IQ大神博客阅读心得2

hardware interpolation：采样插值；由于硬件的限制，手动进行双线性滤波。
Gamma Correct Blurring：基础知识；注意不要在SRGB上进行模糊等线性像素操作
rational rendering and floating bar：三角函数的“错误性”；有理数取代浮点数的原因和大致方法
Fast Trisect In GLSL：三次曲线或四次曲线求解的处理和近似优化
Wavelet Image Compression：图像压缩的非常规方法以及相关细节
lyapunov fractals：利用Lyapunov指数创建分形图；
Mandelbrot：无限的自相似性；分形学之父发明的一种点集合的计算；分形的一种形成方法
Budhabrot：在Mandelbrot set的基础上进行的神奇方法，产生了“佛陀”，所以科学的尽头是神灵吗
Julia Set：和Mandelbrot很像；也是个在复平面上形成分形的点的集合
Geometric Orbit Traps：几何轨道陷阱？在Mandelbrot集上对于轨迹的几何性质进行分析处理

hardware interpolation

承接之前阅读心得1的improved texture interpolation，如IQ所言，当时的GPU大多采用24.8结构，使用8位存储小数部分，因此在两个像素之间最多有256个插值结果，这对于某些场合是不对的（纹理通常不仅编码表面属性，而且还用作查找表（LUT），高场（用于地形渲染）），例如作者举例的：地形高度场，阶梯状的数值变化，会导致很多问题

改进的方法，也很简单，手动进行双线性滤波，

```
vec4 textureGood( sampler2D sam, vec2 uv )
{
    vec2 res = textureSize( sam );
```

```
    vec2 st = uv*res - 0.5;

    vec2 iuv = floor( st );
    vec2 fuv = fract( st );

    vec4 a = texture( sam, (iuv+vec2(0.5,0.5))/res );
    vec4 b = texture( sam, (iuv+vec2(1.5,0.5))/res );
    vec4 c = texture( sam, (iuv+vec2(0.5,1.5))/res );
    vec4 d = texture( sam, (iuv+vec2(1.5,1.5))/res );

    return mix( mix( a, b, fuv.x),
                mix( c, d, fuv.x), fuv.y );
}
```

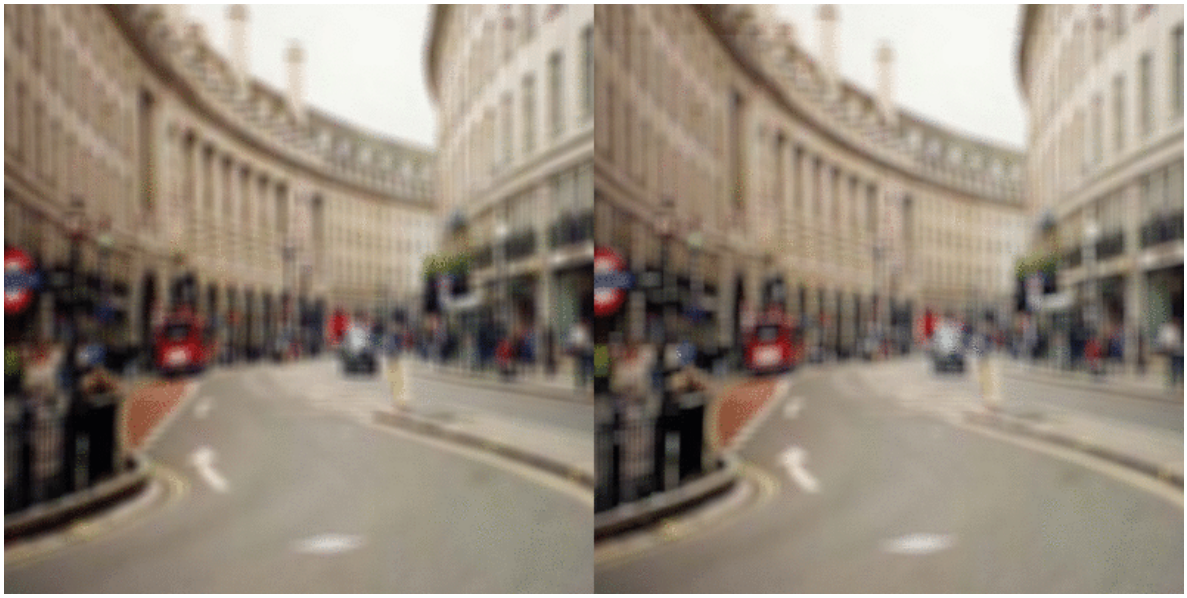
Gamma Correct Blurring

我们大多数人在某个时候所犯的一个错误是在照片上执行了图像模糊缩小（或其他线性像素操作）而没有考虑伽玛或srgb编码的问题（两种不同，但表现上足够相似）。这是因为图片在srgb色彩空间编码可以获得最佳显示质量（相对于人眼）。理想情况下，您的图像模糊算法应保持图像亮度不变，并且仅对其中的细节进行模糊处理，换句话说，如果模糊内核要保持亮度，则它必须在线性空间而不是伽马空间中运行。因此，在模糊照片时，应先对图像像素进行反Gamma处理，然后再进行线性运算，然后平均后进行Gamma处理。

大多数GPU硬件现在可以自动完成这个，如果我们为纹理选择正确的内部格式。大概的伪代码如下：

```
//Gamma(x)=pow(x,1.0/2.2) or sqrt(x)
//DeGamma(x)=pow(x,2.2) or x*x
vec3 col = vec3(0.0);
for( int j=-w; j<=w; j++ )
for( int i=-w; i<=w; i++ )
{
    col += DeGamma( src[x+i,y+j] );
}
dst[x,y] = Gamma( col / ((2*w+1)*(2*w+1)) );
```

