

Notes on Computer Graphics and OpenGL

Thomas Strathmann

April 21, 2010

Preface

The purpose of this document is to server as a sort of logbook in which I put down all the interesting math and assorted trivia I encounter while occasionally dabbling in computer graphics. Thus, the contents are not in any meaningful sense of the word exhaustive or suitable as an introductory text. Only rasterization techniques and related topics (such as interpolation or procedural geometry/textures) are discussed.

I hope that others will find this information not only useful, but will be compelled to extend the contents of this little "book". Because, in order to really understand any subject matter one not only has to absorb the knowledge of other people, but rather think it through and try to explain it to others. This could be the start of a beautiful project ...

Thomas Strathmann (December 2008)

Contents

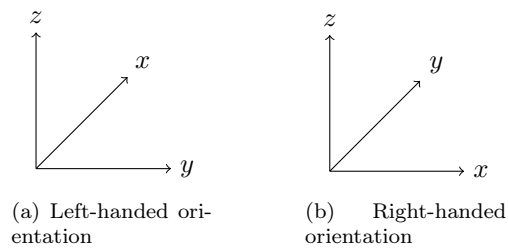
1	Models and Geometry	1
1.1	Coordinate Systems	1
1.1.1	OpenGL	1
1.1.2	Homogeneous Matrices	2
1.1.3	Orientation	3
1.2	Calculating Normals	3
1.2.1	Calculating Normals for Triangles	3
1.2.2	Calculating Normals for Subdivision Surfaces	4
1.2.3	Calculating Normals for Analytical Curves	4
2	Interpolation	7
2.1	Linear Interpolation	7
2.1.1	Bilinear Interpolation	7
2.2	Cosine Interpolation	8
2.3	Catmull-Rom Splines	8
2.4	Smooth Step	8
3	Illumination	9
3.1	Attenuation	9
3.1.1	Point Light	9
3.1.2	Spot Light	9
3.1.3	Directional Light	9
3.2	Phong Shading	9
A	Notation	11

Chapter 1

Models and Geometry

1.1 Coordinate Systems

Orthonormal basis of a vector space, usually \mathbb{R}^3 .



TODO: arc with arrow pointing clock-wise for left-hand, ccw for right-hand

1.1.1 OpenGL

In OpenGL the y -axis points upwards and the x -axis points towards the right. The most peculiar part of the convention is that the positive z -axis points towards the viewer, i.e. the z -component of any vector becomes negative as it moves farther away from the camera.

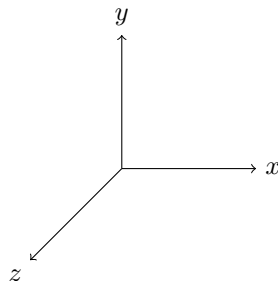


Figure 1.1: OpenGL texture coordinates

Texture coordinates are another matter as figure 1.1.1 depicts.

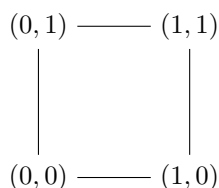


Figure 1.2: OpenGL 2D texture coordinates

Thus, to draw a textured Quad in the xy -plane the following OpenGL code may be used. Note, that the winding convention for front facing polygons mandates that the vertices be drawn in counter-clockwise order.

```
glBegin(GL_QUADS)
glNormal3i(0, 0, 1);
glTexCoord2i(0, 0); glVertex3i(-1, -1, 0);
glTexCoord2i(1, 0); glVertex3i(1, -1, 0);
glTexCoord2i(1, 1); glVertex3i(1, 1, 0);
glTexCoord2i(0, 1); glVertex3i(-1, 1, 0);
glEnd();
```

1.1.2 Homogeneous Matrices

Linear transformations in n dimensions can be described by a $n \times n$ matrix, but affine transformations can only be described by $n + 1 \times n + 1$ matrices. We use so-called homogeneous 4×4 matrices to describe affine and other kinds of transformations in 3D space. Such a Matrix can be understood as transforming one coordinate system A into another coordinate system B . This matrix is then denoted ${}^A T_B$ and consists of a linear part (the rotational component) and an affine part (the translation component).

$${}^A T_B = \begin{pmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The upper left 3×3 submatrix is the linear part of the transformations. Its constituent column vectors x , y , and z are the three basis vectors of the system B . Thus, they need to be linearly independent. The remaining column vector p is exactly the affine portion of the matrix. The last row is invariantly the same for physically motivated transformations.

