In short words: 3d

- Chapter 1: basic 3d calculations -

1 The Vanishing Point

The first thing to understand is the so-called vanishing point. This is a point on a 3-dimensional image which indicates the depth of the things on the picture. Here is a brief example:

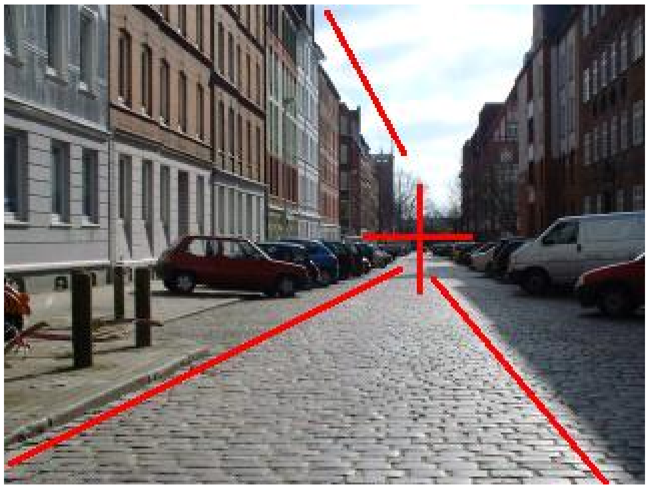

Illustration 1: The vanishing point

We see, that – with increasing z-coordinate – the edges of the things in the picture converge against the vanishing point. Based on this idea, we can provide two formulas, which indeed model a realistic 3d view.

2 3D → 2D projection

2.1 First formulas

More formalized, we have to provide a function which gets something like a 3d-point with x-, y- and z-coordinate as a parameter and returns a 2 dimensional projection of this 3d-point on the screen with x- and y-coordinate. We will now assume, that the vanishing point is located at the coordinates $(0,0)$ - that means, it is located at the top left os the screen. Based on the idea that everything moves closer to the vanishing point, we could try a function which implements the following formulas:

$$x_{2d} = \frac{x_{3d}}{z_{3d}} \qquad y_{2d} = \frac{y_{3d}}{z_{3d}}$$

Why do they work? The vanishing point is at $(0,0)$. If we let the x- and y-coordinate of some point untouched and increase the z-coordinate of this point, x_{2d} and y_{2d} will get smaller, because they are made up from a division by the z-coordinate.

For the rest of this document, let us assume, that we have some kind of array named `pts` which holds the x-, y- and z-coordinates of a traditional cube. The vertexes and edges of this 100x100x100 cube look like the following:

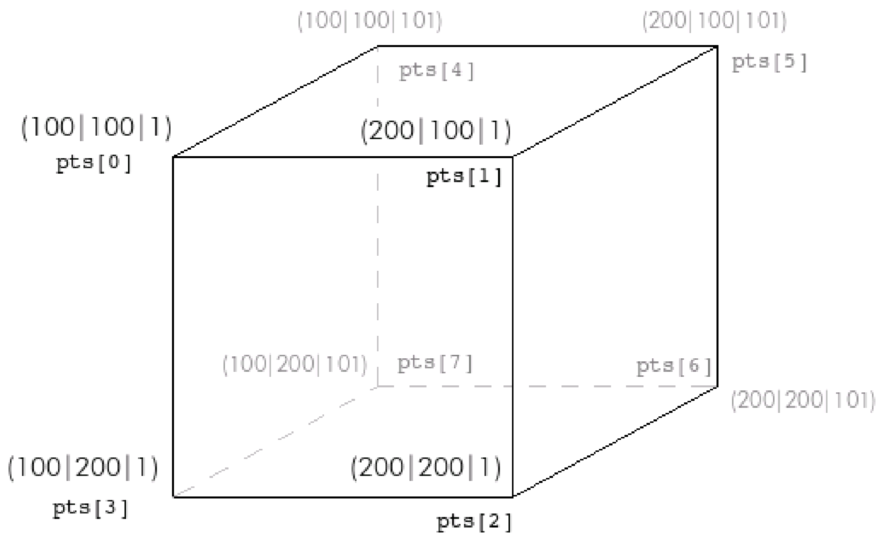

Illustration 2: Coordinates of the 3 Dimensional cube

Let us try to project this cube using the formula provided above. The plan is to set up the array of 3d-coordinates, then project them and connect them according to Illustration 2 (eg. draw a line from projected `pts[0]` to projected `pts[1]`, from projected `pts[1]` to projected `pts[2]`, ...) If you are familiar with C and LibSDL, then you can download sdl3d_chapter1_1_basic_3d.zip and watch the result for yourself. If you are not, however, it does not matter, because you can implement it in any language you want and i will present the expected results, so here is the first one:

