Computer Graphics Notes

by Tyler Wright

github.com/Fluxanoia fluxanoia.co.uk

These notes are not necessarily correct, consistent, representative of the course as it stands today, or rigorous. Any result of the above is not the author's fault.

These notes are in progress.

Contents

1 Data Types

1.1 Objects

Objects are stored in `.obj` files with texture and colour information stored in `.mtl` files. Object files define sets of points and, from these points, elements consisting of triangular faces. Similarly, texture points can also be defined and attached to the faces. Elements can also be assigned materials defined in an inherited `.mtl` which should define the material, giving it a texture path or a colour.

1.1.1 Object-space to World-space Coordinates

We convert object-space coordinates to world-space coordinates by translating, scaling, and rotating our object. This can be done with matrices.

1.2 Colour

Colours can be represented in multiple ways, in `.mtl` files they are three floats ranging from `0.0` to `1.0` to represent red, green, and blue. It's also useful to have the same representation but from `0` to `255` instead with an additional alpha value for transparency.

1.2.1 Packing

We can pack a colour represented by four 8-bit values (`r`, `g`, `b`, `a`) into a 32-bit integer `c` in `C++` with:

```
c = (a << 24) + (r << 16) + (g << 8) + b;
```

This representation is used to set the colour of pixels on the screen.

2 Camera

2.1 Rotation Matrices

We can perform an anti-clockwise 3D rotation of θ degrees using the 3D rotation matrices R_x, R_y, and R_z corresponding to rotations around the x, y, and z axes respectively:

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix},$$
$$R_y = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix},$$
$$R_z = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The 2D anti-clockwise rotation is represented by the matrix R:

$$R = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}.$$

2.2 Camera Position and Orientation

We can simply represent the position of the camera with a 3D vector, which we can translate quite easily. The orientation, however, is more complex and is represented with a 3D matrix C, consisting of vectors r, u, and f, corresponding to the direction 'right' from the camera, 'up' from the camera, and 'forward' into the camera:

$$C = \begin{pmatrix} \uparrow & \uparrow & \uparrow \\ r & u & f \\ \downarrow & \downarrow & \downarrow \end{pmatrix}.$$

This matrix can be rotated using the standard rotation matrices.

2.2.1 Look At

We can set our camera orientation matrix to look at a point by setting the vectors inside it appropriately. The 'forward' vector is the vector from the desired point to the camera. The 'right' vector is the cross-product of $(0, 1, 0)$ and the 'forward' vector. The 'up' vector is the cross-product of 'forward' and 'right'.

2.3 World-space to Camera-space Coordinates

We can apply the translation of our camera by translating vertices by the camera position multiplied by -1. The orientation of the camera can be applied by taking the vector from camera to the vertex and right-multiplying it by C.

2.4 Homogenous Coordinates

The homogenous coordinate system adds a dimension to the camera orientation matrix in order to embed translation information. This allows for all camera information to be stored in a single matrix and the composition of translations and rotations. We can map to homogenous coordinates as follows:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix},$$

$$\begin{pmatrix} \uparrow & \uparrow & \uparrow \\ r & u & f \\ \downarrow & \downarrow & \downarrow \end{pmatrix} \mapsto \begin{pmatrix} \uparrow & \uparrow & \uparrow & x \\ r & u & f & y \\ \downarrow & \downarrow & \downarrow & z \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where x, y, and z store the translation information of the matrix, by default, this would be all zeroes (in rotation matrices for example). We can still perform in-place rotations by simply translating the matrix to the origin before the rotation and then translating it back afterwards.

3 Rasterising

Rasterising is about converting world-space images into screen-space images to draw objects to the screen.

3.1 Interpolation

We can interpolate in one dimension between two floating point values x and y in $s \geq 2$ steps by defining $v_1, \ldots, v_s$:

$$v_i = x + (i-1)\left(\frac{y-z}{s-1}\right),$$

we will write the sequence $(v_i)_{i \in [s]}$ as $I(x, y, s)$ as the interpolation from x to y in s steps.

