1 Going backwards

Before we can start thinking about how to calculate the point of view of an image, we must know how to project a given point in space to a given plane and how to draw a picture out of that projection. The following text gives a brief overview about that subject and tries to impart the necessary knowledge.

Preliminaries

- If P is a point, $\vec{p}$ is the corresponding position vector.
- $\vec{a} \circ \vec{b}$ denotes the scalar product of $\vec{a}$ and $\vec{b}$.

Let us suppose we know the point of view P and a point Q on the image plane E on which we want to project the object. The vector connecting P and Q is in the same time the normal vector $\vec{n}$ of E. The point A is the one we want to project onto the plane; its projection is called A'. A sketch of this you can see in figure 1.

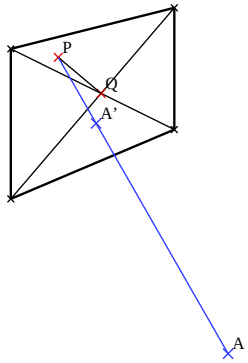

Figure 1: The scene and its most important objects

We want to find out the coordinates of A', so at first we need the normal form of E:

$$E : (\vec{x} - \vec{q}) \circ \vec{n} = 0 \tag{1}$$

We also need the equation describing the straight line g from A to P:

$$g : \vec{x} = \vec{a} + r \cdot (\vec{p} - \vec{a}) \tag{2}$$

Now we have g intersect E to determine A', so we put together both equations:

$$\begin{aligned} 0 &= (\vec{a} + r \cdot (\vec{p} - \vec{a}) - \vec{q}) \circ \vec{n} \\ 0 &= r \cdot (\vec{p} - \vec{a}) \circ \vec{n} + (\vec{a} - \vec{q}) \circ \vec{n} \\ (\vec{q} - \vec{a}) \circ \vec{n} &= r \cdot (\vec{p} - \vec{a}) \circ \vec{n} \\ r &= \frac{(\vec{q} - \vec{a}) \circ \vec{n}}{(\vec{p} - \vec{a}) \circ \vec{n}} \end{aligned}$$

The r thus obtained can be inserted in (2) to get the following formula:

$$g : \vec{a}' = \vec{a} + \frac{(\vec{q} - \vec{a}) \circ \vec{n}}{(\vec{p} - \vec{a}) \circ \vec{n}} \cdot (\vec{p} - \vec{a}) \tag{3}$$

If we would try to put this in a first little Python script, this could look as follows:

```
#!/usr/bin/python

from Numeric import array, dot

def proj(p, q, a):
     n = q-p
     a_ = a + float(dot((q-a),n))/dot((p-a),n)*(p-a)
     return a_
```

Let us test this method by some sample data:

```
>>> p = array([-10, -10, 20])
>>> q = array([-8, -8, 16])
>>> a = array([2, 5, 3])
>>> print proj(p, q, a)
[ -7.63934426  -7.04918033  16.6557377 ]
```

So we can already easily determine the projection of A onto E. This is a good thing, but unfortunately, this is not exactly what we want. Later on, we must transform the position in E to the position in our image I. So we actually don't need the coordinates of A' but its position in relation to a certain point (let us take Q) in E. Since the coordinates obtained by the procedure above only have finite precision, it might occur that when we try to transform $\vec{a'}$ into a form such as $\vec{q} + r \cdot \vec{d_1} + s \cdot \vec{d_2}$ (where $\vec{d_1}$ and $\vec{d_2}$ are the linearly independent vectors that span E) there is no solution because of the loss of precision.

So we have to calculate r and s so that

$$\vec{a} + t \cdot (\vec{p} - \vec{a}) = \vec{q} + r \cdot \vec{d_1} + s \cdot \vec{d_2}$$

which is equivalent to

$$0 = r \cdot \vec{d_1} + s \cdot \vec{d_2} + t \cdot (\vec{a} - \vec{p}) + \vec{q} - \vec{a} \tag{4}$$

Splitting this into the vectors' components, we get a linear equation system consisting of three equations:

$$\begin{aligned} 0 &= r \cdot d_{1x} + s \cdot d_{2x} + t \cdot (a_x - p_x) + (q_x - a_x) \\ 0 &= r \cdot d_{1y} + s \cdot d_{2y} + t \cdot (a_y - p_y) + (q_y - a_y) \\ 0 &= r \cdot d_{1z} + s \cdot d_{2z} + t \cdot (a_z - p_z) + (q_z - a_z) \end{aligned}$$

