# In short words: 3d

## - Chapter 2: Texture Mapping -

# 1 What? Why? Where?

## 1.1 What?

Texture Mapping can be expressed as the following equation:

*Illustration 1: Texture Mapping as equation*

Given is a picture and 4 3D-points which mark the "side" of the cube, the picture should be mapped on. The process of clipping this picture on these points is known as Texture Mapping. Normally, there can be more or less than 4 points, but we will simplify and start with exactly four 3D-points.

## 1.2 Different types of Texture Mapping

First, one has to say, that texture mapping costs a lot of time. The computer has to calculate very much in order to make the whole scene look realistic. Therefor, there are different types of Texture Mapping. In this document, we will analyze the main ones: "completely 2D-interpolated" and "completely correct". Interpolation means, that the computer calculates numbers which are not always 100% correct, but usually much faster to estimate. For example: if you search a number between 1 and 10, the computer may give you a 12.23. This is incorrect, but "correct" enough for us.

- Using interpolation, one can calculate **very fast**
- Though, the results are **not fully correct** sometimes

What does this mean: "the results are not fully correct"? It means, that our scene will look unrealistic. A quick example: We consider the following picture, which should be mapped:

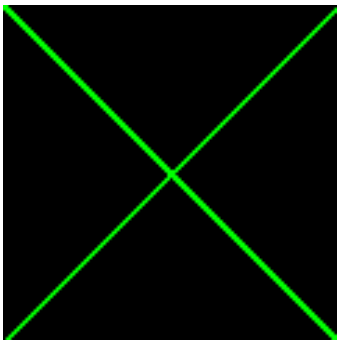


*Illustration 2: Picture which is going to be mapped in two different ways*

If we are using interpolation, the whole thing speeds up, but the scene looks unrealistic: on the left side, you can see the picture correctly mapped. On the right, an interpolation algorithm is applied.

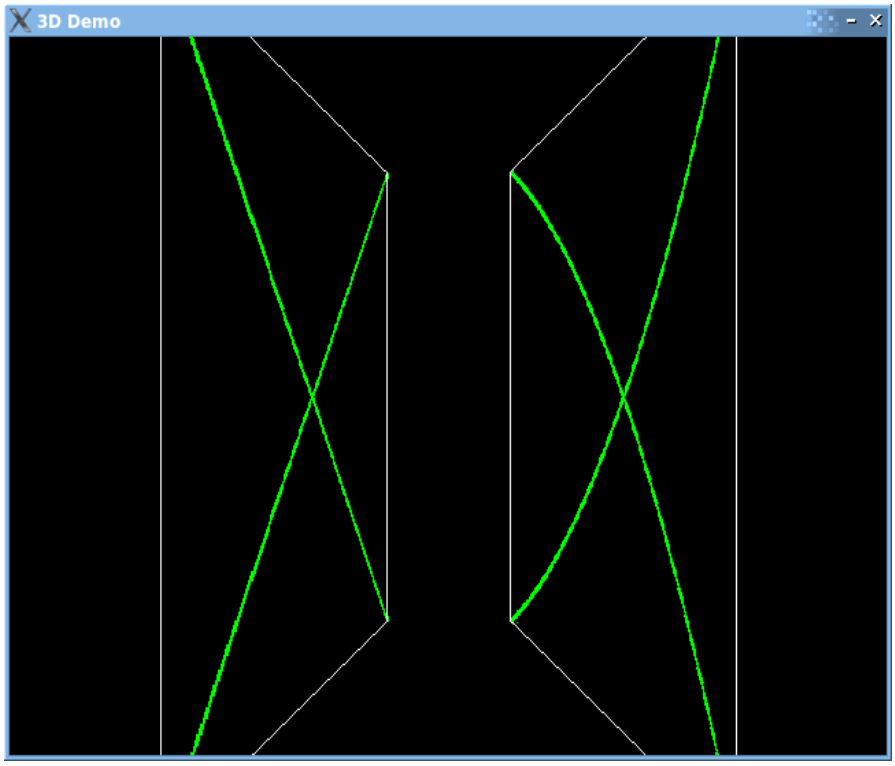



*Illustration 3: A simple cross - mapped using perfect mapping (on the left) and 2D-interpolation (on the right)*

One could remark, that this example is only very fictive, and indeed: in some cases, there is no visible difference between both of the algorithms...

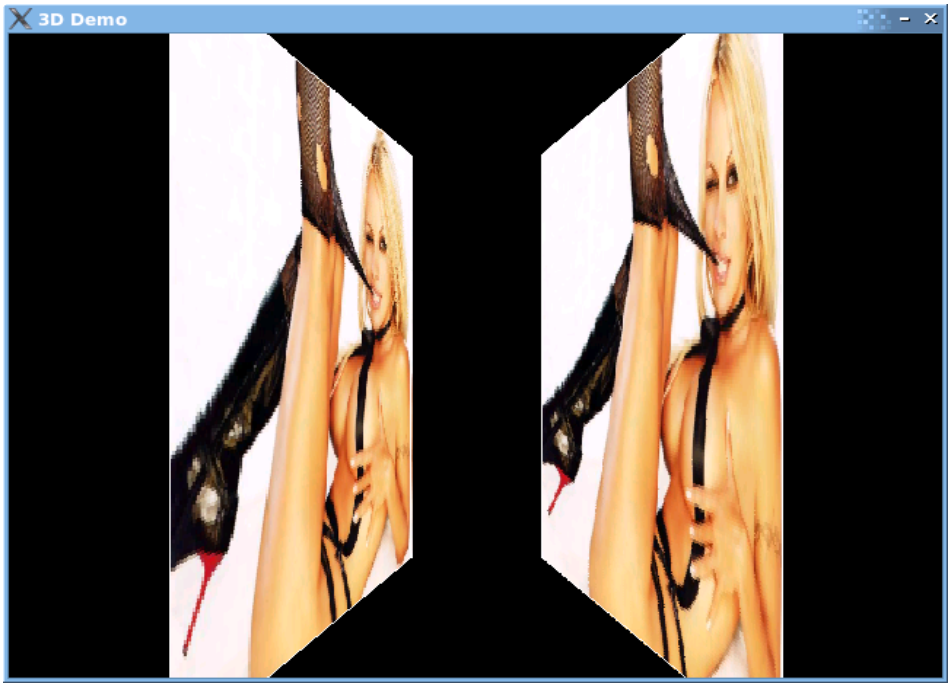



*Illustration 4: Special case, in which interpolation produces no visible difference*

- Developing Texture Mapping algorithms is about finding a Trade-off between speed and accuracy
- In **some** cases, interpolation-effects are invisible

So let's get our feet wet. We will start with the perfect 3D-based mapping algorithm.

# 2 Perfect 3D Mapping

## 2.1 Algorithm

For easing up the process we will start with a very simple mapping picture which has the size 10x10 pixels and looks like this:

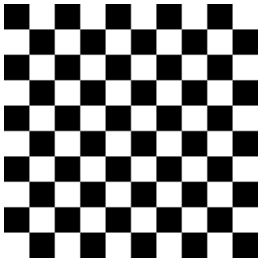


*Illustration 5: First mapping picture*

Please note, that the picture was enlarged for showing how it looks, in reality, it is a lot smaller.

We want to project this picture on the front side of our cube. This means, that four 3D-points are given; we will call them `leftUpper`, `leftLower`, `rightUpper` and `rightLower`. Our first goal will be to discover the following points on the cube:

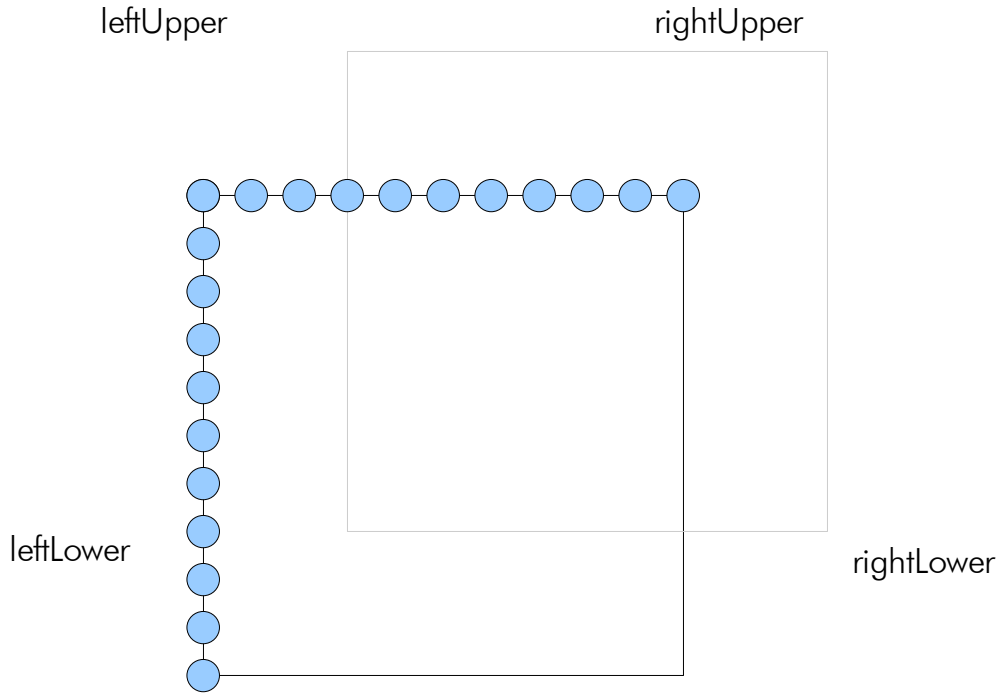



*Illustration 6: Cube with points to calculate*

Why the hell exactly these points? We do this in order to estimate the position (and more precise: the borders) of the "pixels" of the image (shown at illustration 5) on the cube. After we estimated the position and the borders of each "pixel",

we can start to project the pixels onto the cube. These projected (and often scaled pixels) will be called Texels. So how can we calculate the position of the dots? We somehow have to break up the way from `leftUpper` to `rightUpper` into smaller parts. These smaller parts will be called $\vec{v}$ from now on:

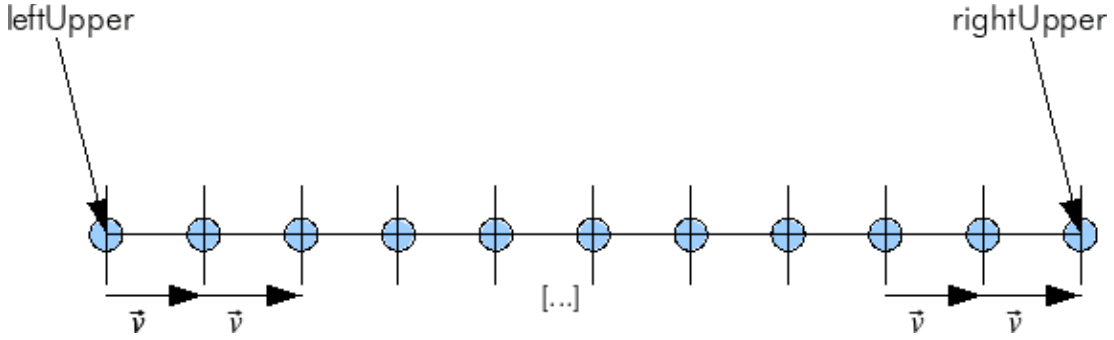



*Illustration 7: Splitting up the way from `leftUpper` to `rightUpper`*

So, how would we do it if we had only one dimension? If we want to break the number 50 into 5 parts, then we do it like the following:

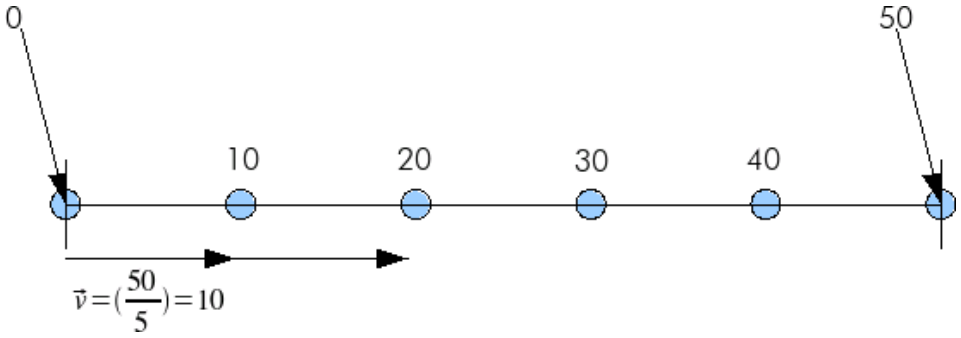



*Illustration 8: Splitting up a "way" from 0 to 50 into 5 parts*

As you can see, we simply divide 50 by 5 (because we want to break up the way in 5 parts) and as the result, we get 10, which is the length of one part. So if we have a number x and we want to break it up into y parts, then we divide x/y and get y parts of the length (x/y). But what happens, when we want to go from 10 to 60?

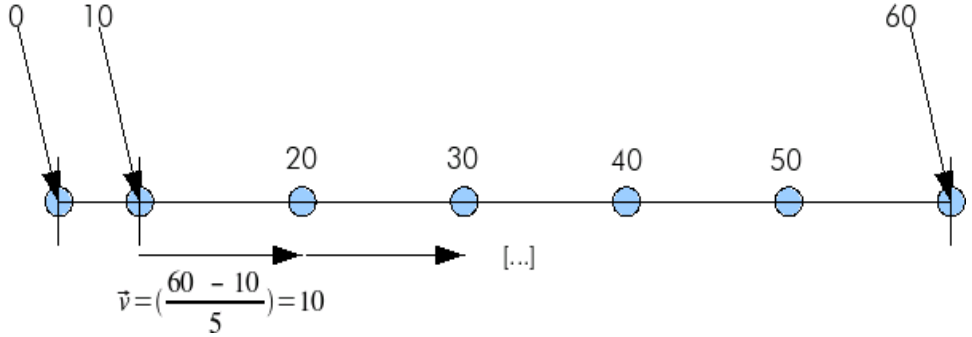



*Illustration 9: Splitting up the same way as above, just translated by 10*

Then we do not simply divide 60 by 5, because this would be wrong. So we do not want to have the absolute position of each point, but the distance between both of them. Let's assume we have some point "ptLeft" and some point "ptRight"

and we want to break up into "numParts" parts, then $\vec{v}$ is calculated as follows:

$$\vec{v} = \frac{ptRight - ptLeft}{numParts}$$

In 3 dimensions, it is exactly the same. One just has to notice, that "ptRight" and "ptLeft" now have three values which must be substracted. "numParts" will be th width of the image which has to be clipped onto the cube, because we want to break up the way from `leftUpper` to `rightUpper` into this amount of parts for placing the pixels on the surface. All in all, the formula for 3 dimensions looks like the following:

$$\vec{v} = \frac{1}{imageWidth} * \begin{pmatrix} rightUpper.x - leftUpper.x \\ rightUpper.y - leftUpper.y \\ rightUpper.z - leftUpper.z \end{pmatrix}$$

We will save the position of the points in an array called `upperPts`, `leftPts` for the points on the left side of the cube respectively.

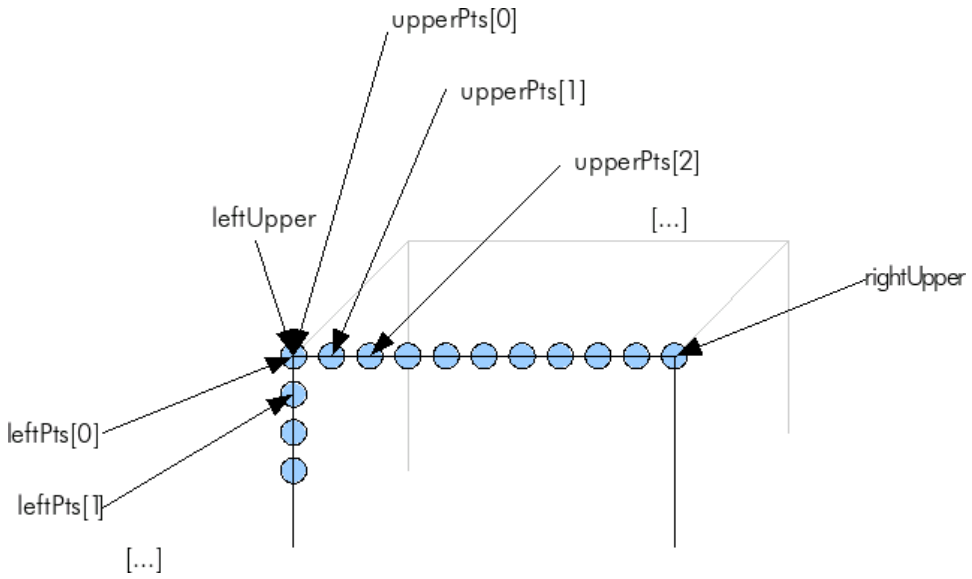



*Illustration 10: Saving the coordinates of the points (speeds up the process)*

Important: The picture above is adequate, if and only if `leftUpper` is at $(0,0,0)$ of the coordinate system. `upperPts` contains coordinates relative to the "beginning" of the cube, not of the actual position. This means, that `upperPts[0]` will always be initialized with zeros:

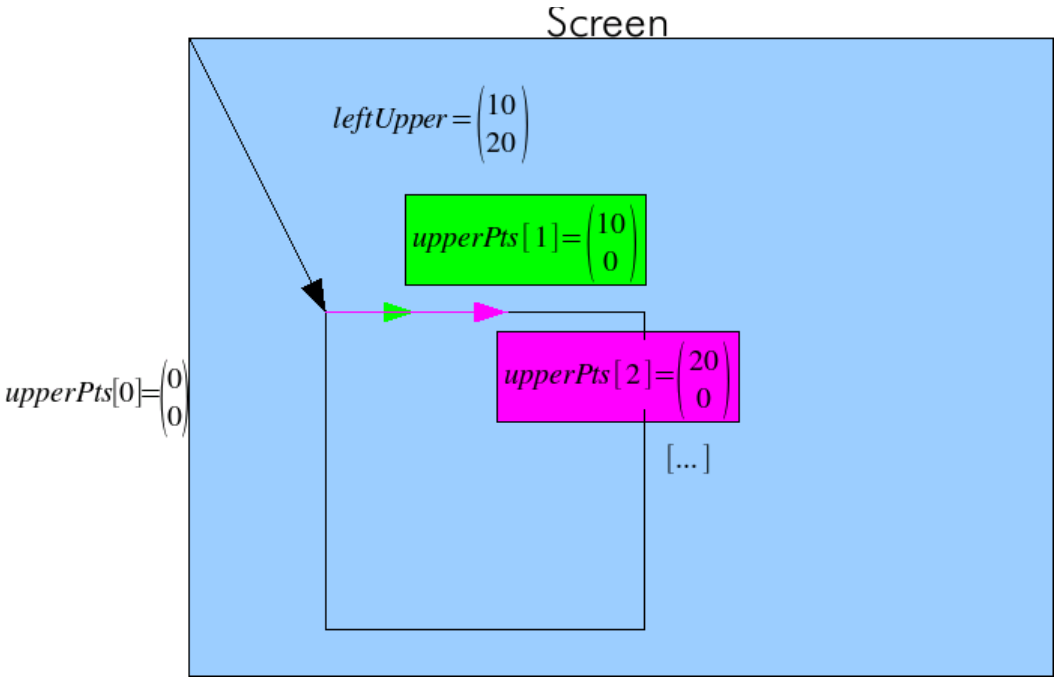



*Illustration 11: `upperPts` is located relative to `leftUpper` (fictive 2D example)*

So how should your code look like? It should implement the following formula:

$$upperPts[i] = (i * \vec{v})$$

For optimization, we will let the computer iterate and calculate this in a more efficient way, because why not using the old result again? So we will implement the following formula which does basically the same, but in a more efficient way:

$$upperPts[i] = upperPts[i - 1] + \vec{v}$$

This results into the following piece of code:

```
function textureMapping_3d(mapPicture [Type: Picture],
                           leftUpper, [Type: 3d-point which provides x, y and
                                              z-coordinate as a floating
                                              point value]
                           rightUpper, [Type: same as leftUpper]
                           leftLower,  [Type: same as leftUpper]
                           rightLower  [Type: same as leftUpper]) {

     // INITIALIZATION
     declaration upperPts [imageHeight+1]; // [Type: array of 3d-points]
     declaration leftPts [imageHeight+1];  // [Type: array of 3d-points]
     declaration v [Type: 3d-point with floating point values x, y and z];

     /*
      * Set up "upperPts"
      */

     upperPts[0].x = 0;
     upperPts[0].y = 0;
     upperPts[0].z = 0;
```

```
      // in c, c++, etc, you may have to cast imageWidth to a
      // floating-point value if you have stored it as an integer...
      v.x = (rightUpper.x – leftUpper.x) / (float) imageWidth;
      v.y = (rightUpper.y – leftUpper.y) / (float) imageWidth;
      v.z = (rightUpper.z – leftUpper.z) / (float) imageWidth;


      for (int i = 1; i <= imageWidth; i++) {
            upperPts[i].x = upperPts[i-1].x + v.x;
            upperPts[i].y = upperPts[i-1].y + v.y;
            upperPts[i].z = upperPts[i-1].z + v.z;
      }

      /*
       * Set up "leftPts"
       */

      leftPts[0].x = 0;
      leftPts[0].y = 0;
      leftPts[0].z = 0;

      v.x = (leftLower.x – leftUpper.x) / (float) imageHeight;
      v.y = (leftLower.y – leftUpper.y) / (float) imageHeight;
      v.z = (leftLower.z – leftUpper.z) / (float) imageHeight;

      for (int i = 1; i <= imageHeight; i++) {
            leftPts[i].x = leftPts[i-1].x + v.x;
            leftPts[i].y = leftPts[i-1].y + v.y;
            leftPts[i].z = leftPts[i-1].z + v.z;
      }

      /*
       * to be continued ...
       */
}
```

Now we have the position of all the pixels on the image. We can find the position of the pixel located at $(xPixel, yPixel)$ on the original picture by doing some basic maths:

$$positionOf(xPixel, yPixel) = \begin{pmatrix} upperPts[xPixel].x + leftPts[yPixel].x \\ upperPts[xPixel].y + leftPts[yPixel].y \\ upperPts[xPixel].z + leftPts[yPixel].z \end{pmatrix} + \begin{pmatrix} leftUpper.x \\ leftUpper.y \\ leftUpper.z \end{pmatrix}$$

We have to add `leftUpper`, because the position of each `upperPts` was addressed relatively to `leftUpper`. This means, that we pretend the cube to be located at $(x, y, z) = (0, 0, 0)$ which is not the case mostly.

Using the formula provided above, we are able to estimate the left upper boundary point for each pixel; how do we estimate the position of the other three boundary points? We simply calculate the left upper point of some other pixel:

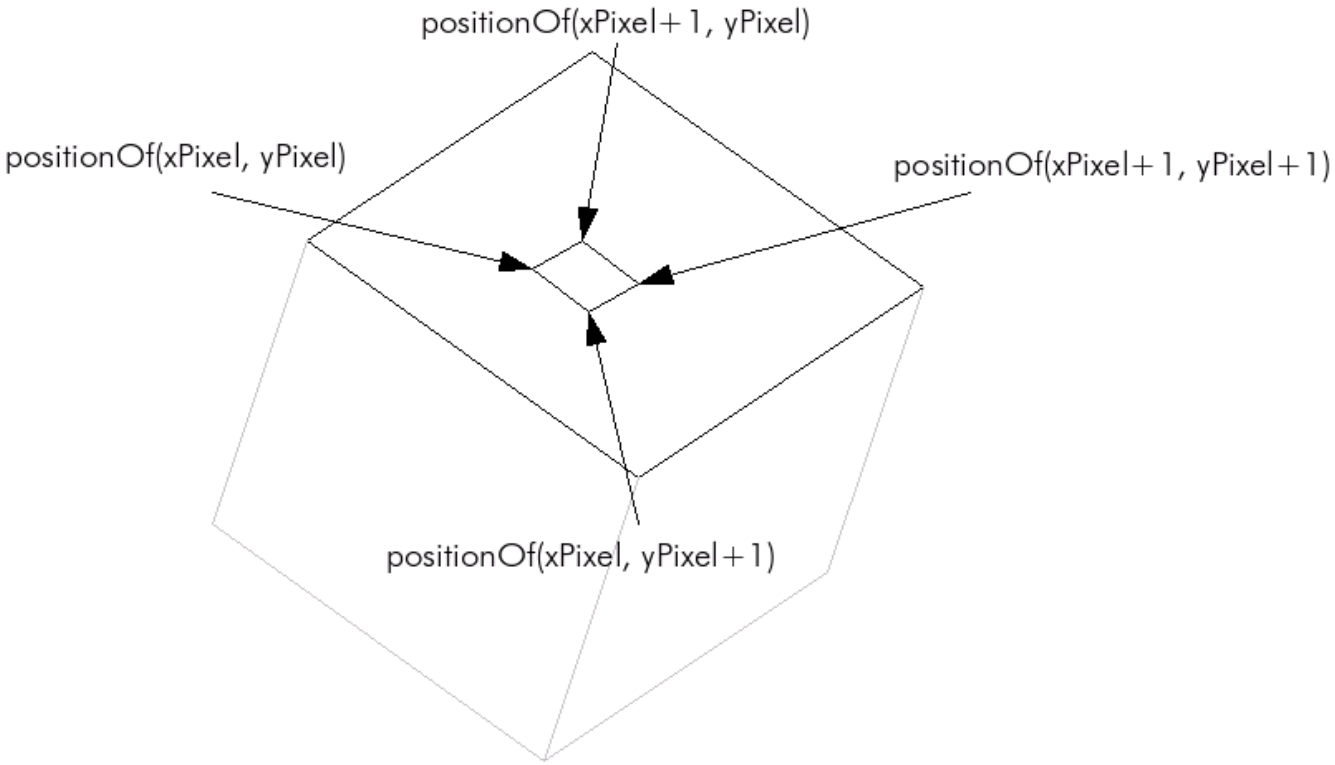



*Illustration 12: Mapping one pixel onto the surface of the cube*

Ok, now we have the boundary points of all pixels and we mapped them onto the cube, but they still have 3 coordinates. So how to get them onto the screen?

