MIT 6.837 作业实验报告

1. 实验内容概述

1.1 分形（预备作业）

这是一个预备作业，不涉及到课程的核心内容。作业的具体内容是利用 IFS(Iterated Function System)画一些自相似的分形的图形。

1.2 光线追踪（Assignment 1~7，核心内容）

光线追踪算法的实现和优化是整个课程的核心内容。其基础包括根据相机投射射线，射线与空间物体求交，交点处利用 Phong shading 光照模型计算光照以及计算阴影。这样实现的光线追踪不能模拟自然中光线的反射和折射，不能实现镜面或者透明的物体的效果，为解决这一问题，改进了算法采用递归的方法，射线与物体求交后，该交点处的颜色由三部分组成，第一部分是自身用光照模型着色的颜色，第二部分是反射光线的颜色（结果需要乘反射的系数），第三部分是折射光线的颜色（需要乘折射的系数），这三部分组合起来才构成一个交点的颜色。后两部分需要继续进行光线追踪，因此是一个递归的调用。

第二个改进是速度的改进。由于场景中很可能有大量三角形面片，如果每一次都全部求交将涉及大量运算，而这些运算有很大一部分是不需要的。作业中涉及了一种优化方法，将空间划分成均匀大小的立方体（三维网格），然后将物体放入与之相交的格子中，这样，与一条射线有交点的物体必须存在于他经过的格子中，于是只需要对路径上的格子中的物体求交。

除了上述改进，另外一个优化不仅仅属于光线追踪，而是对信号采样都有效的优化——反走样。光线追踪得到的是一个二维的连续函数，这个函数根据光线追踪算法来定义，并且作为光线追踪算法的结果其实是已经完成了光线追踪的工作了，剩下的就是将这个函数显示出来，这就必须涉及到采样，于是立刻就有走样的问题，需要进行反走样。

1.3 OpenGL 的使用和 rendering pipeline（Assignment 3）

OpenGL 是一套图形 API，在作业中使用 OpenGL 的方式类似于传递命令，一部分函数用来设置状态，一部分函数用于传输数据，例如顶点坐标。使用的模式一般就是，打开（或关闭）某些开关，设置参数，传输数据。这一部分了解还比较浅，作业中也一般以基础代码的形式提供，需要下个作业的时候更加深入学习。

渲染管线是对场景中每一个对象做变换得到屏幕空间中的对象，同时能够计算顶点的属性值并对三角形面片进行插值计算像素。

1.4 曲线和曲面（Assignment 8）

作业中主要使用到两种曲线：Bezier 曲线和 B-spline 曲线。这两种曲线性质有一些不同，作业中主要讨论的是两种曲线的 3 次的版本。三次 Bezier 曲线有 4 个控制点，如果需要更多的控制，需要一段一段地拼接多条 Bezier 曲线，每两段曲线之间只共享一个点。而且多端拼接的 Bezier 曲线只能保证 C0，如果在拼接处让控制点中心对称，可以保证 C1 连续性，但是 C2 以及更高的连续性不能保证；B-spline 曲线也是 piece-wise 多段曲线拼接成的，每一段有 4 个控制点，但是相邻两段曲线共享 3 个控制点，增加控制点和删除控制点都非常方便，而且 B-spline 曲线可以保证更好的连续性。

不过实际使用的时候 Bezier 曲线由于经过第一个和最后一个控制点，因此操作起来比较方便，相比而言，B-spline 并不经过控制点，因此调整起来比较麻烦。生成曲面的时候，这个特征也很方便，因此 Bezier 曲线很容易拿来生成曲面。

1.5 粒子系统

粒子系统的基础是解微分方程，例如根据牛顿定律加速度是速度的导数，加速度根据人为的定义在空间中是已知的，给定初始条件求解时刻 t 的速度就是解一个微分方程。粒子的状态，包括某一时刻的位置、速度等其实都由微分方程的解确定。因此推动粒子系统其实就是要计算包含状态的微分方程的解，并随时间更新这些状态。

作业中的粒子系统包括几个基本组成部分，第一是粒子生成器，粒子系统会每隔一段时间调用例子生成器产生粒子，第二是力场，立场决定了每一个位置的加速度，因此决定了速度，从而决定了例子的位置，最后体现在视野中的就是例子位置的移动。第三是为了近似微分方程解使用的积分器，作业中使用到了 4 种积分器，分别是 Euler, midpoint, trapezoid 以及 Runge Kutta 积分器，他们的作用就是近似微分方程的解，而微分方程的解就描述了粒子的状态，包括位置、速度、颜色等。

2. 原理分析

2.1 利用 IFS 画分形图形

虽然是第一个作业，但是其原理却很困难，由于这个作业本意是在熟练使用 C++，所以这一部分忽略。

2.2 Ray Casting

Ray Casting 指的是从相机投射射线（加入反射和折射后，也可以从反射位置和折射位置），并且求射线与场景中物体的交点。求射线是非常简单的，只需要将屏幕坐标系中的点映射点空间中对应像平面上的点，正交投影只需要加上方向，透视投影只需要和相机位置做减法求出方向，然后直接构造射线即可。

真正要实现的是几种 Primitive（基本几何图形）的求交算法。

2.2.1 球

球是里面最容易求交的一种几何图形了，球的方程和直线联立之后可以得到一个一元二次方程，直接使用公式求解即可。若是判别式 $\Delta<0$，说明没有交点，如果判断式>0，有两个交点，=0 有一个交点，也就是处于相切的位置。解出解之后需要和 tmin 作比较，tmin 指的是合法的交点最小的 t 值，丢弃掉<tmin 的解后，如果还剩两个交点，那就需要找出较小的一个了；或者如果解都<tmin，那么则不存在交点。

