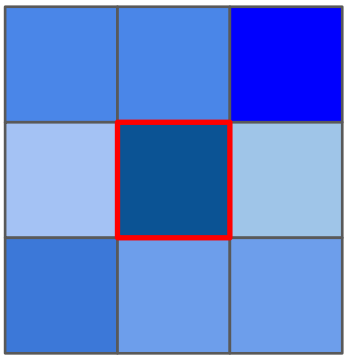
Texture Filtering

Jim Van Verth

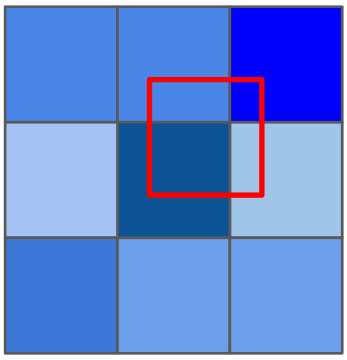
Some definitions

Pixel: picture element. On screen.

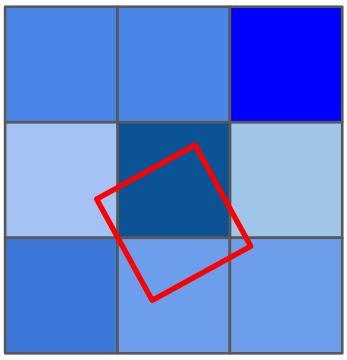
Texel: texture element. In image (AKA texture).



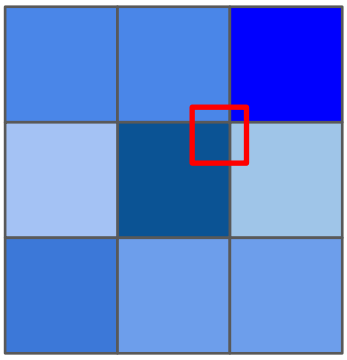
Suppose we have a texture (represented here in blue). We can draw the mapping from a pixel to this space, and represent it as a red square. If our scale from pixel space to texel space is 1:1, with no translation, we just cover one texel, so rendering is easy.



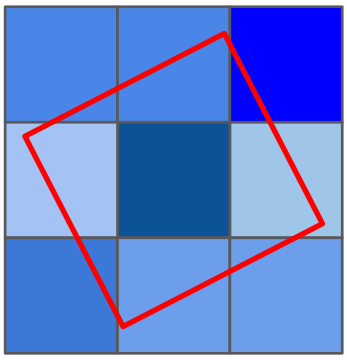
If we translate, the pixel can cover more than one texel, with different areas.



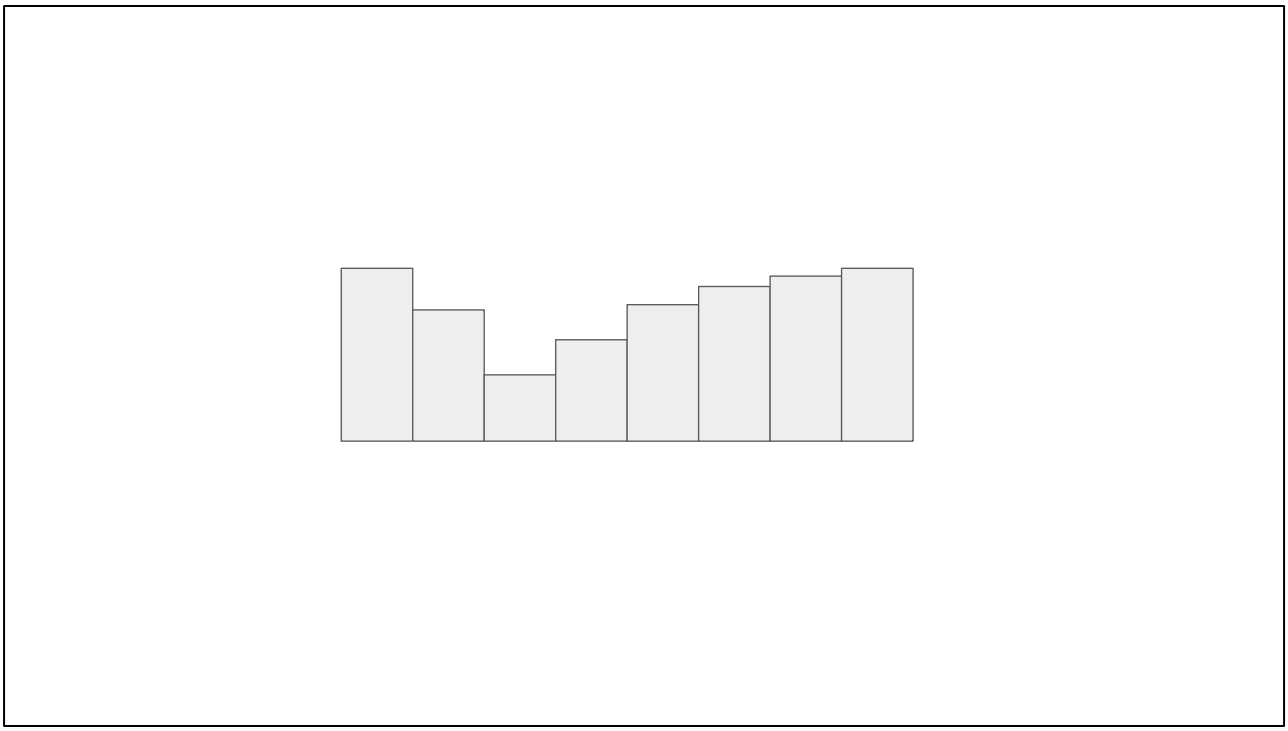
If we rotate, then the pixel can cover even more texels, with odd shaped areas.



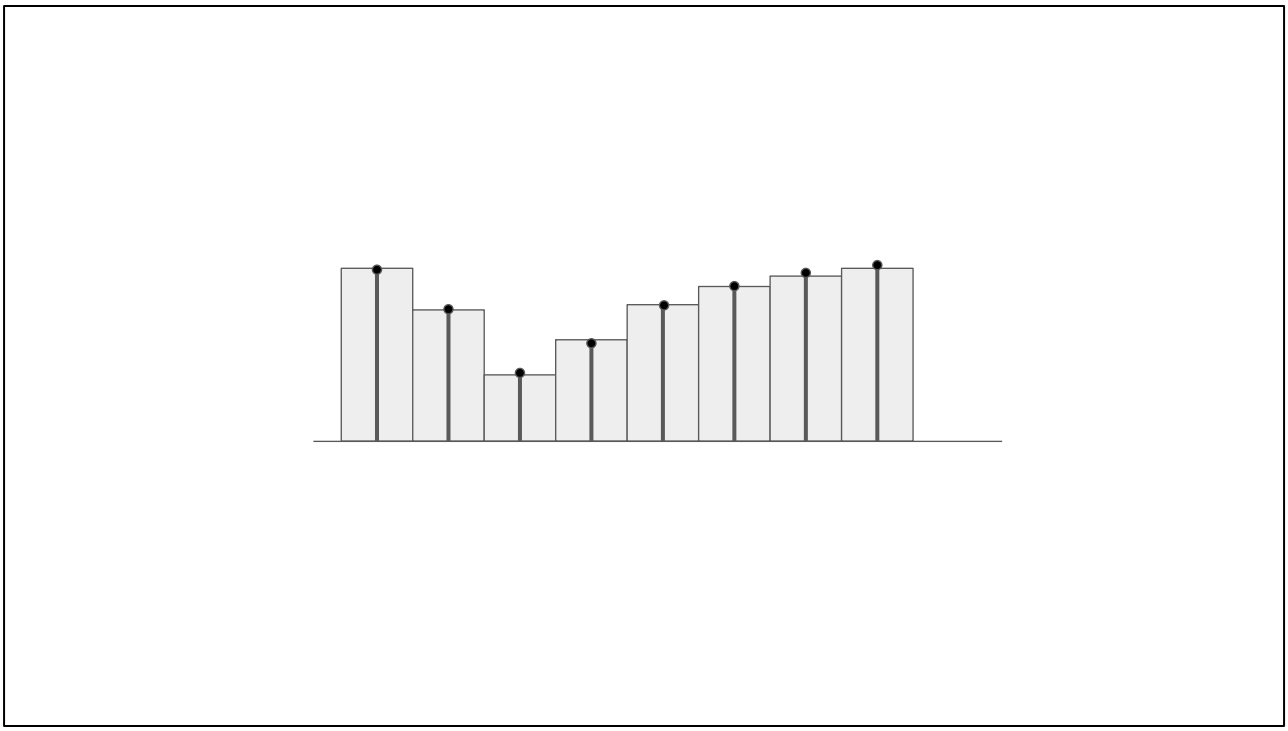
And with scaling up the texture (which scales down the pixel area in texture space)...



