A Crash Course on Texturing

Li-Yi Wei

Microsoft Research

not textured

textured

Figure 1: *The effect of texturing.*

Abstract

Texturing is a fundamental technique in Computer Graphics, allowing us to represent surface properties without modeling geometric or material details. In this course, we describe the basic ideas of texturing and how we could apply textures to objects in real applications. We concentrate on the three main components of texturing: texture acquisition, texture mapping, and texture sampling. For each component, we present the fundamental algorithms as well recent innovations. We also describe implementation issues and additional application of texturing on graphics hardware.

Keywords: Texture Mapping, Texture Synthesis, Texture Sampling and Filtering

1 Introduction

Modeling surface details is one of the most important tasks for rendering realistic images. One way to achieve this is to meticulously represent the detailed micro-geometry and BRDF (bidirectional-reflection-distribution-function) over a dense sampling of the surface. This can certainly be done, but this approach has two major problems. The first problem is that it can be tedious and labor intensive to acquire the detailed geometric and BRDF data, either by manual drawing or by measuring real materials. The second problem is that, such a detailed model, even if it can be acquired, may take significant time, memory, and computation resources to render.

Fortunately, for Computer Graphics applications, we seldom need to do this. If we are simulating a physical for a more serious purpose of say, designing an aircraft, an artificial heart, or a nuke head, then it is crucial that we get everything right without omitting any detail. However, for graphics, all we need is something that looks right, for both appearance and motion. As research in psychology and psychophysics shows, the human visual system likes to take shortcuts in processing information (and it is probably why it appears to be so fast, even for slow-witted people), and this allows us to take shortcuts in image rendering as well. To avoid annoying some conservative computer scientists (which can jeopardize my academic career), I will use the term *approximation* instead of *shortcut* hereafter.

Texturing is an approximation for detailed surface geometry and material properties. For example, assume you want to model and render a brick wall. One method is to model the detailed geometry and material properties of all the individual bricks and mortar layers, and send this information for rendering, either via ray tracer or graphics hardware. However, in a typical application such as games, few people would care much about the details of the brick wall (unless you know it contains a hidden door to a treasury room), and would normally view the wall from a distance. In this situation, we probably dont need to bother with all the details of the brick wall. Instead, we can simply model the wall as a big flat polygon, and paste onto the polygon a brick wall image so that the polygon looks like a real brick wall. This image, acting as an *approximation* to the real brick wall, is called *texture* in graphics, and the process of applying the texture to an object surface is called *texturing*.

Texturing resolves the two problems for modeling and rendering surface details, as described earlier. First, by representing the surface as a texture image, you dont have to painfully model all the geometric and material details. This saves users time and resources, so that you can spend more time doing more useful stuff, like reading this paper. Second, by rendering a rough polygonal model (e.g. a single square polygon for a brick wall) and a texture instead of a detailed model with different BRDFs, the rendering can be done much more efficiently, either via ray tracer or polygonal rasterizer. This saves computers time and resources, so you can add other fancy rendering effects within the same timing and resource budget.

The usual practice of texturing can be roughly divided into three major components. First, before you can do anything, you need to **acquire** the texture image. This can be done by manual drawing or photographing, but these approaches have their limitations. Later, we will discuss algorithmic methods that overcome these limitations. Second, given the texture image and a 3D model, you need to figure out how to **map** the texture onto the model. This is actually a very difficult problem. Think, for example, of wrapping a gift paper around a basketball (or a baseball autographed by Barry Bonds, if you are a San Francisco Giants fan). Essentially, the gift paper is a texture and the ball is the 3D object, and our goal is to wrap the paper around the ball as nicely as possible, so that we dont see too much crumpling and folding of the paper, and we would definitely want the ball to be entirely wrapped up. Third, after you have de-

cided how to map the texture onto the object, you will have to carefully **sample** the texture in the rendering process, otherwise, you may see some undesirable artifacts rooted in the signal processing theory.

In the rest of this paper, we will describe in detail how to perform each of these three operations. We will describe the basic ideas and algorithms first, followed by more recent innovations. We will then describe various applications of texturing, beyond the original intention we just described. In fact, texturing is one of the most fun field to play with in Computer Graphics, and you can potentially achieve unexpected effects if you are creative enough.