Solving this system with `maxima` gives the following solution:

$$
\begin{aligned}
r = &-(d_{2x}(a_y(q_z - p_z) + p_y(a_z - q_z) + (p_z - a_z)q_y) + p_x(d_{2y}(q_z - a_z) - d_{2z}q_y + a_y d_{2z}) \\
&+ a_x(d_{2y}(p_z - q_z) + d_{2z}q_y - d_{2z}p_y) + (d_{2y}(a_z - p_z) + d_{2z}p_y - a_y d_{2z})q_x)/(d_{2x}(d_{1y}(p_z - a_z) \\
&- d_{1z}p_y + a_y d_{1z}) + d_{1x}(d_{2y}(a_z - p_z) + d_{2z}p_y - a_y d_{2z}) + (d_{1z}d_{2y} - d_{1y}d_{2z})p_x \\
&+ a_x(d_{1y}d_{2z} - d_{1z}d_{2y})) \\
s = &(d_{1x}(a_y(q_z - p_z) + p_y(a_z - q_z) + (p_z - a_z)q_y) + p_x(d_{1y}(q_z - a_z) - d_{1z}q_y + a_y d_{1z}) \\
&+ a_x(d_{1y}(p_z - q_z) + d_{1z}q_y - d_{1z}p_y) + (d_{1y}(a_z - p_z) + d_{1z}p_y - a_y d_{1z})q_x)/(d_{2x}(d_{1y}(p_z - a_z) \\
&- d_{1z}p_y + a_y d_{1z}) + d_{1x}(d_{2y}(a_z - p_z) + d_{2z}p_y - a_y d_{2z}) + (d_{1z}d_{2y} - d_{1y}d_{2z})p_x \\
&+ a_x(d_{1y}d_{2z} - d_{1z}d_{2y})) \\
t = &(d_{2x}(d_{1y}(q_z - a_z) - d_{1z}q_y + a_y d_{1z}) + d_{1x}(d_{2y}(a_z - q_z) + d_{2z}q_y - a_y d_{2z}) \\
&+ (d_{1z}d_{2y} - d_{1y}d_{2z})q_x + a_x(d_{1y}d_{2z} - d_{1z}d_{2y}))/(d_{2x}(d_{1y}(p_z - a_z) - d_{1z}p_y + a_y d_{1z}) \\
&+ d_{1x}(d_{2y}(a_z - p_z) + d_{2z}p_y - a_y d_{2z}) + (d_{1z}d_{2y} - d_{1y}d_{2z})p_x + a_x(d_{1y}d_{2z} - d_{1z}d_{2y}))
\end{aligned}
$$

OK, I see, this is an ugly solution... However, we need r and s as they are the factors which determine the length of the vectors from Q to A' as shown in figure 2. We build a second Python script that uses this method to compute the coordinates of A' once more.

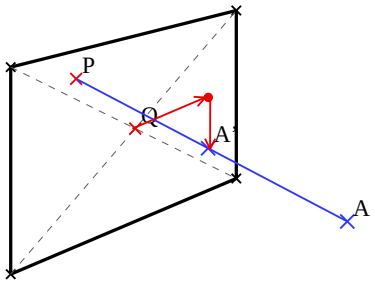

Figure 2: Projection using the parameter form

```
#!/usr/bin/python

from Numeric import array

def proj2(p,q,d_1,d_2,a):
     p_x, p_y, p_z = p
     q_x, q_y, q_z = q
     d_1_x, d_1_y, d_1_z = d_1
     d_2_x, d_2_y, d_2_z = d_2
     a_x, a_y, a_z = a
     r = - float(d_2_x*(a_y*(q_z-p_z)+p_y*(a_z-q_z)+(p_z-a_z)*q_y)\
         + p_x*(d_2_y*(q_z-a_z)-d_2_z*q_y+a_y*d_2_z) \
         + a_x*(d_2_y*(p_z-q_z)+d_2_z*q_y-d_2_z*p_y) \
         + (d_2_y*(a_z-p_z)+d_2_z*p_y-a_y*d_2_z)*q_x) \
         /(d_2_x*(d_1_y*(p_z-a_z)-d_1_z*p_y+a_y*d_1_z) \
         + d_1_x*(d_2_y*(a_z-p_z)+d_2_z*p_y-a_y*d_2_z) \
         + (d_1_z*d_2_y-d_1_y*d_2_z)*p_x+a_x*(d_1_y*d_2_z \
         - d_1_z*d_2_y))
     s = float(d_1_x*(a_y*(q_z-p_z)+p_y*(a_z-q_z)+(p_z-a_z)*q_y)\
```

```
        + p_x*(d_1_y*(q_z-a_z)-d_1_z*q_y+a_y*d_1_z) \
        + a_x*(d_1_y*(p_z-q_z)+d_1_z*q_y-d_1_z*p_y) \
        + (d_1_y*(a_z-p_z)+d_1_z*p_y-a_y*d_1_z)*q_x) \
        /(d_2_x*(d_1_y*(p_z-a_z)-d_1_z*p_y+a_y*d_1_z) \
        + d_1_x*(d_2_y*(a_z-p_z)+d_2_z*p_y-a_y*d_2_z) \
        + (d_1_z*d_2_y-d_1_y*d_2_z)*p_x+a_x*(d_1_y*d_2_z \
        - d_1_z*d_2_y))
    a_ = q + r*d_1 + s*d_2
    return a_
```

