IQ大神博客阅读心得1

the game of life：一个简单有趣的算法
Plane deformations：较为古老，但效果不错，基于UV坐标重定位的特效技术
CellEffect：有机会深入研究
Simple color palettes：简单的调色板技术，用于物质的色变（草、木，石头等）
clever normalization of a mesh：网格中点的法向量的获取与优化分析
improved texture interpolation：主要介绍了SM的原理和应用背景。实用的小技巧
Catmull-Rom splines：简单的样条，应用于曲线拟合
Ray and Polygons：基本内容，主要介绍引擎中的深度含义和计算（图形学基础）
Avoiding rigonometry：以旋转为例介绍用点，叉乘代替三角函数的基本思路
fixing frustum culling：对于大物体进行视椎体剔除的小技巧

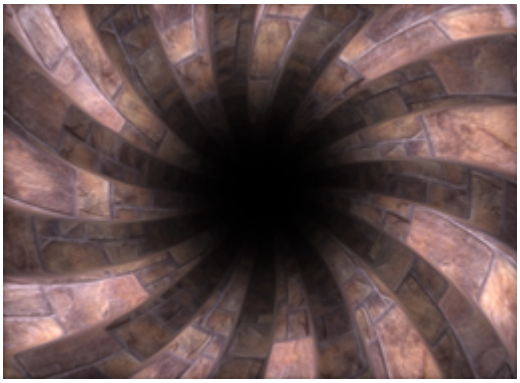
1. simple effects

the game of life

网格中的每个像元可以是无效的（0）或有效的（1）。现在，我们将执行一次迭代（生成），每个单元的状态将根据其当前状态和其周围单元的状态而变化。有一些简单的规则可以驱动该过程：如果一个单元还活着，而八个周围的单元中有两个或三个还活着，则保持活动，否则死亡。如果该单元已经死亡，但八个相邻单元中有三个处于活动状态，则该单元将恢复活动。网格中的所有单元都更新后，我们便有了新一代，可以重新开始了。这个过程永远重复，因此零和一的模式会演变，成型结构等。当然，您需要两个缓冲区，一个缓冲区拥有当前代的单元格，另一个缓冲区将要计算新的单元格

Plane deformations

最好的老式效果之一是二维LUT变形和隧道。这些都是非常快速的实时计算，也非常出色，因此在当时非常普遍。这个想法是取一个纹理，将笛卡尔坐标转换为极坐标，再依靠数学公式修改极坐标，并随着时间的推移以几种方式变形，以创建各种形状，例如花朵，假的3D风景，隧道，水坑，洞等。我也经常使用它们，在1999年和2003年， 2000年，在《*Rare*或*Storm*》等几个演示中。

