软件光栅渲染器

从软件光栅化角度深入解析图形渲染管线

Yiwei Gong

Contents

前言

在我读大学的时候，3D 游戏引擎还没有那么流行，学校也没有开设相关的课程。事实上，作为计算机系的学生，依旧很难接触到计算机图形学。我接触的第一个游戏框架是苹果 iOS 平台的 SpriteKit，而当时市场上大多数游戏则使用了 Cocos2D。这些 2D 引擎给我造成了极大的误解，单纯地以为游戏的画面是由图片和颜色构成。因此在我创业之前，我根本没有意识到图形学的困难所在。2016 年正值虚拟现实的萌芽阶段，在学习了两个星期的 Unity 引擎之后，我在新加坡创立了VRcollab，https://vrcollab.com 公司，并着手研发建筑虚拟现实软件。

我们制作了一款建筑模型一键生成 VR 场景的软件，同时我们还允许多个用户在场景中共同交流。起初，我并不了解着色的概念，只是简单的写了一个材质转换器，将建筑软件中的材质转换成 Unity 材质。最初的版本只有颜色和颜色贴图。然而很快，我们在制作转换器的过程中就遇到了难题。Unity 的表面着色器使用的参数是光滑度，而少部分建筑软件采用的是粗糙度。由于并不知道着色器的原理，我们在导入过程中，将这些值全部归一转换为光滑度。对于数值的处理是相对简单的，然而当遇到光滑度贴图和粗糙度贴图的时候，我们不得不针对每一个像素进行处理并生成新的贴图，这一操作不仅增加了内存的占用，同时也拖慢软件的整体运行速度。从那时起，我才真正开始接触着色器和计算机图形学。

事实上，图形学满足了我个人的所有爱好。我小时的梦想是成为一个电影特效工程师。我当时并不知道电影特效是如何制作的，只是简单地认为，如果能把科技和艺术相结合，会是很美好和极具意义的。另一方面是对数学的热爱，我总是能感觉到数学里的逻辑美感和形式的艺术感，在计算机图形学之前，我并不能将编程，数学和艺术完美结合一起，而之后，图形学不仅是我的工作，还是我快乐和价值的主要来源。

关于这本书，是来自于一个朴素的想法。我的一位好朋友在 2020 年的时候，出版了《自己动手写 Python 虚拟机》。我突然觉得除了工作之外，应当做一些有意义的事情，不仅仅是对自己知识的一次整理，也是为更多还没有接触图形学的人介绍这一个美妙的世界。

对于第一次接触图形学的人来说，即使是多年的程序员，依旧是非常困难的。从知识层面上，图形学需要有不错的数学功底和对物理模型的抽象能力，至少线性代数和微积分是不可少的。甚至仅仅编程本身，不像是手机 App 或是前端开发，早已有了成熟的框架和抽象层，图形学编程往往涉及到非常底层的硬件知识，OpenGL 和 DirectX 都有着晦涩难懂的 API。着色器的编程则涉及到大量的并行算法和技巧。在游戏引擎层面，则还会涉及虚拟机，Lua 脚本嵌入，资源管理，内存管理等一系列琐碎的内容。因此，一个成熟的游戏引擎的难度已经不亚于一款操作系统。

我看的第一本书是Physically Based Rendering:From Theory To Implementation。直到今天我也没有完全明白里头的技术细节。另一点，离线渲染器和实时渲染器的实现方式又截然不同。对于实时渲染器来说，如今市面上的教程大多以 Unity 或是虚幻引擎为多。这两款游戏引擎已经将绝大部分的实现封装完美，就开发者而言，仅仅需要关心的是上层的业务逻辑而非底层的实现。

正是如此，我想写一本关于图形学底层渲染管线的书。但我并不想从 OpenGL 或者是 DirectX 入手。硬件和 API 的抽象对初学者来说并不友好。甚至，在某种程度上会让人产生相当的困惑。在我学习的过程中，我花了很长时间去了解顶点着色器，片元着色器的工作原理。然而由于存在硬件抽象层，很多的细节只能靠黑盒测试的方式，不断修改参数来推断其中的实现逻辑。

我萌生了一个想法。以纯粹软件的角度，实现渲染管线，以便更好地调试和理解管线中的任何一点细节。事实上，虽然在 GPU 已经成为市场不可或缺的主导，软件渲染器依然在某些剔除算法上，发挥极大的作用。

让我们带着现在的知识，回到上个世纪 90 年代，那个 GPU 还未诞生的年代。我们会以纯粹的 C++ 代码，实现一个支持有现代 GPU 绝大多数功能的软件渲染器。另外，我们也会效仿那个年代的游戏，嵌入一个 Lua 虚拟机，并使用 Lua 脚本实现顶点着色器，曲面细分着色器，片元着色器。一些应用逻辑，业务逻辑的部分，我们也会通过 Lua 脚本的方式进行调用，最终形成一个较为完整的游戏引擎。麻雀虽小，五脏俱全，希望通过如此的方式为所有对图形学感兴趣的读者带来一个不一样的视角。

我希望我能在 30 岁生日以前完成本书，并且作为生日礼物送给自己。本书创作过程以纯粹开源的方式进行,本书的原稿发布在 GitHub 共享社区,https://github.com/SoftwareRenderer/book。对本书中文字和技术细节的错误，欢迎在 GitHub Issue 页面提交和指正。