2 Texture Generation

The first task for texturing is to acquire the texture image. Usually, you will have at least a rough idea in mind on what kind of texture you need, such as a marble table or a wooden chair. Our goal is to generate a texture so that it has the desired visual appearance and properties of the texture you have in mind.

There are a variety of methods to generate the texture image. If you are an artist, you could simply draw a texture by hand. However, manual art is usually limited to artificial textures, as it is very difficult to manually draw a texture with realistic appearance. Besides, not everybody is an artist.

Another option is to photograph the material you would like to texture. For example, to acquire a texture of orange skin, you can simply photograph an orange. This approach is very easy to deploy, and you can easily obtain photorealistic textures. However, this approach has some limitations. To acquire a high quality texture, usually we would like to avoid lighting or curvature bias in the acquired texture image, because the texture can be applied to an object with different shape from the original object, and rendered under a different lighting condition. The lighting problem can be resolved by photographing under a carefully controlled studio lighting to avoid bias. The curvature problem is more serious, however, as it is usually infeasible to flatten a real object, such as an orange.

Texture synthesis is designed as a possible solution for the curvature problem. First, you obtain a texture sample. The sample can be small, and this would allow you to photograph a small region of the original object so that the region is more or less flat. This small sample is usually useless for your texturing application, as you might need a larger texture to adequately cover the entire target object. Fortunately, from the small texture sample, texture synthesis would produce an arbitrarily large result automatically for you, and this allows you to apply the result to your object with minimum visual artifacts.

The goal of texture synthesis can be defined as follows. Given a sample image, texture synthesis would produce an output texture that looks like the input. This can be achieved by making assumptions about the statistical properties of the texture images, and different assumptions will yield algorithms with different quality and computation speed. We will cover some of these algorithms later.

The major limitation of all texture synthesis algorithms is that they can only operate on more or less homogeneous patterns. In fact, the term *texture* in image processing and computer vision literature refers to images with regularly or stochastically repeating patterns. In contrast, the term *texture* in computer graphics usually means any image that you applied for texturing objects. The specific meaning of the term can usually be disambiguated from the context, but in case of possible confusions, we will refer to the former as *narrow* while the later as *broad*, definition of textures.

For *narrow* sense of textures, they can be generated by texture synthesis algorithms. This will be the main focus for the rest of this section, as we will describe two major flavors of texture synthesis algorithms: procedural and example-based. For *broad* sense of textures, since they can be arbitrary images, there is no single algorithmic way to incorporate all of them. They are usually either rendered manually by hand or automatically by render-to-texture on a GPU, or can be assembled from photographs by any artistic tools such as photomontage. We will not describe all the possibilities, as this itself is a separate art project.

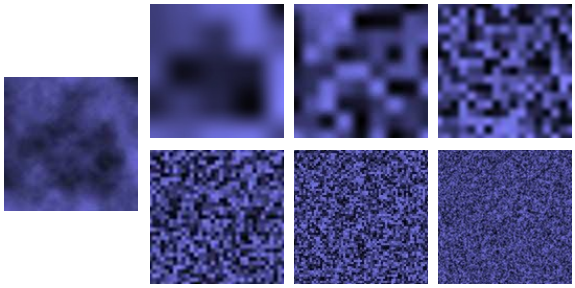


Figure 2: *2D Perlin noise example. Left: the Perlin noise image. Right: the individual noise bands, from low to high frequencies. Image courtesy of [Elias 2003].*

2.1 Procedural Synthesis

One method to synthesize textures is to write special procedures that simulate either the physical formation process or simply the appearance of the material. For certain patterns such as marble or wood, they can be emulated by very simple function called Perlin noise. For more complicated patterns, they could be simulated by fancier procedural process. For example, some animal skin patterns or structures can be simulated by a chemical process called reaction diffusion. The rusting of metals and the formation of flow lines can also be simulated by detailed physical modeling.

The advantage of these procedural texture codes is that they can be very compact, as the only storage requirement is the procedure itself and associated parameters. Another advantage is that by changing the parameters of these procedures, you can easily change the appearance of the resulting textures, providing excellent controllability.