The first thing you should notice is, that neither `leftPts`, nor `upperPts` nor `positionOf(...)` gives you a concrete point on the screen, because they all have 3 coordinates and they only exists in a virtual 3D-coordinate-system. In order to get them onto the screen, one has to break down this 3D-system into a system which has only two coordinates, namely the screen. This is exactly, what the projection formula from chapter 1 does. This results in the following piece of code:

```
function textureMapping_3d(mapPicture [Type: Picture],
                           leftUpper, [Type: 3d-point which provides x, y and
                                              z-coordinate as a floating
                                              point value]
                           rightUpper, [Type: same as leftUpper]
                           leftLower,  [Type: same as leftUpper]
                           rightLower  [Type: same as leftUpper]) {

     // [preparation of upperPts[] and leftPts as seen above]

     declaration color;      // [Type: some color variable which stores
                             //    r (red), g (green) and b (blue) values
                             //    between 0 and 255]

     // [Type: 2D-point structure which provides an x- and y-coordinate]
     declaration leftUpper_2d, leftLower_2d, rightUpper_2d, rightLower_2d;
```

```
      // [Type: 3d-point structure which provides an x- ,y-, and z-coordinate]
      declaration leftUpper_3d, leftLower_3d, rightUpper_3d, rightLower_3d;


      // loop through all the pixels on the original image
      for (int yPixel = 0; yPixel < imageHeight; yPixel++) {
         for (int xPixel = 0; xPixel < imageWidth; xPixel++) {

                  // get the color of the pixel at the current position
                  color = get_pixel(mapPicture, xPixel, yPixel);

                  leftUpper_3d = upperPts[xPixel] +
                                    leftPts[yPixel] +
                                    leftUpper;

                  leftLower_3d = upperPts[xPixel] +
                                    leftPts[yPixel+1] +
                                    leftUpper;

                  rightUpper_3d = upperPts[xPixel+1] +
                                    leftPts[yPixel] +
                                    leftUpper;

                  rightLower_3d = upperPts[xPixel+1] +
                                          leftPts[yPixel+1] +
                                    leftUpper;



                  // project the boundary points from 3d to 2D
                  // known from chapter 1, "basic 3d calculations"
                  // see [http://happy-werner.de/howtos/isw/parts/3d/chapter_1/
                  // chapter_1_basic_3d.pdf]

                  // this function saves the 2D-coordinates of the point *_3d
                  // into *_2d
                  project_3d_2d(leftUpper_3d, leftUpper_2d);
                  project_3d_2d(leftLower_3d, leftLower_2d);
                  project_3d_2d(rightUpper_3d, rightUpper_2d);
                  project_3d_2d(rightLower_3d, rightLower_2d);

                  // draw a simple square using the given color
                  drawSquare(leftUpper_2d, leftLower_2d,
                              rightUpper_2d, rightLower_2d, color);
         }
      }
}
```

Obviously it works and gives us the following result:

Since filling out a polygon is not a simple thing, we skip it here. Take a look at Drawing squares is that difficult? for the discussion of this problem.

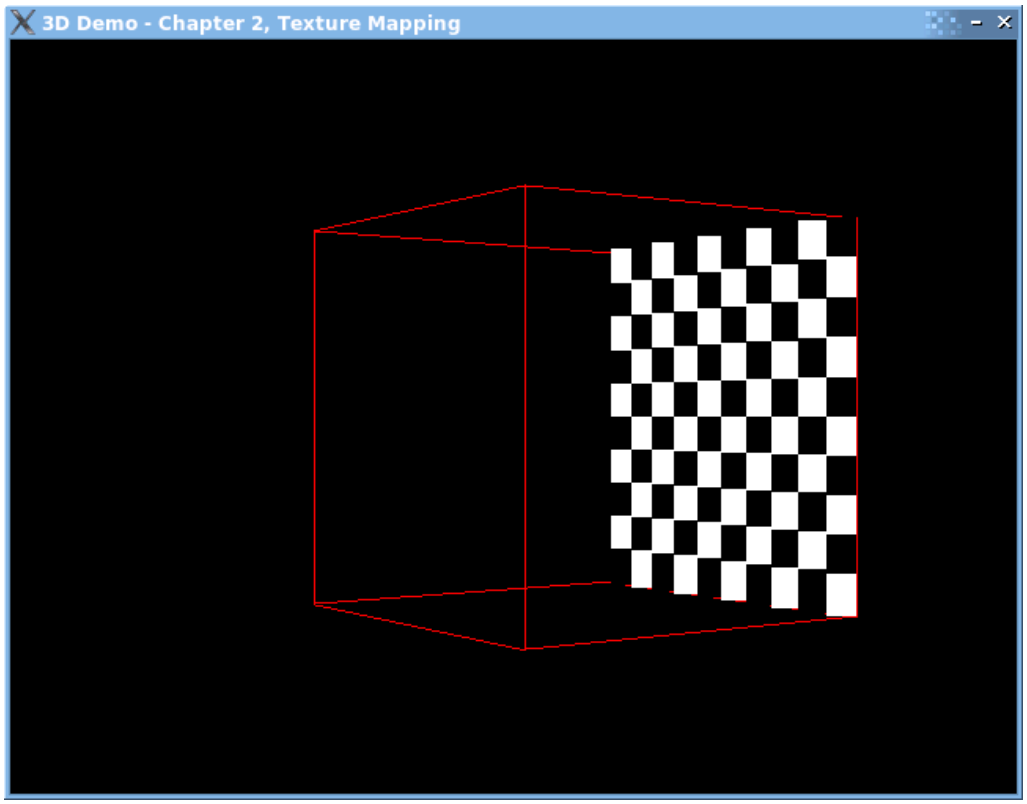



*Illustration 13: 3D texture mapping result*

| Download the source which supports perfect 3D texture mapping here: sdl3d_chapter2_1_3d_perfect_mapping.zip |
|---|

## 2.2 Analysis

☺ The algorithm does, what we want, it calculates the correct position of each pixel

☺ Moreover, the position is perfect, there is no other algorithm with such an accuracy

☺ It is relatively simple to implement and the provided source has a lot of possibilities where it could be optimized

☹ This algorithm has the worst performance of all texture algorithms, therefor it is not useful for developing real-time 3D-games

Do you wonder why this algorithm has such a bad performance? Let us take a closer look on the following lines:

```
// project the boundary points from 3d to 2d
// see [http://happy-werner.de/howtos/isw/parts/3d/chapter_1/
// chapter_1_basic_3d.pdf]
project_3d_2d(&leftUpper_3d, &leftUpper_2d);
project_3d_2d(&leftLower_3d, &leftLower_2d);
project_3d_2d(&rightUpper_3d, &rightUpper_2d);
project_3d_2d(&rightLower_3d, &rightLower_2d);
```

As you may know from chapter 1, this projection formula does nothing else but dividing the x3d-coordinate by the z3d-coordinate. See: dividing a floating-point number by some other floating-point number is nearly the most expensive and complex basic operation. Hence, these lines of source cause the most trouble.

## 2.3 Optimization

The algorithm provided above has a lot of optimization possibilities. If you called it every time, the main loop runs, you would calculate `upperPts` and `leftPts` and the sum of `upperPts[xPixel]` and `leftPts[yPixel]` everytime. But these values never change for a picture that does not change (because there are made up by some operation which depends on the height and width of the picture). So one could calculate `upperPts` and `leftPts` at the beginning for each picture, one wants to map. Then, a lot of operations at the middle of the source are completely gone. The only thing, which remains is adding the values with `leftUpper` which is necessary, because these values (x-, y- and z-coordinate of `leftUpper`) actually change.

All in all, there is no possibility to avoid the lines above, so the algorithm stays complex and slow at all.