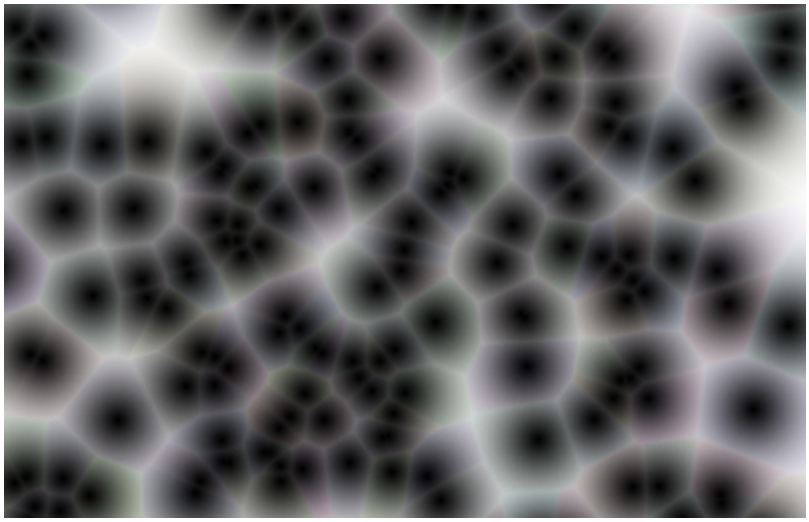


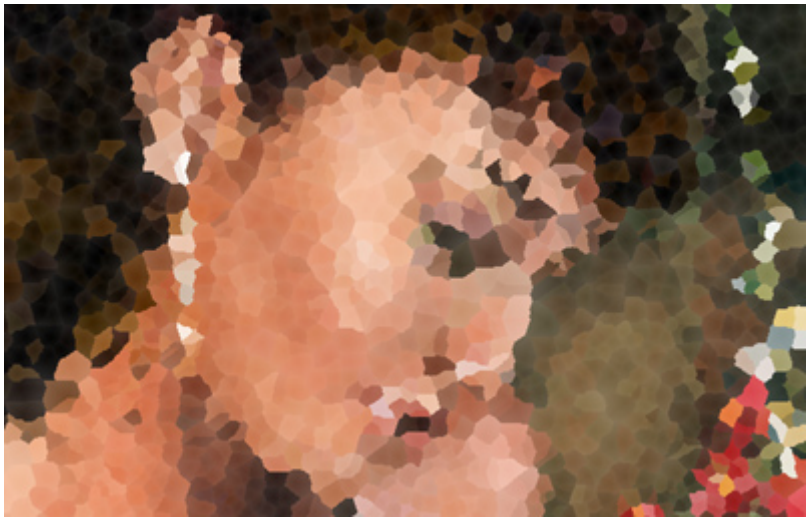
以下是一些简单的UV变化函数：

$$
\begin{aligned}
u &= x * cos(2 * r) - y * sin(2 * r) \\
v &= y * cos(2 * r) + x * sin(2 * r) \\
u &= 0.3/(r + 0.5 * x) \\
v &= 3 * a/pi \\
u &= 0.02 * y + 0.03 * cos(a * 3)/r \\
v &= 0.02 * x + 0.03 * sin(a * 3)/r \\
u &= 0.1 * x/(0.11 + r * 0.5) \\
v &= 0.1 * y/(0.11 + r * 0.5) \\
u &= 0.5 * a/pi \\
v &= sin(7 * r) \\
u &= r * cos(a + r) \\
v &= r * sin(a + r) \\
u &= 1/(r + 0.5 + 0.5 * sin(5 * a)) \\
v &= a * 3/pi \\
u &= x/abs(y) \\
v &= 1/abs(y)
\end{aligned}
$$

CellEffect

蜂窝模式，或更确切地说是Voronoi图，经常出现在计算机图形学和机器人技术中。给定一组特征（点，线，圆，对象等），此图表示给定距离度量（例如规则的欧几里得距离）到最近的特征的距离。当用颜色表示距离时，将获得通常的细胞纹理，这对于生成过程纹理（用于岩石，瓷砖，树木，皮肤等）非常有用。





2 Coding Tricks

Simple color palettes

在进行过程图形处理时，最好将颜色变化添加到图像元素中。从草叶到草丛，再到岩石，树木或山丘，所有尺度的细节都受益于某些颜色变化。为了实现这一目标，当然可以采用不同的方法，但是在最便宜和最易于编码的方法中，有一些是基于简单（硬）编码公式的方法。通常，这些颜色变化将是微妙而重要的，并且可以通过使用可更改元素（草，岩石或其他元素）基色的调色板来实现。简单的加法或调制（乘法）就足够了-通常，无需先将颜色转换为HSV并在转换回RGB之前在该空间中进行色相或饱和度处理。创建程序调色板也很有趣，可以对灰度级且没有自然着色的信号进行着色，例如密度图或分形。本文介绍了一种可能的方法，可以以一种便宜的方式（如前所述）用一个简单的公式来计算用于调制或可视化的调色板。

