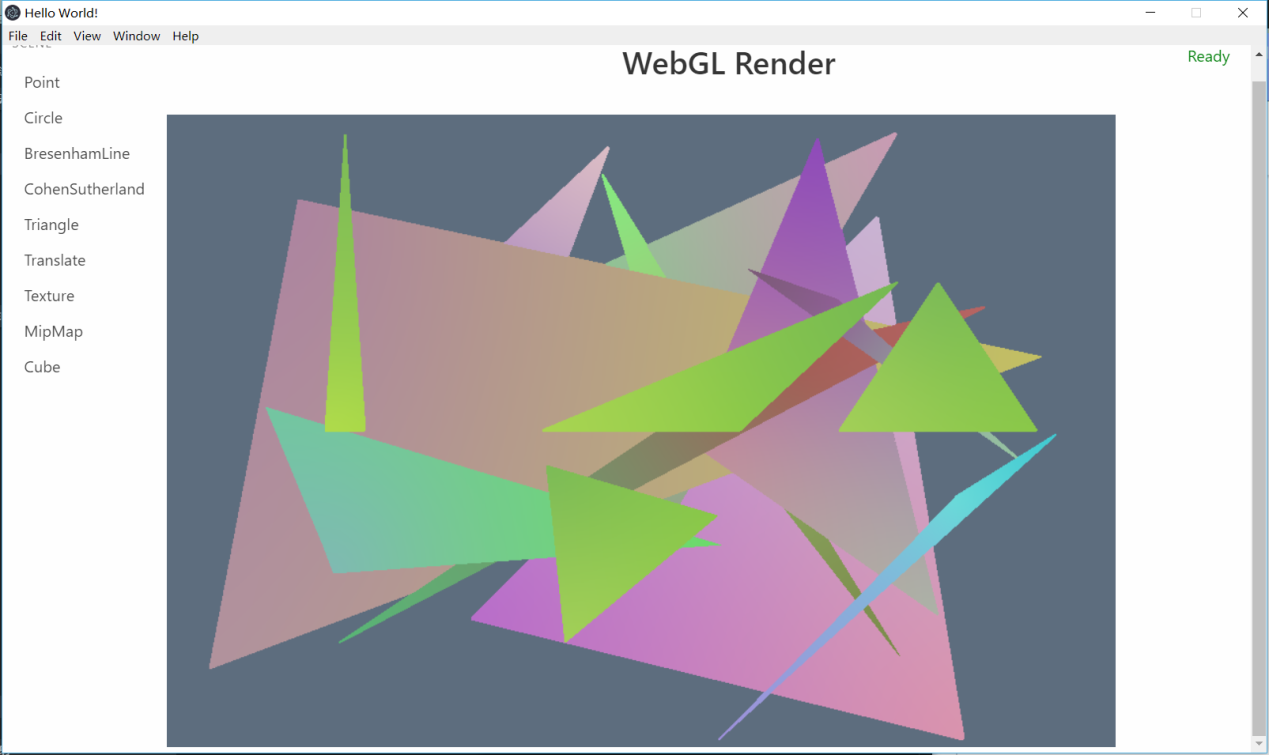
**图 4-4 三角形像素填充功能展示**



**图 4-5 三角形权重颜色计算部分代码展示**

综上，一个基本的三角形填充功能就已经完成了，这个功能涉及到后期的两个大的要点，仿射变换和 3D 图形。

c.最后这一节再提及一下我们的数据优化方案，优化内容为数据遍历次数减少对比数量，因为绘制大量的三角形需要大量的像素数据，虽然目前我们仍然没有涉及到 3D 部分的计算，但是我们仍然需要清楚 3D 场景中 2D 所不具有的概念 “深度”。在 webgl 中有深度检测的概念，为了是 3D 场景物体排除绘制的先后顺序概念，能够以实际深度进行像素数据覆盖。到目前位置，假设我还没有设计以及实现我下文将会提到的数据优化方案，先想象一下我又十数个三角形在当前的场景中，那么大量的三角形填充像素数据将会在对比深度值时会严重的影响到渲染器性能，这也是我实际体会过，十个三角形大约计算时间为 n ^ 2 以总像素点个数为 n 那么这个消耗无疑是巨大的，那么具体的优化方案是什么呢？下午 Cube 一章的时候会详细介绍。下面查看一下 Triangle 场景最终显示效果（见图 4-6）