感谢关注，让我们就此开始。

1 颜色和着色器

对于刚接触图形学的人来说，第一个想要了解的问题，图形学研究的是什么？简要的回答是，图形学研究渲染器中的理论，算法和工程架构。所谓的渲染器不过是一个简单的程序，输入一个描述 3D 场景的场景文件，输出这个场景所对应的一张图片。

大多数人对于图片的概念，不过是电脑上的 JPG 和 PNG 文件。如果使用 Photoshop 之类的软件打开任何一张图片，就会发现图片由一组带颜色的小格子构成. 每一个格子称之为**像素**。对于每一个像素来言，一般来说由三种颜色构成，或者称为三个通道，即红绿蓝通道。每一个通道的取值均为 $[0,1]$。对于一些半透明的图片来说，还会引入第四个通道，即透明通道。同样透明通道的的取值范围也是 $[0,1]$。

如果一个像素的颜色为 $(1,0,0,1)$，像素的四个通道一般情况以 RGBA 的顺序排列，那么很显然这个像素为红色。同样的，如果是 $(1,1,0,1)$，那么该像素呈现红色和绿色的混合颜色黄色。

对于每一个像素来说，由于 $[0,1]$ 是一个浮点数，如果我们单纯使用单精度浮点类型`float`进行存储，对于一个包含 RGBA 四通道的像素来说，其大小为 $32*4$，128 个 bit，即 16 个 byte。对于一张 1920x1080 分辨率的图片来说，这需要 $16*1920*1080=33177600$ 个 byte，相当于 32MB 的大小。事实上，人眼分辨颜色的能力是有限的，如果我们仅仅使用 8 个 bit 来表示一个通道，那么对于每个通道我们将会获得 256 种颜色。红色，绿色，蓝色的混合可以得到 $256*256*256$，大约一千六百万种颜色。8bit 相对 32bit，每个像素仅仅需要 4 个 byte 即可存储，通过一些合适的压缩算法，我们就能将一张 1920x1080 分辨率的图片合理压缩到 2~3MB 的大小。

因此，绝大部分的图片和颜色都采用 8bit 的方式来存储。由于 8bit 刚好可以用 $[0,255]$ 区间的整数或者是两个 16 进制数表示，很多程序，比如 HTML，就会使用 16 进制颜色编码来，例如`#2980b9ff`，来表示颜色。在 C 语言中，我们则可以使用`char`类型来表示一个通道的数值。

1.1 第一个实验：C++ 生成单颜色图片

有了对图片和像素的基本了解，我们不妨在 C++ 做一个小实验来生成一张只有颜色的图片。

既然要生成图片，首先我们要考虑图片的格式。对于绝大多数的图片格式来说，其像素都是由一个一维数组构成。其数组的长度等于图片的宽度乘以图片的高度乘以每个像素的通道数量，即：

$$len = width * height * channels$$

stb是一个小巧的 C++ 头文件库，提供了基础图片格式的读写操作。我们可以通过 GitHub 获取，https://github.com/nothings/stb/blob/master/stb_image_write.h。

在此，我们将使用stbi_write_png(char const *filename, int w, int h, int comp, const void *data, int stride_in_bytes)函数将我们的图片数据保存为 PNG 格式。

stbi_write_png函数由六个参数构成，第一个参数是文件路径，第二和第三个参数是图片的宽度和高度。第四个参数表示图片数据的通道数量。由于不需要使用透明通道，因此三通道图片即可满足要求。第五个参数表示图片数据。由于我们的图片数据是一个一维数组，stride_in_bytes则表示每行的像素数据的长度。

了解了stbi_write_png函数的参数，在我们的例子中，我们可以将其简化为stbi_write_png(char const *filename, int w, int h, 3, const void *data, w*3)。

下面关注数据部分。我们只需要生成一张不透明的颜色图，假定其为 200x200 分辨率，同时使用 8 Bit 颜色格式。因此，我们只需要声明一段像素数组：

```
1  char pixels[200*200*3];
```

最后我们将其组装在一起，得到我们的第一个小程序：

```
1  // single-color.cpp
2
3  // This line is required for stb image library
4  #define STB_IMAGE_WRITE_IMPLEMENTATION
5  #define STB_IMAGE_WRITE_STATIC
6  #include <stb/stb_image_write.h>
7
8  int main()
9  {
10     int w = 200;
11     int h = 200;
12     int c = 3;
13
14     char pixels[w * h * c];
15     for (auto x = 0; x < w; x++)
16         for (auto y = 0; y < h; y++) {
17             pixels[(y * w + x) * c + 0] = (char)255;
18             pixels[(y * w + x) * c + 1] = (char)0;
19             pixels[(y * w + x) * c + 2] = (char)0;
20         }
21
22     stbi_write_png("single-color.png", w, h, c, pixels, w * 3);
23     return 0;
24 }
```

我们编译并运行该程序，即可获得一张名为single_color.png的红色图片。

接下来，我们将在这个基础上，做一些简单的调整，让我们输出的图片更加的漂亮。

1.2 第二个实验：屏幕空间坐标和着色器

在第一个实验中，我们通过设置像素数组的数值来实现输出颜色的目的。我们不妨把程序做一个稍稍的整理。

首先是颜色，由于每一个像素都由三个通道构成，我们可以如此定义颜色：

```
1  struct Color {
2    char r;
3    char g;
4    char b;
5  };
```