所以，请记住。如果我们在GPU中，请对按原样显示的纹理使用sRGB。如果您在CPU中，请记住应用反伽马校正。



rational rendering and floating bar

IQ开篇提出这样一个问题：**“如果我们用有理数替换浮点数并尝试渲染图像，将会发生什么？”**，这里举了一个例子：判断一个点是否属于三角形内部是困难的，原因在于——通过行列式或某些叉积来检查四个点的共面性（实际是一个东西）永远不会导致这些数学方法所需的值恰好为零（哪怕共面性计算是精确的，但输入的精度折衷也将几乎以1.0的概率保证四个点本身不会共面，毕竟我们取的小数点位是有限的）。因此，考虑将输入改为有理数。

典型的渲染操作（边界测试，射线-三角形相交，射线反射）基于叉积，点积和标量除法，它们基于加减乘除四个基本运算，而基本运算的结果也是有理数。而开根号和三角函数是不包含这个规则的（现在，反过头来想想之前IQ大神在旋转举例Avoiding rigonometry那节表达自己对三角函数的厌恶，读者更加认同了）

IQ大神又谈到了无处不在的Normalize问题：

> For the former, the square root, except for conics sections (spheres, cylinders, etc) and actual shading/brdfing/coloring, one doesn't really need to normalize the rays and surface normals as often as we usually tend to. Certainly not for ray generation, traversal, intersection, reflections, etc. Sadly, often times I find programmers normalizing things for no good reason other than "I do it to be safe". In practice, you rarely need to normalize things in the geometry tracing part of the rendering, so I had hopes one could indeed trace a whole scene never leaving rational numbers - what I'd call "rational rendering".

具体实现过程还有很多细节，一些细节归纳：1.实际运算，可以其转化为分母分子的四则运算组合。2.需要额外存储符号位。3.使用GCD（辗转相除法）确定最大公因数，缩放分母分子。具体见博客。

Fast Trisect In GLSL

计算机图形学中存在涉及三次方程或四次方程的多个问题。从反转平滑步函数，计算到抛物线的距离，绘制二次贝塞尔曲线，或计算与四次曲面（包括圆环）的交点。但一般来说，许多开发人员都避免分析处理这种问题，因为求解鲁棒性高的求解器是不容易的。

通常在每个三次曲线的中心（Center）进行以下操作：

$$y = cos(acos(x)/3)$$

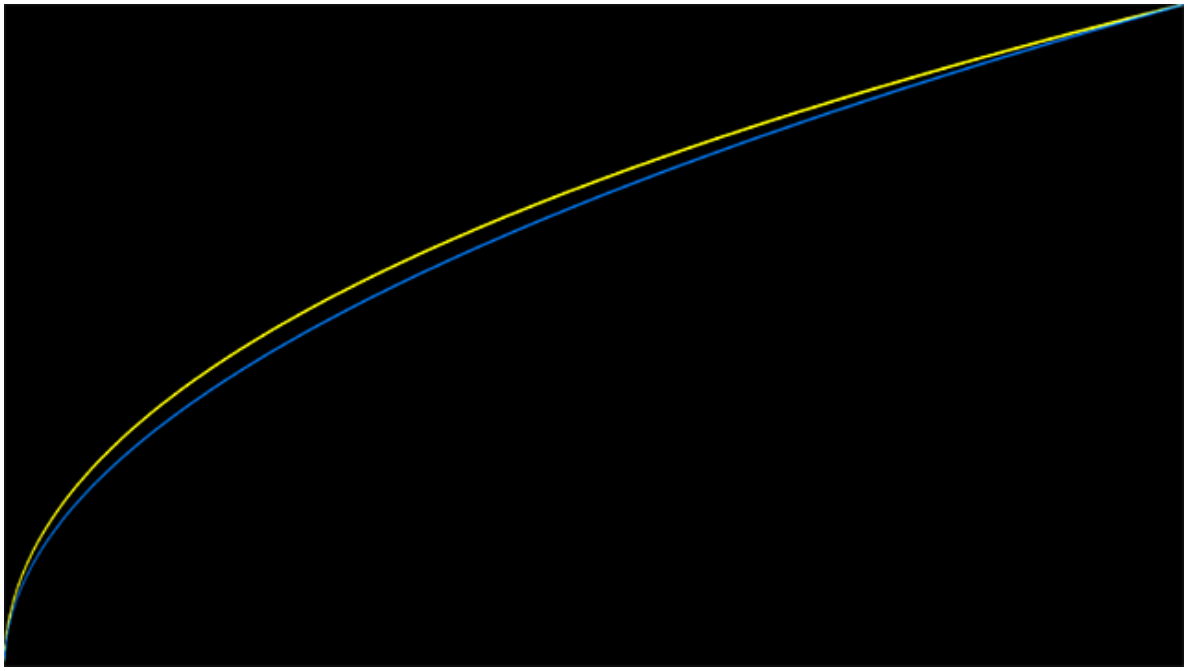
IQ分析，这个操作基本上是三等分，这是计算单位的立方根所需要的，这是问题的核心。这个操作在二次方程的解析中所起的作用与平方根相同，后者平分一个角(想想平方根对复数的辐角所起的作用)。让我们将实现功能的函数称为“trisect(x)”

IQ指出，Cos操作是在GPU中运行最快的操作之一，而ACOS则相反（GPU上是利用多项式进行逼近）

> 角度：避免他们。它们很少是您真正想要的，它们倾向于在代码中引入很多三角函数，您突然需要担心parametrization artifacts (e.g. dealing with wraparound），并且使用角度的代码通常更难于阅读/理解/调试。比使用向量的等效代码要慢（而且速度也较慢）。

$$Yellow : trusect(x)$$

$$Blue : g_1(x) = \frac{1}{2} + \frac{1}{2} \cdot sqrt(\frac{1}{2} + \frac{1}{2}x)$$



用二次方，三次方和四次方替换平方根输出中的偏移和偏置（这是线性变换），看看是否可以将曲线向目标trisect（x）弯曲。让我们称这些二次，三次和四次近似$g2(x)$，$g3(x)$和$g4(x)$，IQ使用随机系数测试其性能，就比内部用三角函数实现的方法快1.6到2.2倍。