Inverting a Homogeneous Matrix

The inverse of a given 4×4 homogenous matrix ${}^A T_B$ is given by

$$({}^A T_B)^{-1} = {}^B T_A = \begin{pmatrix} x_1 & x_2 & x_3 & -\langle \vec{p}, x \rangle \\ y_1 & y_2 & y_3 & -\langle \vec{p}, y \rangle \\ z_1 & z_2 & z_3 & -\langle \vec{p}, z \rangle \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.1.1)$$

1.1.3 Orientation

The orientation of an object is determined by the linear component of the transformation matrix from inertial space to the world space (`GL_MODELVIEWMATRIX`). A number of different approaches beside transformation matrices exist to describe the orientation of an object.

Rotation Matrices

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \quad (1.1.2)$$

$$R_y = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix} \quad (1.1.3)$$

$$R_z = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (1.1.4)$$

Euler Angles

TODO

- Definition, different conventions and names
- Converting to homogenous matrices and back
- Problem of gimbal lock (yields motivation for use of quaternions)

Quaternions

TODO

1.2 Calculating Normals

1.2.1 Calculating Normals for Triangles

Given a triangle consisting of three vertices v_1, v_2, v_3 oriented in counter-clockwise fashion the surface normal of the triangle mesh is

$$N = \frac{(v_2 - v_1) \times (v_3 - v_1)}{\|(v_2 - v_1) \times (v_3 - v_1)\|} \quad (1.2.1)$$

The normal at a vertex v in the triangulated mesh is then calculated by averaging the surface normals of the k triangles that share that vertex:

$$N_v = \frac{1}{\|\sum_{i=1}^k N_i\|} \sum_{i=1}^k N_i \quad (1.2.2)$$

Instead of using formula (1.2.1) for the computation of the surface normals, one could use the following formula instead

$$N = (v_2 - v_1) \times (v_3 - v_1) \quad (1.2.3)$$

which takes into account the area of the involved triangles.

1.2.2 Calculating Normals for Subdivision Surfaces

e.g. a grid of QUADS or a TRIANGLE_STRIP

```

for (int z=0; z<sizez -2; z++) {
    for (int x=0; x<sizeX -2; x++) {
        float tx = (float)x/sizeX;
        float tz = (float)z/sizez;
        float dx = 1.0/sizeX;
        float dz = 1.0/sizez;
        glTexCoord2f(tx, tz);
        glNormal3f(height[z][x+1] - height[z][x],
                  1.0,
                  height[z+1][x] - height[z][x]);
        glVertex3f(tx, height[z][x], tz);

        glTexCoord2f(tx+dx, tz);
        glNormal3f(height[z][x+2] - height[z][x+1],
                  1.0,
                  height[z+1][x+1] - height[z][x+1]);
        glVertex3f(tx+dx, height[z][x+1], tz);

        glTexCoord2f(tx+dx, tz+dz);
        glNormal3f(height[z+1][x+2] - height[z+1][x+1],
                  1.0,
                  height[z+2][x+1] - height[z+1][x+1]);
        glVertex3f(tx+dx, height[z+1][x+1], tz+dz);

        glTexCoord2f(tx, tz+dz);
        glNormal3f(height[z+1][x+1] - height[z+1][x],
                  1.0,
                  height[z+2][x] - height[z+1][x]);
        glVertex3f(tx, height[z+1][x], tz+dz);
    }
}

```

1.2.3 Calculating Normals for Analytical Curves

Given a differentiable parametric curve $p : \mathbb{R} \rightarrow \mathbb{R}^3$ with

$$p(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix},$$

calculate the Frenet frame as follows

$$v_{tangent} = \frac{\begin{pmatrix} \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{pmatrix}^T}{\left\| \begin{pmatrix} \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t} \end{pmatrix}^T \right\|} \quad (1.2.4)$$

$$v_{normal} = v_{tangent} \times \frac{\begin{pmatrix} \frac{\partial^2 x}{\partial^2 t} & \frac{\partial^2 y}{\partial^2 t} & \frac{\partial^2 z}{\partial^2 t} \end{pmatrix}^T}{\left\| \begin{pmatrix} \frac{\partial^2 x}{\partial^2 t} & \frac{\partial^2 y}{\partial^2 t} & \frac{\partial^2 z}{\partial^2 t} \end{pmatrix}^T \right\|} \quad (1.2.5)$$

$$v_{binormal} = v_{tangent} \times v_{normal} \quad (1.2.6)$$

These three vectors form an orthonormal basis which is aligned to p at time t . The problem of odd behaviour at inflection points of p (where $p'(t) = 0$ and $p''(t) = 0$) can be mitigated by calculating the normal vector as the cross product of the tangent with a vector which is guaranteed to never be parallel to the normal vector as follows:

$$v_{normal} = v_{tangent} \times \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T \quad (1.2.7)$$