那么我们像素数组即变成：

```
1  Color pixels[width*height];
```

得益于 C++ 的特性，虽然我们增加了Color类型，但数据在内存排布上和之前的数组则没有任何差别。

我们再将生成颜色的代码整理成一个函数，以便我们更好的控制颜色的生成过程。

```
1  Color PixelColor(int x, int y, int width, int height);
```

这里我们注意到，我们需要传递四个参数。然而在渲染过程中，我们大多只关心渲染的结果，而忽略渲染输出的分辨率。同一张图片，只要长宽比相同，1080P 和 4K 应该仅仅只有清晰度的

区别，而不会有任何内容上的不同。

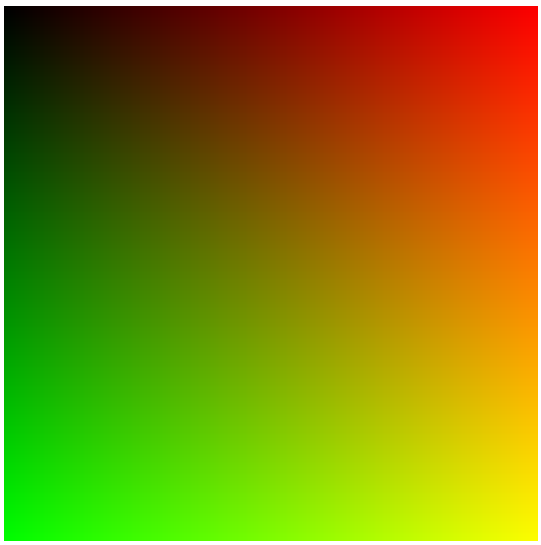
对此，我们可以将PixelColor函数做一个简单的简化：

```
Color PixelColor(float u, float v);
```

其中u = x/(float)width, v = y/(float)height。经过此变换，一个像素的颜色仅仅只和这个像素在屏幕上的位置相关，和屏幕的分辨率的大小不相关。

最后，我们将这些组件拼装在一起，并生成一张具有渐变颜色的图片。

```
// gradient-color.cpp

// This line is required for stb image library
#define STB_IMAGE_WRITE_IMPLEMENTATION
#define STB_IMAGE_WRITE_STATIC
#include <stb/stb_image_write.h>

struct Color {
  char r;
  char g;
  char b;
};

Color PixelColor(float u, float v) {
  Color c;
  c.r = (char)(u * 255);
  c.g = (char)(v * 255);
  c.b = 0;
  return c;
}

int main() {
  int w = 200;
  int h = 200;
  int c = 3;

  Color pixels[w * h];
  for (auto x = 0; x < w; x++)
    for (auto y = 0; y < h; y++) {
      auto u = x / (float)w;
      auto v = y / (float)h;
      pixels[y * w + x] = PixelColor(u, v);
    }

  stbi_write_png("gradient-color.png", w, h, c, pixels, w * 3);
  return 0;
}
```



让我们来回顾一下我们在实验中用到的函数和参数。

首先我们建立起了屏幕空间的坐标系，这一个坐标系在左上方为 $(0,0)$，右下方为 $(1,1)$。图片的颜色仅仅和坐标系的坐标有关，而和屏幕的大小无关。我们在之后空间变换的部分还会重新回顾屏幕空间坐标系。

第二个重要的改变是我们把具体的颜色生成过程移动到了`PixelColor`函数中去。该函数会生成每一个像素的颜色。我们把计算像素颜色的过程称之为**着色**，而负责着色的函数则称之为**着色器**。

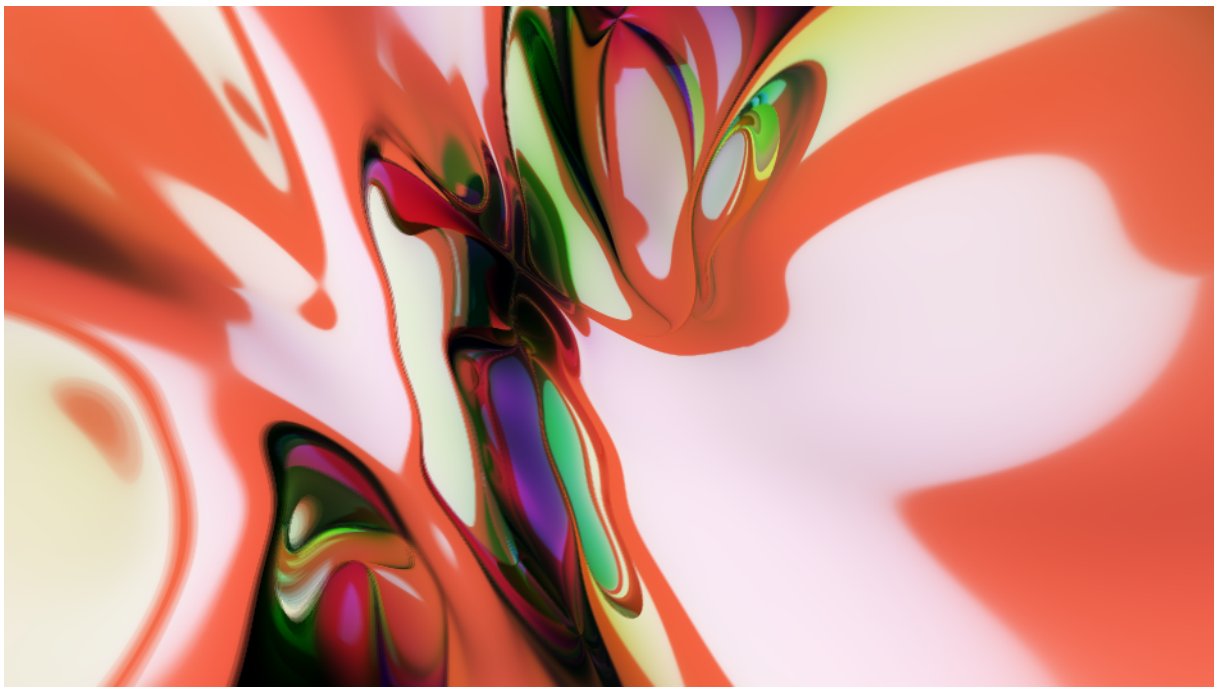
我们会在后续的章节继续介绍**着色器**的概念。如果你曾经接触过一些图形学的书籍，事实上，在此我们所写的`PixelColor`函数就是一个简化的像素着色器。

