A Unifying Paradigm for Ray-Tracing and Rasterization

Frédo Durand
MIT CSAIL

Abstract

Ray casting and rasterization are not so different.

Keywords: Rendering

1 Introduction

The world of rendering is often presented as a battleground between two irreconcilable approaches: ray tracing and rasterization. In this document, we show that they are actually quite similar.

2 Background

2.1 Ray casting

originally: all sorts of primitives. Now: mostly triangles.

Various ray-triangle, one of the good ones: barycentric (see also Tomas's stuff)

Ray casting often not used for primary rays

Discuss Marks and Hunt.

2.2 Rasterization

Old style: scanline, etc. New style: edge equation. Lately, homogeneous rasterization.

3 Main loop

We first analyze the raw algorithm with no acceleration data structures. We only consider primary rays, that is, the computation of what is visible from the viewpoint, ignoring shadows and global illumination. In a nutshell, the main difference between ray casting and rasterization in this context is the order of their main loop.

Ray casting loops over pixels in the image. For each pixel, it generates a ray that gets intersected with all scene primitives. A value `tmin` is maintained to select only the closest intersection:

```
for each pixel
    generate ray
    for each triangle
        intersect(ray, triangle)
        if t<tmin, keep intersection
```

In contrast, rasterization (a.k.a. the graphics pipeline) renders one triangle at a time and displays all the pixels that cover this triangle. Modern rasterization is based on edge equations that test if a pixel is inside the triangle or not. In its simplest form, all pixels are tested against the edge equations. To handle hidden surfaces, a z buffer is used that stores for each pixel the distance to the closest surface. When rendering a triangle, a pixel is updated only if the new z value is closer than the stored one:

```
for each triangle
    for each pixel
        test if pixel passes all three edge tests
        if z<zbuffer[x,y], keep
```

The main difference between ray casting and rasterization for primary visibility is that the two main loops (over pixels and over triangles) are swapped. Incidentally, this means that rasterization needs to store and access randomly the equivalent of the tmin values, which we do with a z buffer. Now let's look closer at the details of teh ray-triangle intersection and that of the edge equations.

4 Ray-Casting

4.1 Ray-Triangle Intersection

A triangle is described by its three vertices A, B, and C. A ray is described by its origin O and its direction D

We use the popular barycentric approach to ray-triangle intersection [**?**]. A point on the plane of the triangle can be described by its barycentric coordinates $(1-\beta-\gamma), \beta, \gamma$. $P = A+\beta AB+\gamma AC$. Points inside the triangle respect $\beta > 0$, $\gamma > 0$ and $\beta+\gamma < 1$. A point along the ray is described by a parameter t so that $P(t) = O + t * D$.

The intersection between the plane and the ray is given by $P(t) = O + t * D = A + \beta AB + \gamma AC$ Or in matrix form:

$$(-AB \; -AC \; D)\begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix} = (OA)$$

(i.e. the leftmost term is a matrix M whose first column is the vector $-AC$, etc.)

we can use Cramer's rule to solve this 3×3 system:

$$\beta = \frac{|OA \; -AC \; D|}{|M|}$$

$$\gamma = \frac{|-AB \; OA \; D|}{|M|}$$

$$t = \frac{|-AB \; -AC \; OA|}{|M|}$$

where $|M|$ denotes the determinant of M One convenient expression of the determinant is the cross product of the two first columns, followed by a dot product with the third column.

This gives us

$$\Delta_M = BA \times CA \cdot D \quad (1)$$
$$\Delta_\beta = OA \times CA \cdot D \quad (2)$$
$$\Delta_\gamma = BA \times OC \cdot D \quad (3)$$
$$\Delta_t = BA \times CA \cdot OA \quad (4)$$

Which gives us the code `intersectWithBarycentric` in Fig. **??**

4.2 Ray Casting main loop

Ray casting loops over the pixels and for each pixel it loops over the triangles and performs ray-triangle intersection.

See the code `raycast` in Fig. **??**

5 Rasterization

5.1 Edge equation test

We start from the triangle-ray intersection code from above and not that when the ray origin is shared by many queries, we can precompute many of the quantities, namely most of the cross products. In what follows I assume that the origin is at (0, 0, -z) but it does not matter much.