In the numerical case the tangent vector $v_{tangent}$ can be approximated from two computed points p_1 and p_2 by using either the *forward*, *backward*, or *central difference* method using the equation

$$v_{tangent} = \frac{p_2 - p_1}{\|p_2 - p_1\|} \quad (1.2.8)$$

Let δ be the step size of the approximation and t the current time. Then p_1 and p_2 are given by

$$p_1 = p(t) \quad p_2 = p(t + \delta) \quad \text{for forward difference,} \quad (1.2.9)$$

$$p_1 = p(t - \delta) \quad p_2 = p(t) \quad \text{for backward difference,} \quad (1.2.10)$$

$$p_1 = p\left(t + \frac{\delta}{2}\right) \quad p_2 = p\left(t + \frac{\delta}{2}\right) \quad \text{for central difference.} \quad (1.2.11)$$

Chapter 2

Interpolation

2.1 Linear Interpolation

Given two points p_0 and p_1 and an index t linear interpolation between p_0 and p_1 is accomplished by the formula

$$(1 - t) \cdot p_0 + t \cdot p_1 \quad (2.1.1)$$

2.1.1 Bilinear Interpolation

Often one needs to calculate the value of a function f of two variables on a two dimensional grid. The most commonly used procedure to achieve this is called bilinear interpolation. It is used to interpolate 2D textures, for instance. Let the values of f at four points which form a rectangular region in which the point $p = (x, y)$ lies be known as $q_{00} = (x_0, y_0)$, $q_{01} = (x_0, y_1)$, $q_{10} = (x_1, y_0)$, $q_{11} = (x_1, y_1)$. First, we interpolate in the x direction:

$$f((x, y_0)) \approx \frac{x_1 - x}{x_1 - x_0} f(q_{00}) + \frac{x - x_0}{x_1 - x_0} f(q_{10}) \quad (2.1.2)$$

$$f((x, y_1)) \approx \frac{x_1 - x}{x_1 - x_0} f(q_{01}) + \frac{x - x_0}{x_1 - x_0} f(q_{11}) \quad (2.1.3)$$

Then, we interpolate in the y direction by using the previously computed values:

$$f(p) = f(x, y) \approx \frac{y_1 - y}{y_1 - y_0} f((x, y_0)) + \frac{y - y_0}{y_1 - y_0} f((x, y_1)) \quad (2.1.4)$$

Combining the two steps into one yields the following complete formula for the bilinear estimate of $f(x, y)$:

$$\begin{aligned} f(x, y) \approx & \frac{f(q_{00})}{(x_1 - x_0)(y_1 - y_0)} (x_1 - x)(y_1 - y) + \\ & \frac{f(q_{10})}{(x_1 - x_0)(y_1 - y_0)} (x - x_0)(y_1 - y) + \\ & \frac{f(q_{01})}{(x_1 - x_0)(y_1 - y_0)} (x_1 - x)(y - y_0) + \\ & \frac{f(q_{11})}{(x_1 - x_0)(y_1 - y_0)} (x - x_0)(y - y_0) \end{aligned} \quad (2.1.5)$$

2.2 Cosine Interpolation

A variation on linear interpolation is cosine interpolation where the index t in formula 2.1.1 is substituted by t' .

$$t' = \frac{1}{2}(1 - \cos(\pi t)) \quad (2.2.1)$$

2.3 Catmull-Rom Splines

Catmull-Rom splines are a special case of cubic splines, thus requiring four points for interpolation. Given four points p_0, \dots, p_3 where p_1 and p_2 are the points we want to interpolate between and p_0, p_3 are additional points, Catmull-Rom interpolation takes an index t and is computed as follows:

$$\begin{aligned} \frac{1}{2}(2p_1 + (-p_0 + p_2)t + (2p_0 - 5p_1 + 4p_2 - p_3)t^2 \\ + (-p_0 + 3p_1 - 3p_2 + p_3)t^3) \end{aligned} \quad (2.3.1)$$

Originally conceived to interpolate camera motion from a set of key points Catmull-Rom splines yield extremely pleasing results provided that the first and second derivative of the curve defined by the keypoints are bounded.