更重要的是，`PixelColor`函数只关心一个像素的颜色，而不关心周围像素的颜色，也不使用任何其他像素的颜色作为其计算的基础。因此该函数没有任何的依赖关系。正是如此，像素着色器能够被高度的并行优化，而在 GPU 上，每一个像素着色器都是并行计算，从而实现快速实时渲染。

1.3 像素着色器的魔法

虽然我们的程序很短，实际上我们已经实现了一个完整的像素着色的过程。仅仅依赖于像素着色器，和 GPU 的并行计算的优势，我们就可以创建无数优美的图画。

ShaderToy是一家专注于发布和分享像素着色器的平台。无数的艺术家、程序员在上面创建了各种魔法般的图案。



图片来自https://www.shadertoy.com/view/ltfXzj

1.4 小结

本章节中，我们介绍了图片和颜色的格式，并且通过 C++ 生成了简单的图片。在我们正式进入到渲染管线之前，我们都会通过图片的形式来演示光栅化的过程。

我们还引出了屏幕空间的概念，在本章节的屏幕坐标系中，是一个仅有二维坐标，并且取值范围在 $[0, 1]$ 的空间。

紧接着，我们讨论了着色过程和像素着色器。像素着色器真正揭示了 GPU 的并行原理。因为每个像素着色器互不依赖，因此 GPU 才可以同时计算每个像素的颜色。

整个图形学研究的内容就是如何快速地，更逼真地对场景完成整个着色过程。下一章节，我们将从最简单的几何图形出发，为我们的渲染器增加图元渲染的能力。

1.5 思考和扩展

1. 如何绘制一个带有半透明背景的图片。
2. 如何绘制一个红色的圆形。

2 线段绘制

> 本章节算法和代码引用了 Dmitry V. Sokolov 的 tinyrenderer 第一章线段光栅化过程 (Sokolov 2022a)。在原算法的基础上，本章节做了整理、改进和重构。感谢 Dmitry V. Sokolov 将这一段算法整理成册，并开源地发布在 GitHub 供所有人使用。

在上一单元，我们了解了颜色的表示方式，基础颜色的绘制，以及基础的着色过程。对于着色过程，我们还引入了现代 GPU 的着色器的概念，并简单地描述了像素着色器。在这一章，我们将通过代码绘制最基础的几何图形，线段，的绘制。

在数学上，线段由两个端点和端点的连线构成。对于任何一个线段，我们可以通过线段的方程 $f(x) = ax + b$ 来进行描述，或者我们也可以通过线段的一个端点，线段的方向，以及线段的长度来描述一个线段。

假设存在一根线段，其端点为 (x_1, y_1)，(x_2, y_2)，我们试着找出所有在该线段上的点。

我们设向量 $\vec{L} = (x_2, y_2) - (x_1, y_1)$，那么很显然，线段上的点满足参数方程

$$y = \vec{L} * t + (x_1, y_1)$$

其中，t 的取值范围为 $[0, 1]$。

现在让我们结合 C++ 代码，在屏幕空间上绘制绘制一根红色的，端点为 $(0.2, 0.2)$ 和 $(0.6, 0.6)$ 的线段。

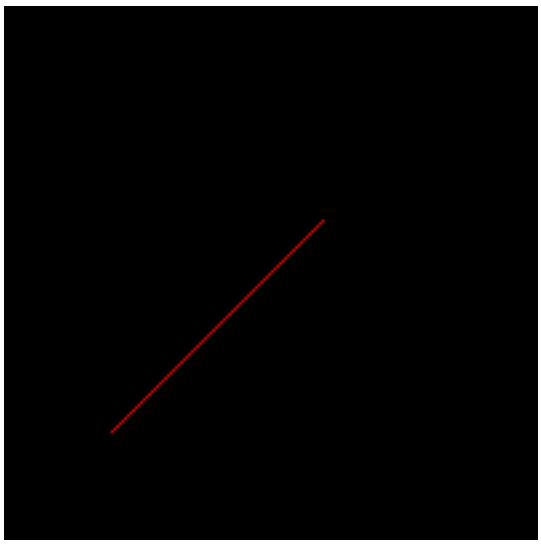
2.1 线段的参数方程绘制

首先要说明的是我们的坐标系，由于 PNG 图片的特殊性，左上角为坐标原点，右下角为坐标的 $(1, 1)$ 点，我们为了更好地贴合数学上的笛卡尔坐标系，我们将图片上下翻转，使得做左下角为原点，右上角为 $(1, 1)$ 点。

```
1 // Pre-defined data types
2 #include "datatypes.hpp"
3
4 using namespace SoftwareRenderer;
5
```