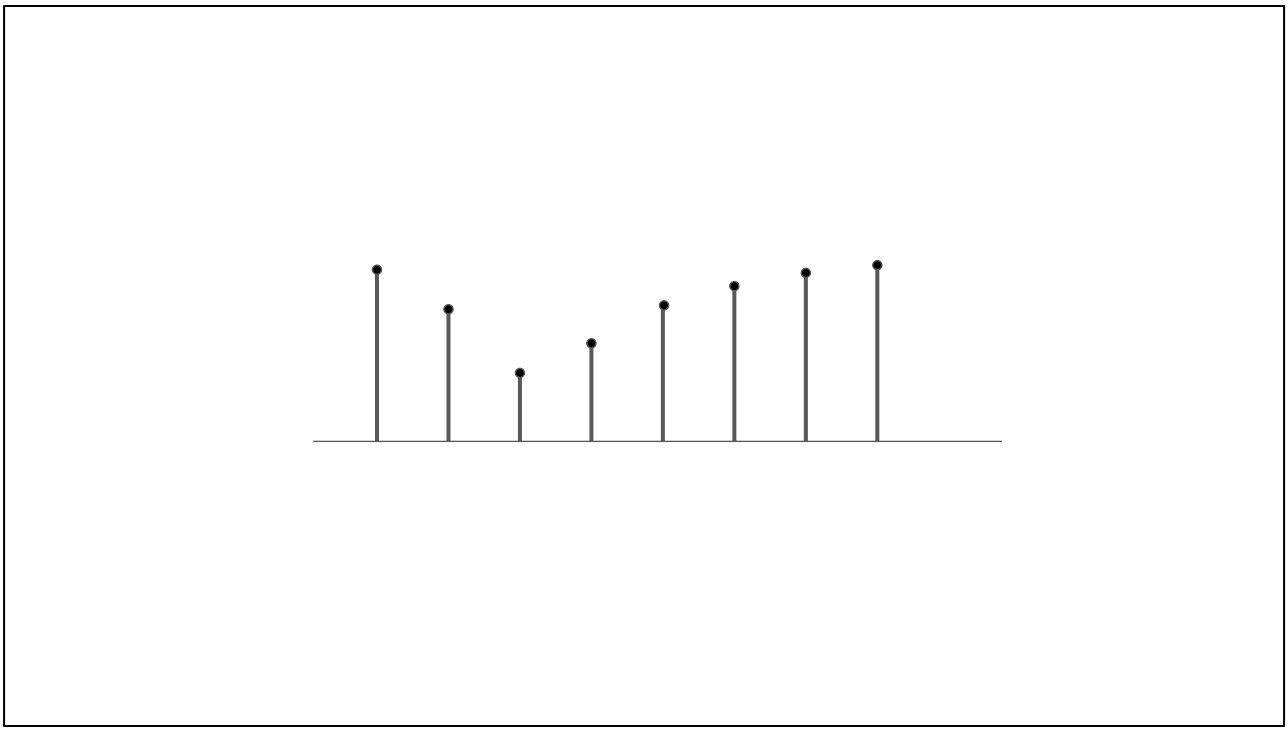
...or scaling down the texture (which scales up the pixel area in texture space), we also get different types of coverage.



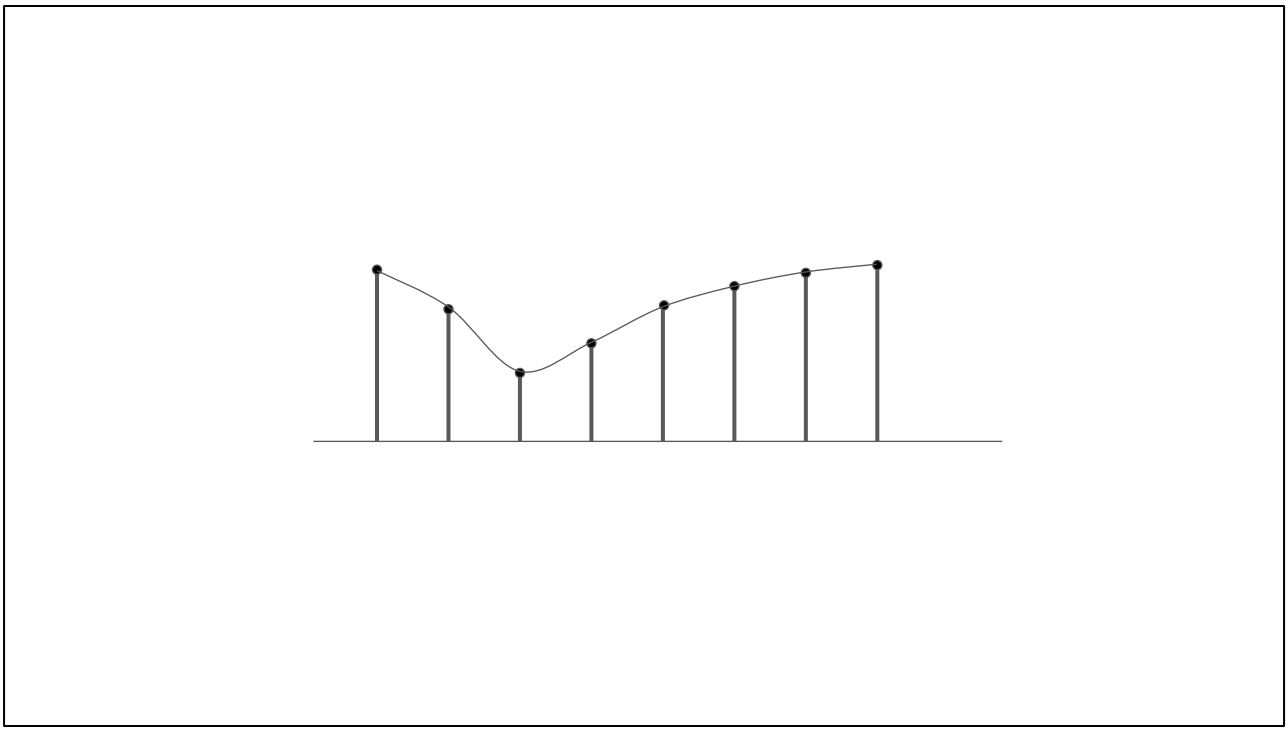
So what are we trying to do? We often think of a texture like this -- blocks of “color” (we’ll just assume the texture is grayscale in these diagrams to keep it simple).