For a given pixel, the ray direction is simply $D = (x, y, 1)$

Recall the determinant equations 1 through 4. Only D varies on a per pixel basis. We compute and cache all the rest. Δ_t is a constant.

$$Eq_M = BA \times CA \quad (5)$$
$$Eq_\beta = OA \times CA \quad (6)$$
$$Eq_\gamma = BA \times OC \quad (7)$$
$$\Delta_t = BA \times CA \cdot OA \quad (8)$$

And we get

$$\Delta_M = Eq_M \cdot D \quad (9)$$
$$t = \Delta_t / Delta_M \quad (10)$$
$$\beta = Eq_\beta \cdot D / \Delta_M \quad (11)$$
$$\gamma = Eq_\gamma \cdot D / \Delta_M \quad (12)$$

See the code `setUpTriangle` for the preprocessing part, and `testPixel` for the per-pixel part.

In fact, these equations define perspective-correct (screen-space) interpolation of depth and edge equations. If one removes the division by Δ_M, this is essentially Olano's homogeneous rasterization, except that we have only two edge equations. In the perspective-correct version, we can get away with only two equations because those equations are setup so that they are equal to 1 at the third vertex.

5.2 Rasterization main loop

See the routine `rasterize` in the code to see how rasterization makes use of `setUpTriangle` and `testPixel`. Because the for loops over triangles and pixels are reversed compared to ray casting, we can afford to factorize the preprocessing more easily and do it once per triangle. Note that the same could be done for ray casting, at the cost of additional per-triangle storage. In contrast, rasterization requires per-pixel storage for the z buffer and frame buffer.

Discuss streaming of triangles vs random access. On the other hand, graphics hardware also likes to have the scene in memory.

6 Acceleration

6.1 Space hierarchy

image vs. object hierarchy. See Greene's hierarchical z buffer

Bounding volume hierarchy

Look at hit rate (both image and object).

7 Misc.

Add multipass;: pre-Z, deffered shading, per light pass (deferred lighting)

displacement, tesselation

shading: object vs. image space, especially with derivative

Add discussion of sort last vs sort middle to unified rendering

Either frame buffer or bin data needs to go off chip.

Cacheability, look at last reference of an object, histogram

```
def intersectWithBarycentric (self, triangle, orig, D):
          detM=triangle.BA.cross(triangle.CA)*D
          if fabs(detM)<epsilon: return False, 0
          OA=triangle.A-orig
          detBeta=OA.cross(triangle.CA)*D
          beta=detBeta/detM
          detGamma=triangle.BA.cross(OA)*D
          gamma=detGamma/detM
          detT=triangle.BA.cross(triangle.CA)*OA
          t=detT/detM
          if beta<-epsilon or gamma<-epsilon or beta+gamma>1+epsilon:
               return False, 0
        else: return True, t

def setUpTriangle (self, triangle, orig):
        self.detMEq=triangle.BA.cross(triangle.CA)
        OA=triangle.A-orig
        self.detBetaEq=OA.cross(triangle.CA)
        self.detGammaEq=triangle.BA.cross(OA)
        self.detT=triangle.BA.cross(triangle.CA)*OA

def testPixel(self, D):
        detM=self.detMEq*D
        if fabs(detM)<epsilon: return False, 0
        detBeta= self.detBetaEq*D
        beta=detBeta/detM
        detGamma=self.detGammaEq*D
        gamma=detGamma/detM
        t=self.detT/detM
        if beta<-epsilon or gamma<-epsilon or beta+gamma>1+epsilon:
               return False, 0
        else: return True, t
```

Figure 1: *Code. In yellow are the parts that only depend on the ray origin, while green parts also require the direction (pixel position). In blue are the terms that are preprocessed and cached by the setup in rasterization. * denotes the dot product while* `cross` *is the cross product.*

```
def raycast(scene, width, height):
    im=Image.new('RGB',(width,height))
    for y in range(height):
        for x in range(width):
            dir=vec3(2.0*x/width-1.0, 1.0-2.0*y/height, 1.0)
            tmin=infinity
            for T in scene.triangles:
                test, t=inter.intersectWithBarycentric (triangle, orig, dir)
                if test and t>0 and t<tmin:
                    im.putpixel((x,y), T.shade())
                    tmin=t
    return im

def rasterize(scene, width, height):
    im=Image.new('RGB',(width,height))
    tmin = [[infinity for col in range(width)] for row in range(height)]
    for T in scene.triangles:
        inter.setUpTriangle(T, orig)
        for y in range(height):
            for x in range(width):
                dir=vec3(2.0*x/width-1.0, 1.0-2.0*y/height, 1.0)
                test, t=inter.testPixel(dir)
                if test and t>0 and t<tmin[x][y]:
                    im.putpixel((x,y), T.shade())
                    tmin[x][y]=t
    return im
```