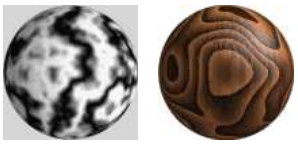
However, procedural synthesis can only be applied for a specific class of textures that you know how to simulate procedurally. For a texture that has no known procedural code, we will not be able to synthesize it procedurally. Even for textures with known procedures, it can still be tricky to choose the proper parameters, as the mapping from the parameters to the final texture appearance might not be straightforward or intuitive.

Nevertheless, procedural texturing has enjoyed great success in the film rendering community, as many feature animations heavily utilize procedural textures. Many artists prefer procedural synthesis over other approaches due to its controllability. Devising new procedural code for simulating textures can also be a fun and challenging activity.

To give you a flavor of what procedural texturing is all about, we will provide a high level overview of Perlin Noise [Perlin 2002], which is arguably the most popular form of procedural synthesis. For more information about procedural texturing, we recommend [Ebert et al. 1998], which provides an excellent introduction as well comprehensive literature survey.

2.1.1 Perlin Noise

Since its inception in 1985 [Perlin 1985], Perlin Noise has been widely adopted as the standard for procedural texturing, especially in the digital film industry.



marble wood

Figure 3: *Example textures generated from Perlin noise.*

The basic idea of Perlin noise is surprisingly simple and elegant. Before introducing Perlin noise, we will first describe what a white noise is. A white noise is a signal that has uniform energy across all frequency bands; i.e. the Fourier transform of a white noise will show a rough flat spectrum. You can see a white noise when you tune your TV to a non-broadcasting channel. A white noise can be simulated on a computer via a uniform random function. For example, to generate a 2D white noise with amplitude in the range $[0, 1]$, all you need to do is to fill in the noise with pixel values drawn from a uniform random function in the range $[0, 1]$.

Unlike a white noise, a Perlin noise is a band-limited signal. It can be constructed as a summation of white noises at different frequency bands, as shown in the following equation:

$$perlin = \sum_{i=0}^{n-1} interpolate(white_i) \times p^i \qquad (1)$$

where n is the total number of band, p is the persistence, and i is the band number, with $i = 0$ being the lowest frequency band.

It is actually technically incorrect to talk about white noise at different frequency bands, but we are talking about how to compute the noise procedurally, not the rigorous mathematical meaning. A white noise at a specific band is simply a white noise with specific image size. In the equation above, white noise at band i has size 2^i. Because the bands have different sizes, we need to properly interpolate them to the final noise size before taking the summation. The summation is taken over all the frequency bands, with each band weighted by the power of a quantity called *persistence*. Persistence is a user-specified parameter, which simply controls the relative weight of the frequency bands. Usually, persistence is within $[0, 1]$, so that the weight decreases with the increasing of band frequency. A visual example of 2D Perlin noise and the constituting bands can be found in Figure 2.

Based on Perlin noise, a variety of textures can be synthesized by proper procedures. Some examples are shown in Figure 3. Their synthesis formulas are as follows:

$$marble = cosine(x + perlin(x, y, z)) \qquad (2)$$

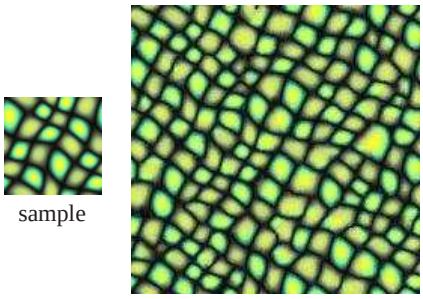
$$g = perlin(x, y, z) * scale$$

$$wood = g - int(g) \qquad (3)$$

Exercise Implement the Perlin noise as described above. Try to generate the marble and wood textures by finding out the proper parameters, such as persistence, scale, and colors. This will give you a feeling of how it is like to tune the procedures in order to generate the desired textures. Hint: try to use a low persistence value for wood, and normal persistence value in $[0, 0.5]$ for marble. If you find it difficult to tune procedural textures, you are not alone. In fact, working with procedural textures is such a demanding task that there is a unique profession created in the game developer community for these people whose job is to write procedural textures, they are often called *texture artists* or *texture designers*.

In the next part, we will describe alternative methods to generate textures that do not require artistic skills.

