Interpolation

Contents

Introduction

Bilinear Filtering

Trilinear Interpolation

Source Code

Bilinear Interpolation

Bilinear interpolation is used when we need to know values at random position on a regular 2D grid. Note that this grid can as well be an image or a texture map. In our example we are interested in finding a value at the location marked by the green dot (c which has coordinates cx, cy). To compute a value for c we will first perform two linear interpolations (see introduction) in one direction (x direction) to get b and a. To do so we will linearly interpolate c00-c10 and c01-c11 to get a and b using tx (where tx=cx). Then we will linearly interpolate a-b along the second direction (y-axis) to get c using ty (ty=cy). Whether you start interpolating the first two values along the x-axis or along the y-axis doesn't make any difference. In our example we start by interpolating c00-c10 and c01-c11 to get a and b. We could as well have interpolated c00-c01 and c10-c11 using ty then interpolated the result (a and b) using tx. To make the code easier to debug and write though it is recommended to follow the axis order (x, y and z for trilinear interpolation).

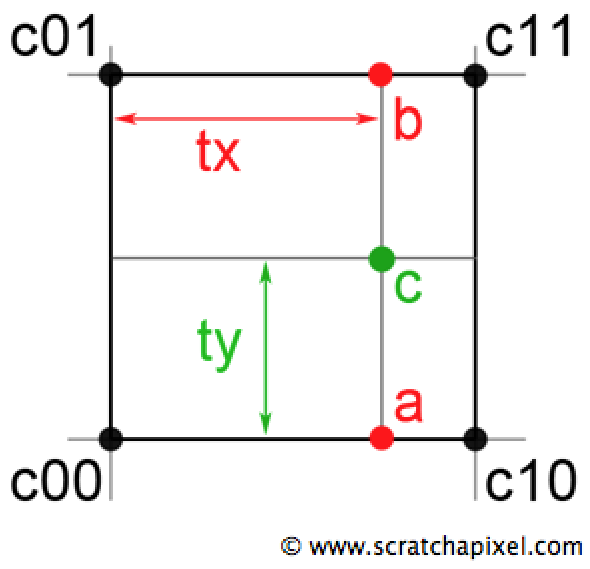


Figure 1: bilinear interpolation. We perform two linear interpolations first to compute a and b and then we interpolate a and b to find c.

evaluate is not outside the limits of your grid (if the grid has a resolution NxM you may need to create (N+1)x(M+1) vertices or NxM vertices and assume your grid has a resolution of (N-1)x(M-1). Both techniques work it is a matter of preference).

Contrary to what the name suggests, bilinear interpolation is not a linear process but the product of two linear functions. The function is linear if the sample point lies on one of the edges of the cell (line c00-c10 or c00-c01 or c01-c11 or c10-c11). Everywhere else it is quadratic.

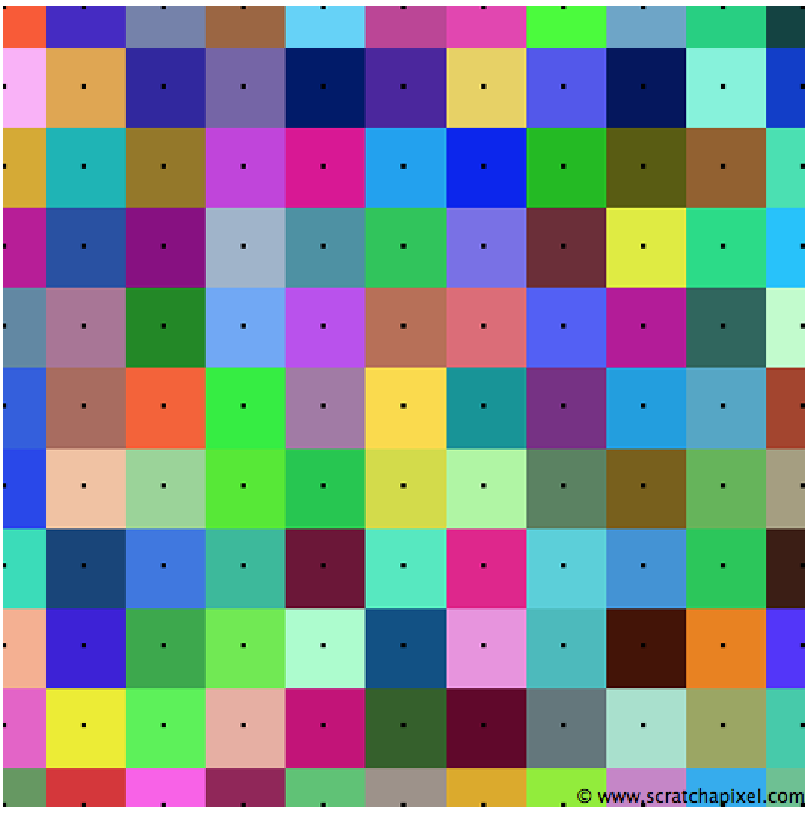
In the following example (complete source code is available for download) we create an image by interpolating the values (colours) of a grid for each pixel of that image. Many of the image pixels have coordinates which do not overlap the grids coordinates. We use a bilinear interpolation to compute interpolated colours at these "pixel" positions.

```
float bilinear(
   const float &tx,
   const float  &ty,
   const Vec3f &c00,
   const Vec3f &c10,
   const Vec3f &c01,
   const Vec3f &c11)
{
#if 1
    float  a = c00 * (1 - tx) + c10 * tx;
    float  b = c01 * (1 - tx) + c11 * tx;
    return a * (1) - ty) + b * ty;
#else
    return (1 - tx) * (1 - ty) * c00 +
        tx * (1 - ty) * c10 +
        (1 - tx) * ty * c01 +
        tx * ty * c11;
#endif
}

void testBilinearInterpolation()
{
    // testing bilinear interpolation
    int imageWidth = 512;
    int gridSizeX = 9, gridSizeY = 9;
    Vec3f *grid2d = new Vec3f[(gridSizeX + 1) * (gridSizeY + 1)]; // lattices
    // fill grid with random colors
    for (int j = 0, k = 0; j <= gridSizeY; ++j) {
        for (int i = 0; i <= gridSizeX; ++i, ++k) {
            grid2d[j * (gridSizeX + 1) + i] = Vec3f(drand48(), drand48(), drand
        }
    }
    // now compute our final image using bilinear interpolation
    Vec3f *imageData = new Vec3f[imageWidth*imageWidth], *pixel = imageData;
```

```
                // convert i,j to grid coordinates
                T gx = i / float(imageWidth) * gridSizeX; // be careful to interpol
                T gy = j / float(imageWidth) * gridSizeY; // be careful to interpol
                int gxi = int(gx);
                int gyi = int(gy);
                const Vec3f & c00 = grid2d[gyi * (gridSizeX + 1) + gxi];
                const Vec3f & c10 = grid2d[gyi * (gridSizeX + 1) + (gxi + 1)];
                const Vec3f & c01 = grid2d[(gyi + 1) * (gridSizeX + 1) + gxi];
                const Vec3f & c11 = grid2d[(gyi + 1) * (gridSizeX + 1) + (gxi + 1)]
                *(pixel++) = bilinear(gx - gxi, gy - gyi, c00, c10, c01, c11);
            }
        }
        saveToPPM("./bilinear.ppm", imageData, imageWidth, imageWidth);
        delete [] imageData;
    }
```