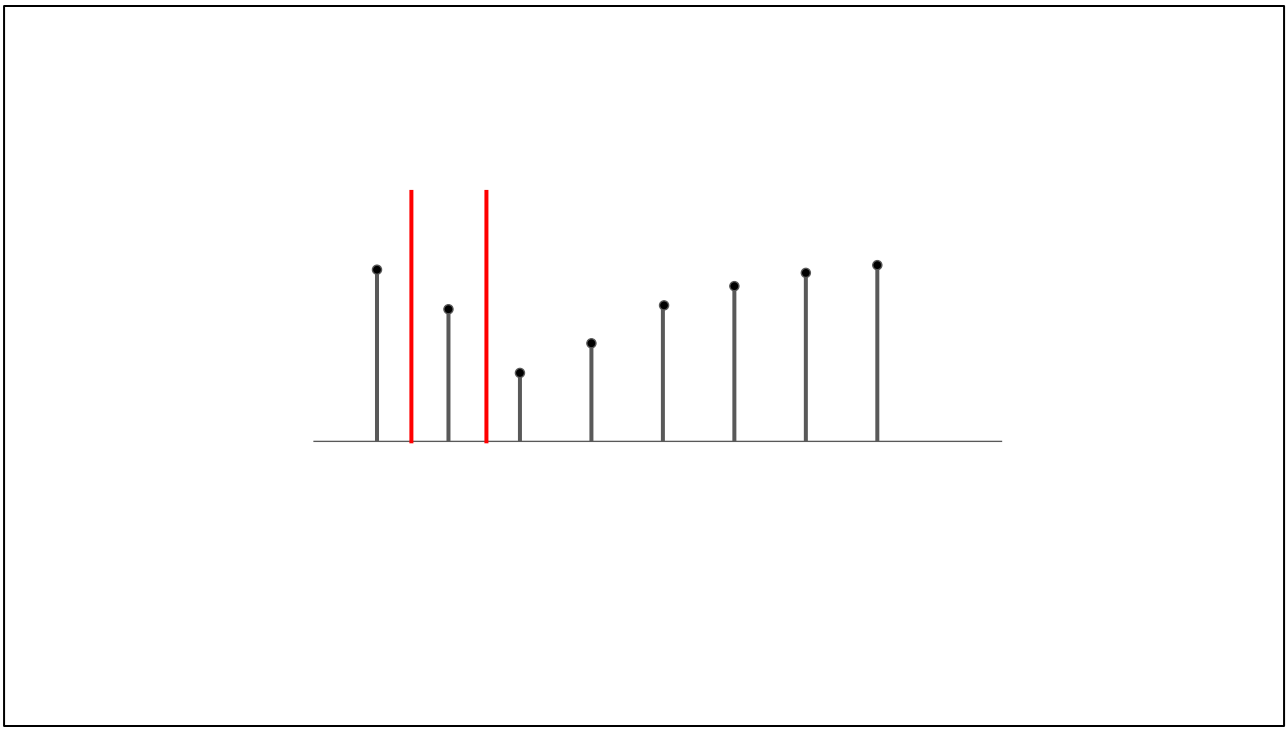
But really, those blocks represent a sample at the texel center.



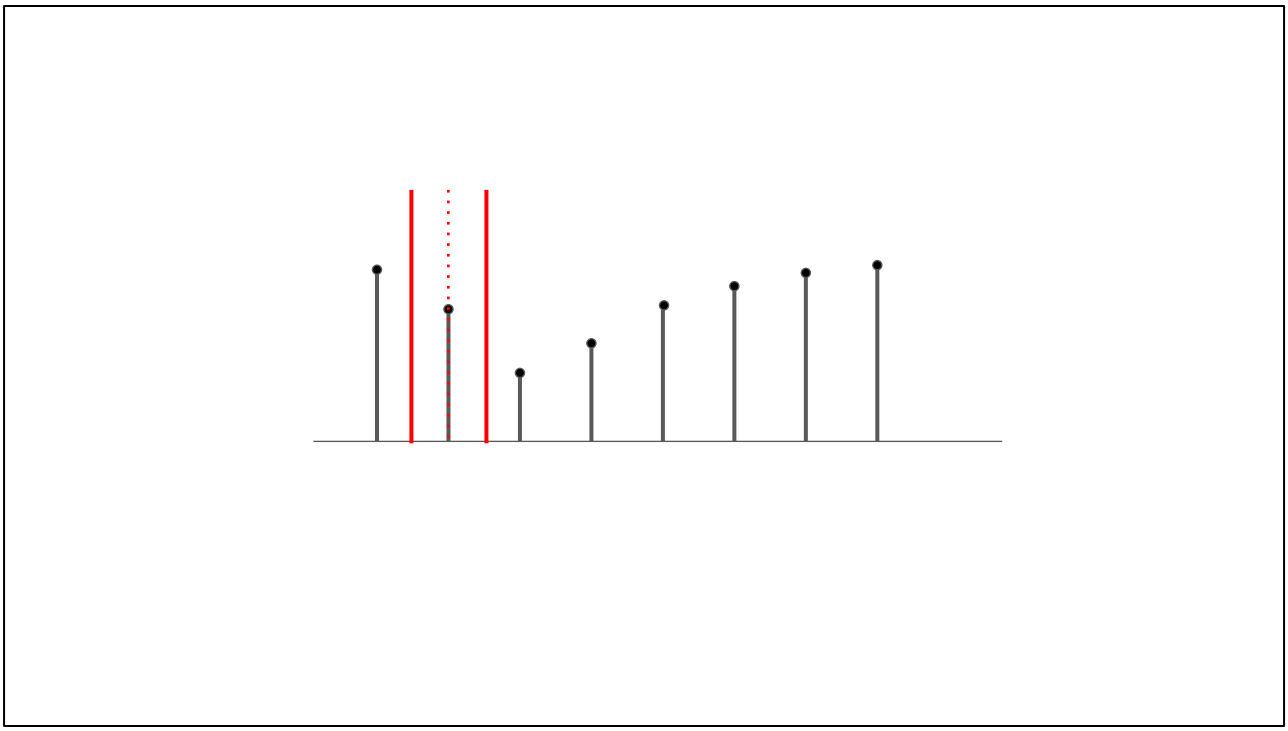
So what we really have is this. Those samples are snapshots of some function, at a uniform distance apart.



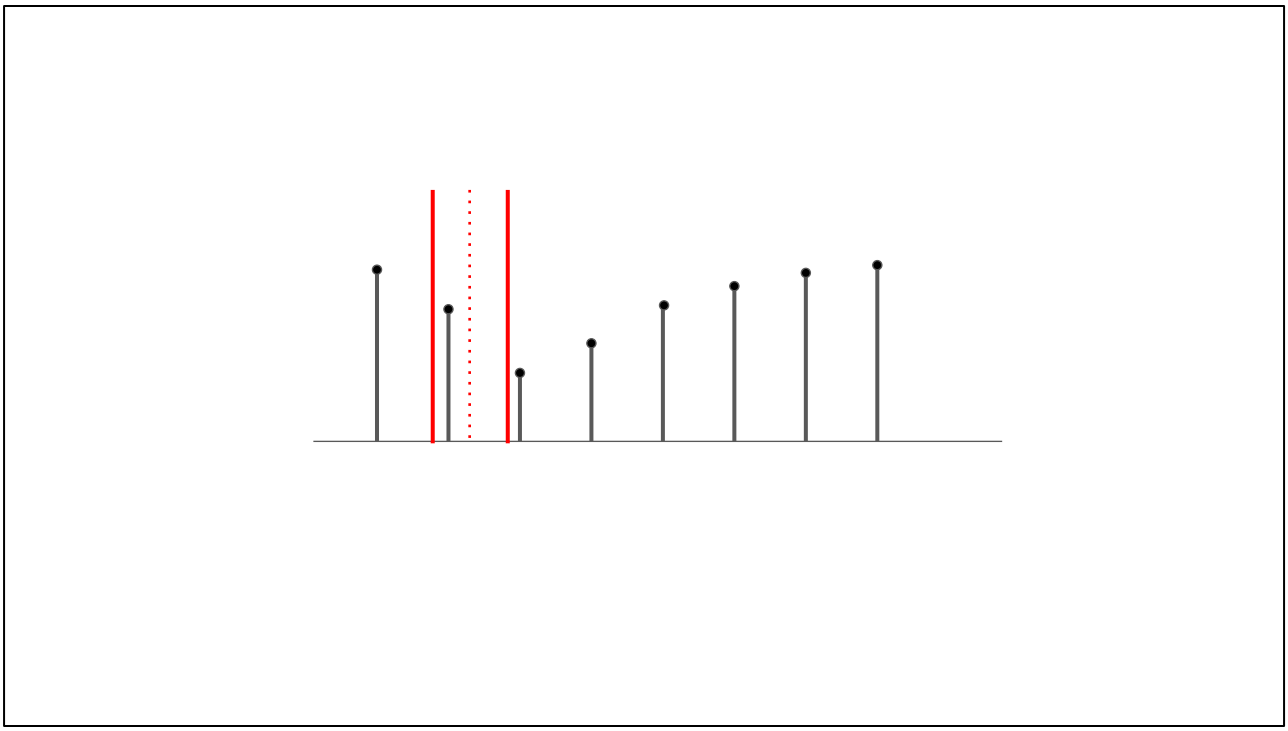
We don’t really know what the function is, but assuming it’s fairly smooth it could look something like this. So our goal is to reconstruct this function from the samples, so we can compute in-between values.