$$color(t) = a + b \cdot cos[2\pi(c \cdot t + d)]$$

```
vec3 pal( in float t, in vec3 a, in vec3 b, in vec3 c, in vec3 d )
{
    return a + b*cos( 6.28318*(c*t+d) );
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 p = fragCoord.xy / iResolution.xy;

    // animate
    p.x += 0.01*iTime;

    // compute colors
    vec3                col = pal( p.x,
vec3(0.5,0.5,0.5),vec3(0.5,0.5,0.5),vec3(1.0,1.0,1.0),vec3(0.0,0.33,0.67) );
    if( p.y>(1.0/7.0) ) col = pal( p.x,
vec3(0.5,0.5,0.5),vec3(0.5,0.5,0.5),vec3(1.0,1.0,1.0),vec3(0.0,0.10,0.20) );
```

```
    if( p.y>(2.0/7.0) ) col = pal( p.x,
vec3(0.5,0.5,0.5),vec3(0.5,0.5,0.5),vec3(1.0,1.0,1.0),vec3(0.3,0.20,0.20) );
    if( p.y>(3.0/7.0) ) col = pal( p.x,
vec3(0.5,0.5,0.5),vec3(0.5,0.5,0.5),vec3(1.0,1.0,0.5),vec3(0.8,0.90,0.30) );
    if( p.y>(4.0/7.0) ) col = pal( p.x,
vec3(0.5,0.5,0.5),vec3(0.5,0.5,0.5),vec3(1.0,0.7,0.4),vec3(0.0,0.15,0.20) );
    if( p.y>(5.0/7.0) ) col = pal( p.x,
vec3(0.5,0.5,0.5),vec3(0.5,0.5,0.5),vec3(2.0,1.0,0.0),vec3(0.5,0.20,0.25) );
    if( p.y>(6.0/7.0) ) col = pal( p.x,
vec3(0.8,0.5,0.4),vec3(0.2,0.4,0.2),vec3(2.0,1.0,1.0),vec3(0.0,0.25,0.25) );


    // band
    float f = fract(p.y*7.0);
    // borders
    col *= smoothstep( 0.49, 0.47, abs(f-0.5) );
    // shadowing
    col *= 0.5 + 0.5*sqrt(4.0*f*(1.0-f));
    // dithering
    //col += (1.0/255.0)*texture( iChannel0,
fragCoord.xy/iChannelResolution[0].xy ).xyz;

    ragColor = vec4( col, 1.0 );
}
```

clever normalization of a mesh

一般来说，计算点法线的方法是：对于网格中的一个点，获取由它组成的所有的面的面法向量（这个容易，一个三角形面片abc的法向量n：a × b），累积法线，然后除以累积中使用的面数。但实际上，思考一下，法线的长度并不重要，而法线累加之后，方向已经确定，因此最后的除法可以舍去。需要均一化吗？回到叉乘，两条边叉乘的法线的长度为$absin\theta$，而这个长度很明显是正比于三角形面片的面积，那么，按照一般的经验，面积越大的三角形面片的法向量对于点法向量的贡献是不是更大呢？因此，我们只需要在最后对结果进行均一化，而单独的每次面法向量计算，则无需。

```
void Mesh_normalize( Mesh *myself )
{
    Vert     *vert = myself->vert;
    Triangle *face = myself->face;

    for( int i=0; i<myself->mNumVerts; i++ ) vert[i].normal = vec3(0.0f);

    for( int i=0; i<myself->mNumFaces; i++ )
    {
        const int ia = face[i].v[0];
        const int ib = face[i].v[1];
        const int ic = face[i].v[2];

        const vec3 e1 = vert[ia].pos - vert[ib].pos;
        const vec3 e2 = vert[ic].pos - vert[ib].pos;
        const vec3 no = cross( e1, e2 );

        vert[ia].normal += no;
        vert[ib].normal += no;
        vert[ic].normal += no;
    }
```

```
    for( i=0; i<myself->mNumVerts; i++ ) verts[i].normal = normalize(
verts[i].normal );
}
```