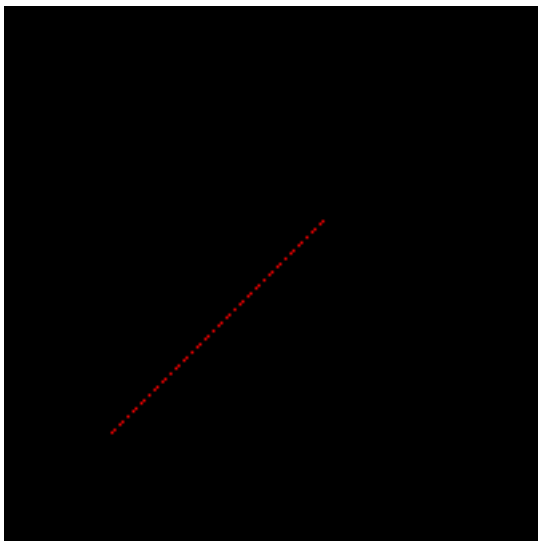
```
 6  void DrawLine(Image& img, Vector2f p1, Vector2f p2, float step, Color
        color)
 7  {
 8      float lx = p2.x - p1.x, ly = p2.y - p1.y;
 9      for (auto t = 0.0f; t <= 1.0f; t += step) {
10          auto x = p1.x + t * lx;
11          auto y = p1.y + t * ly;
12          // 将屏幕坐标转换到像素空间
13          img.SetColor(img.Width() * x, img.Height() * y, color);
14      }
15  }
16
17  int main()
18  {
19      auto img = Image(200, 200);
20      DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.6f), 0.01f,
            Color::Red());
21      img.SaveAsPNG("line-0.01.png");
22  }
```

编译并运行上面的代码，得到



我们观察到`for (auto t = 0.0f; t <= 1.0f; t += step)`这一层循环。由于计算机只能表示离散地数列，而无法表示连续的取值范围，我们只能通过一个极小数，0.01，的步进来拟合连续的表达。如果我们把这一个步进的步长取得过小，则会大幅度增加计算机的计算负担，实际上，当步长过小时，而我们整张图片的大小却只有 200x200 像素，这就意味着，每一小步增长甚至没有超出一个像素，那么有大量的像素被重新计算和重新着色。

如果我们将步长取得过大，那么线段则会发生断裂的现象。



很显然，对于这个算法，我们必须精确控制参数 t 的取值，才能得到完整的，同时又不过多绘制的线段。

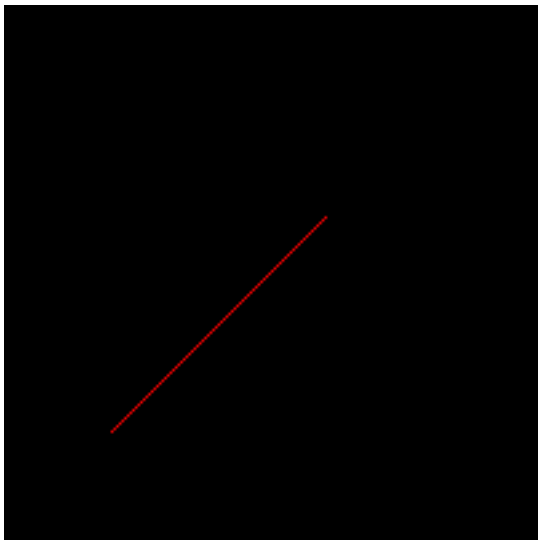
2.2 像素空间绘制

我们首先尝试解决第一个问题，如何避免参数 t 的取值影响我们最后的呈现效果。

从数学的角度出发，t 的取值越小，我们获得的结果一定越精确，当我们用微分 dt 作为变化的时候，我们则可获得完整的连续的线段。另一方面，将数学定义的世界空间的线段转换成我们的图片的时候，最终的线段只会用有限个像素表达。由于线段是连续的，那么在其两个端点间，$[x_1, x_2]$，每一个 x 一定有一个 y 相对应。而在像素层面上来说，x 的取值个数是有限的，因此我们可以用如下方式绘制：

```
// Pre-defined data types
#include "datatypes.hpp"

using namespace SoftwareRenderer;

void DrawLine(Image& img, Vector2f p1, Vector2f p2, Color color)
{
    // 将屏幕坐标转换到像素空间
    int px1 = p1.x * img.Width(), px2 = p2.x * img.Width();
    int py1 = p1.y * img.Height(), py2 = p2.y * img.Height();
    for (auto x = px1; x <= px2; x++) {
        auto t = (x - px1) / (float)(px2 - px1);
        auto y = (int)(t * (py2 - py1) + py1);
        img.SetColor(x, y, color);
    }
}
```

