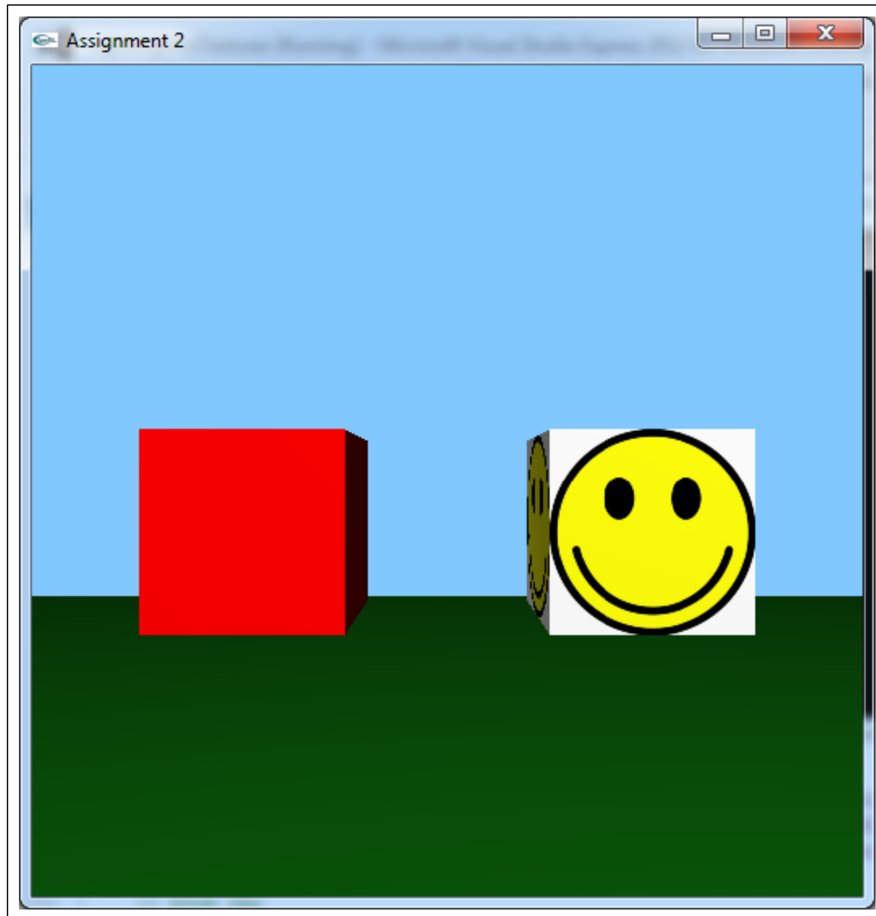
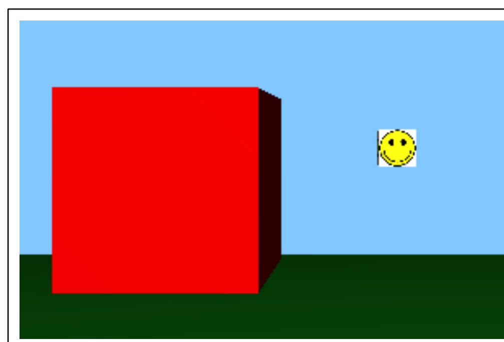
A problem

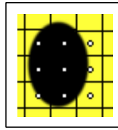
The picture below is taken from an example program I used in an earlier iteration of this course. The program displays a pair of cubes, one of which has a solid color and the other of which has a texture applied to it.



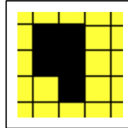
By holding down both the right mouse button and dragging, I can change the z position of a cube. If I do this to move the textured cube far away from the view position, the texture on the cube appears to degrade.



Why does this degradation take place? The picture below illustrates why this is happening.



In the picture we have a grid of pixels overlaid on the image. The center of each pixel is represented as a white dot. In the simplest form of texture application, we simply sample the texture where the dots are and assign the color we sample to the entire pixel. This results in a blocky image as shown here:



Worse yet, this image is not stable. If we adjust the pixel grid by a tiny amount relative to the underlying texture image, the sampled pixels may change in a disconcerting way.

Mip mapping

A standard solution in OpenGL to the problem I pointed out above is to use *mip mapping*. "mip" is an acronym for the Latin *multem in parvo*, which means "many things in a small space."

In this technique we use the original texture image in its full size

but then also precompute a series of ever smaller replicas of the texture using a high quality method to average together blocks of pixels.

With such a collection of precomputed images at our disposal, we can match the size of the image we need to one of the precomputed smaller images. When we go to apply a texture to a region, OpenGL will automatically select the version of the texture that most closely matches the size of the region on the screen where we are applying the texture.