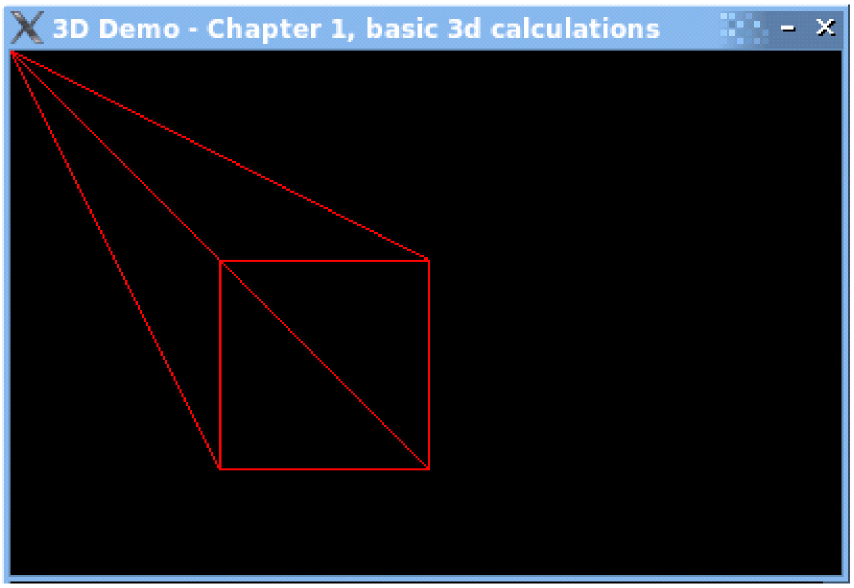

Illustration 3: 3 Dimensional cube (perspectively not correct yet)

As you can see: our little world is not fully correct yet. The dimensions seem to be wrong, because the thing you actually see should look like a cube. It looks like a cube with an infinite depth. This comes from an early mistake concerning the formulas: one has to manipulate them in order to get a more realistic perspective.

Download the first version which supports only basic projection here: sdl3d_chapter1_1_basic_3d.zip.

2.2 The Z-Factor

In order to get this more realistic view, one has to introduce a "z-factor" to the formulas. The idea is to simply stretch the world a little:

$$x_{2d} = \frac{x_{3d} * zFactor}{z_{3d}} \qquad y_{2d} = \frac{y_{3d} * zFactor}{z_{3d}}$$

The next question is: how to choose this ominous z-Factor. Well, the answer depends on some more sophisticated parameters like the height and the width of the screen resolution. For the rest of the document, a zFactor of 1000 and a screen resolution of 640x480 is assumed. As a rule of thumb - which we strictly ignore here :) - , one can say that a zFactor equally to the width of the screen is adequate.

If you know some common 3D-games, you probably know, that they have a parameter called "field of view". This parameter corresponds directly to this zFactor, because everything you set, when you set this field of view, is the zFactor.

So your formulas should look like the following:

$$x_{2d} = \frac{x_{3d} * 1000}{z_{3d}} \qquad y_{2d} = \frac{y_{3d} * 1000}{z_{3d}}$$

Using this zFactor, the cube looks more realistic:

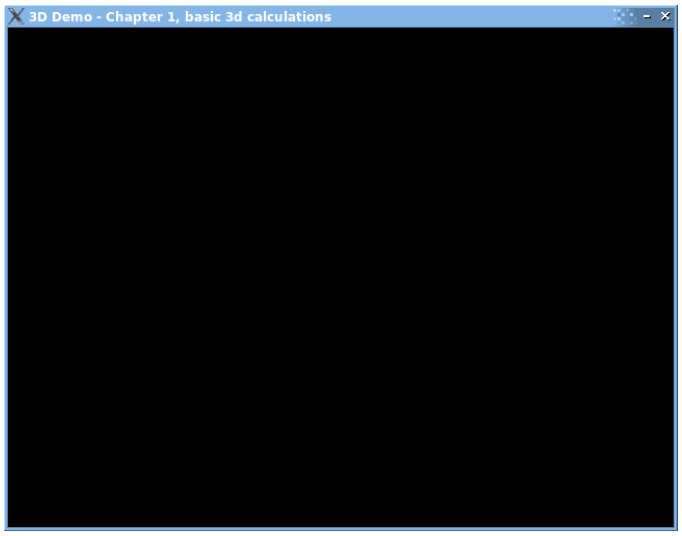

Illustration 4: The original cube, displayed using a zFactor of 1000

Oops, why can't we see anything? If you think of the new formulas, then you will discover, that the world has been translated (moved) a bit, due to this new multiplication. If we simply reset the z origin of the cube to something like 1000, we see the following:

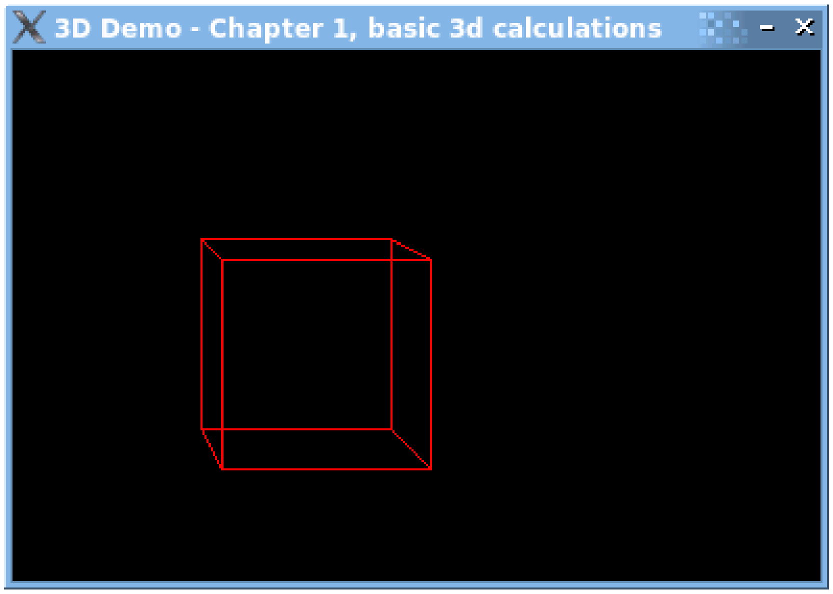

Illustration 5: The cube with a z origin of 1000

The thing, we see, looks definitely more like a cube than the first construction, shown in Illustration 3.

You can download the whole project as a zip-file here: sdl3d_chapter1_2_basic_3d_with_zFactor.zip. (supports basic projection using the zFactor)

3 Moving around

3.1 Translations

After we set up a cube, we want to move around in the 3d space. The first thing, we will implement is moving in a straight way to or away from the cube. Therefor, we have to translate (move) all the points in 3d and re-render them.

For a better point of view, we will move the vanishing point from the upper left of the screen into the middle. We do this by finalizing the projection formulas:

$$x_{2d} = \frac{x_{3d} * zFactor}{z_{3d}} + \frac{Screenwidth}{2} \qquad y_{2d} = \frac{y_{3d} * zFactor}{z_{3d}} + \frac{Screenheight}{2}$$

Now, we want to do a so-called translation of all the points. Therefor, we simply add or subtract a given value from each of the points' coordinates:

```
function translate( xTranslation [Type: floating point],
                    yTranslation [Type: floating point],
                    zTranslation [Type: floating point]) {

  for (i = 0 to maxPoints) {
    pts[i].xCoordinate =  pts[i].xCoordinate + xTranslation;
    pts[i].yCoordinate =  pts[i].yCoordinate + yTranslation;
    pts[i].zCoordinate =  pts[i].zCoordinate + zTranslation;
  }
}
```

Due to performance reasons, one does not calculate the position of the 3d-points every time, the main loop runs. The common tactic is to wait until the user wants to do something (like translating the scene, for example). Then, the main program is going to ask the 3d library for help in order to recalculate the position of each point. After the translation is done, the main program switches back to its normal mode, where no unnecessary calculation is being done.

If you are following the development of the C-code, then you have to know, that there are some other changes on the file "sdl3d.c", because LibSDL has to listen to the arrow keys from now on.