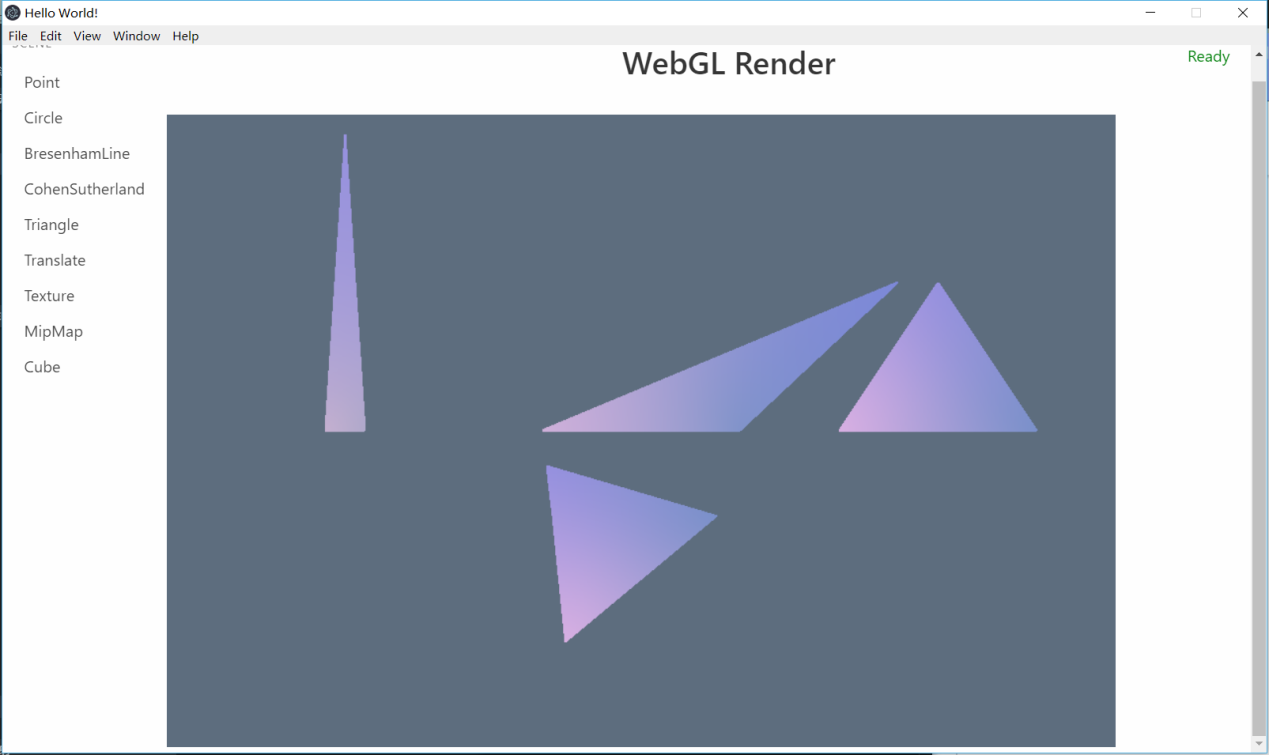
**图 4-6 Triangle 场景正确显示效果**

5. 绘制多个三角形，仿射变换

到目前为止实际上整个项目还没有开始涉及 3D 部分的内容计算，但是 matrix 的数据格式已经给了我们很大的提醒，这里我参考了 《webgl 编程指南》整本书的内容，这本书也作为我入门渲染的基础书籍教会了我很多的基本概念，这里我使用了该书的 matrix 库（详见 cuon-matrix.js）， 计算是以 4 元方阵为单位，那么考虑到 3D 内容的仿射变换，提醒我们当前的数据需要意识到 3D 格式的存在了。因此我将前文的 Vec2 数据改为了 Vec3 数据。

下面还需要介绍一下仿射变换原理。详细内容可以参考网上很多优秀的文章博客。这里不做链接了。仿射变换根源是线性代数，很长一段时间在学校里面，我只知道矩阵计算方法，但是经常搞不明白矩阵与数据之间的线性关系，研究了渲染之后发现实际上涉及 x,y,z 的变化都是线性的变化，而矩阵的存在只是方便的存储数据并根据计算原理获得正确的线性表达式。那么为什么 3D 的计算使用 4 元方阵计算呢？这里我自己认为的解释是 w 的引用，我参考了一篇相当不错的 [blog](https://blog.csdn.net/popy007/article/details/1797121)。三维的世界里，Vec3 我们无法判断它是向量还是顶点，那么引入了 w 的概念，作为辨识。这是理由之一，当随着计算机图形学的不断进步，最后发现 w 的作用还很大。尤其是在下文中我们涉及的投影矩阵计算中，正确的数据就需要 w 存储数据作为必要依据。（这个下文再提）

最后来看一下 Translate 场景效果。（见图 5-1）



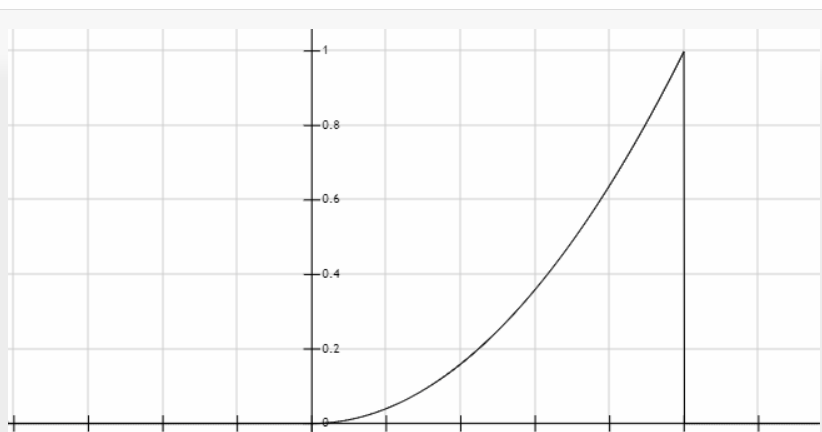
**图 5-1 Translate 场景正确显示效果**

上图中变换包含了size, rotate, skew, translate 这里我同称使用 Translate 代替了，上图中的显示效果并不包含透视投影效果，但是这里的计算为后序的 Cube 绘制时计算顶点数据提供了基本数学运算保证。

6. 绘制 Cube

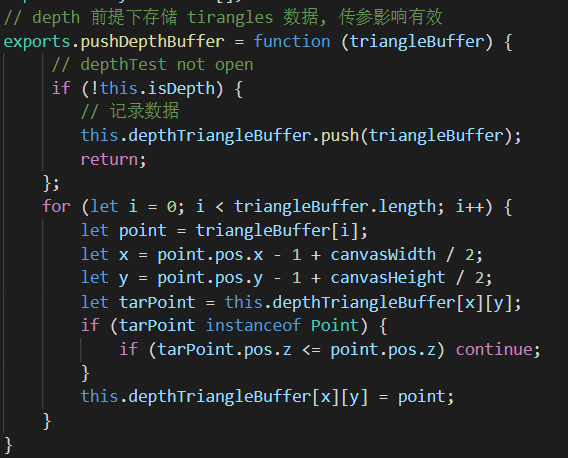
这一个功能涉及的部分比较庞大，我们分要点进行简略阐释。

a.首先是上文中提及的数据填充优化方案，关于优化方案有两个不同的版本，两个方案都各有利弊，首先是方案一，覆盖三角遍历方案。每当绘制一个三角形时就需要判断已经存在的三角形中是否存在与之相交的三角形，若存在就需要遍历两个三角形的所有像素数据进行深度判断，这里不做代码解析，详情可以到我的[项目页面](https://github.com/ColinCollins/RenderingWork)进行查看。这个方案的最大的弊端就是遍历耗时，比起最初的方案只是减少了部分的遍历数量，在三角形过多的情况下，遍历耗时大概的变化图形是倒数变化（见图 6-1）。明显这个方案并不优秀，但是初次之外也没什么其他的缺点了。