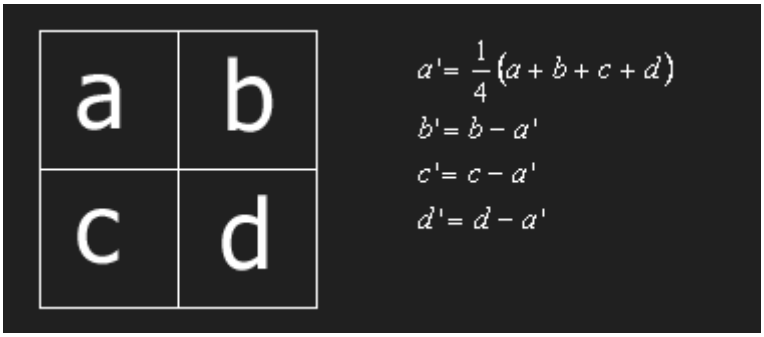
```
float f( in float x )
{
    return cos(acos(x)/3.0);
}

float g1( in float x )
{
    x = sqrt(0.5+0.5*x);
    return x*0.5+0.5;
}

float g2( in float x )
{
    x = sqrt(0.5+0.5*x);
    return x*(-0.064913*x+0.564913)+0.5;
}

float g3( in float x )
{
    x = sqrt(0.5+0.5*x);
    return x*(x*(x*0.021338-0.096562)+0.575223)+0.5;
}

float g4( in float x )
{
    x = sqrt(0.5+0.5*x);
    return x*(x*(x*(x*-0.008978+0.039075)-0.107071)+0.576974)+0.5;
}
```

Wavelet Image Compression

与许多其他基于信号处理的压缩技术一样，wavelet 利用了这样一个事实：对人类来说，大多数信号都是平滑的。这意味着通常一个样本值(比如一个像素的颜色)与邻近的样本(比如像素周围的颜色)相似。然而，大多数基于傅里叶的压缩技术，比如声音的mp3或图像的JPG，甚至像PNG这样的无损技术，都只能在非常简单的层次上利用这种平滑性。实际上，所有这些技术都将信号(声音或图像)划分为采样块或像素块(例如，以8x8像素为一组)，并且仅利用该区域内的平滑度，而没有利用样本/像素可能相似的事实跨越那些边界，而且规模更大（想想几乎是完全蓝天）。wavelet压缩使我们能够在水平上利用平滑度，无论是2x2样本的微小群组，还是一次平面图像的所有像素。因此，与基于块的方法相比，压缩率得到了进一步提高。这意味着，在需要极端压缩的情况下，wavelet确实可以超越传统技术。

这个想法是首先从一个灰度图像开始，然后像处理PNG图像压缩器那样:选择缓冲区并将像素分组到2x2的块中。现在，如果您只存储每个tile的四个像素的平均颜色，那么您已经压缩了1:4。好。当然，图像分辨率降低了。让我们通过以紧凑的方式存储4个像素的实际值来修复它。因为这些像素在物理上相互接近，我们可以很有把握地假设它们的颜色与我们已经编码的平均颜色相似。因此，与其将这些像素存储为完整的灰度值，不如只存储它们与平均颜色之间的差异。事实上，我们只需要存储其中的三个，因为第四个可以从其他三个推导出来因为我们知道它们是平均值。因此，让我们将这三个值(可以是正的或负的，但也可能很小)存储到一个单独的数组中，这样我们就有了平均像素数组，以及三个差异数组。这三个数组压缩得非常好，因为它们的值都很小，图像分辨率越大，这个值就越真实。

在我们不止以PNG开头，而是要注意，平均像素阵列实际上是一幅新图像，其像素数量是原始图像的四分之一-只是其下采样版本。这意味着我们可以简单地重复此过程，以生成一个新的三元组细节值和一个新的平均图像，该图像将再次是当前图像大小的四分之一，就像在纹理的mipmap链中一样。这个过程可以反复进行，直到我们得到一个一像素一像素的图像为止



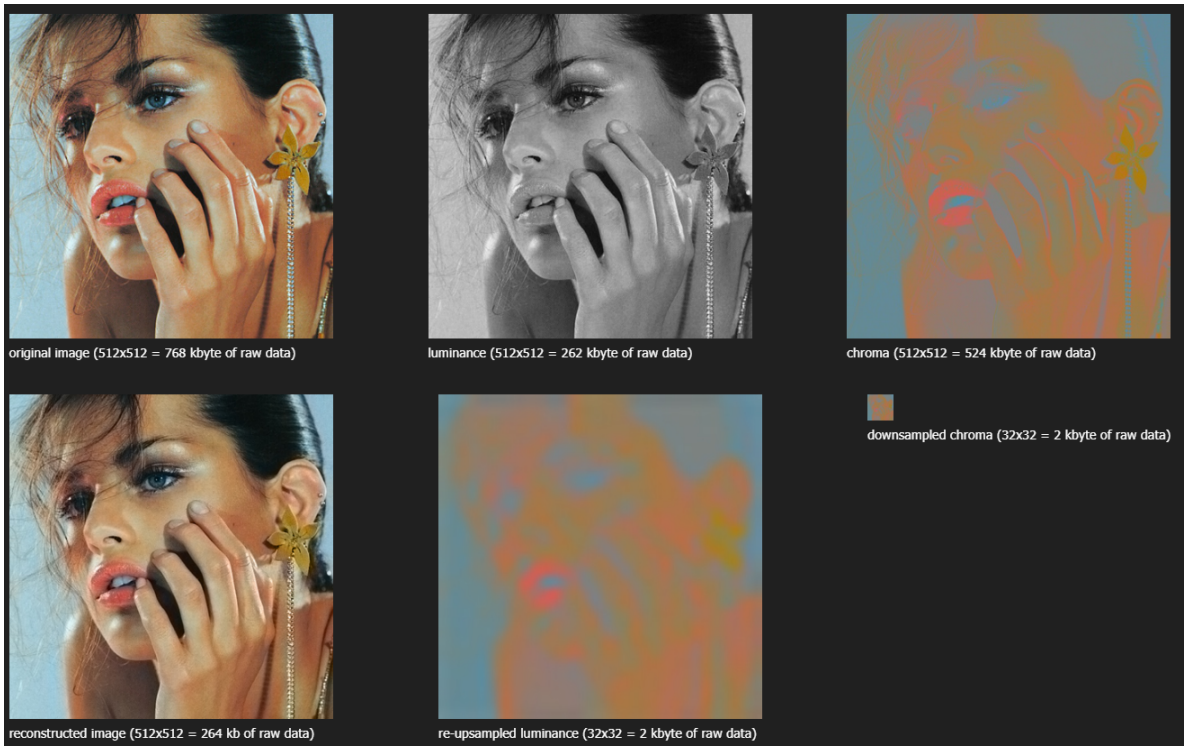
$$\begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 3 & -1 & 1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