You can download newest version of the project as a zip-file here: sdl3d_chapter1_3_basic_3d_with_translation.zip. (supports basic projection and translation)

3.2 Rotations

3.2.1 Considerations

After we are able to move towards and away from the cube, we want to take a look from all possible positions in the 3d-space on it. Therefor, we need some basic rotations. The main idea is as follows:

When you turn left by some positive angle ß, then you have to rotate the scene by -ß.

This is not really intuitive, because the easiest way would be to rotate it by ß, not by -ß. The answer to this is: we are simply not able to simulate a rotation in an other way. If you want to be sure, that this works, go into the middle of some room and watch for the corners. Then turn right. What do you see? The corners turn to your left. That is what we are doing in the computer.

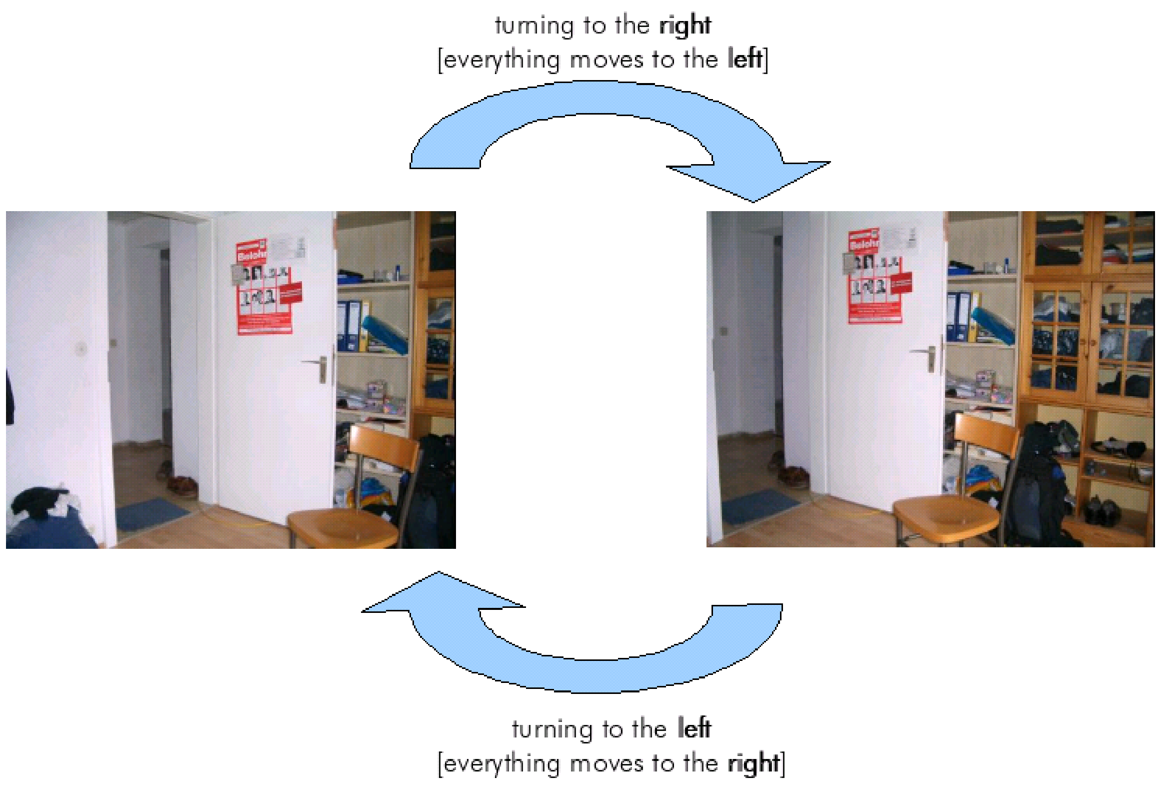
turning to the **right**
[everything moves to the **left**]
turning to the **left**
[everything moves to the **right**]

Illustration 6: Photos taken while turning in a regular room

3.2.2 Connection: sinus/cosine & rotation

In order to rotate the whole scene by some angle ß, we simply have to rotate every point by ß, but how to do this? There are some complicated sinus/cosine formulas which are necessary for rotating points. Why are they based on sinus/cosine? Imagine the following situation: Given is a x/y-point. The task is to rotate that point along an imaginary circle. The claim is, that one is able to model the position of the point using a sinus or cosine. On the left of the following pictures, you will see the point which has to be rotated. On the right, you will see a regular sinus oscillation. The green line marks the amount of degrees and hence the position on the sinus oscillation.