**图 6-1 优化方案一耗时数学大致图形**

优化方案二，提前声明一个depthBuffer 像素数据缓冲区，虽然名称为 depthBuffer 但实际上存储的是整个视窗的像素数据信息，对应的尺寸为 960 \* 640。这个方案要求是在绘制三角形时进行判断是否开启深度检测，开启后将数据传入，只遍历当前三角形数据，并判断当前遍历像素点位置是否已经存在像素点，根据存储的深度 z 信息，记录是否当前的像素数据要被传入绘制数据数组。（代码见图 6-2）

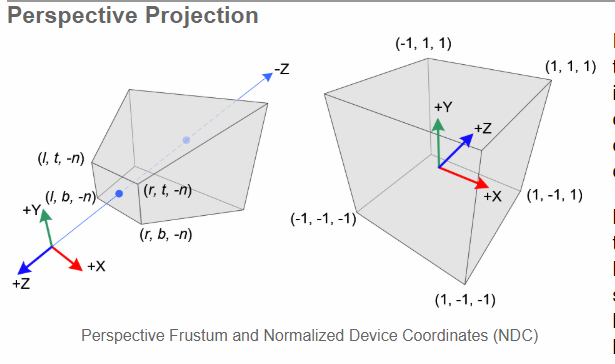


**图 6-2 深度缓冲检测代码部分展示**

这个方案好处是大大的优化了遍历效率，但是缺点是我们不存在超出屏幕视图尺寸的三角形，因为无法定位遍历位置（当然这里还是有优化解决方案的）。

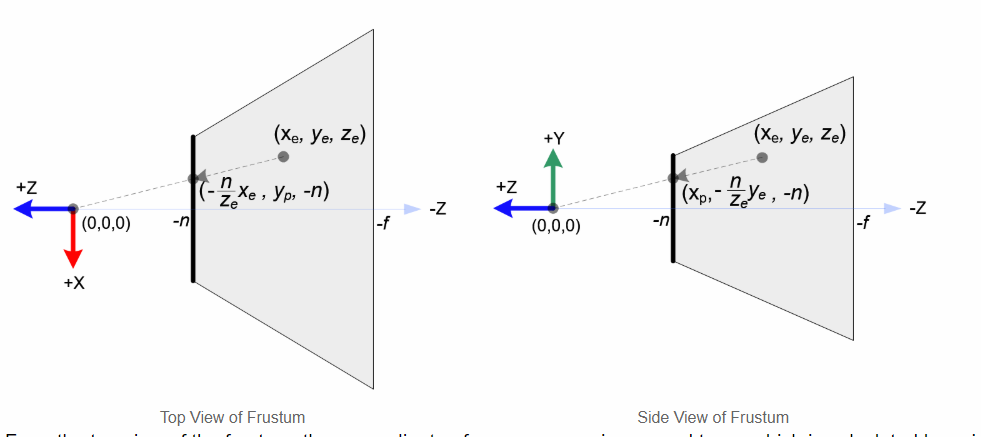
总体来说这个方案大大优于第一个方案，因此我们采用这个方案（似乎这也是个普遍的解决方案）。

b.透视投影矩阵，这是一个相当复杂的矩阵，实际是为了让 3D 空间下的图形投影到我们 2D 的视图窗口上，这里我参考了《计算机图形学》，《webgl 编程指南》，以及一篇相当不错的[文章](http://www.songho.ca/opengl/gl_projectionmatrix.html)。这里借用一张图6-3用于阐释一下投影原理。



**图 6-3透视投影转换原理**

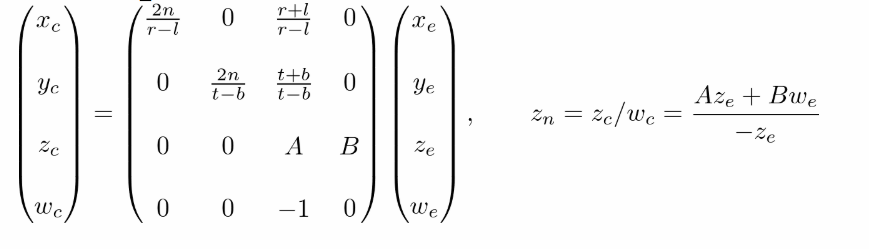
实际上原理还是上文中我们提到的线性变换原理，通过对应平面的线性计算，带入取之区间就能够获得我们想要的线性方程并带入矩阵当中。（见图 6-4）



**图 6-4 平面线性计算**

相对来说比较复杂的情况是关于 z ，起初我纠结了很久的 a + bz / z 为什么会是这个公式，之后我又参考了一篇不错的[文章](https://blog.csdn.net/popy007/article/details/1797121)。