- 3D texture mapping makes the scene look very realistic (it has the best accuracy)
- All in all, the algorithm is not very adequate for developing real-time systems (like games, etc), because it is very slow due to the amount of divisions

# 3 2D Interpolation

## 3.1 Considerations

Since we have seen, that mapping the picture perfectly is not useful for developing a fast real-time application, we have to think about faster ways. Since the projection is the critical point, we try to avoid it. Why did we need this projection? Because we did everything in 3 dimensions. Now we are going to break the problem down to 2 dimensions. In principle, the same technique which was used in 3D is applicable in 2D, so let's begin:

The first step will be to get rid of the third dimension, so we project `leftUpper`, `leftLower`, `rightUpper` and `rightLower` using the well-known projection formula. This means, that we have to do this critical operation (dividing two floating-point values), but only four times per texture, which is much faster than before. From here on, we operate in two dimensions, split up the way from `leftUpper` to `rightUpper`, and so on. The first problem one is faced with, is that applying basic maths is not enough for estimating the positions of each pixel on the surface. A quick example:

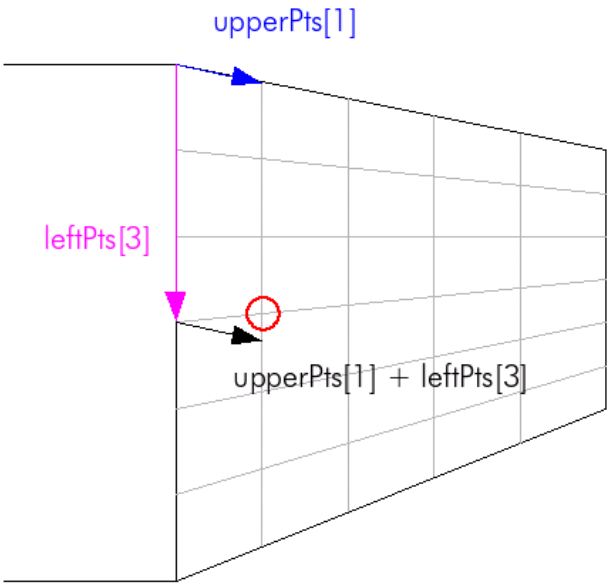



*Illustration 14: Basic vector mathematics fails (`upperPts[1]+leftPts[3]` should point at the cross in the red cycle)*

When we operate in 3D space, it works, because the coordinate system is "correct". If we simply ignore one dimension, we loose information. A part of this information is the "correctness" of the coordinate system. In other words: If we mixed up a virtual 3D-coordinate-system and a real 2D-coordinate-system (the screen), this would be a mistake.

The solution to this problem is to recalculate $\vec{v}$ for every line. We loop then over all "horizontal" lines and walk from the left to the right using this special $\vec{v}$. So the first thing, we have to do is estimating the beginning and the end of each horizontal line. These will be characterized by the points, shown in the following illustration:

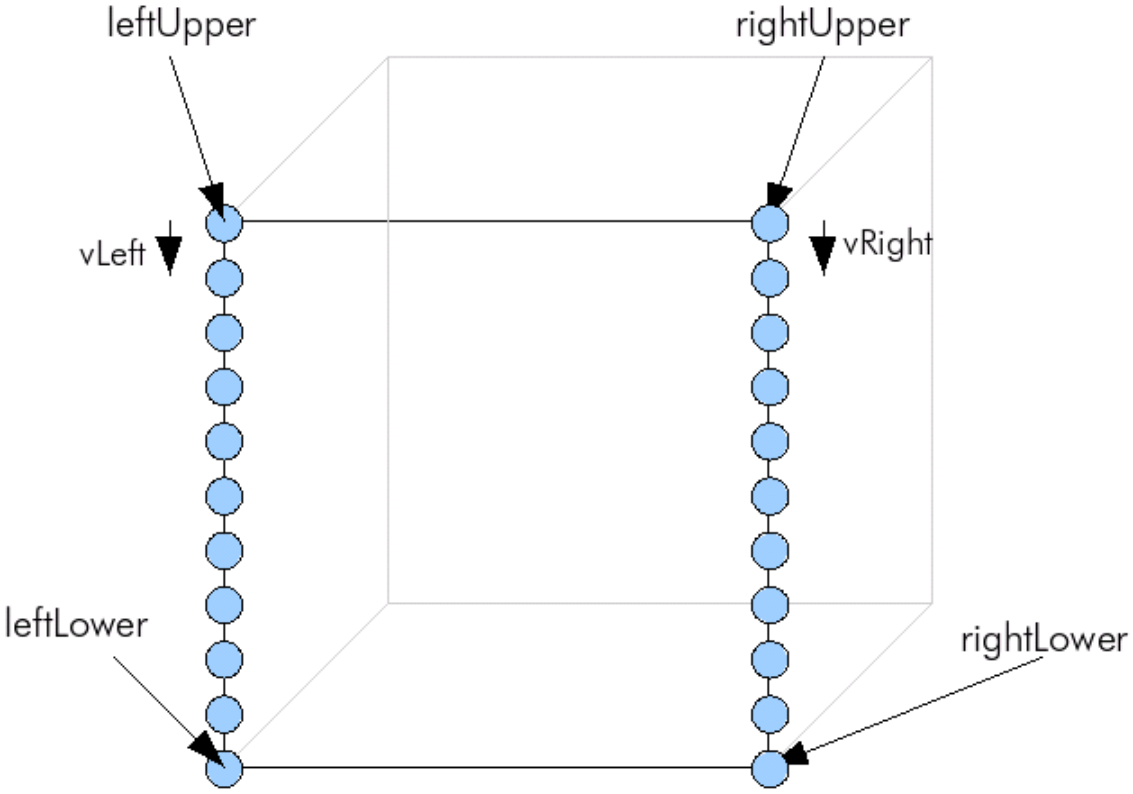



*Illustration 15: Beginning- and endpoints of each horizontal line*

As in perfect 3D mapping, these points will be stored in arrays called **`leftPts`** and **`rightPts`**. The Difference is, that they now consist of 2D-points directly on the screen and not in a virtual coordinate system. For calculating vLeft, vRight and the contents of the arrays, we use the same technique as provided in the chapter above; we split up the way from `leftUpper` to `leftLower`, `rightUpper` to `rightLower` for `rightPts`, respectively. Source:

```
function textureMapping_2d_interpolation(
                            mapPicture    [Type: Picture],
                            leftUpper,    [Type: 2d-point which provides x- and
                                              y-coordinate as integer]
                            rightUpper,   [Type: same as leftUpper]
                            leftLower,    [Type: same as leftUpper]
                            rightLower    [Type: same as leftUpper]) {() {

     declaration imageHeight [Type: integer];
     imageHeight = height of "mapPicture" in pixels;

     declaration rightPoints [imageHeight+1]; // [Type: array of 2d-points]
     declaration leftPts [imageHeight+1];  // [Type: array of 2d-points]

     /* For non-c-guys:
      * this "{" means, that all declared variables will be
      * trashed after the ending "}"
      * makes sense, because we dont need vLeft and vRight afterwards
```

```
 */
{

      // initialize v-vector for the left and the right side
      // (this splits up the way from *Upper to *Lower)

      declaration vLeft, vRight [Type: structure which provides x-
                                        and y-coordinate as
                                        floating point value]

      // in c, c++, etc, you may have to cast imageWidth to a
      // floating-point value if you have stored it as an integer...

      vLeft.x = (leftLower.x – leftUpper.x) / (float) imageHeight;
      vLeft.y = (leftLower.y – leftUpper.y) / (float) imageHeight;

      vRight.x = (rightLower.x – rightUpper.x) / (float) imageHeight;
      vRight.y = (rightLower.y – rightUpper.y) / (float) imageHeight;


      // here, it makes no sense to address everything relative to
      // leftUpper, because we can not do vector-tricks
      // --> here, we address absolute

      // IMPORTANT: here, we also have to use floats!
      declaration currentLeft, currentRight [Type: structure which provides
                                             x- and y-coordinate as
                                             floating point value]

      // since we address absolute, currentLeft != (0,0) but leftUpper!
      currentLeft.x = leftUpper.x;
      currentLeft.y = leftUpper.y;

      currentRight.x = rightUpper.x;
      currentRight.y = rightUpper.y;

        for (int i = 0; i <= imageHeight; i++) {

            // here, we are able to do some rounding, because
            // if the pixel is located at 12,5 or 12,75, its
            // no difference to us
            rightPoints[i].x = (int) currentRight.x;
            rightPoints[i].y = (int) currentRight.y;

            leftPts[i].x = (int) currentLeft.x;
            leftPts[i].y = (int) currentLeft.y;

            /* For non-c-guys:
             * a += b; means the same as a = a + b;
             */

            // but here, we have to use floats again,
            // because the mistakes are getting bigger
            // and bigger otherwise!
            currentLeft.x += vLeft.x;
```

```
                currentLeft.y += vLeft.y;

                currentRight.x += vRight.x;
                currentRight.y += vRight.y;
      }
    }

    [to be continued...]
}
```

Now that we have the beginning and the endpoint of each horizontal line, we can loop through them. For every horizontal line, we are going to split up the way from the beginning to the endpoint in oder to find out the position of the texels. In the following illustration, the upper part of the cube is being shown with the beginning of the first and the second horizontal line:

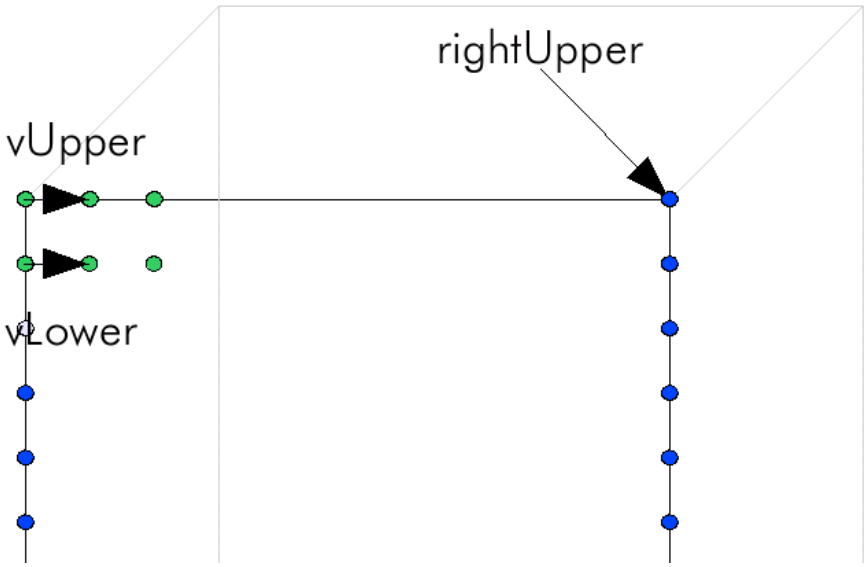



*Illustration 16: Horizontal technique - 2D-interpolation*

```
function textureMapping_2d_interpolation( ... ) {

    // [Initialization and parameters as seen above]

    // [Type: 2D-points providing an x- and y-coordinate as floating point
    //        values]
    declaration vUpper, vLower;
    declaration leftUpper_2d, rightUpper_2d, leftLower_2d, rightLower_2d;

    for (int i = 0; i < imageHeight; i++) {
      // calculate vUpper and vLower


      vUpper.x = (rightPoints[i].x - leftPts[i].x) / (float) imageWidth;
      vUpper.y = (rightPoints[i].y - leftPts[i].y) / (float) imageWidth;

      vLower.x = (rightPoints[i+1].x - leftPts[i+1].x) /
```

```
                                    (float) imageWidth;
        vLower.y = (rightPoints[i+1].y - leftPts[i+1].y) /
                                    (float) imageWidth;
```

So far, we have calculated vUpper and vLower as seen in illustration 16. Now, we are going to initialize the first texel in a horizontal line which looks like this:

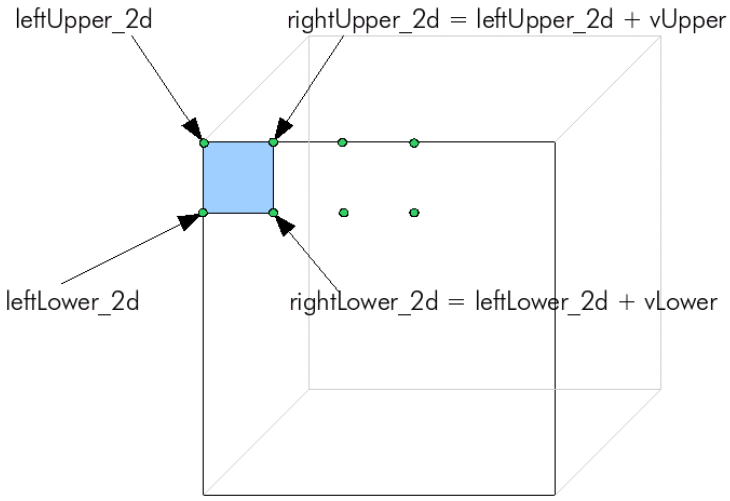



*Illustration 17: Initialization of the first texel in the first horizontal line*

```
        // initialize four points (the boundary
        // points for the first and most left texel)
        // for ONE SINGLE horizontal line
        leftUpper_2d.x = leftPts[i].x;
        leftUpper_2d.y = leftPts[i].y;

        leftLower_2d.x = leftPts[i+1].x;
        leftLower_2d.y = leftPts[i+1].y;

        rightUpper_2d.x = leftUpper_2d.x + vUpper.x;
        rightUpper_2d.y = leftUpper_2d.y + vUpper.y;

        rightLower_2d.x = leftLower_2d.x + vLower.x;
        rightLower_2d.y = leftLower_2d.y + vLower.y;

        // loop through the horizontal line
        for (int j = 0; j < imageWidth; j++) {

                  // get the color of the pixel at the current position
                  color = get_pixel(mapPicture, j, i);

                  // draw the texel
                  drawSquare(leftUpper_2d, leftLower_2d,
                             rightUpper_2d, rightLower_2d, color);
```

Ok, now we are going to move the 2D-points. Afterwards, the situation looks like the following:

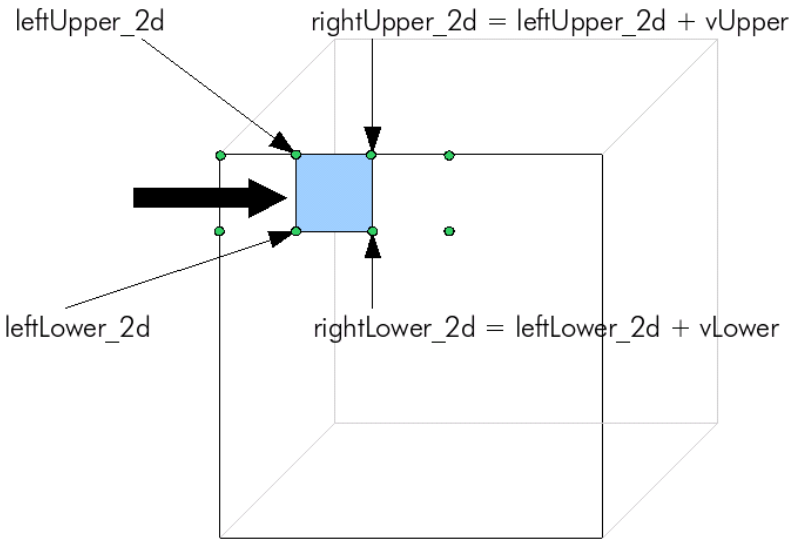



*Illustration 18: The boundary points of the texel have been moved to the right*

```
                    // go to the next texel
                    leftUpper_2d.x += vUpper.x;
                    leftUpper_2d.y += vUpper.y;

                    rightUpper_2d.x += vUpper.x;
                    rightUpper_2d.y += vUpper.y;

                    leftLower_2d.x += vLower.x;
                    leftLower_2d.y += vLower.y;

                    rightLower_2d.x += vLower.x;
                    rightLower_2d.y += vLower.y;

        }
    }
}
```

Download the source which provides 2D-interpolation here: sdl3d_chapter2_2_2d_interpolation.zip

## 3.2 Analysis & Optimization

☺ The algorithm provided above is much faster than perfect 3D mapping

☺ For pictures like "walls", Pamela Anderson, etc, the 2D-interpolation produces a relatively good accuracy

☹ There are many floating-points operations which could be optimized

☹ Pictures (like the crate and the cross seen at illustrations 1 and 3) which allow drawing conclusions from the coordinate system and the dimensions do not look realistic

### 3.2.1 Avoiding floating point operations

When you want to avoid floating point operations under the assumption that there exists a proportionality between both input values, one can always try to rape the **Bresenham algorithm**. The problem of floating-point operations come from the calculation of vUpper and vLower. We need them, because we have to find a way from `leftPts[i]` to `rightPts[i]`. The first thing, we have to do is splitting up the way in x- and y-coordinate. vUpper's x-coordinate is calculated like the following:

$$vUpper.x = \frac{rightPts[i].x - leftPts[i].x}{imageWidth}$$

The question which raises up is: how many steps do i want to do afterwards? As we can see, the second for-loop counts like the following:

```
for (int j = 0; j < imageWidth; j++) { [...]
```

So the problem can be reduced to drawing a line from $(leftPts[i].x, 0)$ to $(rightPts[i].x, imageWidth)$ (because after going "imageWidth" many steps, i have to get from `leftPts[i].x` to `rightPts[i].x`).