$$\begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

$$a' = \frac{1}{4}(a+b+c+d) \qquad b' = a' - \frac{b+d}{2} \qquad c' = a' - \frac{c+d}{2} \qquad d' = a' - \frac{b+c}{2}$$

最酷的部分是这个矩阵等于它自己的逆矩阵(1/4的比例因子)，所以在解码的时候，你可以把a'， b'，c'和d插入矩阵，然后得到a, b, c和d。

对于彩色图像，将使用非常标准的方法，使存储颜色非常便宜，几乎是免费的。但将rgb图像分解为三个独立的灰度图像的方法是一个非常糟糕的主意，**永远**不要这样做。相反，将使用流行的亮度/色度分解，就像JPG一样。该技术所依赖的原理是眼睛对亮度/亮度值最敏感，而对颜色/色度则不太敏感。因此，人们可能可以提取图像（灰度版本）及其色度的亮度，以全分辨率存储亮度并降低色度，对色度进行升采样并将其与原始亮度重新组合后，可以获得良好的图像。



简单的颜色变化

```
vec3 rgb2yuv( const vec3 & rgb )
{
    return vec3( dot(rgb,vec3(0.25f, 0.5f,0.25f) ),
                 dot(rgb,vec3(0.00f,-0.5f,0.50f) ),
                 dot(rgb,vec3(0.50f,-0.5f,0.00f) ) );
}

vec3 yuv2rgb( const vec3 & yuv )
{
    return vec3( dot(yuv, vec3(1.0f,-0.5f, 1.5f) ),
                 dot(yuv, vec3(1.0f,-0.5f,-0.5f) ),
                 dot(yuv, vec3(1.0f, 1.5f,-0.5f) ) );
}
```

lyapunov fractals

李亚普诺夫指数是一种数学工具，用来测量数据或轨道序列中的chaos。它经常被用来绘制Mandelbrot and Julia sets，或者研究一维chaos映射。但在90年代，它被德国的马里奥·马库斯教授用来创造一些漂亮的图像。

根据维基百科定义，最大李雅普诺夫指数定义为：

$$\lambda = \lim_{t\to\infty} \lim_{\delta Z_0 \to 0} \frac{1}{t} \ln \frac{|\delta Z(t)|}{|\delta Z_0|}$$

> **李亚普诺夫特征指数**（Lyapunov characteristic exponent）用于量化动力系统中无限接近的轨迹之间的分离率。具体而言，相空间中初始间隔δZ_0的两条轨迹的分离率为（假定分离可按线性近似来处理）：$|\delta Z(t)| \approx e^{\lambda t}|\delta Z_0|$

对离散时间系统（映射或迭代）$x_{n+1} = f(x_n)$和以x_0为起始的轨迹，上式可以转换成：

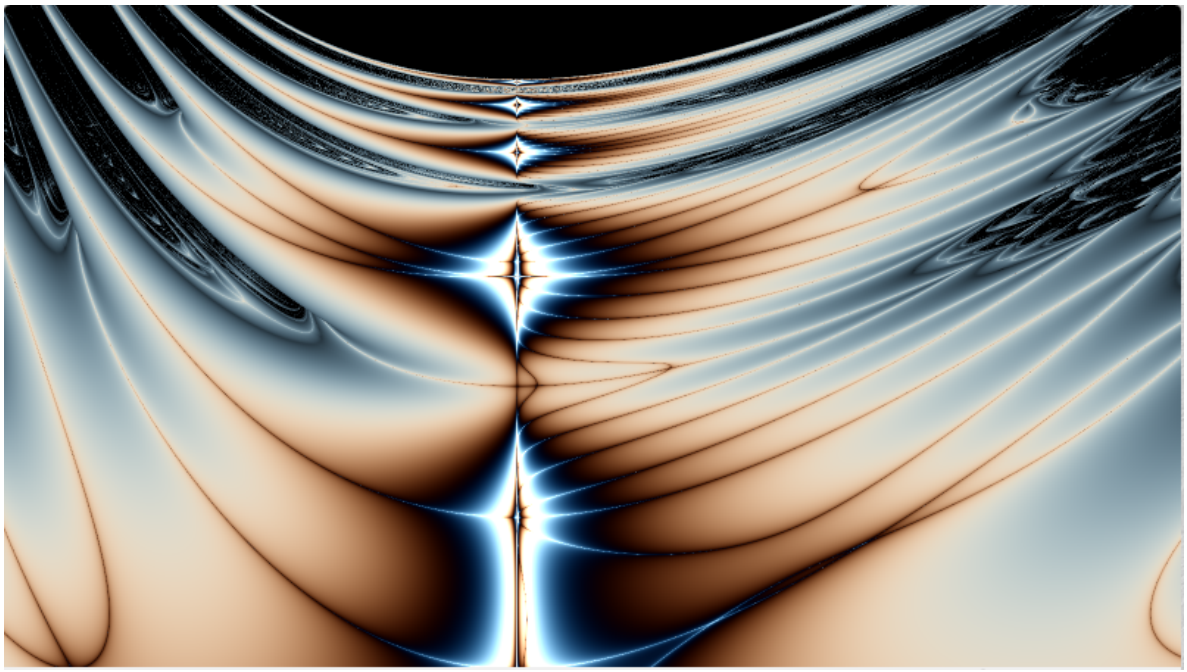
$$\lambda(x_0) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} \ln |f'(x_i)|$$