那么 az + b 实际上应该是为了归一化。最后留得三个对象的线性公式就是透视投影的公式。结合这两句话，查看图 6-5。



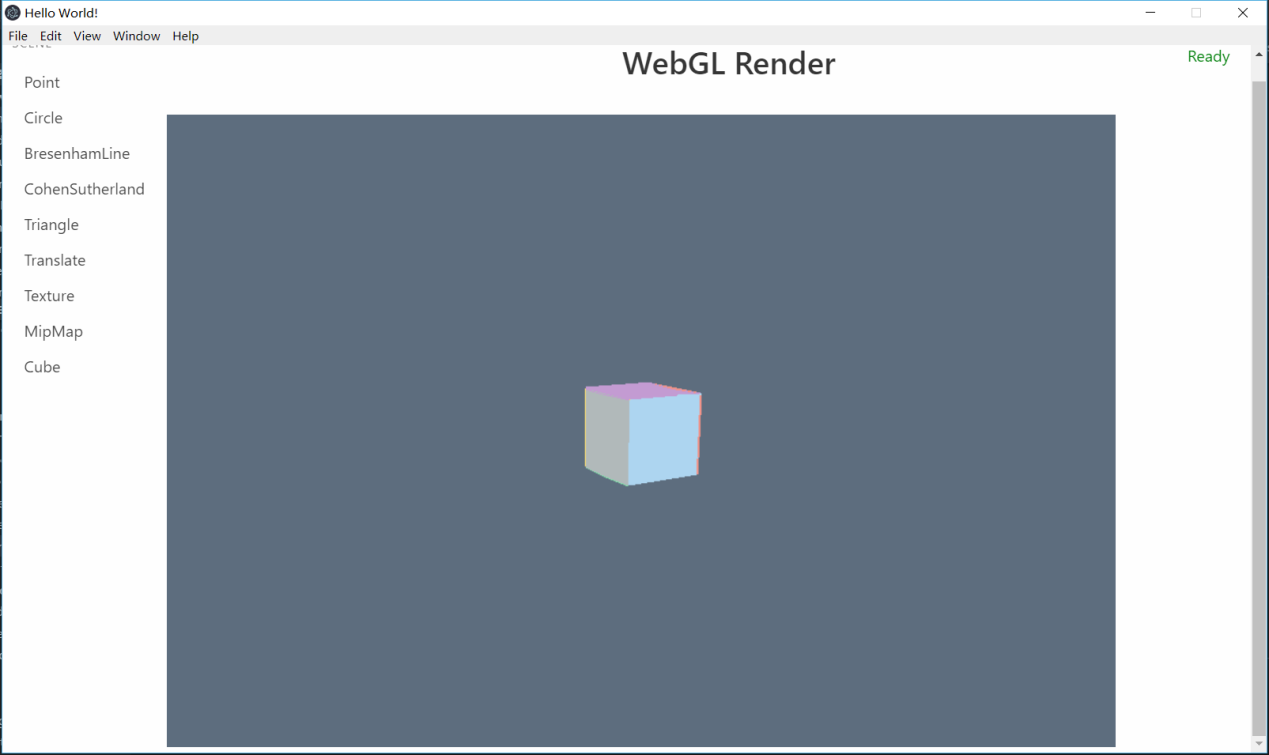
**图 6-5 透视投影矩阵推到演示**

这也是我在重写深度检测是意识到的问题，投影过后的 x, y 信息与 z 不具备对应的线性关系，而矩阵中正好发现根据相似三角形原理 x, y 与 1 / z 有线性关系（1 / z 是因为 near 总是为 1 的所以保留 1），归功于相似三角形，那么根据这两个特性，在一开始获取了 x, y 与 z 的关系之后，再推到 z 数据关系。w 是为了方便存储 z 原始值，因为这在后面的计算有用，所以 [3][3] 的位置保留了 -1 用于存储 z 并且用于线性关系成立的推到。顺便一提深度数据应该是以 [-1, 1] 为区间，这时候不应该按照[0, 1] 的思路计算的，这是错误的，至少从单位上来看是错误的。

接下里考虑矩阵相乘问题，通过参考韦易笑工程（[mini3D 工程链接](https://github.com/skywind3000/mini3d)），可以得知 update\_tranfrom 在创建 box 之前，那意味着矩阵相乘之后的变化不影响 CVV 剪裁检测的，那么得知矩阵之间对于变量的影响是相对独立的，虽然数值上有偏差但是综合矩阵并不会改变最后的数据对应结论。

通过透视投影矩阵的 w 存储的 z 后续我们会引用推导 uv 纹理坐标的差值以及透视纠正问题。

c.最后深度缓冲数据计算，这意味着我们要在 matrix 下计算当前投影像素点的真实深度值。由于上文中我们已经知道了 w 实际上等于 z，因此完全可以使用 w 作为深度检测的判断值，但是在我的代码中你会发现我仍然使用了 z 因为 matrix 与 z 保持的线性关系是一致的，那么排除误差，是不影响深度检测数据的结果的（当然还是建议使用 w 可以使用相对准确值的话还是用 w）。 下面来看一下最终场景效果见图 6-6。



**图 6-6 Cube 场景效果展示**

实际上这是一个动态的场景，建议还是场景内观看。Cube 会旋转，但是因为性能问题，旋转效率较低大概只有 5 – 10 帧左右吧，等待以后继续优化。

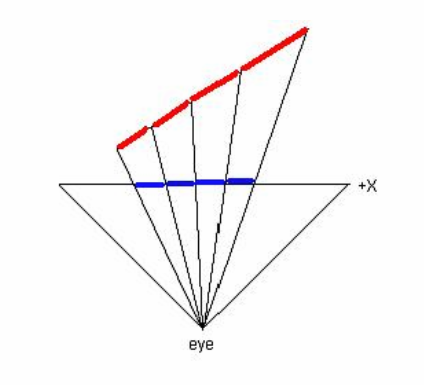
7. 绘制 Texture

这一节建立在正确的透视投影前提下，计算 uv 纹理坐标并根据对应的数据从图片信息中获取像素数据传入图像。这一节存在三个难度要点，图片像素数据的获取，uv 纹理坐标的变换，以及投影像素获取对应真确的 uv 像素数据。

a.首先是获取图片像素数据，这里又体现出 electron 的好处了，可以直接读取本地文件，避免了通过设置浏览器权限获取图片的弊端，也节省了不少代码内容。

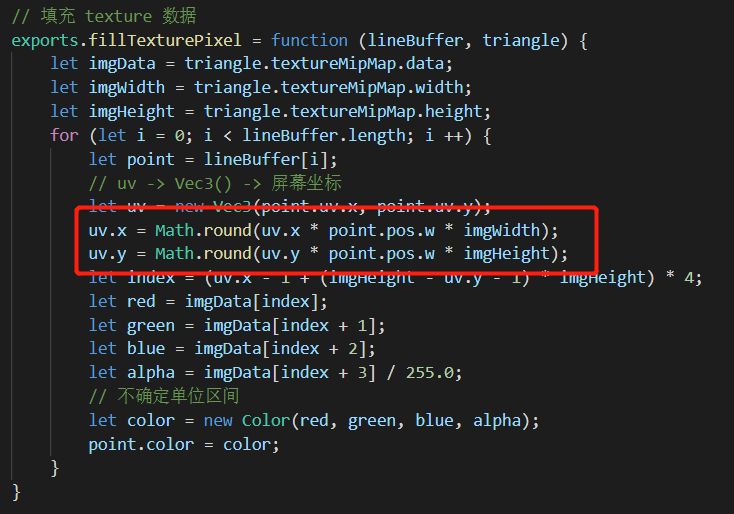
b.纹理坐标是否需要变换？随着点的变化实际上 uv 坐标是不需要变换的，甚至我们需要保留对应的 uv 坐标数据确保最后读取的 index 数据是正确的。

c.绘制 Texture 如何获取对应的 uv 像素数据，这里涉及到一个比较复杂的问题，纹理插值映射，我也从网上找到了不错的[参考文章](https://blog.csdn.net/popy007/article/details/5570803)。实际上在做投影之后，再去获取 uv 纹理坐标时就会遇到这种情况（如图 7-1）