We can draw our pixel (in 1D form) on this graph. Here the red lines represent the boundary of the pixel in texture space.

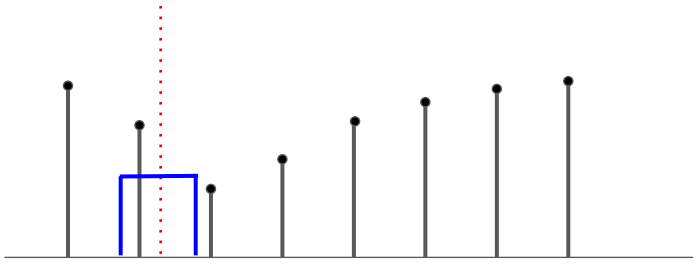


If the pixel is centered on a texture sample, and only includes one sample, then our reconstruction is easy, just return the sample.



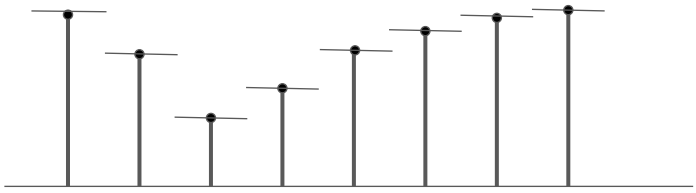
But if we're offset, we need to compute some sort of in-between value. How do we reconstruct this?

Box Filter



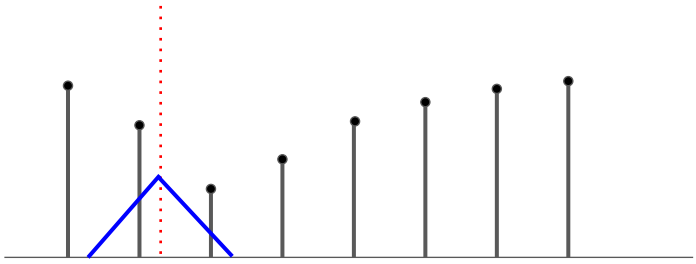
One solution is called a box filter. The box has the same width as the pixel (in texture space). We simply average all the sample values that the box crosses. In this case we always end up with one value so...

Nearest Neighbor



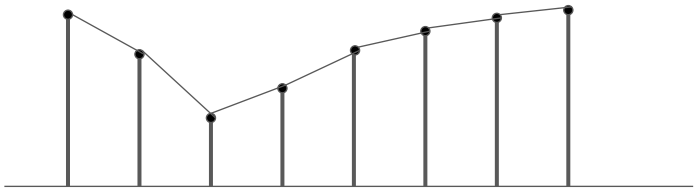
Our resulting function looks like this -- a series of flat areas. This is called nearest neighbor or point sampling. It’s cheap, but ends up with very blocky results, and lots of aliasing artifacts.

Tent Filter

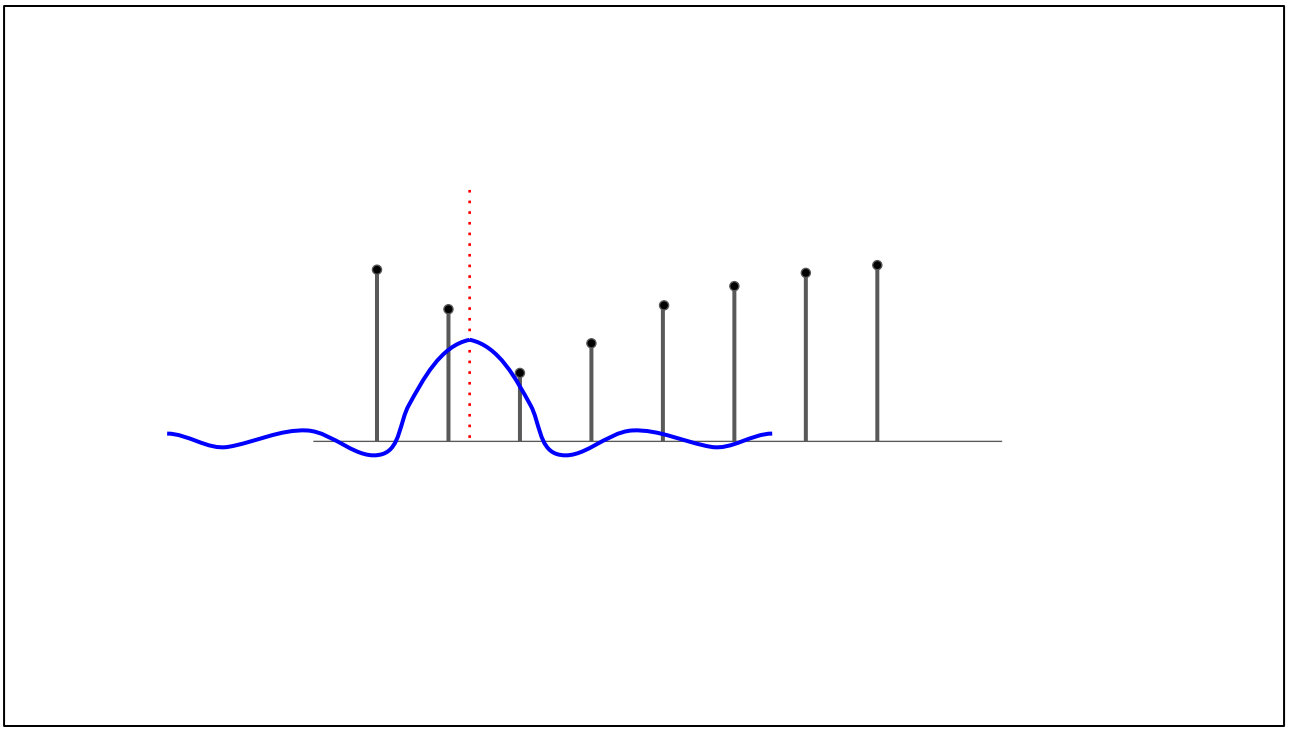


An alternative is a triangular filter called a tent filter. This is twice as wide as the pixel width, in this case with a height of 1 and ramping down to 0. When a sample crosses the filter edge we multiply its value by the filter at that point, and add them all together. So in this case we'd add about 3/4 of one sample and 1/4 of the other.

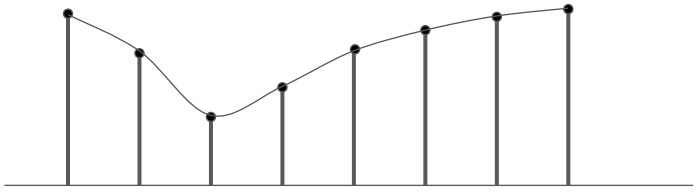
Piecewise Linear



This produces a piecewise linear reconstruction of the function -- each sample is joined by a straight line. This will look much better than nearest neighbor sampling.