2.2.2 平面

平面一般写作 Ax+By+Cz+D=0，其实可以写成点积的形式(A,B,C)*(x,y,z)+D=0。于是将任何一个点代入方程中就可以算出该点到平面的距离（有方向，且有系数，系数是(A,B,C)的模长）。于是平面求交可以有很好的几何方法，将射线端点 O 代入方程除以射线的方向向量在(A,B,C)上的投影，这个时候算出来的结果需要再反转正负号就得到了 t（这个时候问

题其实已经转化为一维的了（例如投影的位置为 x，射线的端点投影到了一维的数轴上，而与平面的交点则在投影在数轴的原点处，所以一维问题可以直接求解，(0-x)/t，于是需要加负号）。我认为这种方式比单纯的代数解法更有规律性。

2.2.3 三角形求交

三角形求交我使用了最直接的代数的方法，联立方程，三角形的平面可以用带有两个自由元的方程表示，射线有一个参数 t，这样就有 3 个未知数，三个方程联立得到了一个 3 元线性方程组，我用了最简单的高斯消元法来计算。

最后可以解出 3 个参数 t,u,v. t 和 tmin 比较判断这个点是不是在相机前方，u 和 v 用来判断交点是不是在三角形内部，如果 u>=0, v>=0, u+v<=1，那么说明该交点在三角形内部。上述条件都满足，才能判断有交点。

2.2.4 变换后图形的求交

基本的图形可以经过一系列变换得到其他形状或不同位置的图形。对于平移旋转这一类的变换由于不改变形状，因此还可以将变换直接应用到图形上得到的还是原来类型的图形。但是对于轴向不均匀的缩放，例如将圆变成椭圆的变换，就没办法直接用变换应用在图形上然后消除变换了。

这个时候就需要反过来思考，不是在世界坐标系中用变换后的图形与直线求交，而是将直线逆变换到物体所在的局部坐标系求交，求完了交点再变换到世界坐标系中。这里就要求直线逆变换后得到的还是一条直线，而不能是一条曲线，而建模使用的大多数变换都是满足这样的条件的（仿射变换）。

变换后的图形求交还有一个问题，就是法向量的计算，法向量不能通过简单在局部坐标求出法向量然后变换到世界坐标系中。要计算两个坐标系中法向量的关系，需要进行一个简单的推导，这里直接使用课件上的推导，如图 1.

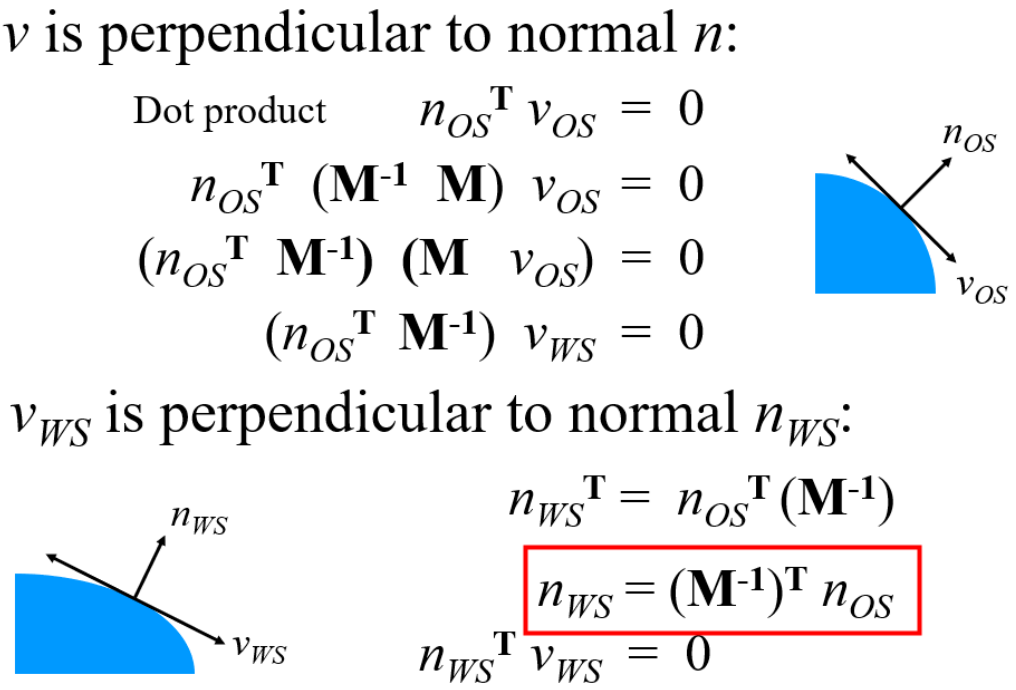


图 1. 局部坐标系的法向量变换到世界坐标系中的推导，M 是变换的矩阵表达，由此可见，这个矩阵必须是可逆的

2.3 Phone shading

Phong shading 光照模型其实非常简单，除了 ambient 之外，就两个部分，第一部分是漫反射光，物体表面一个点的漫反射光只由光源的颜色和光源的方向决定，与视线方向无关，所以有漫反射的效果；第二部分是镜面反射光，其值与光源颜色、方向以及视线的方向都有关系，而且越接近镜面反射的角度亮度就越大，偏离该角度后，亮度下降很快，于是有光亮镜面的效果。

$$L_o = k_s(\cos\beta)^q \frac{L_i}{r^2} = k_s(\mathbf{n}\cdot\mathbf{h})^q \frac{L_i}{r^2}$$