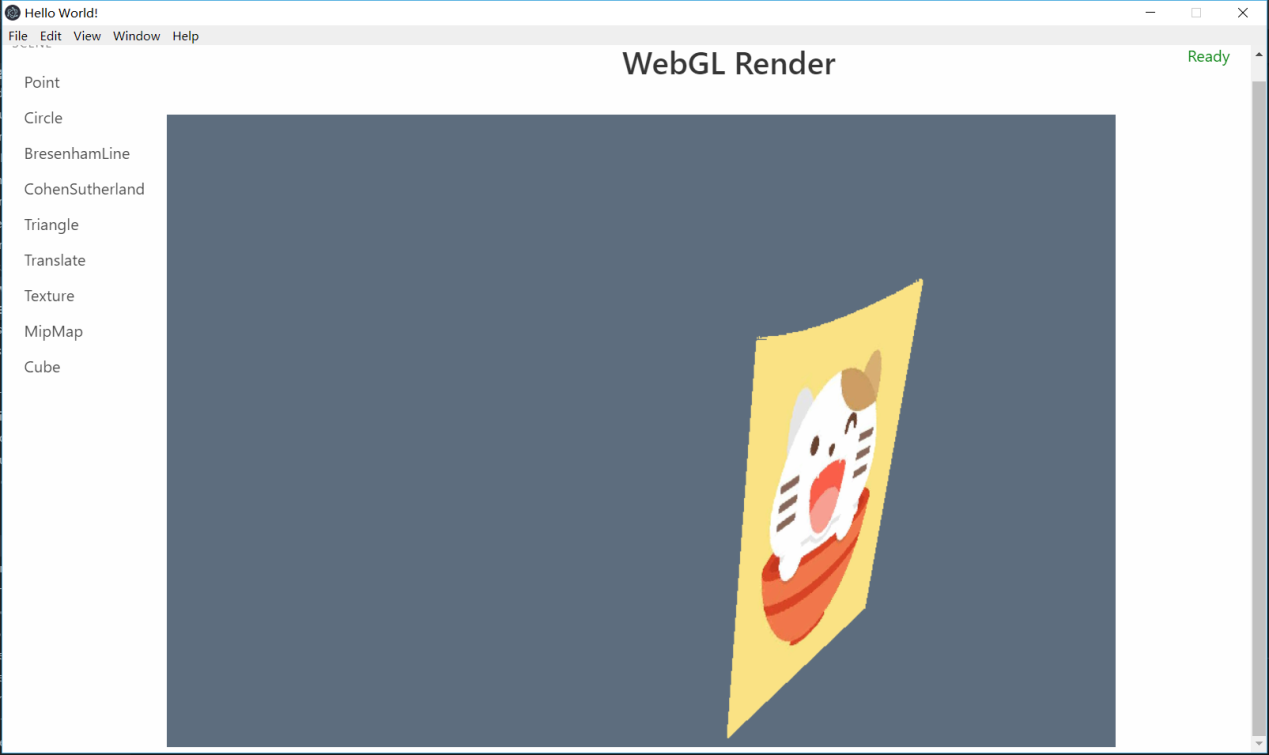
**图 7-1 投影对应抽象图**

那么导致最后获取对应纹理像素时会产生偏差，因为错误的对应关系以及步长，那么纹理插值计算就是解决了当前的这个问题。解决方案很简单还是利用 1 / z 的线性关系，实际上 uv 和 1 / z 也是有线性关系的，那么通过 uv / z 获取转换后投影的 uv 坐标再通过插值取得每个封装对应点的 uv 坐标，我们已经计算过 w 的差值，也就是深度差值，那么再将 uv \* z（或者称w）获取真实 uv 坐标下对应的点，最后乘以图片本身的宽高作为的 index 值获得正确索引获取像素（代码展示见7-2）。



**图 7-2 texture 像素填充方法**

具体插值的计算在 Bresenham 生成线段对应像素的时候已经处理了。那么这样我们就获取了正确的场景效果（见图 7-3）。

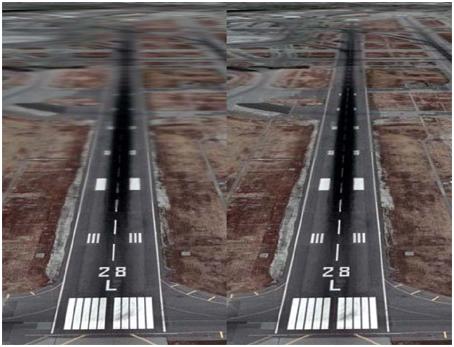


**图 7-3 Texture 场景效果展示**

虽然这张的篇幅不多，但是在纹理映射插值的部分网上介绍五花八门，理解概念就花了不少时间，实际上这一节才是最复杂的一节，前文涉及到的所有数据以及变换都在这一节有所体现。最终我们可以通过这个方案绘制一个 Cube，但是有点懒了，知道效果并且确实可以实现就可以了。

8. 计算 mipmap

这一节首先先确认一下 mipmap 的概念，mipmap 是一组由同一张纹理生成的金字塔式的纹理集。用于处理场景中实际深度过高，导致采样出现绘制模糊现象的一种解决方案基础数据集（见图 8-1）。

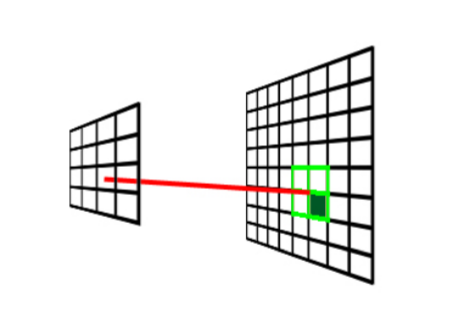


**图 8-1 模糊效果展示**

原理是因为深度值的变换，如上图 uv 插值计算时的那张图，取样 uv 时跨度过大导致图片开始模糊不清。从 mipmap 中获取一些较小尺寸的 uv 对象可以有效的避免这个模糊，因为较小的 uv 相当于拉大的纹理间距，减小了投影与实际纹理的 uv 跨度。

