Minification, Magnification, and Mip-Pyramids

Assume that we have a texture of size 256×256 texels and we want to map it onto a square. If the square is roughly the same equivalent size as the texture, then the image will look fine. However, if the quad is much bigger, we will have to expand the image via a process called *magnification*. Conversely, if the quad is too small, we will need to shrink the image via *minification*. We will now discuss the general techniques for both.

Magnification

The two main techniques for enlarging a texture are the following:

- **Nearest Neighbour:** The idea is to use the value of the nearest neighbour to the fragment. This works as you might expect and results in very *pixelated* images, since we're basically repeating pixels.
- **Bilinear Interpolation:** This works as follows: for each pixel, find the 4 neighbouring pixels and linearly interpolate the two dimensions to find the blended value for the pixel. The result is blurrier, and gets rid of a lot of jaggedness.

An alternative (though more expensive) is to use higher-order interpolations, such as *bicubic interpolation* (or *cubic convolution*). This is not supported by hardware, but it can be implemented in the shader.

Now explain how linear interpolation works. Note that this is called `GL_LINEAR` in code, while nearest neighbour is `GL_NEAR`. A solution to blurriness is to use *detail textures*. These are textures that are overlain onto the magnified texture. Another option is to not use bilinear interpolation alone, but to clamp the values past certain thresholds in order to retain a similar effect.

Minification

We have the same approaches as magnification (and the same commands for OpenGL). Note that bilinear interpolation fails when more than 4 texels cover a pixel.

Mip-Pyramids

Suppose we are trying to model a piece of architecture with gold inlays. The texture for this model will have size $16,000 \times 16,000$. The amount of detail in the texture would only be visible when model is close to the camera. Now suppose that this piece of architecture is far away from the camera. Would it still make sense to use such a detailed texture even if we won't be able to appreciate all the details?

The answer to this question is no, but the problem now is that we would need to be able to scale back the texture as it moves further away. For small textures

this can probably be done on-the-fly, but for larger textures the performance cost would be higher. What we can do instead is to cache sequentially smaller versions of the texture and retrieve them as is needed. This concept is known as a *mipmap* (or mip-pyramids). The idea is then the following: take our $16,000 \times 16,000$ image and scale it down so it is now $8,000 \times 8,000$. Now take that image and scale it again, so it is now $4,000 \times 4,000$. Repeat this process until you have obtained an image that is 1×1. The result is a sequence of increasingly smaller textures where each one is half the width and height of the previous step. If we place them all vertically, we would end up with a pyramid of images, where the original size is at the bottom and the smallest at the top.

The example given here is very extreme, but the general principle for the mipmaps is the same regardless of the size of the texture. The downside to this approach is the amount of memory that would be consumed, especially for large textures. There are better techniques, but these are very case-specific, and tend to be custom implemented, while mipmaps are provided by OpenGL directly. It is also worth noting that the options for scaling down the images are the same as those available for minification.