这是改进后的计算镜面部分的公式，原来的公式是以理想反射光的方向为基础计算衰减的，但是这有个问题，就是角度大与 90° 时这个分量就变成 0 了，这样会影响效果；采样这种改进的方法可以让整个 180° 的范围都有值，因此更加平滑。Lo 表示镜面反射成分，h 向量是视线方向 v+光源方向 L 后向量的单位向量，q 控制光亮衰减的速度，q 越大光亮区域就越小，q 越小光亮区域就越大，q 减为 0 就跟漫反射光的方程一样了。方程最右边$\frac{Li}{r^2}$表示光源强度是平方衰减的，这是符合物理规律的，但是实际上由于衰减太快，往往并不会使用平方衰减。

2.4 包含反射和折射的光线追踪

自然界中广泛存在反射和折射的现象，也就是说表面的颜色不仅仅被光源照亮，有可能他是半透明的，因此折射光对该处颜色有贡献，如果镜面，则反射光有贡献。所以这是一个非常直接的扩展，直接将原来的光线追踪算法增加两条光线追踪这两个成分就可以了，实际上实现起来非常简单。

2.5 光线追踪加速

作业中使用的其实是非常简单的一种空间划分方式，将场景的包围盒用三维网格来划分，于是场景中的物体可以放入与之相交的网格中，而一次光线投射产生的直线只会与少量网格相交，而且还有一个非常重要的性质，三维网格是空间有序的。因此如果顺着射线的方向遇到第一个网格，那么这个网格肯定比后面格子都要靠前，于是只要交点在这个格子内部，肯定可以宣告找到最近的交点了，而不需要继续找下去了，反之则需要继续找下一个格子。考虑原来的无序的对象，先找到交点的对象可能反而在比较远端的位置，因此不得不遍历整个 group 才能判断最近的交点，这样就导致效率非常低。所以，这样一种序的关系对优化提供了便利。

另一个关于投射 shadow ray 的优化作业中并没有提到，但其实可以在一定的条件下使用。假设光源在场景的 boundingbox 之外，那么如果 shadow ray 在于网格中物体相交的过程中发现了交点，不管这个交点在不在该网格内部，他肯定在 boundingbox 内部，所以必然遮挡光源，而原来利用网格加速的算法，如果交点在网格外部，则需要继续判断下一个网格，需要更多的时间才能宣告有交点。但是这个方法必须保证光源在场景的 boundingbox 外部，如果光源在里面则可能产生错误的阴影。

2.5 光线追踪反走样

通过光线追踪算法可以得到一个二维连续图像，但是这个图像的频率有可能很高，如果用较低的频率去采样，就会出现走样的情况，例如锯齿、摩尔纹。其实不仅仅是光线追踪，走样产生的原因是信号采样频率不足导致的，因此这个优化其实都可以用到的解决信号走样的问题。

在采样频率不变的情况下，如果要去除走样，就要想办法去除原图像中的高频成分，只留下低频成分，这样即使是低的采样频率也不会产生走样。要做到这个效果，就需要使用低通滤波器，对原图像做一个卷积，比较常用的比如 box filter，tent filter 以及 Gaussian filter，卷积之后的图像还是一个二维连续函数，再对其采样，就可以减少走样。

但是这明显是无法做到的，因为卷积是一个二元函数的积分运算，这是非常困难甚至无法计算的。在作业中，采用的策略是像素内多个位置不同采样，然后每个像素的颜色值是 filter 的函数计算权重之后对半径内的采样的加权和，我觉得这其实就是一种近似的办法。因为卷积操作不能直接进行，于是就通过采样的方法来近似积分的结果，但其实理论还是一样的，就是先卷积再对卷积后的图像采样。

2.4 渲染管线

光线追踪和渲染管线可以算是两种独立的方法来生成图像，光线追踪是从图像空间出发，通过反向追踪获得图像。而渲染管线是从场景出发，对场景中的每一个三角形，经过一系列变换后得到屏幕空间中的图像。这两种方法有很大的不同。