当然这节我不去实现如何采样，而是实现 mipmap 的生成以及 box 滤波。

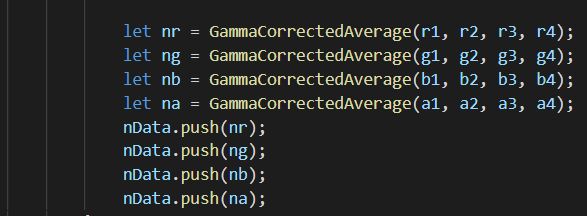
Box 滤波是个很基础的概念，也被广泛的应用在计算机图形学中（见图 8-2）。



**图 8-3 box filter**

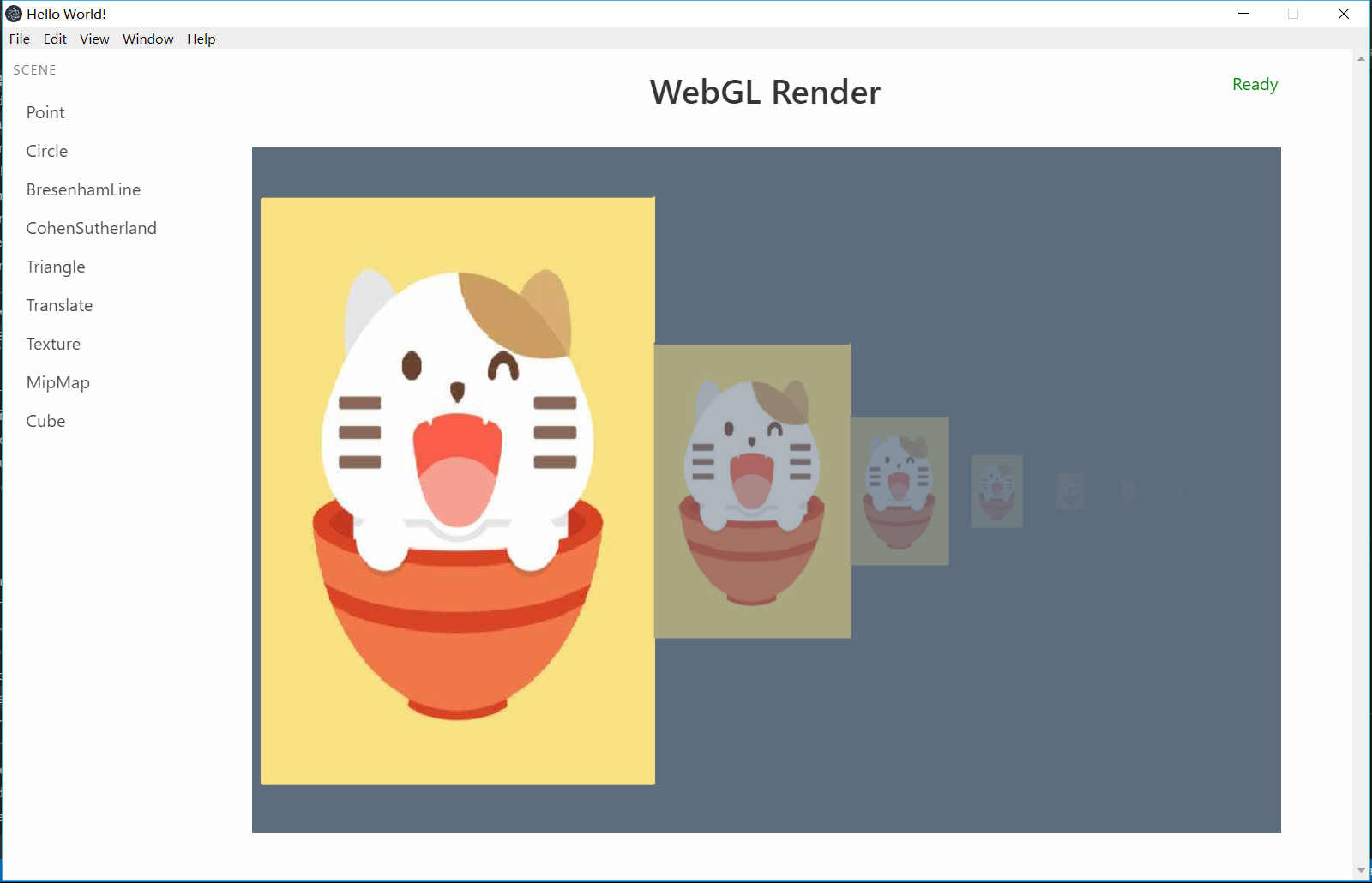
盒子滤波实际上就是像素数据的取平均，深入探讨的话还有非整型像素点取特定非中心位置像素信息时的 box 权重滤波计算。