2.2 Example-based Synthesis



sample result

Figure 4: *Texture synthesis from example. The original sample is shown on the left, while the synthesis result is shown on the right.*

An alternative approach is to synthesize a new texture from a given example. This is certainly more user friendly, because to use the algorithm, all you need to do is to provide an image sample, rather than writing a procedural shader followed by tuning parameters. However, now the key issue shifts to the design of a synthesis algorithm that can synthesize a good result for any given input sample. This is certainly not an easy task, as much research has been devoted to texture analysis and synthesis in both the computer vision and graphics community.

Instead of describing all these previous work, which is definitely beyond the scope of this paper, we will simply describe some recent algorithms in the graphics community. Specifically, we will introduce methods that focus on synthesis quality rather than analytical modeling, which is an interesting topic for recognition or segmentation, but irrelevant to our goal in this paper.

To develop a successful synthesis algorithm, we need to make some assumptions about the properties of textures. A common assumption is Markov Random Field, which states that textures are both local and stationary. A texture is *local* in the sense that each texture pixel only correlates with pixels in a small local neighborhood. A texture is *stationary* because the statistical property is the same for all image pixels. These local and stationary assumptions certainly do not hold for general images. For example, for an image containing a human face, pixels on the left eye correlate with those on the right eye, even though they are far apart. As a result, this image is not local. This human face is also not stationary because the statistical property varies depending on the pixel location at the human face, such as hair, eyes, mouth, or skin.

2.2.1 Pixel-based synthesis

Based on these assumptions, several algorithms exist for synthesis. One possibility is to synthesize a new texture pixel by pixel, where the value of each new pixel is determined by the local neighborhood. Starting from a seed point, [Efros and Leung 1999] generates new pixels outward in a spiral fashion. To avoid sampling not yet synthesized pixels, the authors adopt a variable neighborhood, incorporating only the already synthesized pixels. The value of each output pixel is determined by choosing the input pixel with a similar neighborhood. This approach produces surprisingly good results, and is conceptually simple and elegant. However, due to its use of exhaustive search of input neighborhoods, the technique is quite slow.

[Wei and Levoy 2000] proposed a similar algorithm, but using a fixed neighborhood. This would incorporate garbage pixels during the initial phase of synthesis, but the effect quickly fade out as more

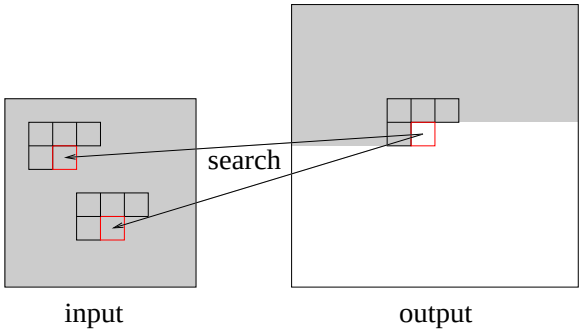


Figure 5: *Pixel-based texture synthesis. The gray region in the output indicates already synthesized portion.*

pixels are synthesized. One good thing about using fixed neighborhood is that it allows acceleration via various techniques such as tree-structured vector quantization. The authors further improved quality and speed via a multi-resolution approach.

These pixel-based algorithms work well for more stochastic patterns, but fail for textures containing regular or large scale patterns, which cannot be preserved by synthesizing individual pixels.

2.2.2 Patch-based synthesis

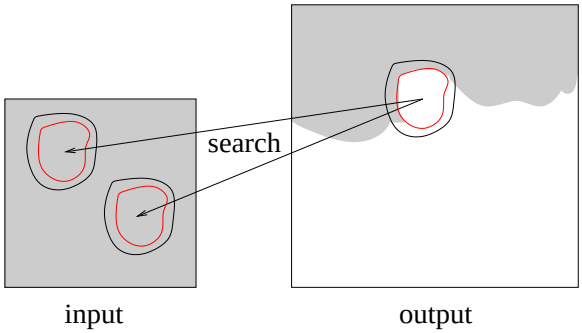


Figure 6: *Patch-based texture synthesis. The gray region in the output indicates already synthesized portion.*