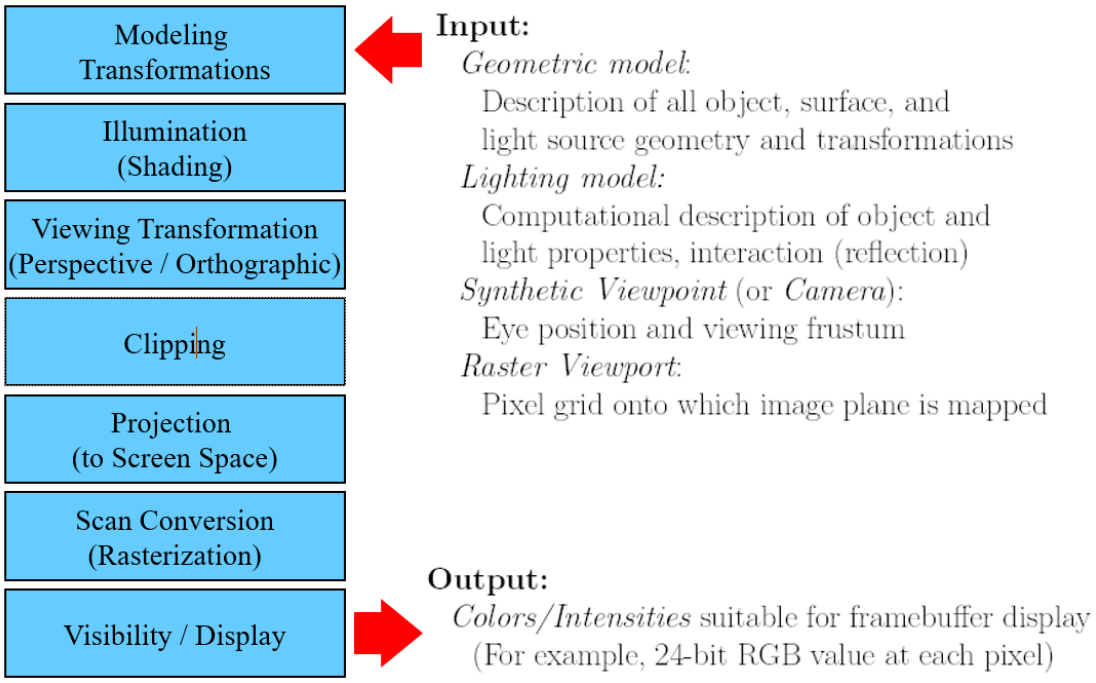


图 2. 图形管线的流程

如图 2 所示，我认为其中最麻烦的一步变换就是 Perspective（透视）变换，因为一般来说 Modeling transformation 都是仿射变换，对空间的变换都是均匀的，因此变换之后可以很方便地进行线性插值。例如有两个顶点 A,B，两个顶点都有颜色， 经过仿射变换后，可以直接在新的坐标系中进行插值，得到中间线段上点的颜色，而进行变换的过程中，颜色等

属性无需特别关注，只需要保存下来就行。

但是如果是透视变换就不行了，因为这个变换不是线性变换，如果还按照上面的方法处理颜色、法向量等信息，屏幕坐标系和原来空间坐标系中的点就不能建立良好的对应关系，就会出现颜色、法向量等效果与实际不符。

解决这个问题的方法也很简单，因为透视变换只会把直线变成直线，平面变成平面（所以说透视矩阵里深度存储为 1/z 是非常有道理的），只是不同的地方伸缩不一样，这是一个非常好的性质，因为这样空间坐标系中的三角形经过透视变换肯定还是一个三角形，于是在屏幕坐标系中也可以通过线性插值的方法获得这个三角形。

如果要计算诸如颜色、法向量、纹理坐标这一类的信息，可以直接将原来三维空间中的点看成是高维空间中的点，包括点的各种属性。而三角形上各个点的属性如果满足线性关系（所以说空间坐标系中各个属性如法向量等都应该是线性插值得到的，这样方便后续处理），那么整个三角形连同这些属性构成的高维向量也是高维空间上的一个平面，于是经过透视变换也是一个高维空间上的平面，这个平面上的点形如(x/z,y/z,1/z,r/z,g/z,…)，这个平面可以通过线性插值得到，得到屏幕空间中的这个三角形上的一个点之后，就得到了形如 x/z 的形式，x 是原始空间坐标系中对应轴的值，由于有 1/z，所以可以很容易恢复出这个点原本坐标中的坐标，这样就可以使用该坐标的属性计算各种信息，解决了透视变换非线性造成的空间中点不对应的问题。

2.5 Bezier 曲线和 B-spline 曲线

这两种曲线都是可以通过控制点来控制其形状，从而用于造型的曲线。比较重要的是如何表示这样的曲线，在课件中使用了一种统一的表示方法 Q=GBT(t)。使用这种方式来表达

$$Q(t) = \begin{pmatrix} Q_x(t) \\ Q_y(t) \\ Q_z(t) \end{pmatrix} = \begin{pmatrix} (P_0) & (P_1) \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} t \\ 1 \end{pmatrix}$$

$$Q(t) = \mathbf{GBT(t)} \ = \text{Geometry } \mathbf{G} \cdot \text{Spline Basis } \mathbf{B} \cdot \text{Power Basis } \mathbf{T(t)}$$

曲线给两种曲线之间的转换带来了很大的方便，而且也更容易看出这些曲线的结构，G 是控制点组成的矩阵，T 是变量 t 的幂次构成的列向量，而 BT(t)通过某种方法把基本的构成多项式的项组合一下，得到了基本的函数，然后通过 G 来组合这些基本函数就得到了需要的函数。可以看到 B 矩阵最后一列元素之和为 1，左边的各列，每一列元素之和都为 0.

作业中还涉及到另一个问题，需要用三角形来表示一张曲面。对于这样的参数表达的平面，这是非常简单的，只需要在(θ,t)的空间中用网格划分曲面，对应到(x,y,z)空间中就是一个曲线网，在(θ,t)空间中的好处就是在这个空间中曲面就是一个简单的二元函数，因此很容易划分曲面，到(x,y,z)空间中就很难确定一个好的曲线网来划分曲面。

2.6 粒子系统

粒子系统的核心就是一个微分方程（包含了粒子的各种属性），整个微分方程定义了粒子的运动状态。然而，实际上无法求解微分方程，所以只能使用一些方法来近似微分方程的解，得到粒子的各种属性，包括位置、速度、颜色等。最简单的也是最容易理解的就是 Euler 方法，通过初始状态计算导数，然后利用导数估计函数值，但这其实完全就是用微分估计一个函数的方法。改进的方法可以降低估计的误差，包括 trapezoid, midpoint 等其他方法。