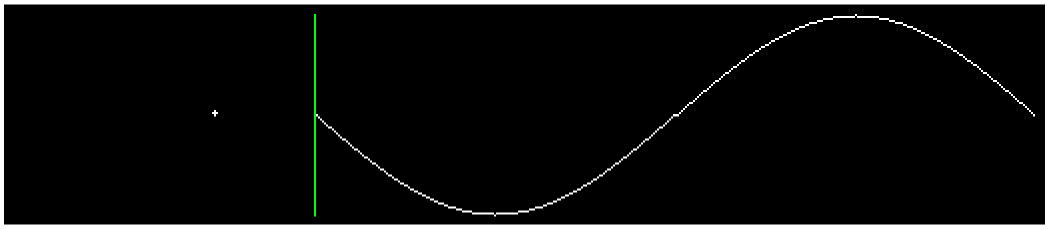

Illustration 7: The point with a rotation of 0 degrees

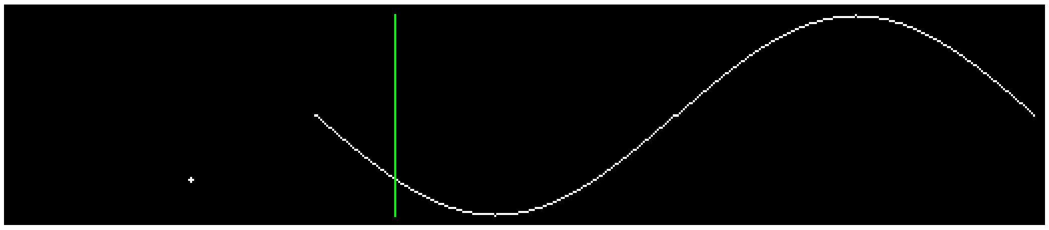
Illustration 8: ... 40 degrees

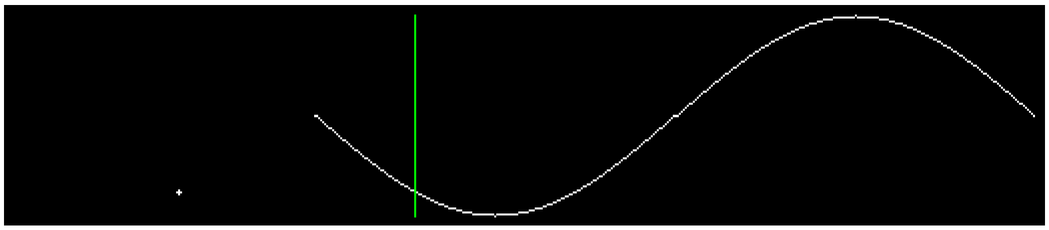
Illustration 10: ... 50 degrees

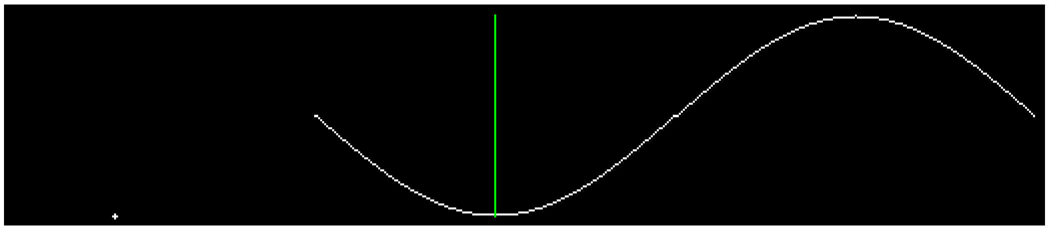
Illustration 11: ... 90 degrees, and so on

As you can see: the sinus function indeed gives us the position of the point. If you want to be sure that this works when it comes to degrees which are higher than 90, then please take a look at animation_of_a_point.gif, where the whole rotation is being shown.

This is the idea, why one models the rotation of a point using sinus and cosine, but this is only a very fictive example, things are more complicated in real life, and it is not easy to understand the connection between the actual formulas and the rotation as it.