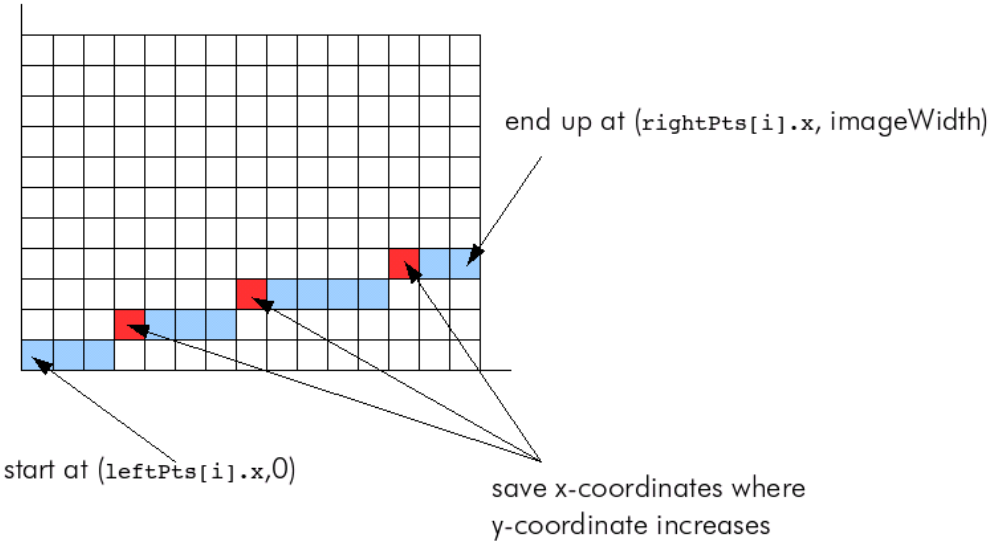



*Illustration 19: Idea of the Bresenham like mapping*

This idea results in a special function which consists of the original Bresenham algorithm basically, but with a few manipulations: Every time, the second variable (the y-coordinate, from the Bresenham algorithm's point of view) changes, it writes down the current x-coordinate in an array. These will be the coordinates of the texels. The function will be called "bresenham_mapping". The lines, which are marked gray like this, belong to the regular implementation of the Bresenham algorithm. The contents of the function was mainly stolen from Wikipedia.de (german: http://de.wikipedia.org/wiki/Bresenham-Algorithmus). Apart from the usual source, the lines which are marked like this were added in order to keep track of the current coordinate.

```
void bresenham_mapping(coordinateStart [Type: integer],
                       coordinateEnd [Type: integer],
                       mapLength [Type: integer],
                       coordinates [Type: array of size mapLength of integers]) {

      // [Bresenham]: variables
      declaration t, dist, xerr, yerr, dx, dy, incx [Type: integer];

      // to keep track, where we are
      declaration currentCoordinate [Type: integer];
      // to keep track of the index of the array
      declaration currentIndex [Type: integer];


      // initialization: coordinates[0] is the startpoint
      coordinates[0] = coordinateStart;
      // current coordinate is the startpoint too
      currentCoordinate = coordinateStart;
      // the current index in the array is 0
      currentIndex = 0;

      // [Bresenham]: calculate differences of dimensions
      dx = coordinateEnd - coordinateStart;
      dy = mapLength;

      // [Bresenham]: estimate sign of the incremental part(s)
      if(dx<0) {
            incx = -1;
            dx = -dx;
      } else
            incx = dx ? 1 : 0;

      // [Bresenham]: which of the distances is bigger?
      dist = (dx > dy)?dx:dy;

      // [Bresenham]: Initialization
      xerr = dx;
      yerr = dy;

      // main loop
      for(t = 0; t < dist; ++t) {

            // [Bresenham]: regular Bresenham operations
            xerr += dx;
            yerr += dy;
```

```
            if(xerr > dist) {
                  xerr -= dist;
                  // keep track of the x-coordinate!
                  currentCoordinate += incx;
            }

            if(yerr>dist) {
                  yerr -= dist;
                  // if the y-coordinate is increased, this means, that
                  // we are switching pixels on the original image NOW!

                  // --> write the current coordinate down, because this is
                  //     one coordinate of the boundary point of
                  //     the current texel!
                  // --> increase currentIndex, because we want
                  //     to address a free cell in the array

                  currentIndex++;
                  coordinates[currentIndex] = currentCoordinate;
            }
      } /* for i = 1 to distance */

      // write down the last coordinate
      coordinates[mapLength+1] = coordinateEnd;
      return;
}
```

Since this function is one-dimensional, we have to call it twice per line we want to split up (for the x-coordinates the first time, then for the y-coordinates the second time). The calculation of `leftPts` looks like the following for example:

```
      // imageHeight is the height of the picture we want to map

      int leftPtsXCoordinates[imageHeight + 1];
      int leftPtsYCoordinates[imageHeight + 1];

      // now split up the way from leftUpper to leftLower to
      // "imageHeight" small steps

      // 1) the X-coordinates
      bresenham_mapping(leftUpper.x, leftLower.x, imageHeight,
                              leftPtsXCoordinates);

      // 2) the y-coordinates
      bresenham_mapping(leftUpper.y, leftLower.y, imageHeight,
                              leftPtsYCoordinates);
```

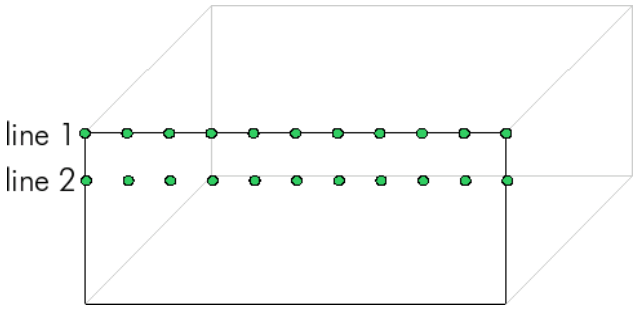
Using this technique for the horizontal calculation too, one can avoid every floating-point operation in the 2D-interpolation algorithm.

### 3.2.2 Some more optimization

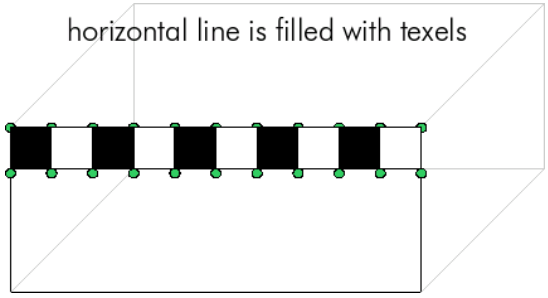
If you keep track of the source, then you may notice some other optimizations of the algorithm. This was done, because in the first version, one calculates the lines two times. This is unnecessary, and so, the coordinates of the "last line" are being saved in order to reuse them.

Before the optimization, the calculation process of the algorithm looked like the following:

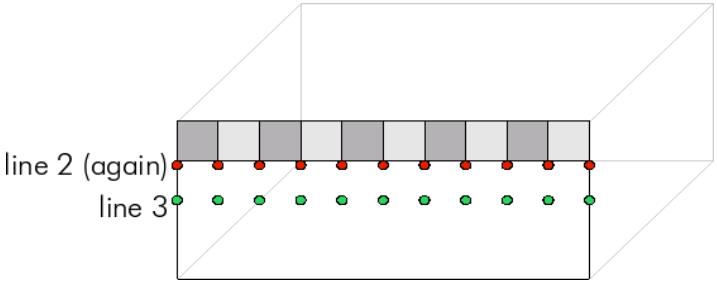
1) line 1 is being calculated as the "upper" line/bound
2) line 2 is being calculated as the "lower" line/bound




3) the texels of the first horizontal line are being drawn




4) line 2 is being calculated as the upper bound (unnecessary)
5) line 3 is being calculated as the lower bound




6) the texels of the second horizontal line are being drawn
7) line 3 is being calculated as the upper bound (unnecessary)
8) [...]

Afterwards, it calculates in a smarter way:

1) line 1 is being calculated
2) line 2 is being calculated
3) “upperPointer” is being set to line 1
4) “lowerPointer” is being set to line 2
5) the texels of the first horizontal line are being drawn
6) upperPointer is being set to line 2
7) line 3 is being calculated
8) lowerPointer is being set to line 3
9) the texels of the second horizontal line are being drawn
10) [...]

Due to this pointer-game, there are no unnecessary calculations anymore.

| |
|---|
| Download the source which provides the fastest possible 2D-interpolation here:<br>sdl3d_chapter2_3_2d_interpolation_bresenham.zip |

### 3.2.3 Making images look more realistic

Ok, so far we have avoided all floating point operation. This costs us a realistic texture mapping. There are many different techniques for avoiding these nasty side-effects, so here is one possible solution to this problem:

If you have a texture and you want to map a picture on it, use both, 3D- and 2D texture mapping. In other words: calculate 9, 16, or more points using the fully correct 3D mapping technique and interpolate in between them using the 2D mapping technique...