3. 算法描述附关键代码

3.1 IFS

```
for "lots" of random points (x0, y0)
    for k=0 to num_iters
        pick a random transform fi
        (xk+1, yk+1) = fi(xk, yk)
    display a dot at (xk, yk)
```

```
for (int i=0;i<points;i++) {
    float x=drand(),y=drand();
    Vec2f p(x,y);
    for (int j=0;j<iter;j++) {
        float dr=drand();
        float l=0.0;
        for (int k=0;k<n;k++) {
            if (dr>=l&&dr<l+prob[k]) {
                trans[k]->Transform(p);
                break;
            }
            l+=prob[k];
        }
    }
```

3.2 Ray casting

For every pixel
Construct a ray from the eye
For every object in the scene
Find intersection with the ray
Keep if closest
Shade depending on light, and normal vector

```
float *ts=new float[width*height];
for (int y=0;y<height;y++) {
    for (int x=0;x<width;x++) {
        Vec2f scrCoord((float)x/L,(float)y/L);
        Ray r=cam->generateRay(scrCoord);
        if (cam->getErrCode()==2) {
            cout<<"error:out of range of [0,0] -> [1,1]"<<endl;
            return -1;
        }
        Hit h;
        if (gro->intersect(r,h,cam->getTMin())) {
            img.SetPixel(x,y,h.getMaterial()->getDiffuseColor());
            ts[getIndex(x,y)]=h.getT();
        }
        else {
            ts[getIndex(x,y)]=MAX_T;
        }
    }
}
```

3.3 变换求交

```
Transform.intersect(ray,hit):
    o=origin of ray
    d=direction of ray
    o2=inverse tranform o
    d2=inverse tranform d
    normalize d2
    intersect with object with Ray(o2,d2), get hit h
    if no hit return no_hit
    p=h.getIntersectionPoint
    p2=tranform p
    calculate t from o and p2
    normal=inverse transpose tranform h.normal
    normalize normal
    set hit with t and normal
    return have_hit
```

```
bool Transform::intersect(const Ray &r, Hit &h, float tmin) {
    Vec3f oinv(r.getOrigin());
    minv.Transform(oinv);
    Vec4f dinv(r.getDirection(),0);
    minv.Transform(dinv);
    Vec3f dinv3f(dinv[0],dinv[1],dinv[2]);
    dinv3f.Normalize();
    Ray rinv(oinv,dinv3f);
    if (!obj->intersect(rinv,h,tmin)) return false;
    Vec3f pt(h.getIntersectionPoint());
    mtrans.Transform(pt);
    float t=(pt-r.getOrigin()).Length();
    Vec4f ntr(h.getNormal(),0);
    minvT.Transform(ntr);
    Vec3f ntr3f(ntr.x(),ntr.y(),ntr.z());
    ntr3f.Normalize();
    h.set(t,h.getMaterial(),ntr3f,r);
    return true;
}
```

3.4 Phong Shading

```
Vec3f PhongMaterial::Shade(const Ray &ray, const Hit &hit, const
Vec3f &dirToLight,
    const Vec3f &lightColor) const {
    Vec3f V = Vec3f(0,0,0)-ray.getDirection();
    Vec3f H = dirToLight +V;
    H.Normalize();
    float specIntensity = max(H.Dot3(hit.getNormal()), 0.0f);
    float diffIntensity = max(dirToLight.Dot3(hit.getNormal()),
    0.0f);
    Vec3f specComp = lightColor*(specularColor * pow(
    specIntensity, exponent));
    Vec3f diffComp = lightColor * (diffuseColor*diffIntensity);
    Vec3f phongColor = specComp + diffComp;
    return phongColor;
}
```

3.5 包含折射和反射的 raytracing

```
traceray(ray):
    Intersect all objects
    color = ambient term
    For every light
        cast shadow ray
        color += local shading term
        If mirror
            color += colorrefl * traceray(reflected ray)
        If transparent
            color += colortrans * traceray(transmitted ray)
```

```
if (bounces < max_bounces) {
    float refWeight = pMat->getReflectiveColor().Length(
    float refraWeight = pMat->getTransparentColor().Leng
    weight;
    if (refWeight > cutoff_weight) {
        Vec3f vRef = mirrorDirection(h.getNormal(), ray.
        getDirection());
        Ray rRef(h.getIntersectionPoint(), vRef);
        Hit h2;
        refColor=traceRay(rRef, UNIT_EPS, bounces + 1, r
        indexOfRefraction, 2, h2);
        refColor = refColor * pMat->getReflectiveColor()
    }
    if (refraWeight > cutoff_weight) {
        Vec3f vRefra;
        float index_t = (whichSide) ? 1.0f : h.getMateri
```

3.6 Grid 加速

```
Grid.intersect(ans_hit):
    if not hit with boundingbox
        return no_hit
    cell=find first cell
    has_hit=false
    while not has_hit and cell in boundingbox:
        check hit with objects in cell
        if no_hit continue
        if hit.t > max_t of this cell
            record the hit to avoid intersect again
            cell=nextCell()
        else
            has_hit=true
            set ans_hit to hit
    if has_hit
        return has_hit
    else
        return no_hit
```

```
//be careful not to change Hit &h, if there is no hit
bool Grid::intersectWithObj(const Ray &r, Hit &h, float tmin,float tmax)
const {
    MarchingInfo march;
    initializeRayMarch(march, r, tmin,tmax);
    while (1) {
        if (march.checkHitObjInCell(h,tmin)) {
            return true;
        }
        march.nextCell();
        if (march.hasEnded()) {
            return false;
        }
    }
    //never come to here
    cerr << "Unexpected error in Grid::intersectWithObj()" << endl;
    return false;
}
```

