

Interpolation

Contents

[Introduction](#)

[Bilinear Filtering](#)

[Trilinear Interpolation](#)

[Source Code](#)

News (August, 31): We are working on Scratchapixel 3.0 at the moment (current version of 2). The idea is to make the project open source by storing the content of the website on GitHub as Markdown files. In practice, that means you and the rest of the community will be able to edit the content of the pages if you want to contribute (typos and bug fixes, rewording sentences). You will also be able to contribute by translating pages to different languages if you want to. Then when we publish the site we will translate the Markdown files to HTML. That means new design as well.

That's what we are busy with right now and why there won't be a lot of updates in the weeks to come. More news about SaP 3.0 soon.

We are looking for native English (yes we know there's a typo here) speakers that will be willing to readproof a few lessons. If you are interested please get in touch on Discord, in the #scratchapixel3-0 channel. Also looking for at least one experienced full dev stack dev that would be willing to give us a hand with the next design.

Feel free to send us your requests, suggestions, etc. (on Discord) to help us improve the website.

And you can also donate). Donations go directly back into the development of the project. The more donation we get the more content you will get and the quicker we will be able to deliver it to you.

5 mns read.

Bilinear Interpolation

Bilinear interpolation is used when we need to know values at random positions 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 c_x, c_y). 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 c_{00} - c_{10} and c_{01} - c_{11} to get a and b using t_x (where $t_x = c_x$). Then we will linearly interpolate a-b along the second direction (y-axis) to get c using t_y ($t_y = c_y$). Whether you start interpolating the first two values along the x-axis or the y-axis doesn't make any difference. In our example, we start by interpolating c_{00} - c_{10} and c_{01} - c_{11} to get a and b. We could as well have interpolated c_{00} - c_{01} and c_{10} - c_{11} using t_y and then interpolated the result (a and b) using t_x . 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).

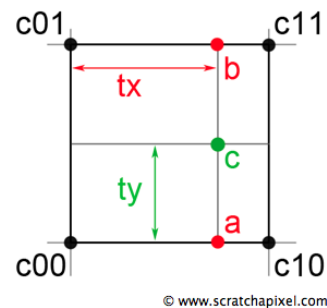


Figure 1: bilinear interpolation. We perform two linear interpolations first to compute a and b and then we interpolate a and b to find c.

When you perform linear interpolation, it is generally a good idea to check in the code that t is not greater than 1 or lower than 0 and to check that the point you are trying to evaluate is not outside the limits of your grid (if the grid has a resolution $N \times M$ you may need to create $(N+1) \times (M+1)$ vertices or $N \times M$ vertices and assume your grid has a resolution of $(N-1) \times (M-1)$. 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 c_{00} - c_{10} or c_{00} - c_{01} or c_{01} - c_{11} or c_{10} - c_{11}). Everywhere else it is quadratic.

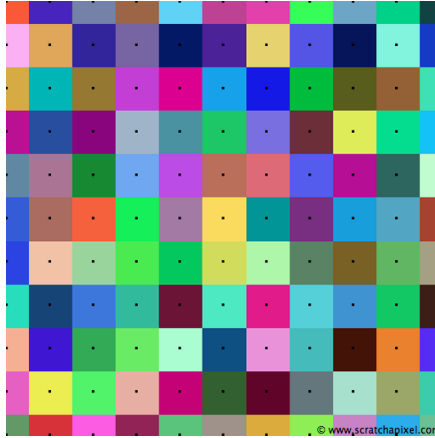
In the following example (complete source code is available for download) we create an image by interpolating the values (colors) of a grid for each pixel of that image. Many of the image pixels have coordinates that do not overlap the grid's coordinates. We use bilinear interpolation to compute interpolated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

```

001 float bilinear(
002     const float &tx,
003     const float &ty,
004     const Vec3f &c00,
005     const Vec3f &c10,
006     const Vec3f &c01,
007     const Vec3f &c11)
008 {
009     #if 1
010         float a = c00 * (1 - tx) + c10 * tx;
011         float b = c01 * (1 - tx) + c11 * tx;
012         return a * (1 - ty) + b * ty;
013     #else
014         return (1 - tx) * (1 - ty) * c00 +
015             tx * (1 - ty) * c10 +
016             (1 - tx) * ty * c01 +
017             tx * ty * c11;
018     #endif
019 }
020
021 void testBilinearInterpolation()
022 {
023     // testing bilinear interpolation
024     int imageWidth = 512;
025     int gridSizeX = 9, gridSizeY = 9;
026     Vec3f *grid2d = new Vec3f[(gridSizeX + 1) * (gridSizeY + 1)]; //latter
027     // fill grid with random colors
028     for (int j = 0, k = 0; j <= gridSizeY; ++j) {
029         for (int i = 0; i <= gridSizeX; ++i, ++k) {
030             grid2d[j * (gridSizeX + 1) + i] = Vec3f(drnd48(), drnd48(),
031             }
032     }
033     // now compute our final image using bilinear interpolation
034     Vec3f *imageData = new Vec3f[imageWidth*imageWidth], *pixel = imageData;
035     for (int j = 0; j < imageWidth; ++j) {
036         for (int i = 0; i < imageWidth; ++i) {
037             // convert i,j to grid coordinates
038             T gx = i / float(imageWidth) * gridSizeX; //be careful to in
039             T gy = j / float(imageWidth) * gridSizeY; //be careful to in
040             int gxi = int(gx);
041             int gyi = int(gy);
042             const Vec3f & c00 = grid2d[gyi * (gridSizeX + 1) + gxi];
043             const Vec3f & c10 = grid2d[gyi * (gridSizeX + 1) + (gxi + 1)];
044             const Vec3f & c01 = grid2d[(gyi + 1) * (gridSizeX + 1) + gxi];
045             const Vec3f & c11 = grid2d[(gyi + 1) * (gridSizeX + 1) + (gxi + 1)];
046             *pixel++ = bilinear(gx - gxi, gy - gyi, c00, c10, c01, c11);
047         }
048     }
049     saveToPPM("./bilinear.ppm", imageData, imageWidth, imageWidth);
050     delete [] imageData;
051 }

```

The bilinear function is a template so you can interpolate data of any type (float, color, 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 depending on what you intend to use the result of the 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 for

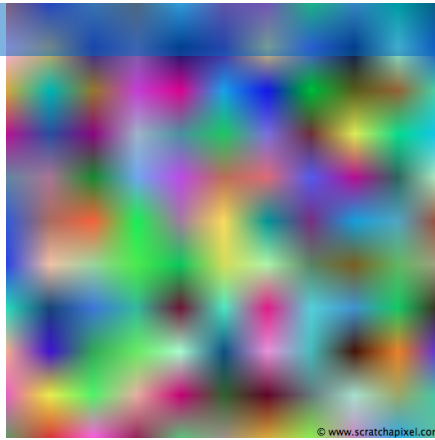


Figure 2: each black dot in the first image represents a vertex on the grid (the resolution of the grid is 10x10 cells which means 11x11 vertices). The second image is the result of interpolating the grid vertex data to compute the pixel colours of a 512x512 image.