其物理意义可以参考知乎问题如何理解李雅普诺夫稳定性分析

对于用来生成图像的李雅普诺夫分形算法，可以参看维基百科，大致过程如下：

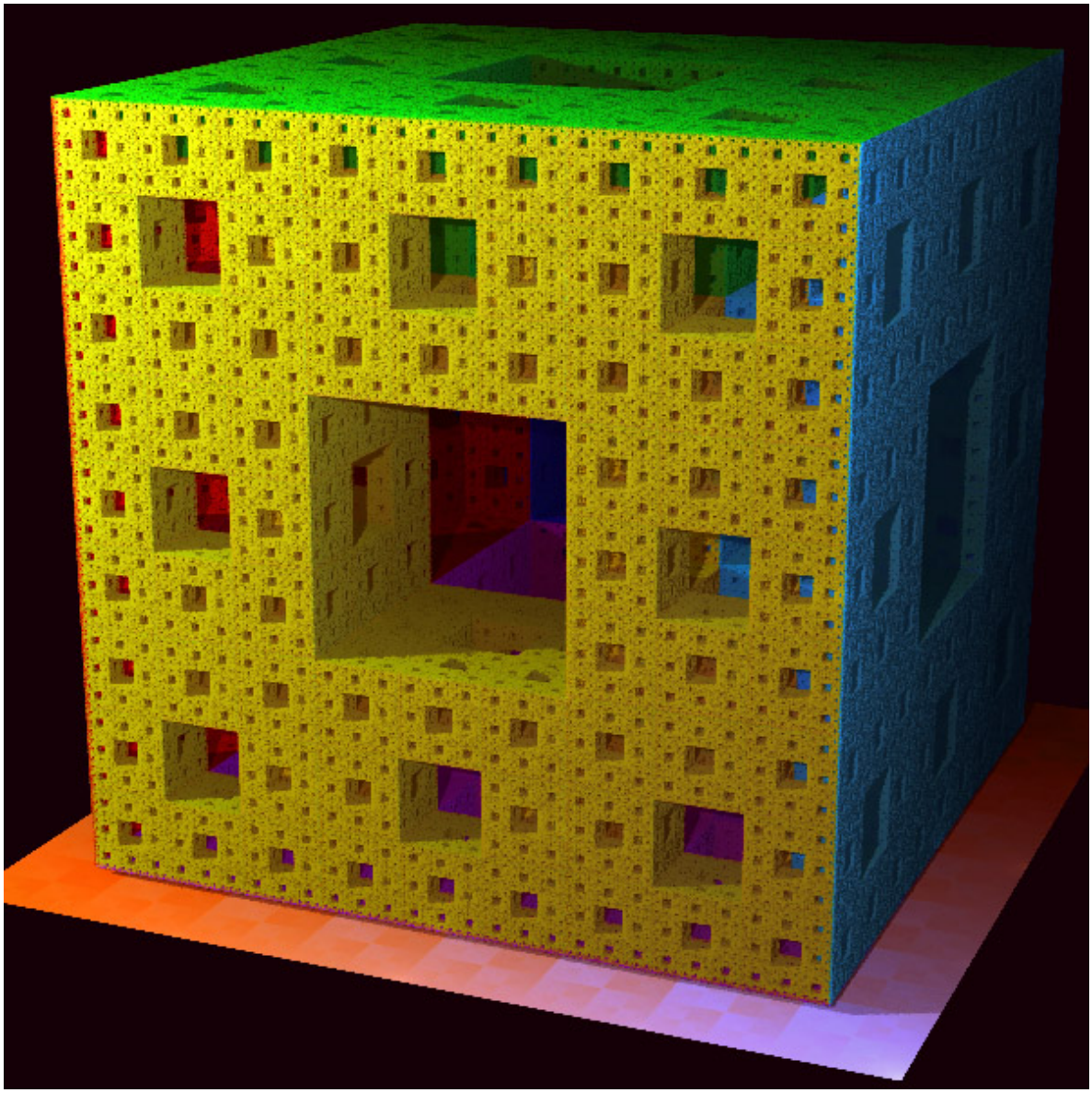
1. 选择一串任意长度的As和B（例如AABAB）。
2. 构建序列 S 由字符串中的连续项形成，并根据需要重复多次。
3. 选择一个点 $(a, b) \in [0, 4] \times [0, 4]$。
4. 定义功能 $r_n = a$ 如果 $S_n = A$和 $r_n = b$ 如果 $S_n = B$。
5. 让 $x_0 = 0.5$，并计算迭代次数 $x_{n+1} = r_n x_n (1 - x_n)$。
6. 计算李雅普诺夫指数：
   $$\lambda = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \log \left| \frac{dx_{n+1}}{dx_n} \right| = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \log |r_n(1 - 2x_n)|$$
   在实践中，λ 通过选择一个适当的大来近似 N 并删除第一个被要求为 $r_0(1 - 2x_0) = r_n \cdot 0 = 0$ 对于 $x_0 = 0.5$。
7. 着色点 (a, b) 根据值 λ 获得。
8. 对图像平面中的每个点重复步骤（3–7）。