3.7 Supersampling

```
traceToFilm:
    for every pixel
        for every sample
            get offset of sample in pixel
            generate ray from offset and pixel
            color=traceray
            film.set(pixel,sample,color)
    return film

filter(film,img):
    for every pixel in film:
        color=(0,0,0)
        sample_cnt=0
        for every pixel pi around(in which samples may be needed):
            for every sample si in pi:
                if dist to si<support radius:
                    calculate weight wi for si
                    color+=wi*si.color()
                    sample_cnt+=1
        color/=sample_cnt
        img.setColor(this pixel,color)

raytrace:
    film=traceToFilm()
    filter(film,img)
```

```
Film* RayTracer::traceToFilm() const {
    int w = width, h = height;
    int ns = sampler->getNumOfSamples();
    Film *film = new Film(w, h, ns);
    for (int i = 0; i < w; i++) {
        if (i % 20 == 0) {
            cout << "progress: " << (double)i / w * 100 << "%" << endl;
        }
        for (int j = 0; j < h; j++) {
            for (int k = 0; k < ns; k++) {
                Vec2f offset = sampler->getSamplePosition(k);
                Vec2f uc = offset;
                uc += Vec2f(i, j);
                uc = Vec2f(uc.x() / w, uc.y() / h);
                Vec3f samColor;
                Camera* cam = pScene->getCamera();
                Ray r(cam->generateRay(uc));
                Hit h;
                samColor = traceRay(r, cam->getTMin(), 0, 1.0f, 1.0f, 1, h
                , accelerate);
                film->setSample(i, j, k, offset, samColor);
            }
        }
    }
    cout << "progress: 100%" <<endl;
```

3.8 Bezier and B-spline

```
convert_bezier_to_Bspline:
    return Gbezier*Bbezier*inverse(Bbspline)

convert_Bspline_to_bezier:
    return Gbspline*Bbspline*inverse(Bbezier)
```

```
Matrix m = Bspline * invBbezire;
float gspline[16] = { 0.0f };
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++) {
        gspline[i * 4 + j] = vertices[i][j];
    }
}
Matrix G(gspline);
G.Transpose();
G = G * m;
for (int i = 0; i < 4; i++) {
    float tmp[3];
    for (int j = 0; j < 3; j++) {
        tmp[j] = G.Get(i, j);
    }
    fprintf(file, "%f %f %f\n", tmp[0], tmp[1], tmp[2]);
}
```

3.9 粒子系统

$$y_{n+1} = y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$
$$t_{n+1} = t_n + h$$

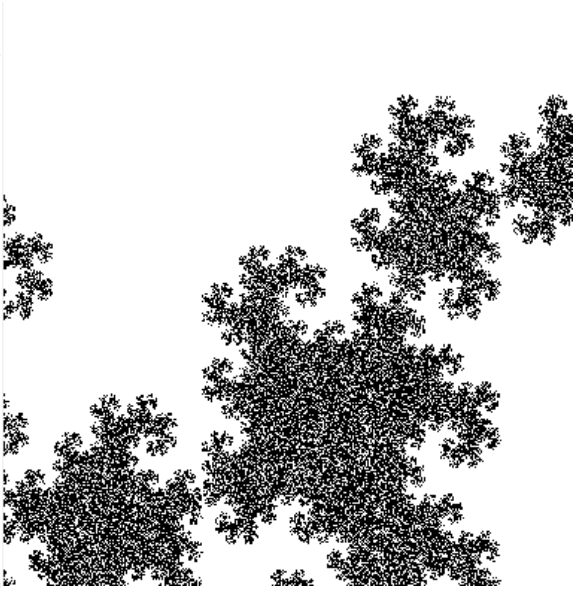
for n = 0, 1, 2, 3, ..., using[3]

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\ k_4 &= f\left(t_n + h, y_n + hk_3\right). \end{aligned}$$

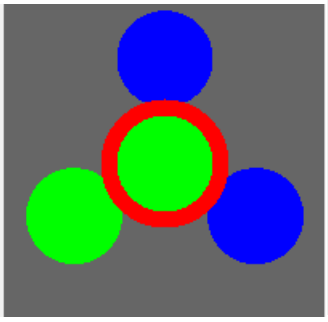
```
void RungeKuttaIntegrator::Update(Particle *particle, ForceField *
forcefield, float t, float dt) {
    //if dead, don't update any more
    if (particle->isDead()) return;
    Vec3f vel = particle->getVelocity();
    Vec3f pos = particle->getPosition();
    Vec3f acc = forcefield->getAcceleration(pos, particle->getMass
    //k1: vel,acc
    //k2
    Vec3f p2 = pos + vel * (dt / 2);
    Vec3f v2 = vel + acc * (dt / 2);
    Vec3f acc2 = forcefield->getAcceleration(p2, particle->getMass
    //k3
    Vec3f p3 = pos + v2 * (dt / 2);
    Vec3f v3 = vel + acc2 * (dt / 2);
    Vec3f acc3 = forcefield->getAcceleration(p3, particle->getMass
    //k4
    Vec3f p4 = pos + v3 * dt;
    Vec3f v4 = vel + acc3 * dt;
    Vec3f acc4 = forcefield->getAcceleration(p4, particle->getMass
```