In the example above we are applying the smiley face texture to the faces of a cube. Each face is a polygon in OpenGL, and as we move the cube closer to the viewer or further away, the sizes of these polygons will change. OpenGL will automatically match the image size to the size of the polygon it is being applied to.

To use mip mapping in OpenGL we load a sequence of textures and assign each texture to a level. The level 0 texture is the original texture, level 1 is half the size of level 0, and so on. Most commonly mip mapped textures start with a level 0 texture who side length is a power of 2. In the example shown above, the level 0 texture is 256 by 256 pixels, level 1 is 128 by 128 pixels, and so on.

Here is some code to illustrate how this will work. The loop below loads a sequence of levels into a texture unit.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texHandle);

for(int level = 0;level <= 8;level++) {
  string fileName = ppmFilename;
  fileName += '_';
  fileName += ('0' + (char) level);
  fileName += ".ppm";
  ppmRead(fileName.c_str(), texWidth, texHeight, pixData);
```

```
glTexImage2D(GL_TEXTURE_2D, level, GL_SRGB,
      texWidth, texHeight, 0, GL_RGB,
      GL_UNSIGNED_BYTE, &pixData[0]);
}
```

This code uses a library function ppmRead to load an image from a file and store its data in an array, pixData. We then pass that image data to the glTexImage2D function.

We have to load enough levels to make it all the way down to a 1 pixel by 1 pixel texture. Once all of the textures are loaded, we also have to use a mip map specific value for the texture minification filter (more about this below):

```
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

You can compare the results with the picture at the beginning of these notes. Using a mip mapped texture collection yields noticably better results.

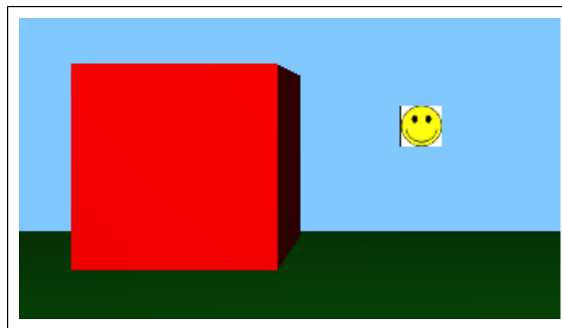
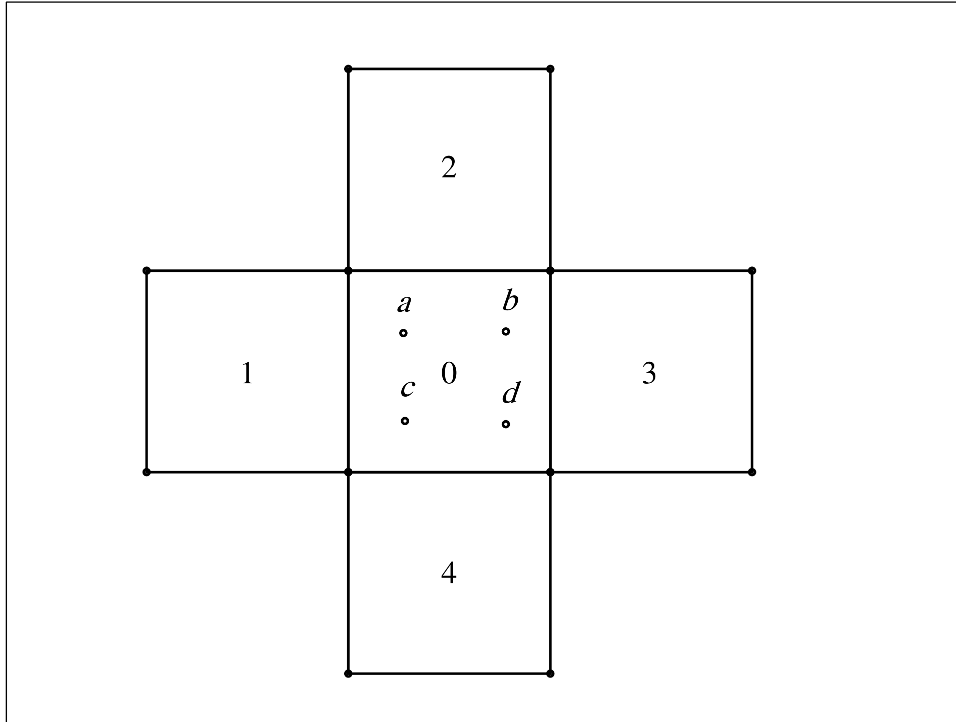


Image scaling

Suppose we have an image and we want to scale up the image to make it have twice its original width and twice its original height.

One obvious option is to replace each pixel with four pixels of the same color. This does not produce very good results.