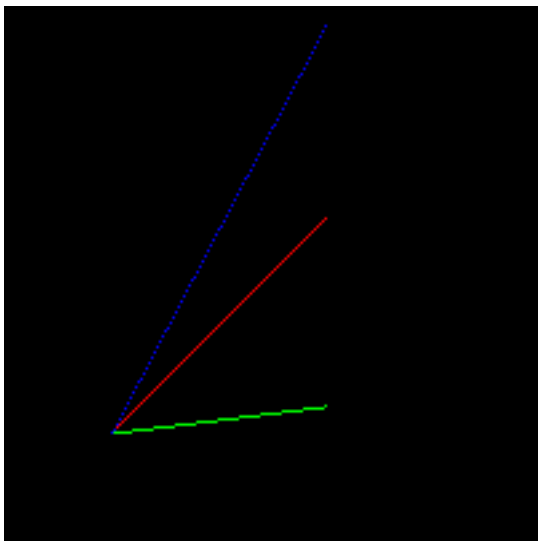
```
17
18  int main()
19  {
20      auto img = Image(200, 200);
21      DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.6f), Color::
            Red());
22      img.SaveAsPNG("line-pixel-space.png");
23      return 0;
24  }
```



在像素空间进行计算和绘制，我们确保了取值范围内的横向的每一个 x 都有一个 y 与之相对应。并且更重要的是，这意味着线段在横轴上的投影一定是连续的。然而，线段在纵轴上的投影却并非如此。

当前，线段的斜率为 1，我们试着绘制两根额外的线段，绿色的线段斜率小于 1，蓝色的线段斜率大于 1。

```
1  // 斜率等于1
2  DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.6f), Color::Red())
       ;
3  // 斜率小于1
4  DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.25f), Color::Green
       ());
5  // 斜率大于1
6  DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.96f), Color::Blue
       ());
```



我们可以发现，红色和绿色的线段可以被连续的绘制，然而蓝色的线段却出现了明显的断点。这是因为当斜率大于 1 时，线段的横轴的投影是连续的，然而线段的纵轴的投影却不连续。从像素层面解释，高度的像素的个数大于宽度的像素个数，如果我们还是依旧宽度进行采样，显然不能覆盖所有的高度像素。一个简单的解决方案是，当斜率大于 1 时，以高度作为循环进行采样，当斜率小于 1 时，以宽度作为循环进行采样。

2.3 Bresenham 直线算法

经过我们上一轮的改进，我们已经能够绘制基本的线段。但是我们的算法还有没有改进的余地，让绘制的效率变得更高呢?

首先我们观察循环中的两次运算

```
1  auto t = (x - px1) / (float)(px2 - px1);
2  auto y = (int)(t * (py2 - py1) + py1);
```

当我们求系数 t 的时候，用到了浮点除法，而大量的浮点运算无疑会降低绘制的效率。将浮点运算转换成整数运算，将除法运算转换成加法和减法运算能有效提高绘制效率。

这里我们将介绍著名的 Bresenham 直线算法。Bresenham 直线绘制算法由 Jack Elton Bresenham 于 1962 年在 IBM 工作时发明 (Bresenham 1962)。由于算法简单易懂，并且绘制效率高效成为线段绘制的标准算法。

我们再来回顾一下线段的绘制过程。

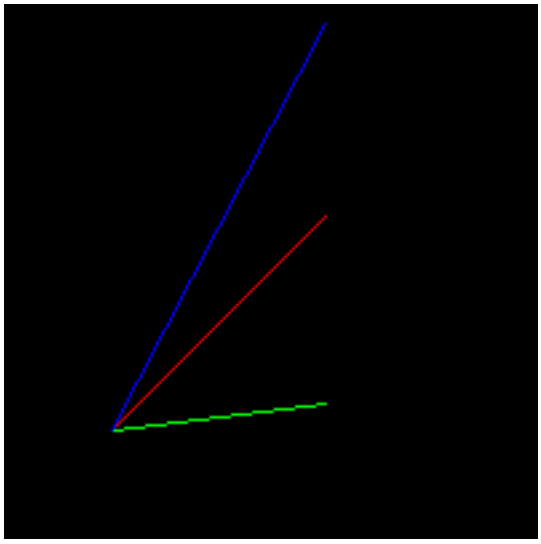
我们从线段的一个端点出发，假使线段斜率小于等于 1，且用横坐标扫描的方式进行绘制（对于斜率大于 1 的线段，我们可以通过纵坐标扫描的形式进行。由于算法是对称的，这里假定横

坐标扫描），对于线段两个端点中的每一个 x，都可以计算出一个 y 坐标。由于线段的斜率小于等于 1，任意两个相邻的 x，其对应的 y 要么相等，要么相差 1。

```cpp
// Pre-defined data types
#include "datatypes.hpp"

using namespace SoftwareRenderer;

void DrawLine(Image& img, Vector2f p1, Vector2f p2, Color color)
{
    // 将屏幕坐标转换到像素空间
    auto p1i = Vector2i(p1.x * img.Width(), p1.y * img.Height());
    auto p2i = Vector2i(p2.x * img.Width(), p2.y * img.Height());

    // 如果斜率大于1，我们将x和y交换，从而统一为横扫描
    auto steep = false;
    if (std::abs(p1i.x - p2i.x) < std::abs(p1i.y - p2i.y)) {
        p1i = Vector2i(p1i.y, p1i.x);
        p2i = Vector2i(p2i.y, p2i.x);
        steep = true;
    }

    // 如果p1点在p2点的右侧，则交换两点，从而统一为从左到右扫描
    if (p1i.x > p2i.x) {
        std::swap(p1i, p2i);
    }

    auto dx = p2i.x - p1i.x;
    auto dy = p2i.y - p1i.y;
    auto derror = std::abs(dy / (float)dx);
    auto error = 0.0f;
    auto y = p1i.y;
    for (auto x = p1i.x; x <= p2i.x; x++) {
        if (steep) {
            img.SetColor(y, x, color);
        } else {
            img.SetColor(x, y, color);
        }
        // 当x步进一格，累积一次error，如果error超过0.5，意味着y需要步进一格
        error += derror;
        if (error > 0.5f) {
            // 如果斜率为正，则向上步进，否则向下步进
            y += p2i.y > p1i.y ? 1 : -1;
            // error-1，重置一个像素偏移，重新计算累积error
            error -= 1;
        }
    }
}

int main()
```

```
48  {
49      auto img = Image(200, 200);
50      DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.6f), Color::
            Red());
51      DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.25f), Color::
            Green());
52      DrawLine(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.96f), Color::
            Blue());
53      img.SaveAsPNG("bresenham.png");
54      return 0;
55  }
```