We can extrapolate this to higher dimensions. For $d > 1$, we have that the interpolation of d-dimensional vectors x and y in s steps is $v_1, \ldots, v_s$ defined as:

$$\begin{aligned} x &= (x_1, \cdots, x_d), \\ y &= (y_1, \cdots, y_d), \\ w_{ab} &= \text{the } b^{\text{th}} \text{ element of } I(x_a, y_b, s), \\ v_i &= (w_{1i}, \cdots, w_{di}). \end{aligned}$$

3.2 Lines

We can draw basic straight lines to the screen by using two one-dimensional interpolations. For (x_1, y_1) and (x_2, y_2) the points we want to draw between, we interpolate from x_1 to x_2 and y_1 to y_2 in s steps where:

$$s = \max(\{2, |x_2 - x_1|, |y_2 - y_1|\}),$$

and fill in the corresponding pixel for each i in $[s]$.

3.3 Triangles

Triangles are important building blocks of computer graphics as we can use them to build any polygon. Stroking triangles is simple, but filling them is a more complex task, which is where we rasterise the triangle.

We want to fill the triangle in 'rakes', top to bottom, left to right. We first take the vertices and order them ascending based on their y coordinate on the screen to get v_1, v_2, and v_3 with $v_i = (x_i, y_i)$ for each i in [3]. We calculate the point v' along (v_1, v_3) with y coordinate equal to that of v_2 by setting:

$$v' = v_1 + \left(\frac{y_2 - y_1}{y_3 - y_1}\right)(v_3 - v_1).$$

We can then calculate $I(x_1, x', y_2 - y_1)$ and $I(x_1, x_2, y_2 - y_1)$, and draw the horizontal lines between them at each i in $[y_2 - y_1]$. We similarly calculate $I(x', x_3, y_3 - y_2)$ and $I(x_2, x_3, y_3 - y_2)$ to fill the bottom section of the triangle.

3.3.1 Texture Mapping

Instead of filling triangles with solid colour, we can also fill them with texture. By having corresponding texture-space coordinates associated with each point on a triangle, we can perform the same interpolation as in the solid colour case simultaneously to find the texture-space points along the edge of the triangle, then instead of drawing each rake normally, we interpolate between the texture points on either side to sample colour from our texture.

It is worth noting that this approach ignores perspective so may give odd results when viewing from certain angles with certain textures. The solution to this is out of the scope of this unit.

3.4 Projecting

We can project the world-space coordinates of an object to screen coordinates by considering the screen plane to be some distance f in front of the camera so that where (u, v) is our desired screen-space coordinate and (x, y, z) is our world-space coordinate, we have similar right-angled triangles formed by:

- $(0, 0, f)$, $(u, v, 0)$, and $(0, 0, 0)$,
- $(0, 0, f)$, (x, y, z), and $(0, 0, z)$,

not accounting for camera position. Using this, we can calculate the desired screen-space coordinate:

$$(u, v, f) = f \cdot \frac{(x, y, f + |z|)}{f + |z|}.$$

We can then convert (u, v) to pixel-space coordinates by inverting the y-axis and translating by half the dimensions of our screen.

3.4.1 Depth

We can calculate the depth of a world-space point (x, y, z) by taking the reciprocal of $|z|$. Thus, a smaller depth value means the point is farther away from the camera, with zero representing a point infinitely far from the camera. This can be used to allow proper layering when rasterising. We note that the positive z direction points into the camera and negative z points to where the camera is facing, which is why we take the reciprocal of the absolute value.

3.5 Culling

Considering how we project our points, things may become strange if we consider points behind the camera. Thus, it makes sense to 'cull' points that have negative depth, to not consider them.