A better strategy is to make each of the new pixels be a blend of the original pixel and its neighbors. This helps to ensure a smoother transition from one pixel to the next in the scaled image. Here is a diagram to illustrate how this might work.



In the grid pixels 0 through 4 have their original colors c_0 through c_4. We would like to split pixel 0 into four pixels by sampling the color grid at points *a* through *d*. Instead of assigning all four of these samples the original color of pixel 0, we can use the locations of the four samples to make blends of the pixel 0's color with the colors of the neighboring pixels.

More concretely, let us imagine that there is some underlying function $C(x,y)$ that assigns a color to any sample point in the diagram, and that $C(x,y)$ evaluates to an original pixel color when we sample at the center of any of the original pixels. That is, for all pixel numbers k we have that

$$C(x_k , y_k) = c_k$$

Another reasonable assumption to make is that our color function acts as some kind of average over all of the existing colors:

$$C(x , y) = \sum_k T_k(x , y) \; c_k$$

For these two assumptions to be consistent with each other, the blending function for any given pixel has to vanish by the time we reach a neighboring pixel.

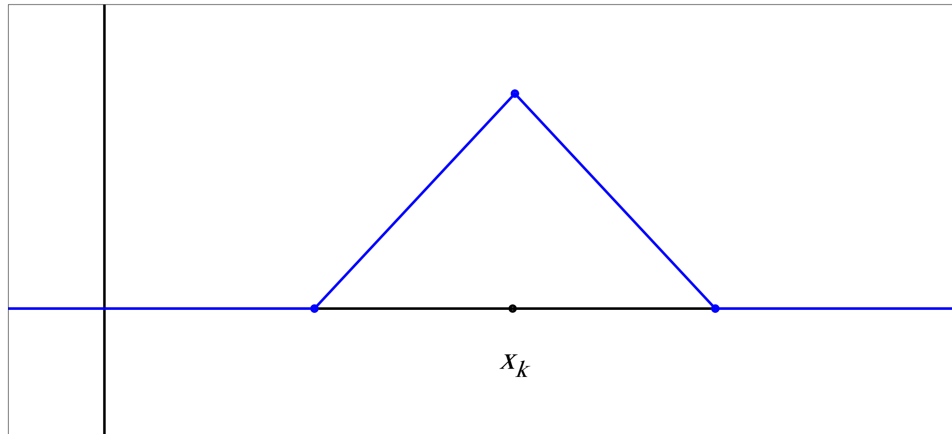
$$T_k(x_n , y_n) = \begin{cases} 1 & n = k \\ 0 & n \neq k \end{cases}$$

The simplest function with these characteristics is a *bilinear tent function*:

$$T_k(x\,,\,y) = Hx_k(x)\;Hy_k(y)$$

where $Hx_k(x)$ and $Hy_k(y)$ are both *linear hat functions*:

$$H_k(x) = \begin{cases} 0 & x \le x_k - 1 \\ x - x_k + 1 & x_k - 1 < x < x_k \\ 1 + x_k - x & x_k \le x < x_k + 1 \\ 0 & x \ge x_k + 1 \end{cases}$$



Bilinear scaling

Using the bilinear tent function to act as the color function for sampling results produces the *bilinear scaling algorithm*. Here is the original image above scaled up via this algorithm.

This image does not have the obvious blocky appearance of the naively scaled version. In particular, the blocky appearance of the string and the specular reflection in the eye are now gone. The trade-off is that portions of the image now look a little more blurry. This is additional blurriness is visible on the peanut.

Scaling and textures

The image scaling problem comes up when we work with textures. In OpenGL applying a texture to a polygon requires us to essentially glue the texture image onto the polygon. In almost every case when we do that, the texture image will have to be scaled to fit onto the polygon.

A more sophisticated way to understand what happens when we work with textures is to say that for each fragment in the polygon we want to apply the texture to we instead have to *sample* the texture image. This is usually done by using a texture lookup function in the fragment shader:

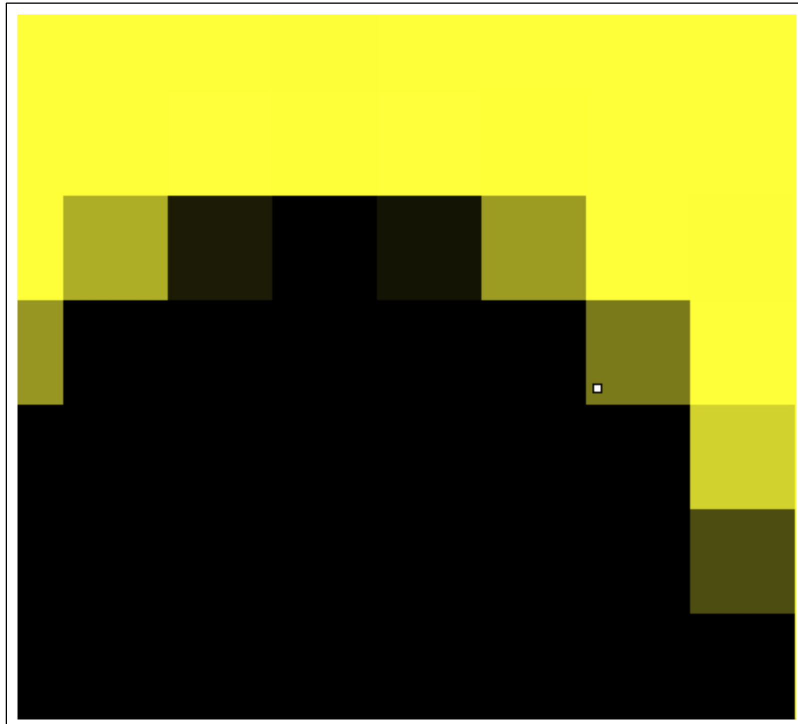
```
#version 430

in vec2 tc;
out vec4 color;

layout (binding=0) uniform sampler2D samp;
```

```
void main(void)
{
  color = texture(samp, tc);
}
```

The texture function here takes two parameters. The first is the sampler that stores the texture and the second is the texture coordinate at which we want to sample the texture. The texture coordinate hardly ever lines up exactly with the center of a pixel in the image we are sampling.



One obvious response is to use the pixel color of the texture pixel whose center is closest to our sample location.

A more sophisticated option is to use the bilinear reconstruction technique we saw above. This performs an intelligent average over the nearby texture pixels.

Magnification and minification in OpenGL

When you load a texture or mip map in OpenGL, you will also need to set texture parameters that tell OpenGL how to scale the texture when scaling is necessary. There are two separate options to set: an option for magnification (called the mag filter) and an option for shrinking (called the min filter).

The two options available for the mag filter are `GL_LINEAR`, which specifies bilinear reconstruction, and `GL_NEAREST`, which simply uses the nearest pixel to the sample point.

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

For the min filter the options depend on whether or not you are using a mip map. If you are not using a mip map, the options for the min filter are the same as the options for the mag filter.

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
```

Mip maps and trilinear filtering

Mip maps can take advantage of a more sophisticated strategy for min filtering, called *trilinear filtering*. In this method, OpenGL first finds a pair of textures to use, one larger than the desired size and one smaller. OpenGL can then do a linear interpolation between the two bounding textures and then sample the interpolated texture using bilinear interpolation. To select trilinear interpolation we use the GL_LINEAR_MIPMAP_LINEAR option:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR_MIPMAP_LINEAR);
```

Another option is GL_NEAREST_MIPMAP_LINEAR, which selects the closest texture level and then does bilinear filtering on that level.

Anisotropy

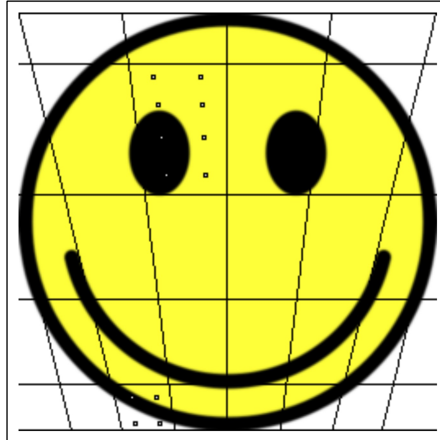
We have one final problem to contend with in applying textures to polygons. Up until this point we have only concerned ourselves with the difficulties of applying textures to polygons that are perpendicular to the view direction. We will of course have to place textures on surfaces that are not perpendicular to the view direction as well. That scenario introduces additional complications.

Here is a picture of a typical situation where we have to apply a texture to a polygon that is not parallel to the view plane.

To apply the texture properly, we have to imagine a grid of pixels superimposed on the image. Although the polygon we are applying the texture to is not parallel to the view plane, the grid of pixels we will superimpose on the polygon is. One important consequence of this is that pixels closer to the front edge of the polygon appear to cover a smaller portion of the polygon, while pixels closer to the back edge of the polygon appear to cover a larger area.

The diagram below illustrates this phenomenon. In the diagram I have superimposed an imaginary grid of pixels on the underlying texture. If we draw the texture in its original, undistorted form, we have to

distort the pixel grid to compensate.



An important consequence of the distortion of the pixel grid is that if we want to do sampling on this pixel grid we will have to sample pixels with greater area more aggressively than pixels that cover a smaller portion of the texture. OpenGL can do this for us automatically, by using a technique called *anisotropic filtering*.