效果图



ifs fractals

IFS即Iterated function system，最常见的分形处理



Mandelbrot

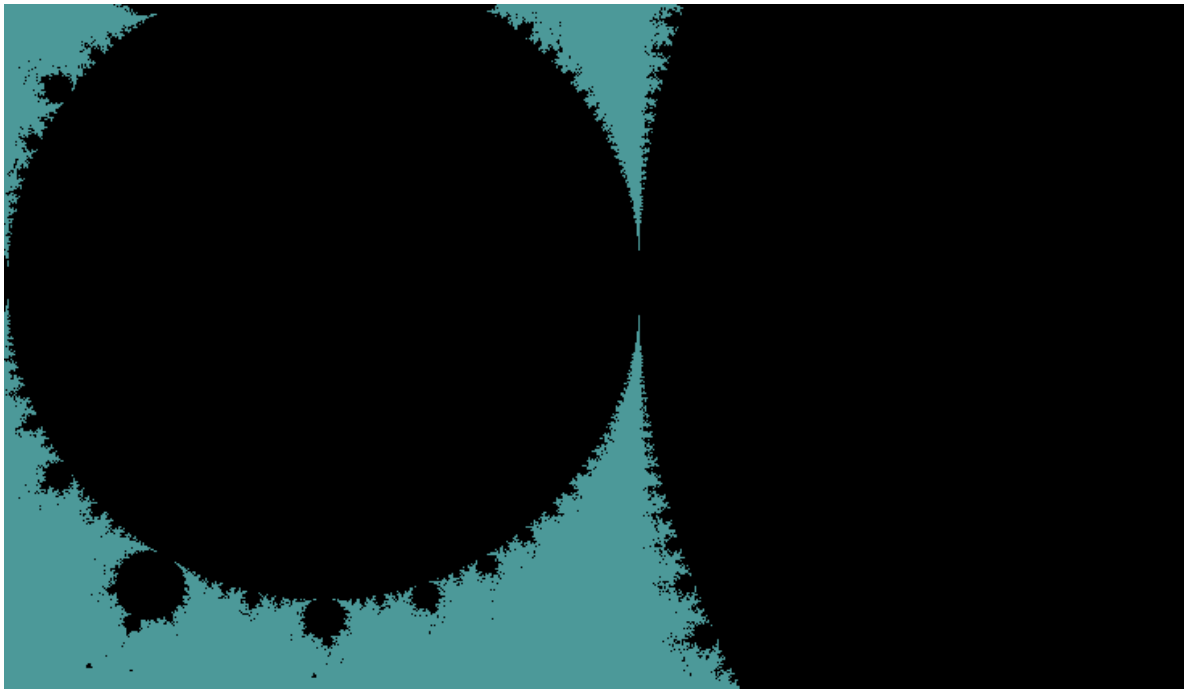
曼德勃罗集合（Mandelbrot Set）或曼德勃罗复数集合。是一种在复平面上组成分形的点的集合，因由曼德勃罗提出而得名。曼德博集合能够使复二次多项式$f_c(z) = z^2 + c$进行迭代来获得。当中，c是一个复参数。对于每个c。从 z = 0 开始对$f_c(z)$进行迭代。序列$(0, f_c(0), f_c(f_c(0)), \dots)$的值，或者延伸到无限大，或者仅仅停留在有限半径的圆盘内（这与不同的参数c有关）。曼德布洛特集合就是使以上序列不延伸至无限大的全部c点的集合。z*z不是向量乘法，而是复数乘法。以及一些性质如下：

> 若$|c| \leq \frac{1}{4}$，则$c \in M$;
>
> 若$c \in M$，则$|c| \leq 2$;
>
> 若$c \in M$，则$|Z_n| \leq 2, (n = 1, 2...)$;

```
#define MAX_NUM 128.0

float Mandelbrot(vec2 p,vec2 point)
{
    vec2 z=vec2(.0,.0);
    vec2 c=p;
    vec2 dis;
    float i=.0;
```

```
    float dist=1e20f;
    for(;i<MAX_NUM && dot(z,z)<1e4;i++)
    {
        z=vec2(z.x*z.x-z.y*z.y,2.*z.x*z.y)+c;
        if(dot(z,z)>2.)return 0.0;
        dis=z-point;
        float s_dis=sqrt(dot(dis,dis));
        if(s_dis<dist)dist=s_dis;
    }
    return dist;
}

void mainImage( out vec4 fragColor, in vec2 p )
{
    //位置偏移+比例*（放大点确定）*缩放比例（最大放大2500倍）
    p = vec2(-.745,.186) + 3.*(p/iResolution.x-.5)*pow(.01,1.+cos(.2*iTime));

    vec2 point=vec2(0.5,0.5);

    float flag=Mandelbrot(p,point);

    fragColor= flag*vec4(0.3,0.6,0.6,1);
}
```

运行效果图:



IQ大神效果图

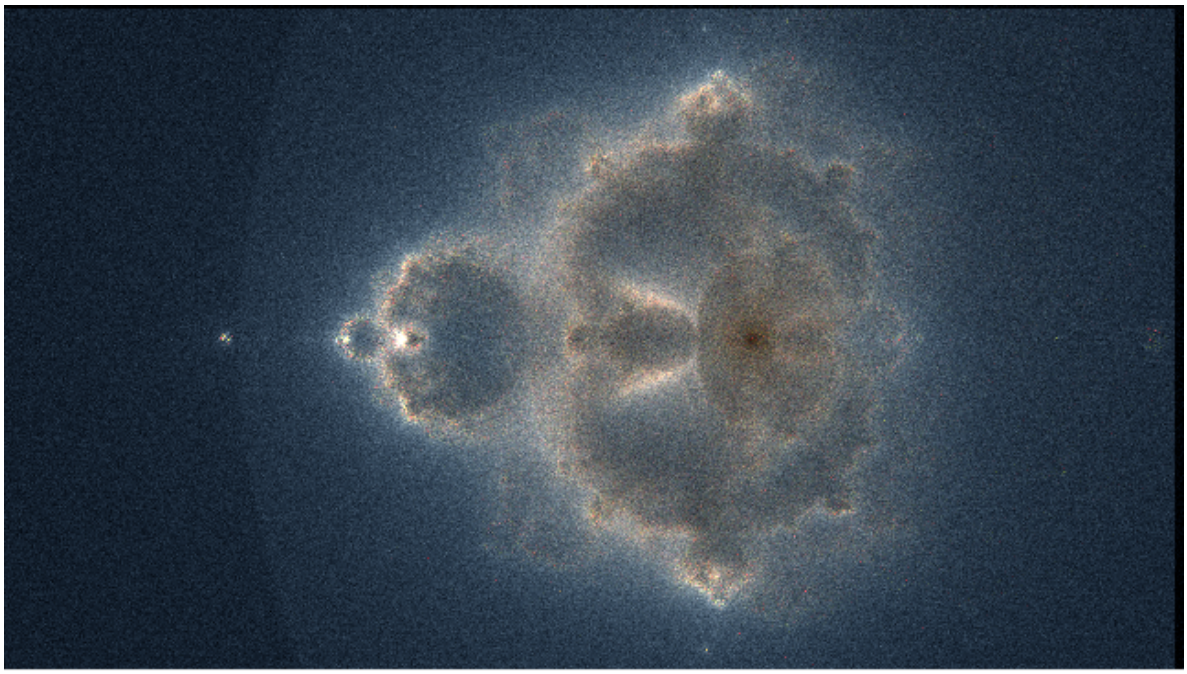
相关链接:

1. 神奇的分形艺术4
2. 维基百科

Budhabrot

这个想法是在复杂平面上迭代每个点，就像在绘制具有给定的最大迭代次数限制的标准Mandelbrot集一样。每个迭代点将照常遵循轨道运行，而我们要做的是检测所有逃逸轨道并将这些轨道的每个点记录在密度图中。因此，与IFS分形一样，我们将缓冲区初始化为零，并且每当一个逃逸轨道的一个点落入缓冲区时，我们就会增加缓冲区的每个像素。理想情况下，我们希望对复杂平面进行统一采样。这意味着在我们完全采样平面之前，我们不会获得正确的图像。您可能想要做的是将其实现为渐进式渲染器，以便在计算外观时可以感觉到最终外观，就像在基于Montecarlo的raytracers中一样。

个人理解：但是要注意的是，这里面我们不关心图片具体的坐标，我们关心的是逃离成功的点的轨迹（指的是每次迭代的值，例如说某个随机值C，带入后，当n=20时逃离，那么轨迹是（Z0，Z1，Z2.....Z20）,然后根据每个轨迹值，映射到具体的像素坐标，对应的值加一，最后依据这些值，确定每个像素的灰度值）



```
int Iterate(double x0,double y0,int *n,XY *seq)
{
   int i;
   double x=0,y=0,xnew,ynew;

   *n = 0;
   for (i=0;i<NMAX;i++) {
      xnew = x * x - y * y + x0;
      ynew = 2 * x * y + y0;
      seq[i].x = xnew;
      seq[i].y = ynew;
      if (xnew*xnew + ynew*ynew > 10) {
         *n = i;
         return(TRUE);
      }
      x = xnew;
      y = ynew;
   }

   return(FALSE);
}
```

Julia Set

朱利亚集合可以由下式进行反复迭代得到：

$$f_c(z) = z^2 + c$$

对于固定的复数c，取某一z值（如$z = z_0$），可以得到序列

$$z_0, f_c(z_0), f_c(f_c(z_0)), f_c(f_c(f_c(z_0))), \ldots.$$

这一序列可能發散于无穷大或始终处于某一范围之内并收敛于某一值。我们将使其不扩散的z值的集合称为朱利亚集合。

一个简单实现如下：

```
#define MAX_NUM 128.0

float Mandelbrot(vec2 p)
{
    vec2 z= p;
    vec2 c=vec2(0.285,0.01);
    float i=.0;
    for(;i<MAX_NUM && dot(z,z)<1e4;i++)
        z=vec2(z.x*z.x-z.y*z.y,2.*z.x*z.y)+c;

    return i/MAX_NUM;
}

void mainImage( out vec4 fragColor, in vec2 p )
{
    //位置偏移+比例*（放大点确定）*缩放比例（最大放大2500倍）
    p = vec2(-.45,.996) + 4.*(p/iResolution.x-.5)*pow(.02,1.+cos(.2*iTime));
    float flag=Mandelbrot(p);
    fragColor= flag*vec4(0.3,0.6,0.6,1);
}
```