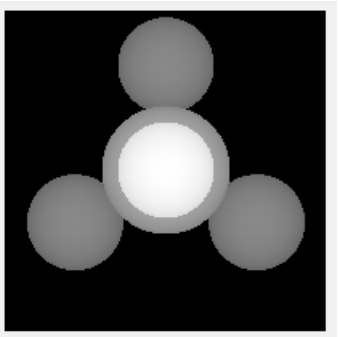
4. 实验结果展示

4.1 assignment 0

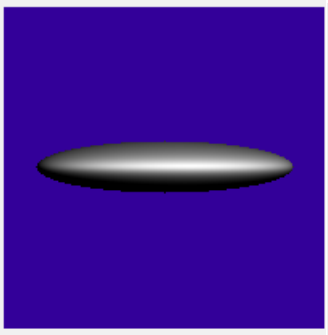


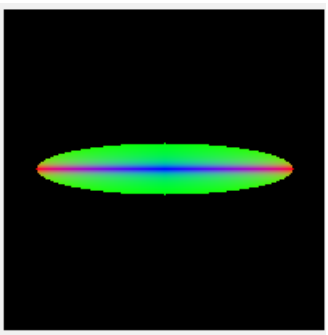
4.2 assignment 1



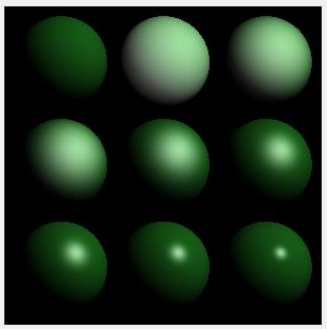


4.3 assignment 2

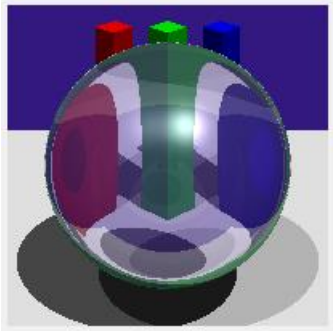




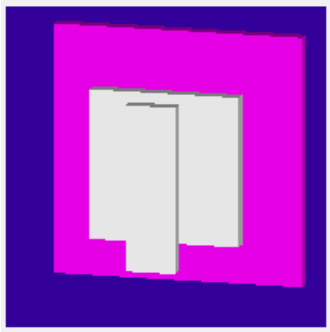
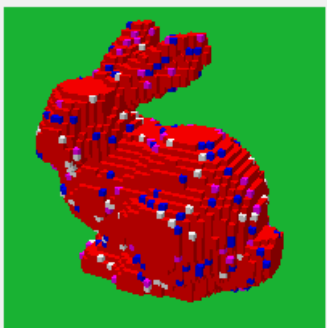
4.4 assignment 3