improved texture interpolation

当前的图形卡功能非常强大，但这仅是因为它们可以完成“简单”的事情。换句话说，GPU仅对那些可以使游戏看起来很棒的事物具有快速实现。当然不用抱怨。但是，当需要其他某些东西，而又有些其他东西超出了游戏的通常要求时，事情就不会那么顺利了。例如，纹理过滤是在游戏中看起来不错的东西之一，但实际上，当前实现的质量对于其他目的而言是很差的。本文介绍如何改进在硬件中实现的线性滤波。

线性插值比对纹理采样的更简单的最近舍入方法要好得多。但是，线性插值只是一阶多项式插值，因此重新采样纹理的斜率/导数是分段常数。这意味着当使用线性插值的高度图纹理生成某些法线/凹凸贴图或浮雕滤镜时，会出现不连续性（因为法线/凹凸贴图或浮雕滤镜是微分算子。通常应该使用三次三次插值（或什至更高阶的插值器）来获得平滑的结果。但这需要额外的硬件，更重要的是，还需要更多的带宽，因为每次像线性插值一样，不仅必须访问四个纹理像素，而且必须访问十六个。如今，GPU几乎受到内存访问时间的限制，因此，将一个纹理提取所需的带宽乘以4听起来并不是一个好主意（您可以在Wikipedia文章中了解三次插值）。因此，我们要做的就是使用GPU提供的双线性过滤：

```
vec4 getTexel（vec2 p）{return texture2D（myTex，p）；}
```

但是有一个技巧是着色器编写人员非常熟悉的，那就是在执行线性插值时使用smoothstep()作为中间步骤。SmoothStep通常用于在输入mix()或lerp()之前 fade the interpolation parameter。因为这是一条平滑的变化曲线，它消除了线性插值的尖锐边缘。实际上，平滑函数的导数在插值边界处是固定的(在本例中为零)

$$sm(x) = x^2(3 - 2x)$$

x = 0和**x = 1**的值均为**0** ，这是通常插值范围（0..1）的极值。因此，如果我们有办法修改硬件的插值因子，那么我们能在纹理插值中应用smoothstep（）技术，而插值因子当然是在硬件的纹理单元中进行深层计算的。我们当然不能访问硬件的这一部分（至少不是现在），但是我们可以人为地修改传递给texture2D（）的纹理坐标以获得相同的效果。

当硬件通过*texture2D（）*函数接收纹理坐标**p**时，它首先计算采样点所在的纹理像素。那通常是介于两者之间像素，这就是插值进入游戏的地方。四个最接近的纹理元件是基于整数部分选择，并在邻近于每个纹理像素的基础的，所述四种颜色混合以得到最终的颜色。直接从采样位置的小数部分**f*** （**p = i + f***），这就是线性关系代表的直接关系，这就是我们要欺骗的东西。我们希望随着采样点的增加，四个纹理像素的影响逐渐消失，但不是呈线性方式。因此，我们必须使用渐变曲线对小数部分进行预失真，这就是平滑步长可以提供帮助的地方。当然可以使用除平滑步长以外的其他曲线。我更喜欢使用五次度曲线，该曲线不仅在插值极值上具有零导数，而且在该位置具有二阶导数。这样可以确保照明（取决于法线）始终保持平滑。

$$sm(x) = x^3(6x^2 - 15x + 10)$$

```
vec4 getTexel( vec2 p )
{
    p = p*myTexResolution + 0.5;

    vec2 i = floor(p);
    vec2 f = p - i;
    f = f*f*f*(f*(f*6.0-15.0)+10.0);
    p = i + f;

    p = (p - 0.5)/myTexResolution;
    return texture2D( myTex, p );
}
```