The ideal construction uses a sync filter, which uses the function sin(x)/x, scaled so that the first lobe is twice the width of the mapped pixel. There are two problems with this -- it is expensive to compute, and it’s infinite. So, impractical. Usually people will use an approximation to this, such as the Lanczos filter.



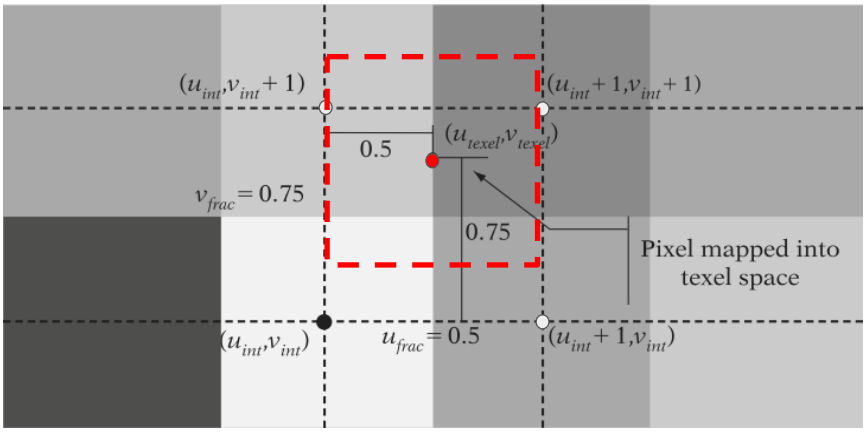
But it will produce an ideal reconstruction.

Bilinear Filtering

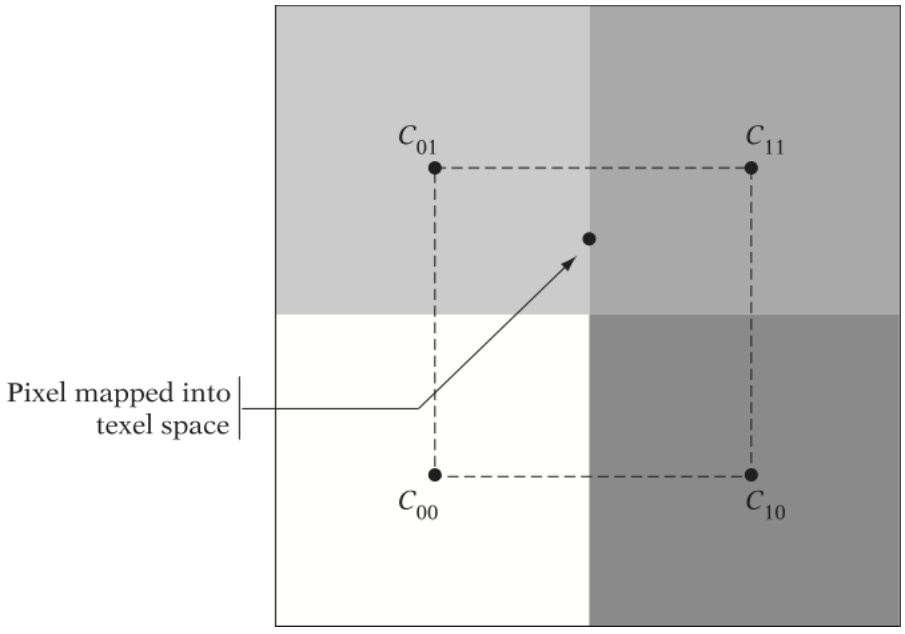
Steps:

- Find texel with min u,v
  - u_int = floor(u-0.5), v_int = floor(v-0.5)
- Find neighbors:
  - (u_int, v_int+1), (u_int+1, v_int), (u_int+1, v_int+1)
- Find fractional value from min u,v
  - u_frac = u - u_int-0.5, v_frac = v - v_int-0.5
- Interpolate twice in u
- Interpolate those results in v

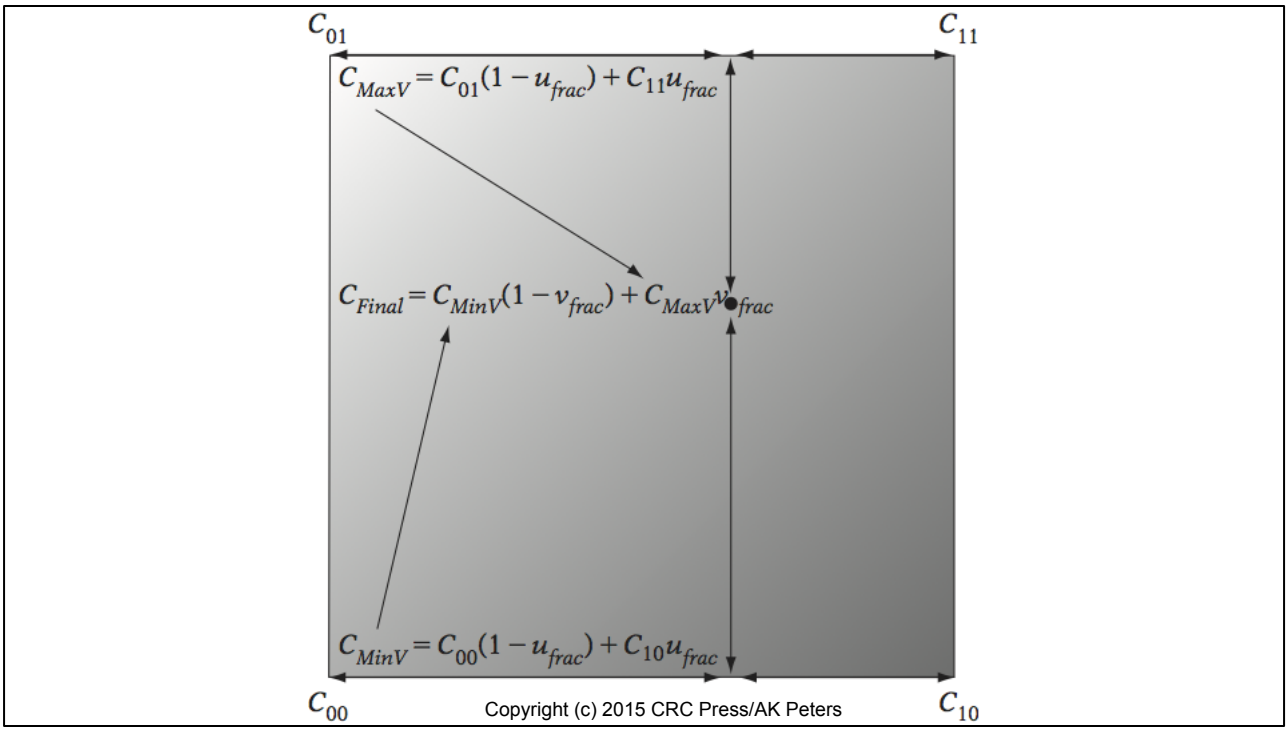
So we'll stick with the tent filter. For a 2D texture, this is called bilinear filtering.



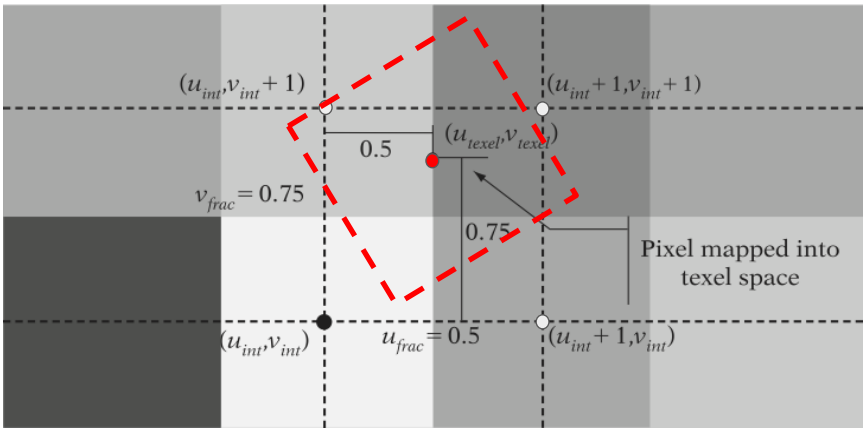
As an example, we can map a pixel onto this texture.