4 Raytracing

Fundamental to raytracing is detecting intersections between rays and the triangles which form our objects. We represent our ray as a start point $s = (x_s, y_s, z_s)$ and a direction vector $d = (x_d, y_d, z_d)$, with every point along the ray representable by $s + kd$ for some k. We take the target triangle to be (p_1, p_2, p_3) with e_1 the edge (p_1, p_2) and e_2 the edge (p_1, p_3), we want some u and v such that:

$$\begin{aligned} s + kd = p_1 + ue_1 + ve_2 &\iff s - p_1 = u(p_2 - p_1) + v(p_3 - p_1) - kd \\ &\iff s - p_1 = \begin{pmatrix} -x_d & \uparrow & \uparrow \\ -y_d & e_1 & e_2 \\ -z_d & \downarrow & \downarrow \end{pmatrix} \cdot \begin{pmatrix} t \\ u \\ v \end{pmatrix} \\ &\iff \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} -x_d & \uparrow & \uparrow \\ -y_d & e_1 & e_2 \\ -z_d & \downarrow & \downarrow \end{pmatrix}^{-1} \cdot (s - p_1). \end{aligned}$$

So, the point of intersection in the plane (p_1, p_2, p_3) lies in is $p_1 + ue_1 + ve_2$ whose distance from s is t. We add the following constraints to u and v to keep them within the triangle:

$$\begin{aligned} 0 \leq u \leq 1, \\ 0 \leq v \leq 1, \\ u + v \leq 1. \end{aligned}$$

We can then cast rays from our camera, through each pixel of the screen-space, to find the closest intersection with an object and use that object's information to correctly colour the pixel.

5 Lighting

5.1 Proximity Lighting

This is simply reducing the intensity of light proportionally to the square of the distance of the surface from the light source. This gives realistic brightness, as this is in accordance with the laws of physics. However, different scales may be used to achieve certain stylistic goals, which may or may not agree with the laws, but the effect is similar.

5.2 Angle of Incidence Lighting

This is calculating the angle between the surface normal and the vector to the light source and varying the light intensity accordingly. If the ray to light is parallel with the normal, light intensity should be maximal, but if it is perpendicular with the normal, or through the surface, the light intensity should be minimal.

The normal can be calculated via a cross-product of the triangle edges (but take care of the order you calculate the product), and the cosine of the angle can be calculated with the well-known dot product formula:

$$\cos(\theta) = \frac{a \cdot b}{|a| \cdot |b|}.$$

5.3 Specular Lighting

We calculate the angle of reflection of light off of a surface and, similarly to angle of incidence lighting, vary the light intensity based on the angle between the reflected ray and the vector to the camera. If the light is reflecting directly into the camera, the intensity should be maximal, and minimal if the reflected ray is pointing away from the camera. We can raise the cosine of the angle between these vectors to a power called the 'specular exponent'. The greater the value, the shinier the surface appears.

The reflected ray r can be calculated with the incident ray i and the normal n via:

$$r = i - 2(i \cdot n)n.$$

5.4 Ambient Lighting

Rays of light in real life reflect multiple times, yet we are only simulating up to one reflection at the moment. This causes our scene to appear dimmer in areas which light may take multiple reflections to reach. Since this would be very computationally expensive to calculate, we emulate the effect with ambient lighting. We either set a minimum threshold for light intensity or add a flat amount of brightness when we calculate the brightness of a pixel.

5.5 Gouraud Shading

Previously, we would use the vertex normal of the whole face to calculate the lighting effects on the whole face. This can lead to blocky lighting as lighting effects do not smoothly transition across the face. Gouraud shading is the process of, instead, generating normals for the vertices of the face (by taking the mean of the normals of the faces the vertex is contained in) and using those to generate the lighting effects at the vertices. These lighting effects can then be interpolated to the point being rendered.

5.6 Phong Shading

Phong shading is similar to Gouraud shading, except we instead interpolate the vertex normals, so we can calculate the vertex normal at any point on a face. We then use this to calculate lighting effects directly, giving a more convincing result than Gouraud shading (at the expense of computational cost).