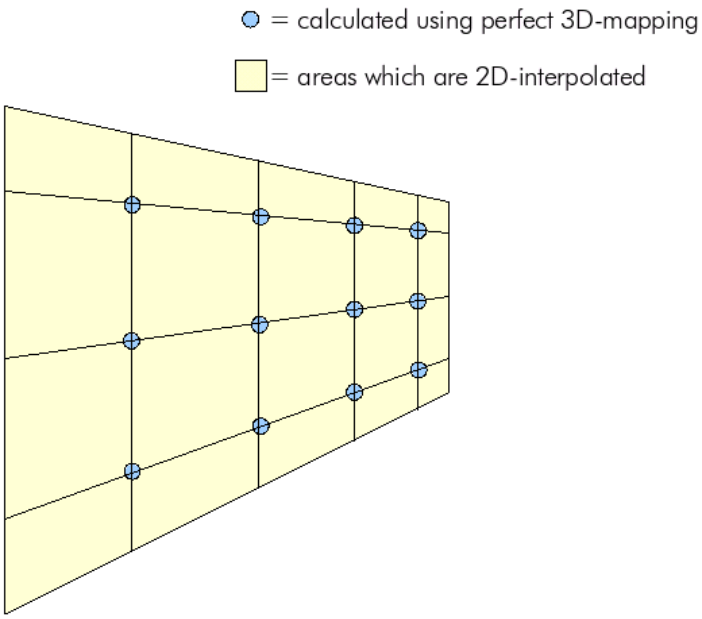



*Illustration 20: Technique which uses 3D and 2D texture mapping in order to avoid side-effects*

- 2D-interpolated texture mapping is much faster than 3D texture mapping
- It is possible to do 2D texture mapping completely without any floating point operation
- Although the algorithm suffers from some flaws, in special situations, it is possible to make the scene look realistic (maybe in addition with 3D-techniques as shown above)

# 4 Drawing squares is that difficult?

Drawing a simple quadrangle is not that easy as it sounds at first. There exists an algorithm called "scanline-algorithm" (http://www.cs.rit.edu/~icss571/filling/index.html provides a very good explanation and manual for implementing it) which is able to fill out a general polygon. At first, one has to say, that this algorithm is very complex. A possible idea for filling squares would be to implement it using some shortcuts for filling out just a simple quadrangle in order to speed up the whole thing, but is it necessary to do that? What if we just used a very simple technique to solve the problem of drawing texels? The solution is to reduce the problem on drawing an even more simple square, and not a general quadrangle. What is the difference between them and how does the solution look like? We simply transform the general quadrangle into a more simple square which is very easy (and therefor fast) to draw:

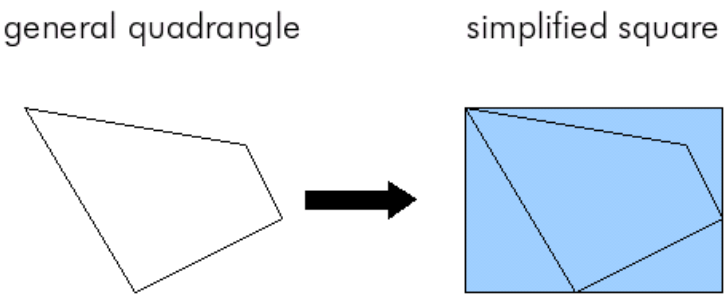



*Illustration 21: Approximation of a general quadrangle trough a square*

So what effects does this have on texture mapping? Let us imagine, we had a 4x4 pixel texture. Then this technique had some serious effects on the scene. On the right, you can see the picture mapped using the correct function for drawing quadrangles. On the left, you can see the picture mapped and drawn using the interpolation technique for drawing the texels much faster:

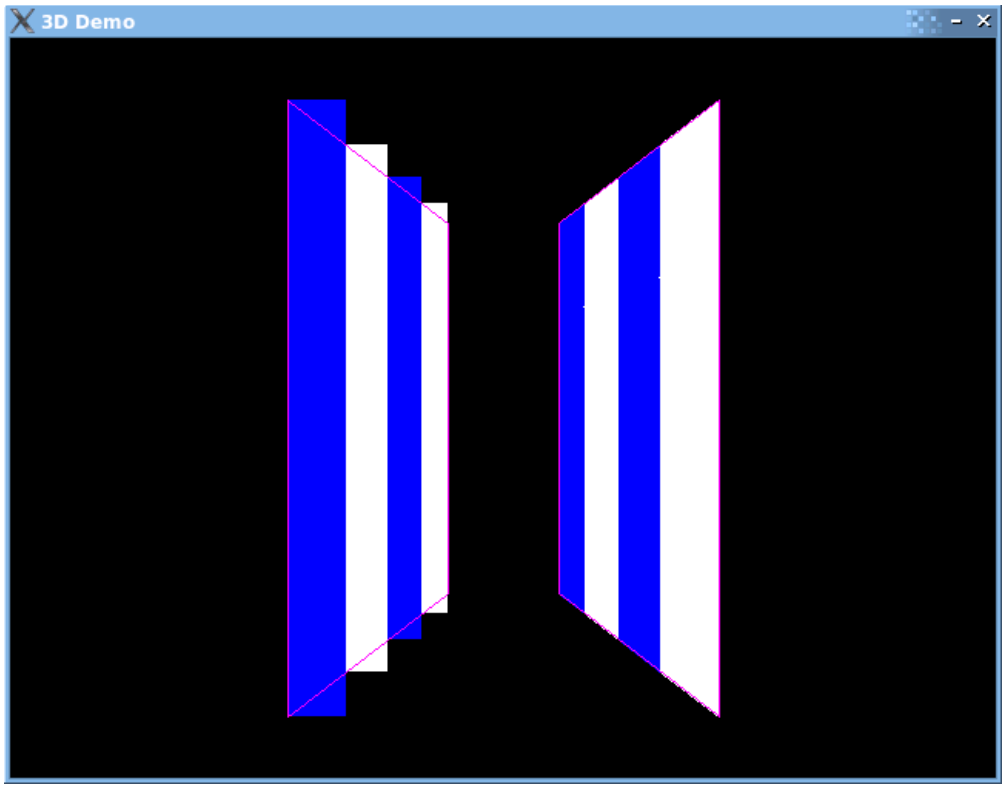



*Illustration 22: Side-effects of square approximation of quadrangles*

As you can see, when having a 10x10 or smaller picture, the mentioned technique is not adequate and one has to implement an algorithm which draws the quadrangle completely correct. But what happens, if we have a picture of a more realistic size, say 64x64 pixels? On the right, you can – again – see the picture mapped using the correct function for drawing quadrangles. On the left, you can see the picture mapped and drawn using the interpolation technique for drawing the texels much faster (can you tell the difference?):

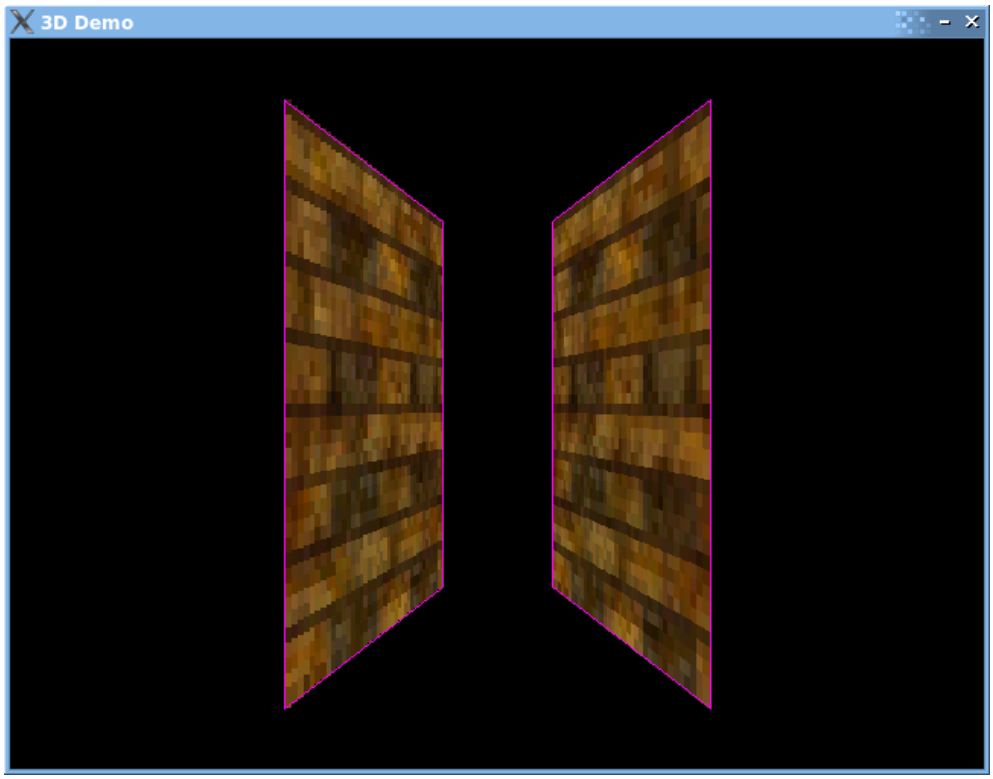



*Illustration 23: Square interpolation has no visible side-effects on pictures larger than 20x20 pixels*