Textures and the Texture Pipeline

Introduction

The texture of a surface is its look and feel. In computer graphics, texturing is a process that takes a surface and modifies its appearance using some image, function, or other data source. Consider the example of a brick wall. In order to render it correctly, one could spend a considerable amount of time manually modelling every single aspect of the wall. This process would not only be incredibly time-consuming, but it also would not scale well if one considers a case where multiple walls need to be used. What we can do instead is define a simple surface (like a quad for instance) and then apply a single image that represents the brick wall. This will work provided that the user does not stare too closely in order to recognize the lack of detail.

That being said, sometimes walls have differing textures throughout. As an example, the mortar that holds the bricks can be more matte, whereas the bricks themselves can be more glossy. To accomplish this, we can apply a second texture that controls the roughness of the brick wall (as opposed to providing different textures for the bricks and mortar). We can further improve this by using a third texture. This time, what we're trying to accomplish is to shift the normals slightly to simulate irregularities along the surface. This texture is known as a *bump map*. Finally, we can apply *parallax mapping* to deform a flat surface in relation to the view (so we can make the bricks stick out higher than the mortar) and *parallax occlusion mapping* (to create the shadows that will surround the bricks). Both these techniques simulate *displacement mapping*, which actually does move the vertices of the surface.

The Texturing Pipeline

To better understand what texturing is, let us take a look at the effects it has on a single pixel. As we have previously seen, we can use shading models to compute the colour of a pixel. These models can take into account lights, materials, transparency, etc. Texturing works by modifying these values. In the brick wall example, the image texture provides the colour of the surface. The pixels in the image are called *texels* to avoid confusion with the pixels that we render to. The roughness texture then modifies the roughness value, and the bump map modifies the normals themselves to produce a different result in the shading equation.

Texturing can be described by a generalized texture pipeline, which can be divided into the following stages:

- Projector function
- Corresponder function
- Sampler
- Value Transform Function

Similar to the graphics pipeline, each stage in the pipeline receives a value from the previous stage, and produces a value that will be consumed by the next stage. To better understand this, let us look at an example.

We begin with a point p on the surface we wish to apply the texture to. The point p can be in world space, but it is usually set in model space so that the texture can move along with the model and is therefore in $\mathbb{R}^3$. The projector function takes p and projects this point onto the surface, converting p into a *texture coordinate* p_t that is now in $\mathbb{R}^2$. This process is also called *mapping* (hence *texture mapping*).

Before we can use this coordinate to access the texture, we must apply a *correspondence function* to transform the texture coordinates into *texture space*. There are two major distinctions: the first one depends on whether we are using OpenGL, DirectX or some other system, while the second one relates to how we sample values beyond the edge of the texture. In the case of OpenGL, it is expected that all texture coordinates be in the range $[0, 1]$ for both width and height, with $(0, 0)$ located on the lower-left corner of the image. Therefore, part of the corresponder function will be to divide the values by the width and height. We will discuss the second part later on. The correspondence function therefore will take p_t and convert it into c_t, which is still in $\mathbb{R}^3$ but now exists in texture space.

The *sampler* now takes c_t and returns the texel at that location in the image, which we will call t. This value can now be modified again by the *value transform function* if needed. Once this is done, the texel can be used to modify some property of the surface.

The main reason for the complexity of the pipeline is that each stage offers the user some form of control, and it should be noted that not all elements of the pipeline need to active at all times. To fully understand, lets go back to our brick wall. Suppose we have a position on the wall (which is in 3D space). First we project that point onto the wall itself, which gives us texture coordinates (uv-space). From here, we convert these coordinates into the image space by multiplying each axes by the width and height, respectively and dropping decimal values. We can now extract the texel value. Assuming that it is in sRGB, we must then transform that value into linear RGB to obtain the final colour we will use for shading.

The Projector Function

The first step is to take the surface location and then project it into texture coordinate space (usually 2D u, v space). Modeling systems typically allow artists to define u, v coordinates per-vertex. These coordinates can be initially created by a projector function or mesh unwrapping algorithms. Artists can then edit these coordinates as needed. A projector function is typically a function that maps 3D space into 2D space. Common functions include spherical, cylindrical, and planar projections.

While these are the most common functions, they are by no means the only types of functions that can be used. An example are parametric surfaces (which themselves have u, v coordinates), creating coordinates from view directions, surface temperature, etc. While most of this is done at the modelling stage, certain effects and animations require texture coordinates to be generated in the vertex or fragment shaders (such as *environment mapping*).

Each type of projection has its regions where distortions occur. Spherical projections usually have problems around the pole areas. Cylindrical has distortion when surfaces are near-perpendicular to the cylinder's axis. Planar projections on the other hand have distortions when surfaces are edge-on.

Texture coordinates don't always have to be 2D. 3D textures such as cube maps can be used to help simulate an environment around a scene (think of the background in a video game), while 1D textures can be used to colour terrain models. Textures can also be used to store data for other computations. As an example, we can simulate a simple cloth using textures in real time.

The Corresponder Function

These convert the texture coordinates into texture-space locations. An example would be converting the texture coordinates to only use a particular portion of an existing texture. Another would be to transform the texture itself (by scaling, moving, rotating, shearing, or projecting). Another type explains how we sample the texture beyond the normal u, v range of $[0, 1]$. Some common corresponder functions are:

- **Repeat:** the image repeats itself across the surface.
- **Mirror:** mirrors the image over each axis.
- **Clamp to Edge:** repeats the final pixel on that axis.
- **Clamp to Border:** texture coordinates outside the normal range are rendered with a separate colour.

These functions provide an inexpensive way of adding more visual detail. However it is very easy for users to pick out the pattern of repetition. A common solution is to combine the texture values with another, non-tiled texture (such as terrain). Another option is to use shaders to implement specialized corresponder functions that randomly recombine texture patterns or tiles.

Texture Values

In the typical case, the returned value from a texture is a triplet of values (or 4 in the case of RGBA). It is worth noting that this isn't always the case. Procedural texturing takes the coordinates provided and then evaluates a function to return the final value. These values can then be further transformed as needed.