The quality problem of pixel-based approaches can be improved by synthesizing patches rather than pixels. Many algorithms have been proposed based on this principle, but for the sake of discussion, we will use [Kwatra et al. 2003], which offered the best quality so far. Even though the algorithm synthesizes an output in patches, the basic principle is still very similar to pixel-based synthesis. In some sense, you can think that a patch is simply a big pixel. Specifically, the output is produced by assembling patches from the input sample. The patch is chosen by matching neighborhoods, which is defined as the ring of pixels surrounding the boundary of the patch. So in some sense, when the patch becomes single pixel, the algorithm would reduce to pixel-based synthesis. However, one major difference between patch-based and pixel-based approaches is that, in a patch-based method, we will need to figure out how to composite a new patch with the already synthesized portion of the output. If this is not done carefully, visible seam artifacts will appear in the output image. [Kwatra et al. 2003] achieves this by finding the minimum error path via graphi-cut. The technique improves the synthesis quality further by allowing pasting patches over already-synthesized regions if this would reduce synthesis error.

2.2.3 Optimization-based synthesis

[Kwatra et al. 2005] provides an interesting twist to the trend of example-based synthesis. Instead of using patches, the algorithm actually considers individual pixels. But unlike previous methods which synthesize pixels one by one in a greedy fashion, this technique considers them all together, and determine their values by optimizing a quadratic error energy function. The error function is determined by mismatches of input/output neighborhoods, so minimizing this function leads to better output quality.

Due to the use of iterative optimization, this technique is slower than previous work, but it would allow novel synthesis effects such as textured flow patterns.

Exercise Implement your favorite example-based synthesis algorithm. Try to run it through a variety of textures. Observe which ones work and which ones fail. Could you explain, by the properties of the textures and the natural of the algorithm, why would they work or fail?

3 Texture Mapping

After acquiring a texture image, the next step is to decide how to map it onto the target object surface. There are several issues to consider here. First, you need to decide roughly how and where you would like to apply the texture. This is more or less an artistic or application choice. Then, you need to figure out exactly how the texture is mapped. Specifically, we would like to minimize distortion or discontinuity.

There is a huge literature in texture mapping, as it involves a lot of math. Many of these papers have title or keywords containing *parameterization*, so they are pretty easy to identify. Basically, parameterization means how to represent the object surface by a 2D function $f(u, v)$, so that each surface point can be reached by a particular pair of (u, v) values. These values are called parameters, and the process of finding $f(u, v)$ is called parameterization. A good parameterization should have low distortion and discontinuity. The major application of parameterization is texture mapping, as well signal mesh processing.

In general, it is impossible to obtain a parameterization that has no distortion or discontinuity, except for simple cases such as a plane, a cylinder, or a cone.

3.1 Volume Texture

One way to circumvent the difficulty of surface parameterization is to synthesize a volumetric, instead of planar texture, and apply texture mapping by embedding the object into the texture volume. This approach would completely avoid discontinuity or distortion in parameterization. In fact, many procedural textures can be synthesized over a 3D domain, and are perfectly suited for this approach.

However, for example-based synthesis, it is usually not possible to synthesize a 3D volume from 2D example, unless the texture is highly random or isotropic, as demonstrated in [Heeger and Bergen 1995]. Even if the volumetric texture can be synthesized or simply drawn manually, they can be expensive to store or render due to the large data size. This data size problem can be reduced by storing only values at voxels near the surface [(grue) DeBry et al. 2002; Benson and Davis 2002], but this would make it tricky to render efficiently on graphics hardware.

Due to these difficulties, volumetric texture mapping is usually employed in conjunction with procedural synthesis.

3.2 Direct Surface Synthesis

Another method to circumvent the difficulty of parameterization is by synthesizing textures directly over the object surface. This can be done by extending 2D texture synthesis via either pixel-based [Turk 2001; Wei and Levoy 2001] or patch-based [Soler et al. 2002] techniques. By allowing only textures with repeating patterns, these techniques bypass the need for a global parameterization. However, they cannot deal with general texture images. In addition, the computed mapping only applies to one texture and cannot be reused for another different texture.