Simplifying this to just the pixel center and the four bounding texels...

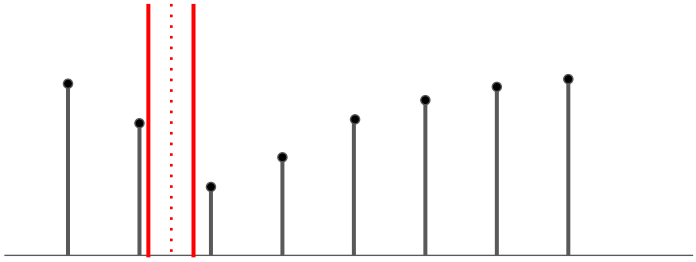


Using the pixel center and the four bounding texel color values, we can compute the bilinear result in three steps.

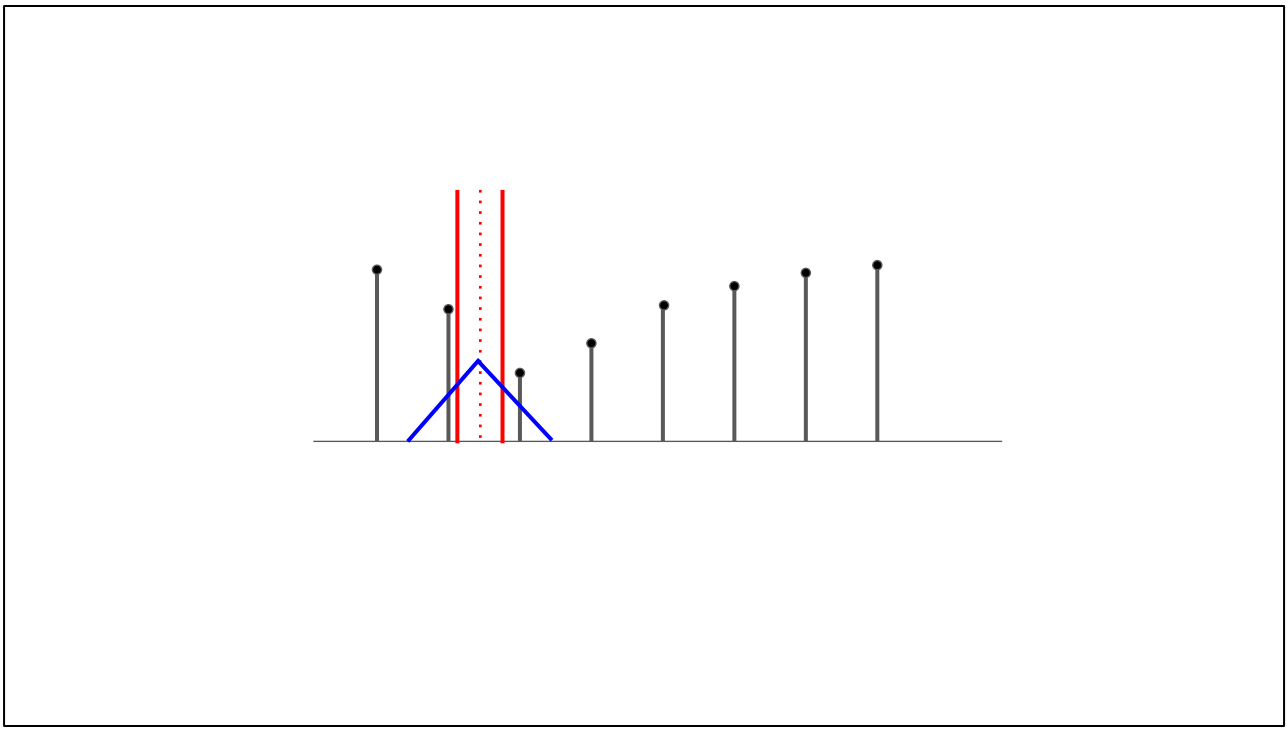


We compute the same value even if the pixel is rotated relative to the texture. All that matters is where the pixel center is.

Magnification

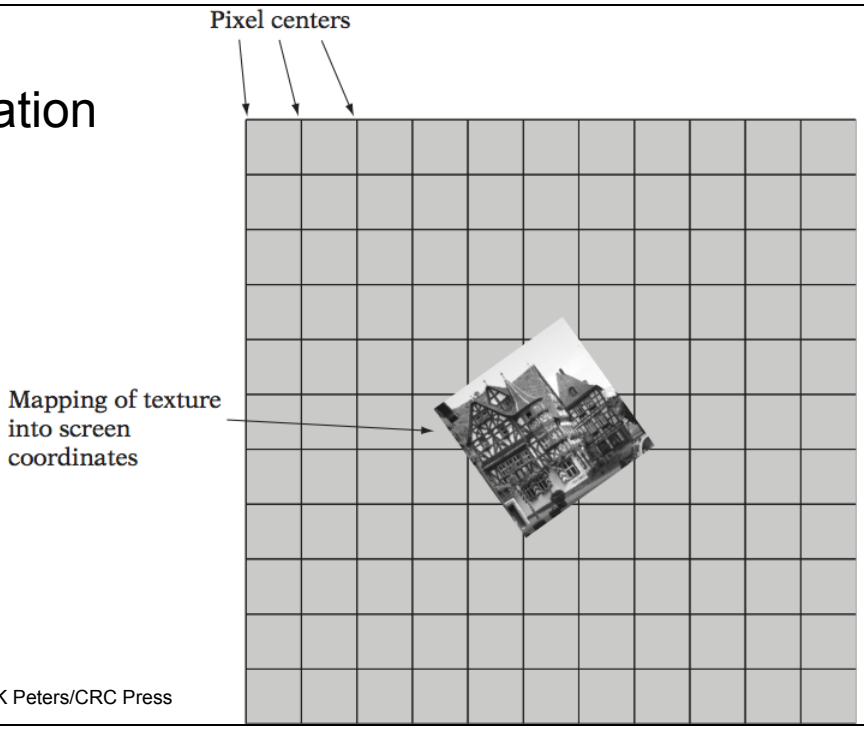


The previous discussion assumed that we were translating or rotating only. Suppose we add scale? If we scale the texture up (zoom in), then the corresponding pixel area in texture space will shrink. If we scale too far, then it's possible the normal tent filter (2x the pixel width) won't hit any texels.

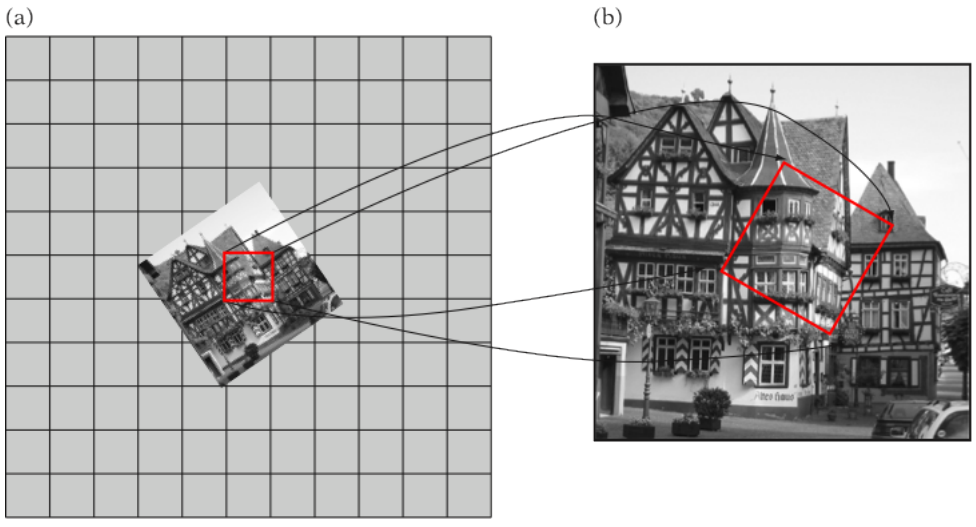


So in practice we just assume a pixel width of 1, and use the normal tent filter.