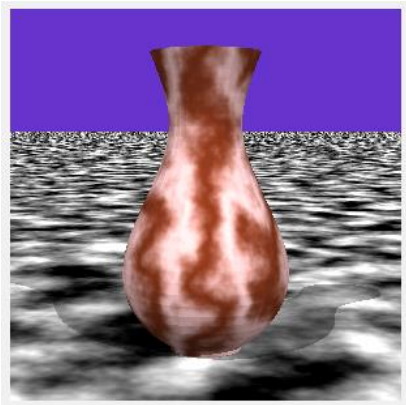
4.5 assignment 4



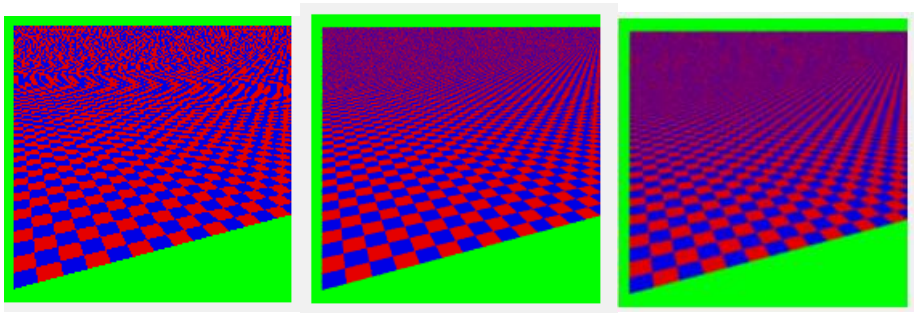

4.6 assignment 5

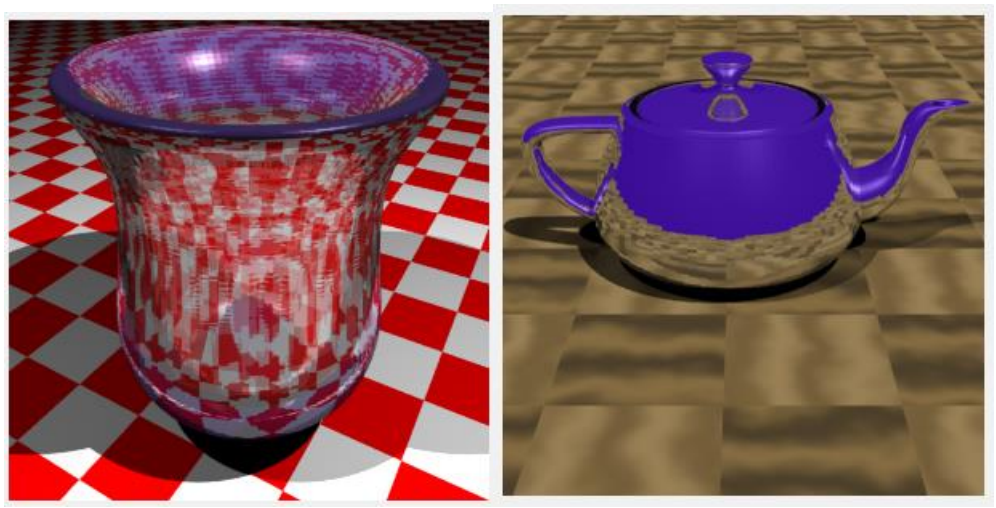
4.7 assignment 6



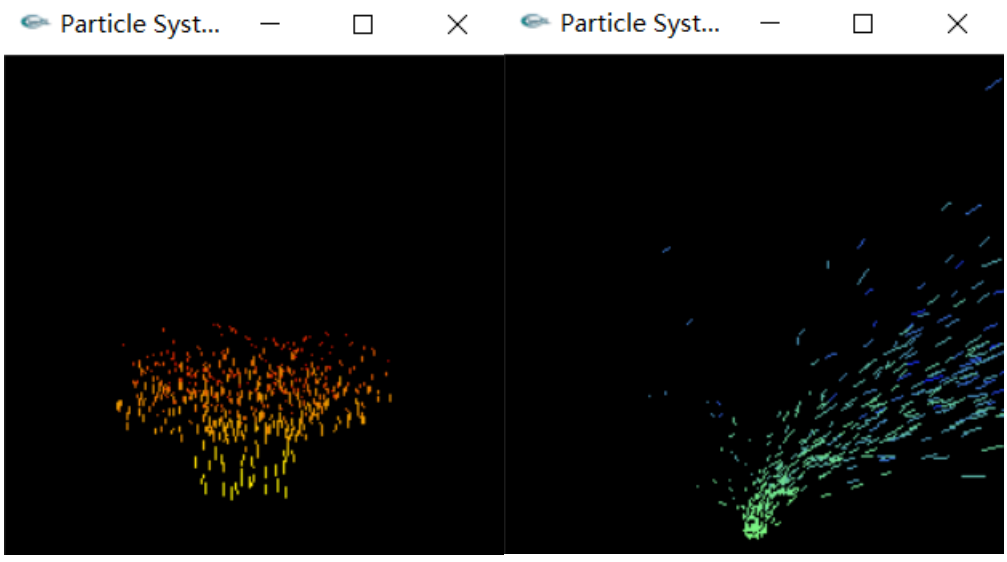

4.8 assignment 7



4.9 assignment 8



4.10 assignment 9



5. 实验遇到的问题

5.1 编译器的问题

在做这系列作业时，遇到的很多问题首先来源于编译器。很重要的一点是 assert 函数非常影响性能，通过性能分析发现，assert 花费的时间甚至比三角形面片求交还要多。还有一点是 fabs 的问题，这个函数应该非常容易实现才对，最多就只需要反转一位符号位就可以了，结果性能分析结果说明 fabs 消耗了大量的 CPU 时间，我把他换成了一个简单的判断加反转正负号的函数，完全就不会消耗多少时间了。

另外一个问题是头文件循环包含的问题，浪费我很多时间，而且编译器给出的问题都是根本没有帮助的，结果导致我一头雾水。最好的方法就是按时间来编写，先写的头文件不能包含后写的头文件，这样肯定能避免循环包含，但是必须加前向声明，同时保持头文件尽量简洁。而且头文件中不能定义函数，定义函数会导致多次定义的问题。

5.2 OpenGL

在使用中发现了几个小问题。

第一，在包含 OpenGL 头文件之前必须包含 windows.h 头文件，否则编译出错。在作业给出的基础代码中有两处似乎有问题。在 assignment 4 中，需要注释掉一行代码。

```
324
325     // Ambient light
326     Vec3f ambColor = scene->getAmbientLight();
327     GLfloat ambArr[] = { ambColor.x(), ambColor.y(), ambColor.z(), 1.0 };
328     glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambArr);
329
330     //glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
331     glCullFace(GL_BACK);
332     glDisable(GL_CULL_FACE);
333
334     // Initialize callback functions
335     glutMouseFunc(mouse);
336     glutMotionFunc(motion);
337     glutDisplayFunc(display);
338     glutReshapeFunc(reshape);
```

另外，有一处代码不符合代码的预期，下图中应当是想要显示为白色（都调用 glColor 了，肯定是想要设置颜色了），但是实际的效果是边框颜色会变成蓝色，通过加入一行代码可以解决这个问题。

```
 8
 9  void BoundingBox::paint() const {
10    // draw a wireframe box to represent the boundingbox
11    glEnable(GL_DEPTH_TEST);
12    glColor3f(1,1,1);
13    glDisable(GL_LIGHTING);
14    glBegin(GL_LINES);
15
16    glVertex3f(min.x(),min.y(),min.z());
17    glVertex3f(max.x(),min.y(),min.z());
18    glVertex3f(min.x(),min.y(),min.z());
19    glVertex3f(min.x(),max.y(),min.z());
```

5.3 perlin noise

在作业中需要使用 perlin noise 生成大理石纹理，使用 perlin noise 计算权重混合两种材质。其中需要调用 noise 函数生成随机值，这个值应该是需要归一化，但是我认为应该除以这个信号的幅值，也就是 1+1/2+1/4+…，但是最后经过反复实验，发现样例的图案只是简单将最后生成的值+0.5，这非常难以理解，因为我发现确实有<-0.5 的随机值产生，这样+0.5 还是小于 0，那不是都超过两种材质的边界了。