3.2.3 Formulas

First, we have to see, that we want to rotate the points in their 3d-space, not in 2d. It would be easy to rotate a point in a two-dimensional space either, but we want them to be perspectively correct, so when the user presses a button like the left-arrow-button, then we want to rotate the scene (and therefor all the points) in a 3d-space. This leaves only one question to us: rotation is good, but around which axis? We have three choices here: the x-, y- and z-axis. Since we want to model a three dimensional world in which we are able to move, we are going to rotate around th y-axis. In a picture's language, this means, we are doing the following:

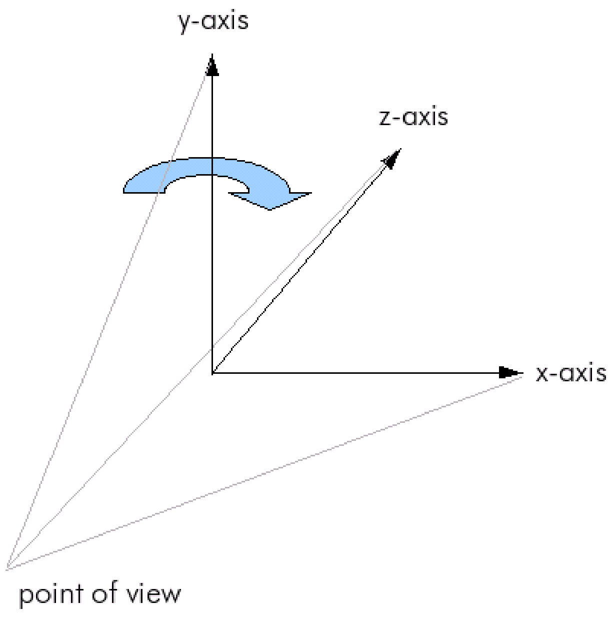

Illustration 12: Direction of rotation

If we programmed a 3d game where the user would be able to fly a spaceship or a plane, then we would rotate around the z-axis, because the plane turns that way. In this case, we want to model a kind of ego-shooter perspective, so we rotate around the y-axis.

Every ordinary math book (and the Internet too, of course) tells you, that you will have to do the following matrix multiplication if you want to rotate a 3d-point around the y-axis by some angle ß:

$$\begin{pmatrix} x_{rotated} \\ y_{rotated} \\ z_{rotated} \end{pmatrix} = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Multiplied completely, this gives us the following formulas:

$$x_{rotated} = x*\cos(\beta) + z*\sin(\beta)$$

$$y_{rotated} = y$$

$$z_{rotated} = x*(-\sin(\beta)) + z*\cos(\beta)$$

The following pseudo-code function executes this rotation:

```
function rotateSceneAlongYAxis( degrees [Type: integer]) {

  for (i = 0 to maxPoints) {
    declaration x3d [Type: double]
    declaration z3d [Type: double]
    x3d = pts[i].xCoordinate;
    z3d = pts[i].zCoordinate;

    // make sure, you are using sinus/cosine correctely, often the
    // natively calculate in rad, not in deghrees!
    pts[i].xCoordinate = (x3d * cos(degrees))  + (z3d * sin(degrees));
    pts[i].zCoordinate = (x3d * -sin(degrees)) + (z3d * cos(degrees));
  }
}
```

The result of all this is, that we can walk around the cube. Therefor, we are able to view it from a completely other perspective:

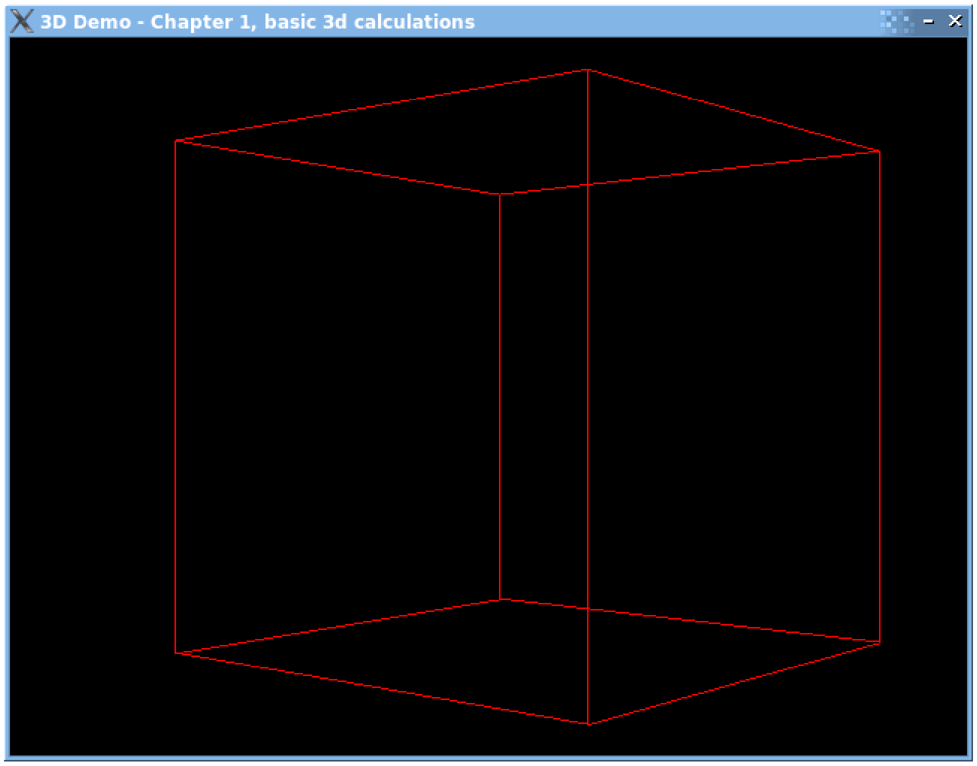

Illustration 13: Cube - seen from an other perspective achieved through rotation and translation

You can download latest version of the project as a zip-file here: sdl3d_chapter1_4_basic_3d_with_rotation.zip.
(supports basic projection, translation and rotation)

4 Some last corrections

Did you take your time exploring the 3d-world you have created? Ok, then you probably may have noticed some strange side-effects when walking around the cube. These side-effects mainly occur, when you are not able to see the cube (when it is "behind" you).

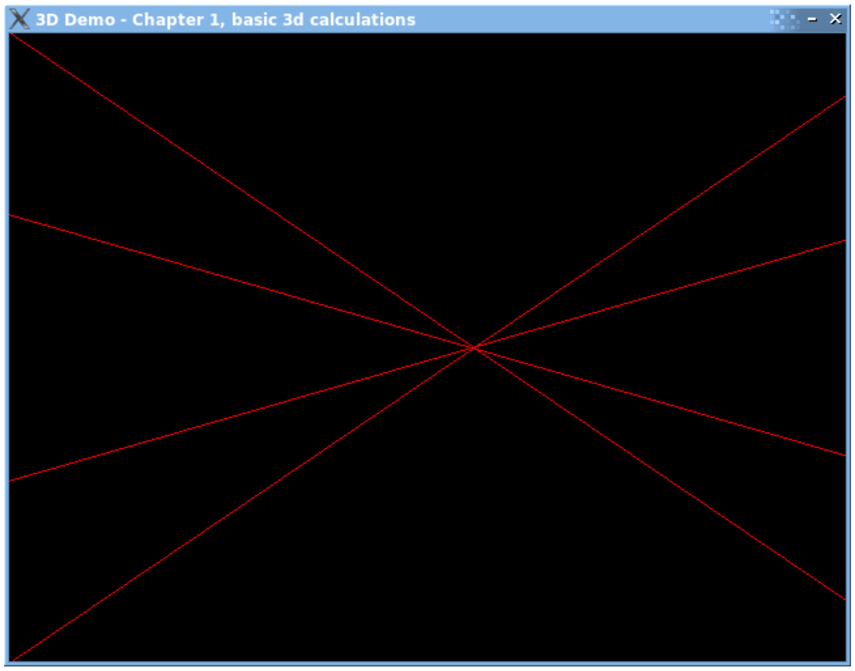

Illustration 14: Occurrence of strange side-effects

These side-effects are caused by the projection formulas. If you remember the formulas from chapter #2.1.First formulas| outline, then you may notice, that there is a division – but the case for denominator = 0 is missing. And what happens, if the z-coordinate of a point is smaller than zero? The solution to all this problems is fairly simple: do, as if the z-coordinate was 1. Why that?