Catmull-Rom splines

本样条不提供切线控制，但足够简单和实用，能够插值3维，甚至n维点，IQ大神提到他用这个对照相机进行控制

1. 建立一个三次多项式：$p(t) = a + b \cdot t + c \cdot t^2 + d \cdot t^3$
2. 导数：$p^{'}(t) = b + 2c \cdot t + 3d \cdot t^2$
3. 确保t=0时，曲线通过第一个点p1
4. 确保t=0时，这个曲线有且切线$(p2 - p0)/2$
5. 确保t=1时，曲线通过第二个点p2
6. 确保t=1时，曲线有切线为$(p3 - p1)/2$
7. 求解参数如下

$$
\begin{aligned}
a &= 2 \cdot p1 \\
b &= p2 - p0 \\
c &= 2 \cdot p0 - 5 \cdot p1 + 4 \cdot p2 - p3 \\
d &= -p0 + 3 \cdot p1 - 3 \cdot p2 + p3
\end{aligned}
$$

Ray and Polygons

基本内容。假设您正在对片段着色器中的某些对象进行光线进给或光线追踪，并且想要将它们与您已渲染或将通过常规栅格化渲染的其他某些几何图形进行合成。唯一需要做的就是在光线跟踪/行进着色器中输出一个深度值，然后让深度缓冲区完成其余的工作。在raytracer / marcher中，您可能可以访问从射线原点（相机位置）到最近的几何图形/相交点的距离。那个距离**不是**您要写入深度缓冲区的内容，因为硬件光栅化程序（OpenGL或DirectX）不存储到相机的距离，而是存储几何相交点的z（下面的Z是点视觉空间的Z坐标）

```
float a = (far+near)/(far-near);
float b = 2.0*far*near/(far-near);
gl_FragDepth = a + b/z;
```

Avoiding Trigonometry

IQ在这里表达了自己不喜欢三角函数的一些原因，并就旋转提出了一些看法

```
mat3x3 rotationAxisAngle( const vec3 & v, float a )
{
    const float si = sinf( a );
    const float co = cosf( a );
    const float ic = 1.0f - co;

    return mat3x3( v.x*v.x*ic + co,       v.y*v.x*ic - si*v.z,    v.z*v.x*ic +
si*v.y,
                   v.x*v.y*ic + si*v.z,   v.y*v.y*ic + co,        v.z*v.y*ic -
si*v.x,
                   v.x*v.z*ic - si*v.y,   v.y*v.z*ic + si*v.x,    v.z*v.z*ic +
co );
}
.....
const vec3   axi = normalize( cross( z, d ) );
const float  ang = acosf( clamp( dot( z, d ), -1.0f, 1.0f ) );
const mat3x3 rot = rotationAxisAngle( axi, ang );
```

对于上述常规的计算，明显存在多余步骤：反cos函数求出角度，后面又使用cos，但可能会有质疑：不求出角度，怎么计算sin值？但仔细想想，对于Sin值可以有：$si = length(cross(z,d))$，因此，我们求旋转等方法可以直接：

```
mat3x3 rotationAlign( const vec3 & d, const vec3 & z )
{
    const vec3  v = cross( z, d );
    const float c = dot( z, d );
    //k=(1-cos)/sin
    //const float k = (1.0f-c)/(1.0f-c*c);
    const float k = 1 / (1.0f + c);

    return mat3x3( v.x*v.x*k + c,     v.y*v.x*k - v.z,    v.z*v.x*k + v.y,
                   v.x*v.y*k + v.z,   v.y*v.y*k + c,      v.z*v.y*k - v.x,
                   v.x*v.z*K - v.y,   v.y*v.z*k + v.x,    v.z*v.z*k + c    );
}
```

fixing frustum culling

一般的视椎体剔除，是将物体的点和视椎体的六个面进行比较判断，这对于小物体的剔除是合适的，但据IQ举例，大物体在某些情况下会导致剔除错误（为什么也没怎么看懂），因此需要额外增加一个简单的点范围测试。