详细实现代码如图 8-4。

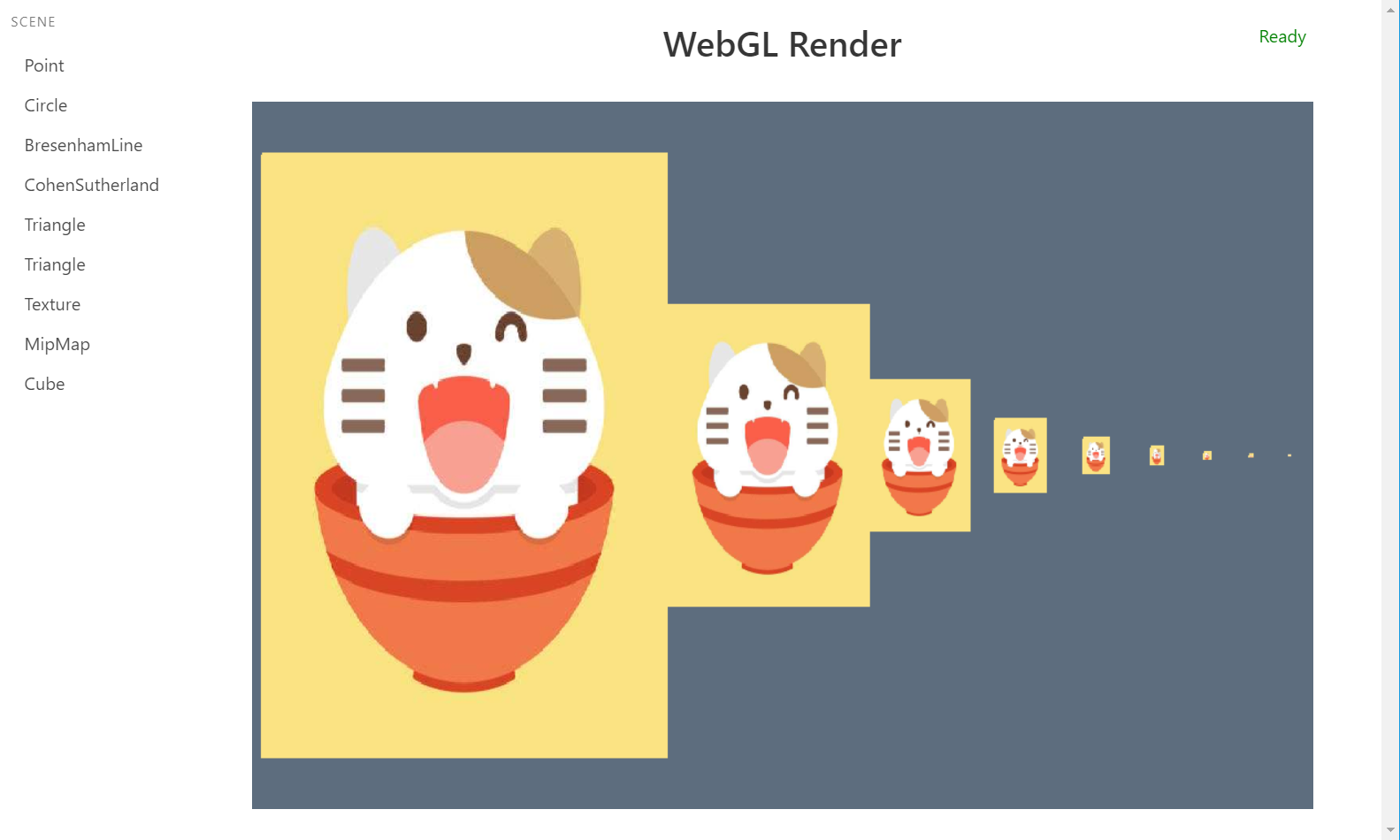


**图 8-4 box 滤波代码部分展示**

这里虽然我方法名称取了 Gamma 校正，但是因为最终效果并不好所以我并没有采用校正方案。下面展示使用 Gamma 校正与不使用 Gamma 校正两种方案的效果图 8-5 和 8-6。