What do we want to achieve? When the endpoints of a line are in the field of view, the line should be visible. This is clear, we did not modify this case. But what happens, when a line has at least one point which is on the outside of the visible space? Then we want to paint the line as long as we can. Setting z3d to 1 does exactly this.

The following illustration should clear things up. Let us consider a line consisting of two points, p1 and p2. p1 is located in the visible part of the 3d space (p1 has a z-coordinate which is higher than zero, black cross on the upper part of the illustration). p2 is located on the outside of the visible part (blue cross on the front), it therefor has a z-coordinate smaller than zero. If we want to draw a line from p1 to p2, we have to project p2 as if it had a z-coordinate of 1 (red cross in the middle).

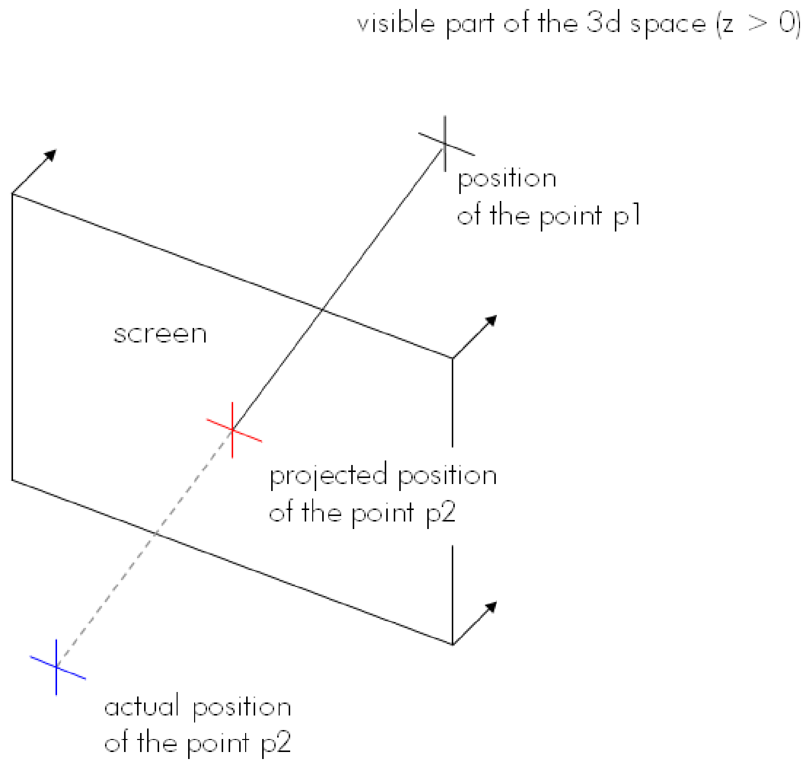
visible part of the 3d space (z > 0)
position of the point p1
screen
projected position of the point p2
actual position of the point p2

Illustration 15: Modified projection gives us more possibilities and avoids side-effects

The complete projection function therefor should look something like this:

```
void project_3d_2d(pt [Type: 3d-point which provides an x, y and
                       z-coordinate],

                   targetPoint [Type: 2d-point which provides x- and y-
                                   coordinate]) {


        // extract 3d values
        declaration x3d [Type: double]
        declaration y3d [Type: double]
        declaration z3d [Type: double]

        x3d = pt.x;
        y3d = pt.y;
        z3d = pt.z;
        // prepare the 2d values
        declaration x2d [Type: integer]
        declaration y2d [Type: integer]

        // the zFactor
        zFactor = 1000;

        if (z3d > 0) {
                // actually project the point with
                // the stretch factor
                x2d = ((int) (x3d * zFactor / z3d)) + (SCREEN_WIDTH / 2);
                y2d = ((int) (y3d * zFactor / z3d)) + (SCREEN_HEIGHT / 2);
        } else {
                // project the point as if it was at z=1
                // in that way, all lines can be drawn normally
                x2d = (int) ((x3d * zFactor) / 1) + (SCREEN_WIDTH / 2);
                y2d = (int) ((y3d * zFactor) / 1) + (SCREEN_HEIGHT / 2);
        }

        // fill in the calculated values in 2d
        targetPoint.x = x2d;
        targetPoint.y = y2d;
}
```

You can download latest version of the project as a zip-file here: sdl3d_chapter1_5_basic_3d_with_corrections.zip. (supports correct basic projection, translation and rotation)