2.4 Smooth Step

The smooth step function is useful in blending two values a and b given an index variable x :

$$\text{smoothstep}(a, b, x) := \begin{cases} 0 & \text{for } x < a \\ 1 & \text{for } x \geq b \\ \left(\frac{x-a}{b-a}\right)^2 \cdot \left(3 - 2\frac{x-a}{b-a}\right) & \text{otherwise} \end{cases} \quad (2.4.1)$$

Chapter 3

Illumination

3.1 Attenuation

3.1.1 Point Light

The intensity I_P of a point light source P at distance d from its position is given by

$$I_P = \frac{1}{k_c + k_l d + k_q d^2} C_P \quad (3.1.1)$$

where C_P is the color of the light source and k_c , k_l , and k_q are the constant, linear, and quadratic attenuation factors respectively.

3.1.2 Spot Light

The intensity I_S of a spot light source S at distance d from its position is given by

$$I_S = \frac{\max(\langle -R, L \rangle, 0)^p}{k_c + k_l d + k_q d^2} C_S \quad (3.1.2)$$

where C_S is the color of the light, k_c , k_l , and k_q are as before, R is the direction in which the spot is pointing, and L is the unit vector between the point under consideration and the light source (cf. section 3.2).

3.1.3 Directional Light

Assumed to be infinitely far away so as to have light rays which are perfectly parallel, the intensity of the light emitted by a directional light source is constant.

3.2 Phong Shading

The Phong lighting equation is due to Bui Tuong Phong, who published in 1973 as part of his Ph.D. thesis. The color of every pixel (fragment) I_f is given by

the equation

$$I_f = I_a + I_d + I_s \quad (3.2.1)$$

where I_a is a four-dimensional RGBA vector which denotes the ambient term, I_d denotes the diffuse term, and I_s the specular term.

The ambient term is defined as

$$I_a = (A_l \cdot A_m) + (A_s \cdot A_m) \quad (3.2.2)$$

where A_l is the ambient color of the light source, A_m the ambient color of the material, and A_s the scene color (`gl_FrontLightModelProduct.sceneColor` in *GLSL*).

The diffuse component of the lighting equation is given by the Lambertian reflection term weighted by the diffuse color of light source D_l and material D_m

$$I_d = D_l \cdot D_m \cdot \max(\langle N, L \rangle, 0) \quad (3.2.3)$$

where N denotes the normal vector of the pixel. The vector L is the normalized vector between the position of the light source and the pixel under consideration. Given the vertex position v in eye coordinates (i.e. the vertex multiplied by the `ModelView` matrix) and the light source position l it is defined as follows:

$$L = \frac{l - v}{\|l - v\|} \quad (3.2.4)$$

The specular term I_s is calculated by the equation

$$I_s = \begin{cases} S_m \cdot S_l \cdot \max(\langle R, V \rangle, 0)^f & \text{if } \langle N, L \rangle > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3.2.5)$$

where S_m and S_l again denote the specular color of the material and light source respectively. R is the reflected light vector given by

$$R = 2 \cdot \langle N, L \rangle \cdot N - L \quad (3.2.6)$$

which is equal to `reflect(L, N)` in *GLSL*. V is the view vector given by

$$V = -\frac{-v}{\|-v\|} \quad (3.2.7)$$

where v again is the current vertex in eye coordinates. Finally, f denotes the specular factor (shininess). Alternatively the specular component can be expressed in a more computationally efficient manner as follows:

$$I_s = \begin{cases} S_m \cdot S_l \cdot \max(\langle N, H \rangle, 0)^f & \text{if } \langle N, L \rangle > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3.2.8)$$

where the halfway vector H between the direction to light vector L and the viewer vector V is given by

$$H = \frac{L + V}{\|L + V\|} \quad (3.2.9)$$

Appendix A

Notation

- $\langle v_1, v_2 \rangle$ – Dot (inner) product of vectors v_1 and v_2
- $v_1 \times v_2$ – Cross product of v_1 and v_2
- $\|v\|$ – magnitude (length) of vector v
- $\langle x \rangle$ – expectation value (average) of x
- $\nabla \vec{x}$ – component-wise derivatives of \vec{x} , i.e. for an n -dimensional vector \vec{x}
$$\nabla \vec{x} = \left(\frac{\partial \vec{x}}{\partial x_1} \dots \frac{\partial \vec{x}}{\partial x_n} \right)$$

Bibliography

- [Bou99] Paul Bourke. Interpolation methods. <http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/>, December 1999.
- [Len04] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, Inc., 2. edition, 2004.
- [SWND06] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Charles River Media, Inc., 5. edition, 2006.