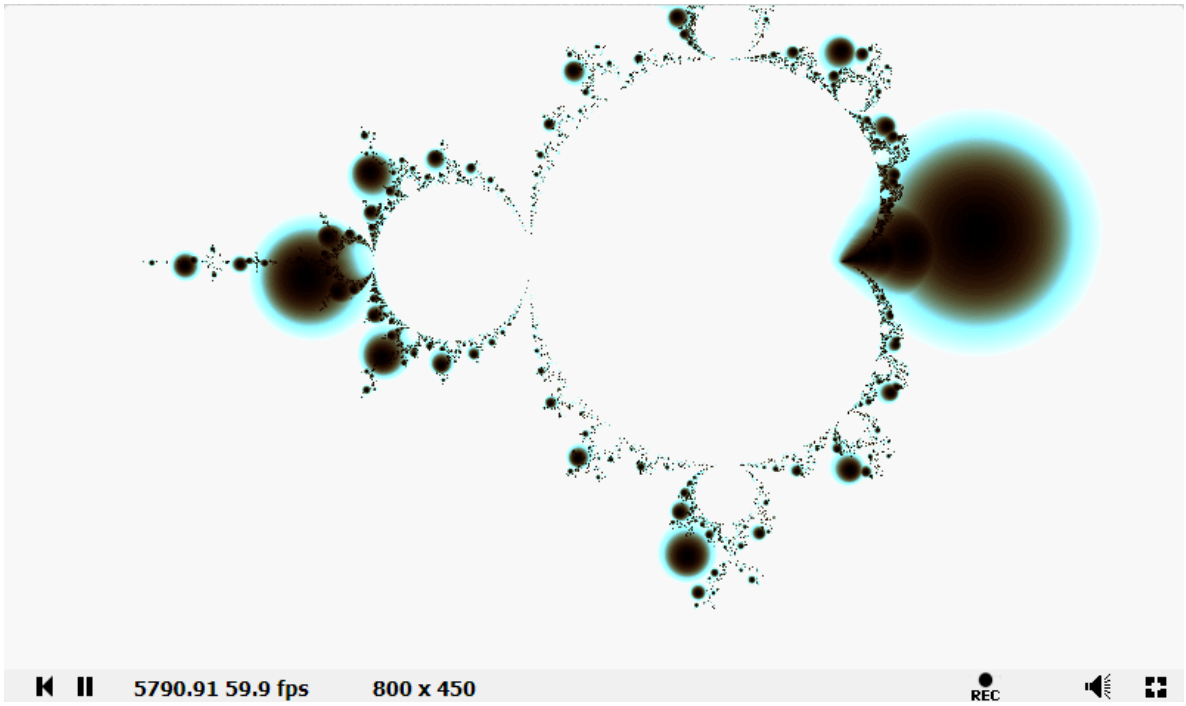
Geometric Orbit Traps

它背后的想法很简单，但是它提供的创建结构和颜色图案的可能性是无限的，给定一个轨道，轨道（c）= {0，c，c 2 + c，...}，我们可以决定研究在平面上绘制时该轨道的样子。确实，这将是某种排序的点云，对于我们开始的每个点/像素c而言，云都是不同的。实际上，如果像素c属于Mandelbrot集（内部），则轨道（c）会很无聊，因为轨道的点将收敛至固定点或将在多个固定点之间振荡，该数量取决于轨道的球面但是，如果被遮挡的c点属于Mandelbrot集的外部，则轨道将变得更加有趣，从某种意义上说，轨道中的点在逃逸至无限。原始点越靠近设置的边界，此轨道将越长。我们正在寻找的是以*某种方式*分析此轨道并为每个轨道提取不同的颜色。

> So, given an orbit orbit(c) = { 0, c, c2+c, ... } we can decide to study how this orbit looks like when drawn in the plane. Indeed, it will be some sort point cloud, a different one for every point/pixel *c* we start with. In fact, if the pixel *c* belongs to the Mandelbrot set (the interior) then orbit(c) will be something quite boring, as the points of the orbit will converge to a fixed point or will oscillate between a number of fixed points, which amount depends in the bulb of the set we are in. However, if the point *c* being shaded belongs to the exterior of the Mandelbrot set, then the orbit will be quite more fun, in the sense that the points in the orbit will fly all over the plane before they escape to infinity. The closer the original point to

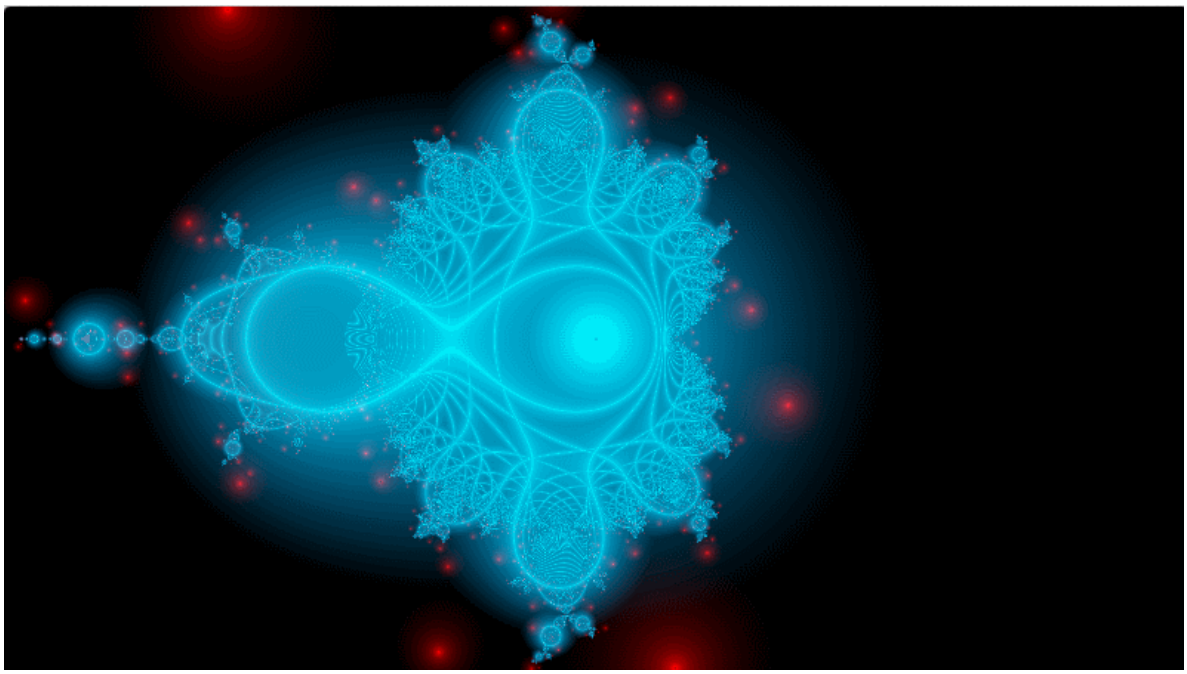
> the set border, the longer this orbit will be. What we are seeking for is to analyze *somehow* this orbit and extract a different color for each.

一个简单的尝试



IQ大神的花纹

代码简单但依然不错的符号



PS：笔记2结束了，但是分形还没有完，感觉笔记3估计全是分形了，哈哈哈哈，什么时候能盲写IQ大神的例子呢

总结：以前看到图形学的分形这块，也就了解皮毛——不就是自相似迭代吗？无知让人自大，哎哎。