Now this seems quite an odd code and if this was the result of an equation I had to solve in a maths exam, I could be sure it was wrong. So let's test it with the values for $\vec{p}$, $\vec{q}$ and $\vec{a}$ from the last test. $\vec{d_1}$ and $\vec{d_2}$ are two arbitrary, non-collinear vectors so that $\vec{d_1} \circ (\vec{q} - \vec{p}) = 0$ and $\vec{d_2} \circ (\vec{q} - \vec{p}) = 0$

```
>>> p = array([-10, -10, 20])
>>> q = array([-8, -8, 16])
>>> a = array([2, 5, 3])
>>> d_1 = array([1, -1, 0])
>>> d_2 = array([1, 1, 1])
>>> print proj2(p, q, d_1, d_2, a)
[ -7.63934426  -7.04918033  16.6557377 ]
```

Hooray! Now this is a result we can really live with, I suppose. Namely, it's the same as some lines above, with the old method. But still, we don't really need the coordinates of A'. What we want in this part is to calculate which pixel in the image I corresponds to A'. Normally, pixels in an image are counted from the upper left corner. Here, it is more useful to count from the center of the image, just as shown in figure 3.

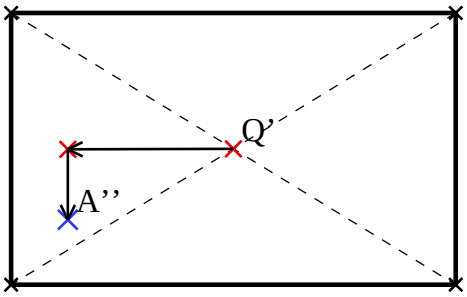

Figure 3: The image with pixel position relative to the center

Comparing figures 2 and 3, one can see that it is quite useful to choose $\vec{d_1}$ and $\vec{d_2}$ spanning E so that

- they have the same length, preferrably 1
- their scalar product is zero (so together with $(\vec{p} - \vec{q})$ they form an orthogonal basis)
- they have the same direction as the x- and y-axes of the image to be generated (i. e. they determine the rotation of I)

In the following we will assume the three statements above to be true for every $\vec{d_1}$ and $\vec{d_2}$.

Now let's say we have determined r and s for all points we are interested in, then we need to choose a transformation factor t that determines how many pixels of the image are used to represent one length unit in our vector space. Do you remember that we can "reach" a point on E by $\vec{q} + r \cdot \vec{d_1} + s \cdot \vec{d_2}$? In our image, it's almost the same, there it is $\vec{m} + r \cdot \vec{q_1} + s \cdot \vec{q_2}$ with M being the center point of the image and $\vec{q_1}$ and $\vec{q_2}$ the vectors parallel to its borders of length t, namely $\vec{q_1} = \begin{pmatrix} -t \\ 0 \end{pmatrix}$ and $\vec{q_2} = \begin{pmatrix} 0 \\ t \end{pmatrix}$. So if $t = 100$ then a point A' with $r = 2$ and $s = 1.5$ will be 200 pixels left and 150 pixels below the center of the image.

At last, I will show an example based on all the knowledge we gained in this part of the text. I first created a list of points representing a simple cube (from (0|0|0) to (10|10|10), with steps of 0.5). I put the point of view to (35|15|25) and chose Q to be (32|14|23) so $\vec{n} = (-3, -1, -2)$. I wanted the image not to be rotated so I had the x- and y-component of $\vec{d_1}$ be the negative of the x- and y-component of $\vec{n}$ and chose z appropriately. $\vec{d_2}$ is obtained by the vector product of $(\vec{q} - \vec{p})$ and $\vec{d_1}$. After shortening both vectors to length 1, this gives $\vec{d_1} \approx (0.507, 0.169, -0.845)$ and $\vec{d_2} \approx (0.316, -0.949, 0)$. I calculated r and s (determining the position relative to Q in E, remember?) for each of the $(10 \cdot 2)^3 = 10000$ points and then transformed this to the position relative to M in the image I. The result of this can be seen in figure 4.

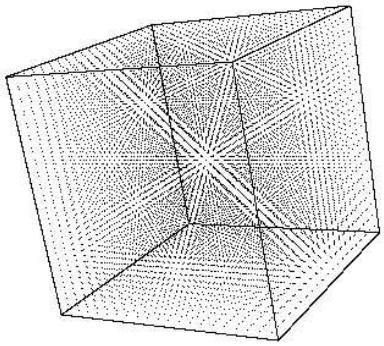

Figure 4: Image of the Cube

Tobias Pfeiffer, g-henna@users.sourceforge.net