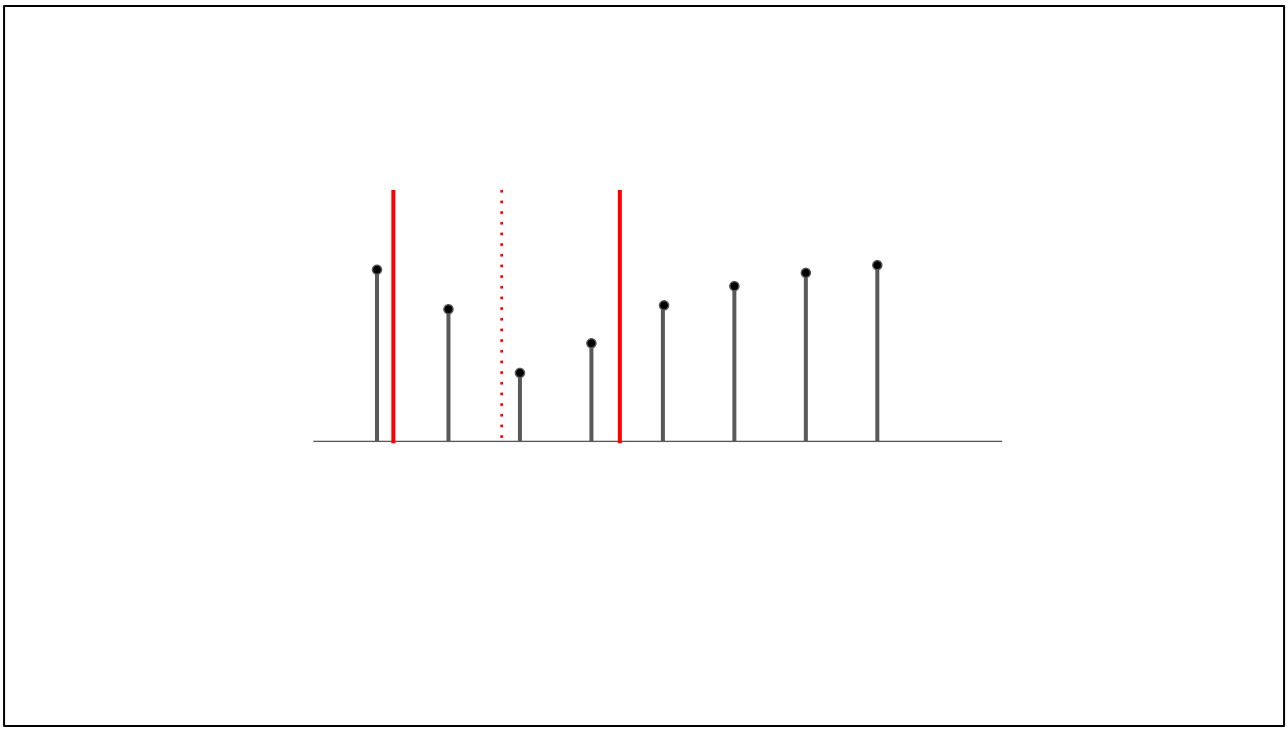
Minification



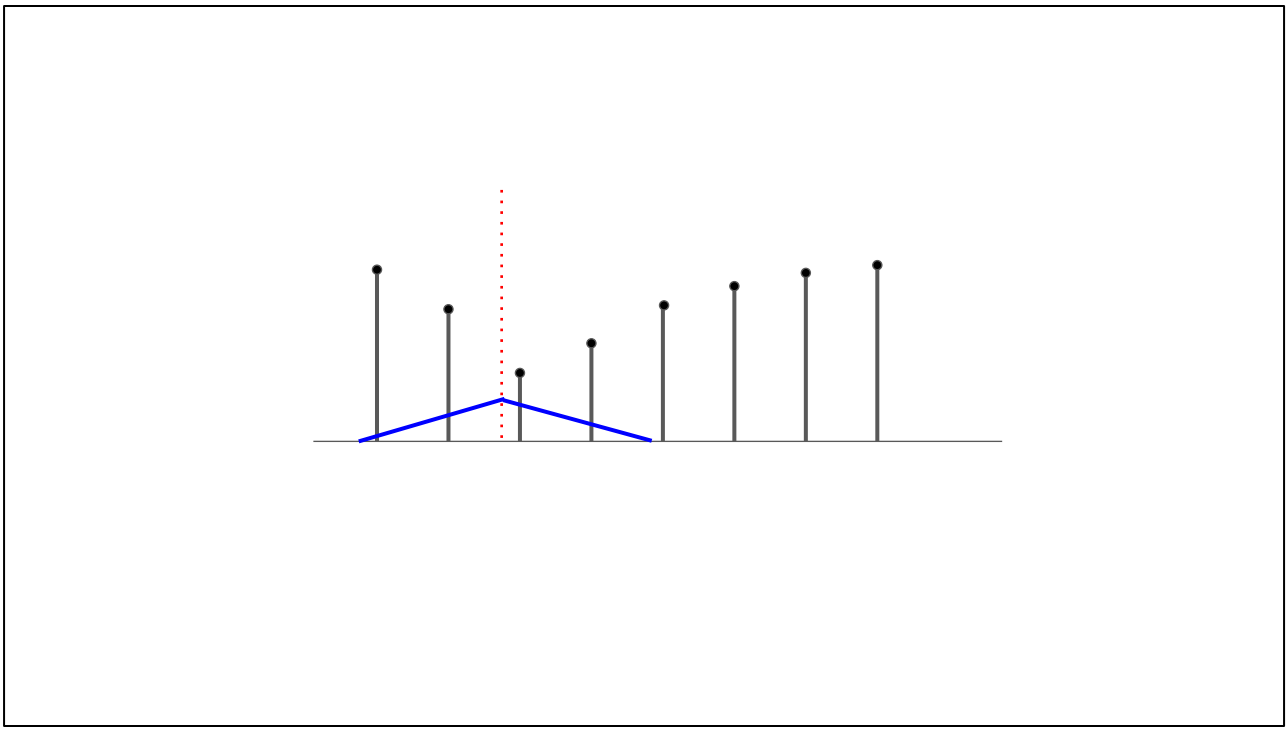
The opposite case is if we're scaling the texture down (zooming out). This shows the texture in pixel space.