编译运行上述程序，我们得到完美的线段渲染。



2.4 小结

在本章节中，我们讨论了一个基本问题，如何绘制一条线段，并且通过几个简单的算法来展示了这一过程。值得注意的是，线段绘制的过程实际上是一个从连续的空间到一个离散空间的映射。由于离散空间的关系，不同的算法会产生不同的线段，有些出现了断点，而有些则没有。然而从数学意义上说，他们都是等价的。由此可见，选取一个好的映射方案能带给我们不一样的绘制效果。最后 Bresenham 算法，通过将浮点乘法和除法运算转换为加法运算的方式，给我们展示了更高效的线段绘制方案。

线段绘制是光栅化基础，在下一章三角形绘制中，我们首先会介绍扫描线算法，并了解如何利用线段绘制来绘制三角形。

2.5 思考和扩展

1. 为什么我们要使用像素空间坐标而不是屏幕空间进行绘制?

3 三角形绘制

> 本章节算法和代码引用了 Dmitry V. Sokolov 的 tinyrenderer 第二章三角形光栅化过程 (Sokolov 2022b)。在原算法的基础上，本章节做了整理、改进和重构。感谢 Dmitry V. Sokolov 将这一段算法整理成册，并开源地发布在 GitHub 供所有人使用。

在上一章节中，我们描述了基本的线段绘制的算法。线段是所有直线，射线，以及曲线的基本构成单元。对于一条曲线来说，我们可以通过利用微积分的思想，将曲线转换为一条一条细小的线段，通过控制线段的数量来控制曲线的平滑精度。而对于一个曲面来说，我们则可以通过渲染无数个细小的三角形来拟合。

我们在此章节中，将会探索在 2D 平面下的三角形渲染。在掌握了 2D 三角形渲染的方法以后，对于一个立体的 3D 图形，无非通过投影变换将三维空间转换为二维平面，然后重复这个渲染过程。

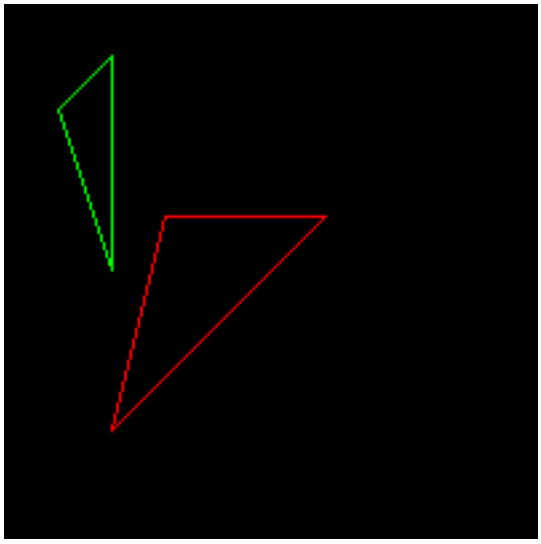
3.1 空心三角形渲染

我们已经利用 Bresenham 算法实现了`DrawLine`函数，对于空心三角形的绘制，无非是三角形三个端点使用`DrawLine`进行三次线段绘制。

```
#include "datatypes.hpp"

using namespace SoftwareRenderer;

void DrawTriangleFrame(Image& img, Vector2f p1, Vector2f p2, Vector2f
    p3, Color color)
{
    DrawLine(img, p1, p2, color);
    DrawLine(img, p2, p3, color);
    DrawLine(img, p3, p1, color);
}

int main()
{
    auto img = Image(200, 200);
    DrawTriangleFrame(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.6f),
        Vector2f(0.3f, 0.6f), Color::Red());
```

```
16      DrawTriangleFrame(img, Vector2f(0.2f, 0.5f), Vector2f(0.1f, 0.8f),
            Vector2f(0.2f, 0.9f), Color::Green());
17      img.SaveAsPNG("triangle-frame.png");
18      return 0;
19  }
```

运行该程序我们就可以得到一个空心三角形。

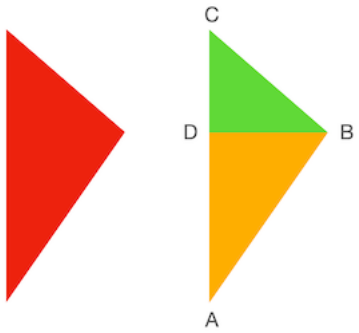


3.2 扫描线算法

通过线段绘制的办法，我们已经得到了一个空心的三角形。而对于实心三角形来说，一个朴素的想法是，我们从下至上，如同填色一样，一条线一条线的补全颜色，直到绘制完成整个三角形。