3.3 Texture Atlas

Texture atlas [Maillot et al. 1993] is the conventional, and perhaps the most popular method for texture mapping in commercial applications.[1] The basic idea is to partition the object surface into several domains, so that each domain can be easily parameterized as a planar chart. The parameter distortion can be reduced by increasing the number of charts. However, care must be taken to avoid discontinuity across chart boundaries. Once created, a texture atlas can be easily rendered as a traditional 2D texture map on graphics hardware.

3.4 Base-domain Parameterization

A planar domain is not the most natural choice for parameterize a complex object. Often, the parameterization can be improved by using a base domain that is more similar to the object shape. One possible choice of the base domain is a simplified mesh [Lee et al. 1998], utilizing triangles on the base mesh as the domain of parameterization. These triangular domains avoid the irregular boundary problem in general texture atlases. Another possibility is to use poly-cube maps as the base domain [Tarini et al. 2004]. This idea is an extension of environment cubemaps, suitable for graphics hardware implementation.

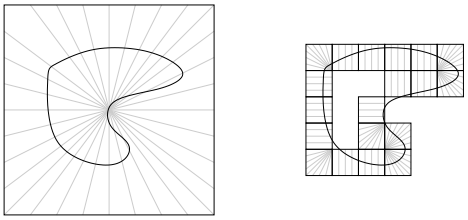


Figure 7: *Cube map and Poly-cube map.*

Exercise Try to run the UVAtlas program [UVAtlas 2005] to produce an atlas for your favorite polygonal mesh. Are you satisfied with the quality? If not, try to improve the quality either manually, by an alternative algorithm.

4 Texture Sampling

Now we know how to generate a texture image and how to map it onto an object surface, we are ready to apply texturing to our rendering tasks. However, if you just do this directly, you might encounter annoying artifacts in the rendered images. An example is shown in Figure 8. These artifacts are caused by signal *aliasing*.

[1] Automatic texture atlas creation is supported in DirectX via UVAtlas.

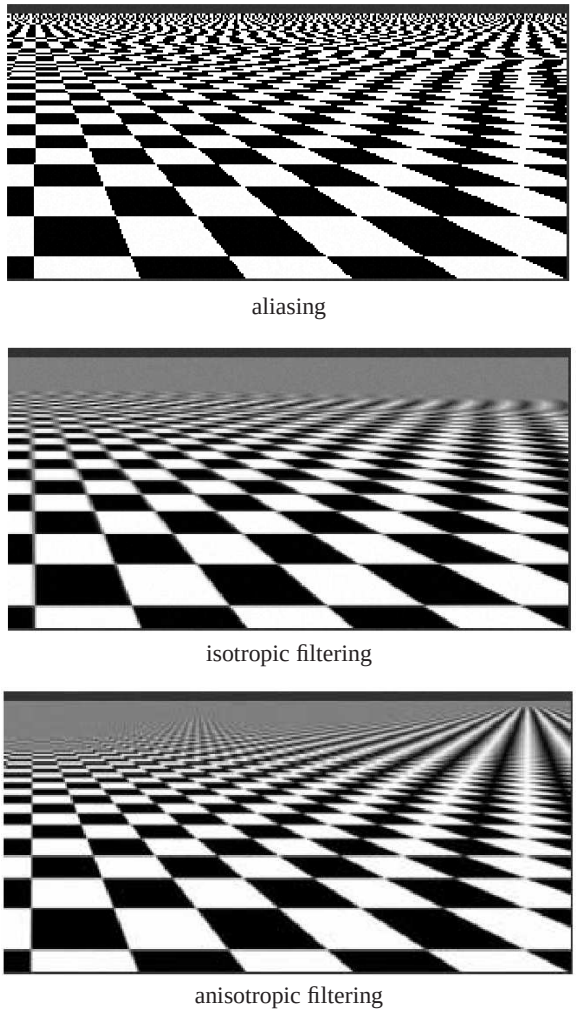


Figure 8: *Aliasing and anti-aliasing for texture mapping.*

In the rest of this section, we describe what aliasing is, and how to prevent them in texturing.

4.1 Signal Aliasing and Anti-aliasing

In a nutshell, aliasing is caused when a signal is sampled at too low a rate. An example of 1D signal aliasing is shown in Figure 9. On the top image, we see a signal that is sampled at a lower rate. Obviously, this sampling rate is too low and many high frequency features of the original signal are missed. As a result, when the sampled points are reconstructed, we obtain a totally different signal, shown in red.