The bilinear function is a template so you can interpolate data of any type (float, colour, etc.). Notice also that the function can compute the same result in two different ways. The first method (line xx to xx) is more readable, but some people prefer to you use the second method (line xx to xx) because the interpolation can be seen as a weighted sum of the four vertices (weighted because c00, c01, c10 and c11 are multiplied by some coefficients. For instance `(1 - tx) * (1 - ty)` is the weighting coefficient for c00).



The advantage of bilinear interpolation is that it is fast and simple to implement. However, If you look at the second image from figure 2, you will see that bilinear interpolation creates some patterns which are not necessarily acceptable interpolation for. If you need a better result you will need to use more advanced interpolation techniques involving interpolation functions of degree two or more (such as the smoothstep function for example which is used in for generating procedural noise as described in the lesson Procedural Patterns and Noise: Part 1).

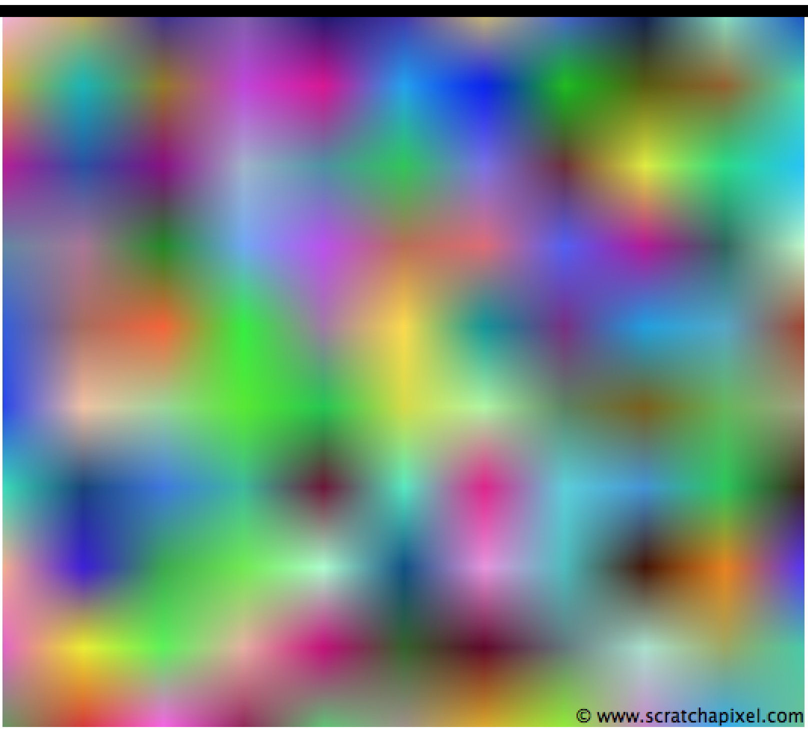


Figure 2: each black dot in the first image represents a vertex on the grid (the resolution of the grid is 10x10 cells which means 11x11 vertices). The second image is the result of interpolating the grid vertex data to compute the the pixel colours of a 512x512 image.

← Previous Chapter Chapter 2 of 4 Next Chapter →