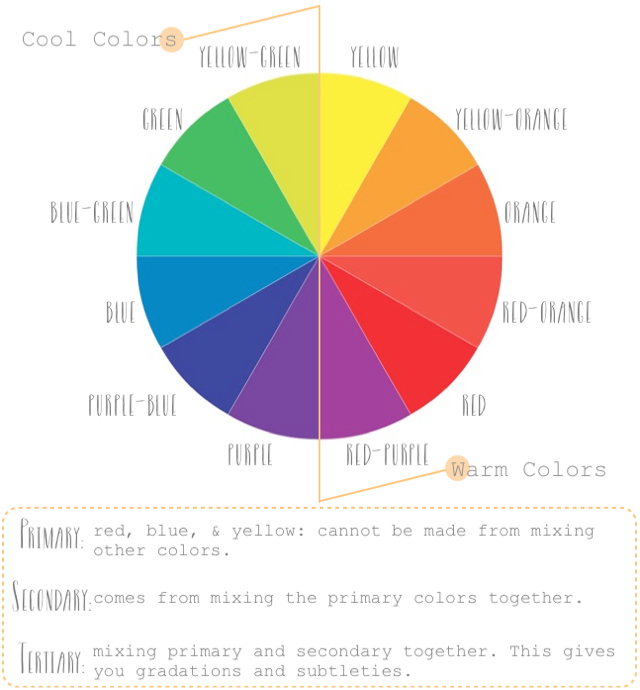
**图 8-5 使用 Gamma 校正**



**图 8-6 不实用 Gamma 校正效果**

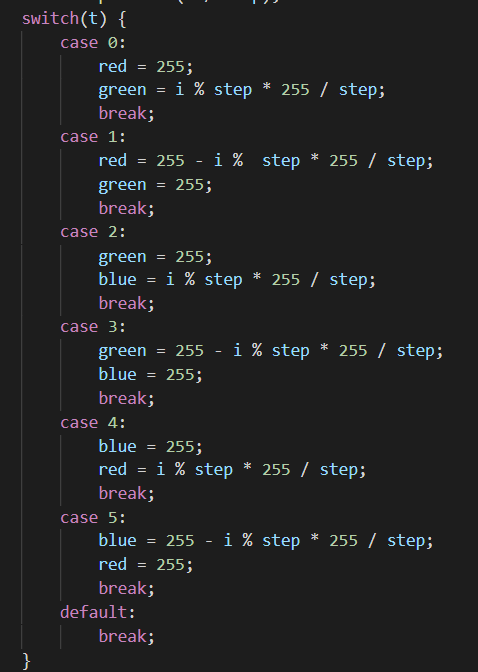
9. 绘制 circle

这一节作为额外内容的非重点，用于绘制圆，采用的方案很简单，填充三角形。我们希望获得的最终效果是一个彩色的圆（见图 9-1）



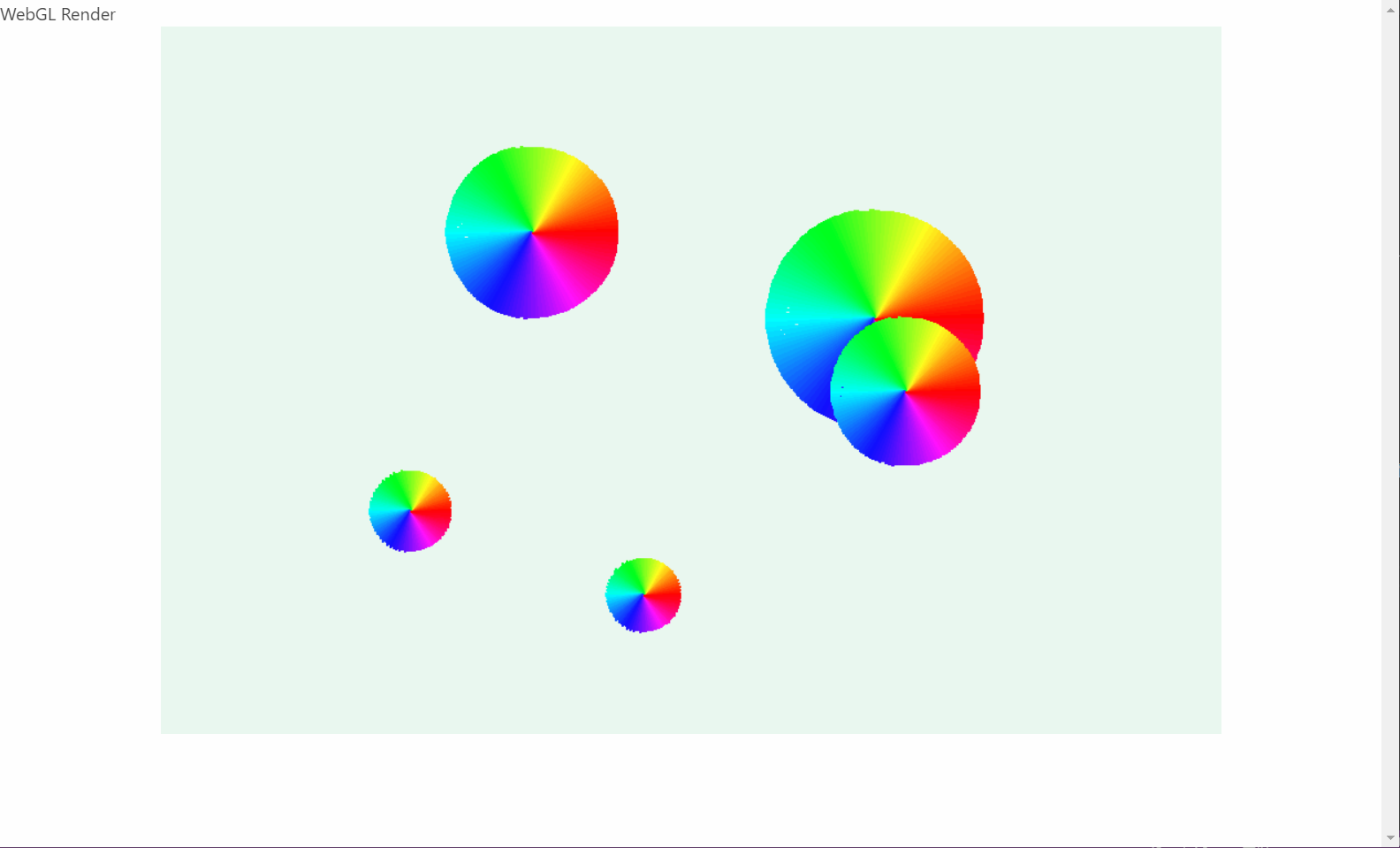
**图 9-1 Circle最终目标**

那么通过上图，也验证了我们的想法通过填充三角形来绘制圆。因为我们已经能够成功的绘制纯色的三角形了，剩下的就是如何获取三角形顶点了。通过最近本的三角函数，循环 2 \* PI 获得一个完整的圆。而颜色可以通过上图发现一定的变换规律这里直接上图吧（颜色填充代码见图9-2）



**图9-2 颜色填充方案**

当然这个方案可以通过 sin 函数优化，毕竟是一组有规律的变化。最后场景展示如图 9-3。

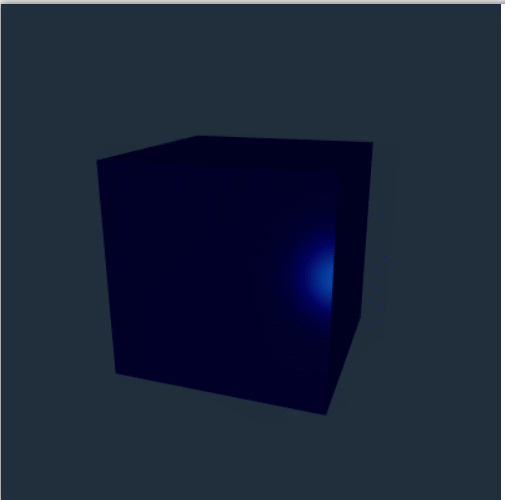


**图9-3 圆形场景绘制**

仔细观察可以发现我绘制的圆外边有毛边，实际上除了我的绘制方案不够理想以外还有是因为视图本身的像素点不足，960 \* 640 在当前的屏幕分辨率下已经属于十分粗糙的分辨率了，尽管我绘制了足够多的三角形保证了三角形内部的过滤平滑，也无法改变像素点不足的情况。解决方案除了优化圆形计算公式外，还有就是增大视图像素数量。

10. material

这一节内容包含下一节的内容已经脱离了我的渲染器代码，数据额外的内容，因此我只是简单的介绍而不做讲解。先来看一下效果图10-1。



**图10-1 material 场景效果展示**

这一张是我通过 webgl 片元着色器实现的 material 内容，其中包括镜面反射specular 自发光diffuse， 漫反射 ambient,以及光泽度 shininess。实际上这是一个动态的场景，我以 gif 的形式保存在工程 res 目录下，可以自行查看。

通过既定公式计算上述四个参数值，调节光泽度，最终达到多种材质效果展示。详细的内容参考了 [learnopengl](https://learnopengl-cn.github.io/02%20Lighting/02%20Basic%20Lighting/)。

11. object file

参考 《webgl 编程指南》实现的 obj 和 mat 文件的加载读取，以及渲染绘制。实现方案我将对应工程放到当前毕设的工程下。只看最后场景。原理实际上就是解析上述的两个文件格式，内容比较单调，直接看场景展示效果（见图 11-1）。



**图 11-1 obj 场景展示效果**

顺便一提若是想通过我的代码查看场景，首先需要本地浏览器做本地文件加载权限处理。其次是改变我代码中的对应文件路径，填入绝对路径，确保能够正确加载模型。

**五、结言**

经过这次的软件渲染器编写，发现了许多问题，比如纹理坐标的透视差值问题，透视投影矩阵的内在数学含义等，但是仍然不敢保证我的理解就是完善的，渲染这条路博大精深，以后还有深入研究探讨。

**主要参考文献**

[1]《计算机图形学》

[2]《webgl 编程指南》

[3]Learnopengl:

<https://learnopengl-cn.github.io/02%20Lighting/02%20Basic%20Lighting/>

[4]韦易笑 mini3D 软件渲染器工程指导:

<https://github.com/skywind3000/mini3d>

[5] 以及各种网络博客