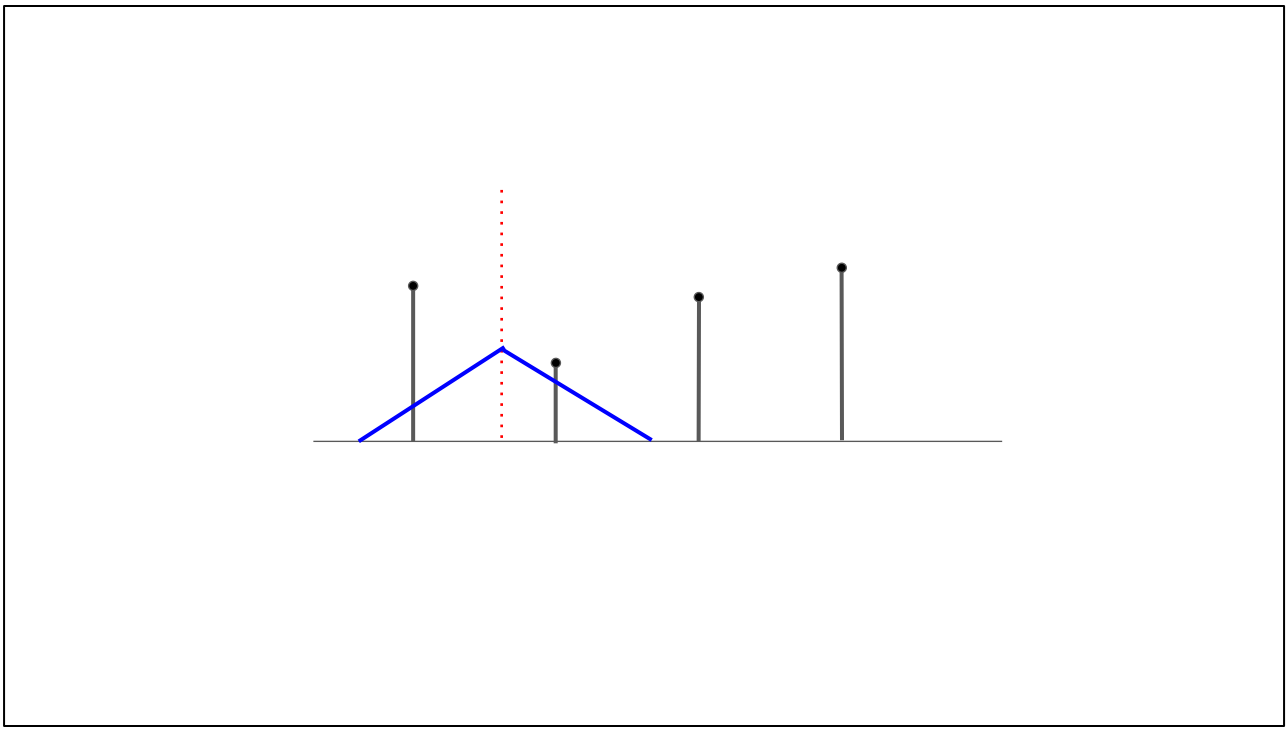
We again can pick a pixel and map it to texture space. We see that the pixel covers many many texels. What do we do in this case?



So here's a simple 1D case, where we've scaled down our texture by 2, so our mapped pixel is twice as big.

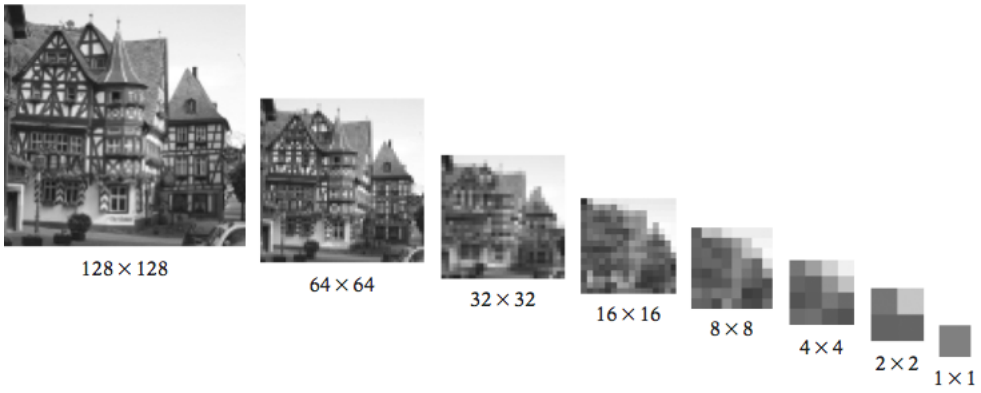


We could do as before, and use a tent filter. Note that the filter is wider (to capture the new pixel width) and shorter (the area under the filter must be 1). This will work, but requires some more computation than simple bilinear filtering.

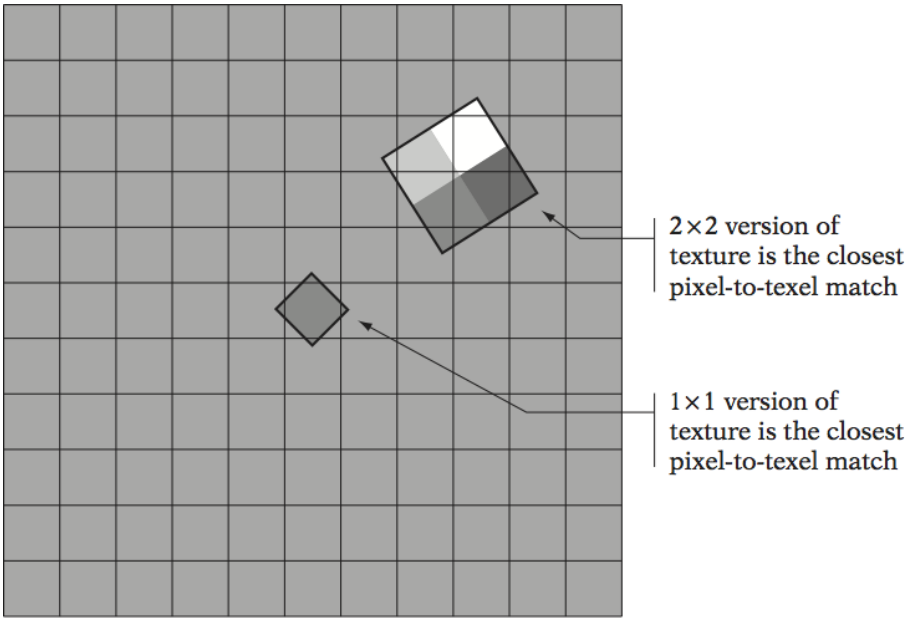


Ideally, it’d better if we had a texture that was half the size, and use our normal bilinear filter. Why not just compute this ahead of time? In fact, why not just compute a whole set of these, down to the smallest size?

Mipmaps



That's the principle behind mipmaps (MIP stands for Multum in Parvo -- "much in little")



Screen-space geometry
(same mipmapped texture applied to both squares)

So now we pick a particular mipmap, or mip level, depending on the area of the texture projected on the screen.

Choosing Mip Level

Try:

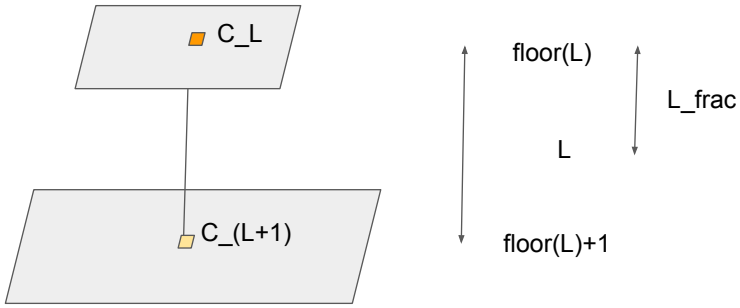
$L = \log_2(1/\text{scale})$

L_int = floor(L) [clamp to be 0 or greater]

0 = largest, mipmap gets smaller as you increase L

If we know our scale (which in a simple renderer, we do) we can compute the mip level this way. In practice, graphics hardware will implicitly calculate the mip level based on differentials computed in the rasterizer. See Essential Math for Games for more details.

Trilinear Filtering



L_frac = L - floor(L)
C_final = (1-L_frac)*C_L + L_frac*C_(L+1)

Even with mipmaps, we can get some aliasing, particularly if our computed mip level lies in the range halfway between. The solution is to compute a bilerped value for both mipmap levels, and then linearly interpolate between them based on fractional mip level.

One last thing

What happens if we scale more in one direction than the other?

(Anisotropic scaling)

- RIPmaps
- Summed Area Tables
- Anisotropic sampling