由于我们不断的在三角形内部从左至右扫描并填，这个算法被称之为扫描线算法。现在让我们来讨论扫描线算法的实现细节。

第一步，我们将三角形切分为上下两部分。



如上图所示，红色的三角形被经过 B 点的水平线，分成了上下两部分。上部分三角形由线段

AC，BC 和水平线 BD 围成，下半部分三角形则由 AC，AB 和水平线 BD 围成。A，B，C 三点的确定只需要根据三角形的三个端点的 y 坐标从低到高排序即可。

如果 A 和 B 处于同一水平线，y 值相等，那么我们可以认为其下半部分 ABD（或上半部分 BDC）为空三角形，而只存在上半部分（或下半部分）三角形。

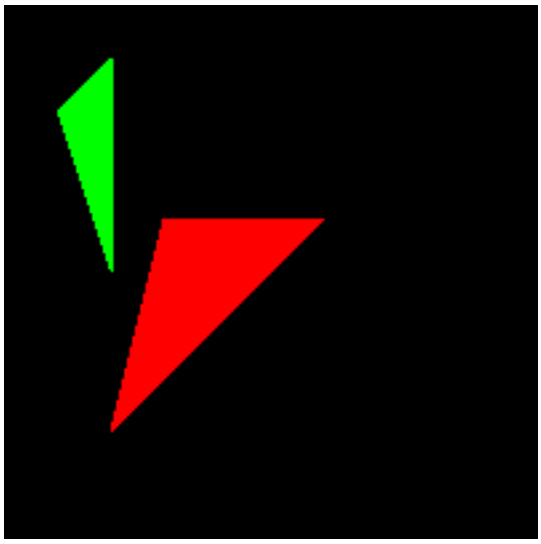
经过如此切分以后，三角形的三条线段的方程都是已知的。那么我们关注三角形 ABD，我们下往上，在线段 AD 上寻找一个点，然后做水平线，与 AB 相交，这条线段即我们的扫描线。这根扫描线明显处于三角形内部，那么我们把其经过的所有像素着色即可。

我们用 C++ 代码来表述这一过程：

```cpp
#include "datatypes.hpp"

#include <iostream>
using namespace SoftwareRenderer;

// 已知线段的两个端点，当一个点在线段上，并已知其y值，求x值
int SolveLineX(Vector2i p1, Vector2i p2, int y)
{
    // p1.y和p2.y应不相等，我们在扫描线循环中，规避了相等情况
    assert(p1.y != p2.y);

    auto t = (y - p1.y) / (float)(p2.y - p1.y);
    return t * (p2.x - p1.x) + p1.x;
}

void DrawTriangle(Image& img, Vector2f p1, Vector2f p2, Vector2f p3,
    Color color)
{
    // 将屏幕坐标转换到像素空间
    auto p1i = Vector2i(p1.x * img.Width(), p1.y * img.Height());
    auto p2i = Vector2i(p2.x * img.Width(), p2.y * img.Height());
    auto p3i = Vector2i(p3.x * img.Width(), p3.y * img.Height());

    // 将三角形三个坐标按y轴从低到高排序
    if (p1i.y > p2i.y)
        std::swap(p1i, p2i);
    if (p1i.y > p3i.y)
        std::swap(p1i, p3i);
    if (p2i.y > p3i.y)
        std::swap(p2i, p3i);

    // 扫描下半部分三角形，如果三角形水平，p1i.y == p2i.y
    // 则该循环被跳过
    for (auto y = p1i.y; y < p2i.y; y++) {
        auto xMin = SolveLineX(p1i, p3i, y);
        auto xMax = SolveLineX(p1i, p2i, y);
        DrawLine(img, Vector2i(xMin, y), Vector2i(xMax, y), color);
    }
```

```
38
39      // 扫描上半部分三角形，如果三角形水平，p2i.y == p3i.y
40      // 则该循环被跳过
41      for (auto y = p2i.y; y < p3i.y; y++) {
42          auto xMin = SolveLineX(p1i, p3i, y);
43          auto xMax = SolveLineX(p2i, p3i, y);
44          DrawLine(img, Vector2i(xMin, y), Vector2i(xMax, y), color);
45      }
46  }
47
48  int main()
49  {
50      auto img = Image(200, 200);
51      DrawTriangle(img, Vector2f(0.2f, 0.2f), Vector2f(0.6f, 0.6f),
            Vector2f(0.3f, 0.6f), Color::Red());
52      DrawTriangle(img, Vector2f(0.2f, 0.5f), Vector2f(0.1f, 0.8f),
            Vector2f(0.2f, 0.9f), Color::Green());
53      img.SaveAsPNG("line-sweeping.png");
54      return 0;
55  }
```

运行代码得到



3.3 包围盒测试算法

Bresenham. 1962. “Bresenham’s Line Algorithm.” Wikipedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm.

Sokolov, Dmitry V. 2022a. “Lesson 1: Bresenham’s Line Drawing Algorithm.” GitHub. GitHub, ssloy/tinyrenderer. https://github.com/ssloy/tinyrenderer/wiki/Lesson-1:-

Bresenham%E2%80%99s-Line-Drawing-Algorithm.

———. 2022b. “Lesson 2: Triangle Rasterization and Back Face Culling.” GitHub. GitHub, ssloy/tinyrenderer. https://github.com/ssloy/tinyrenderer/wiki/Lesson-2:-Triangle-rasterization-and-back-face-culling.