One method to resolve this problem is to sample at a higher rate. Unfortunately, this is not always feasible, and in many applications the sampling rate is pre-determined. For example, in graphics applications, your screen resolution often limits how dense a displayed image is sampled.

Given a limited sampling rate, the other alternative that we could avoid the aliasing problem is to pre-filter the signal into a lower frequency one. This would eliminate the aliasing problem, since now the reconstructed signal will be identical to the original, as shown in the bottom case of Figure 9.

This is probably all you need to know to avoid aliasing problems

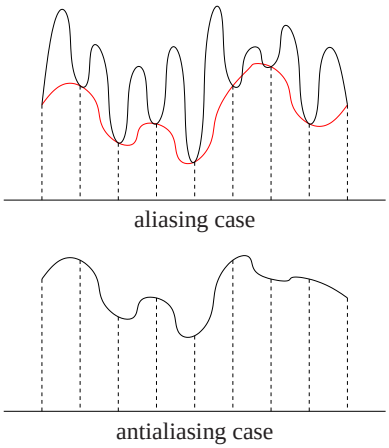


Figure 9: *Signal aliasing and antialiasing. Top: The black curve is the original one, and the red curve is the aliased signal caused by the low sampling rate. The dashed lines indicate sampling locations. Bottom: The filtered signal without aliasing after sampling.*

in texturing. A rigorous explanation of aliasing/anti-aliasing will involve Fourier transform, which we would like to try to avoid here. Instead, we will discuss this issue purely in the spatial domain via concrete examples, as in the next subsection.

4.2 Texture Filtering and Mipmapping

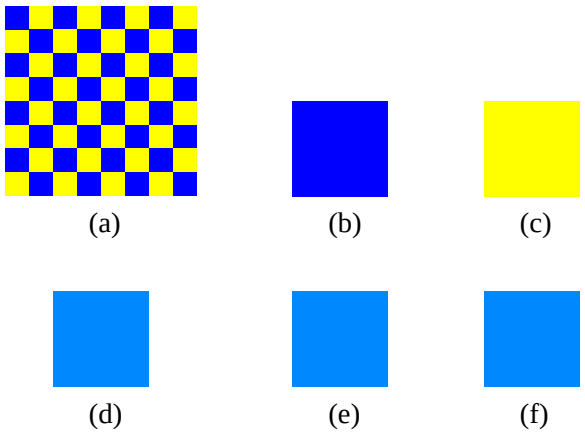


Figure 10: *Anti-aliasing for texture sampling. The original texture is shown in (a), which has 8×8 pixels. When this texture is mapped onto a polygon with 4×4 pixels, the polygon will appear to be either as (b) or (c), depending on how it is sampled. This is an aliasing problem. By filtering the texture into lower frequency (d), we achieve correct texture sampling as shown in (e, f).*

Discussing aliasing and anti-aliasing issues using 1D signal is probably not much fun, so now lets switch to 2D texture images.

Assuming we have a very simple texture image, as shown in Figure 10. The texture has size 8×8, and we apply it to a polygon, rendered frontal-parallel with 4×4 screen pixels. Because the screen resolution of the polygon is smaller than the texture size, we cannot display all the texels simultaneously. So how the polygon will appear? Depending on the numerical precision of your renderer (either in hardware or software) and on where the polygon is located, you can obtain two different results, shown in (b, c). Obviously, both are incorrect. Worse, when the polygon moves across the screen, it might flicker back and forth between the two possibilities. This can be very visually disturbing. This is exactly an aliasing problem, since the sampling rate 4×4 is not sufficient for the texture, which has size 8×8.

To eliminate aliasing, we have to filter the texture so that it does not contain any high frequency content that cannot be sampled in 4×4 resolution. The filtered and down-sampled image is shown in Figure 10 case (d). In this particular case, the texture becomes a constant, so the polygon will have consistent appearance no matter how it is sampled. This is an anti-aliasing technique, commonly referred as mipmapping [Williams 1983]. In this particular case, we build a mipmap with two levels.

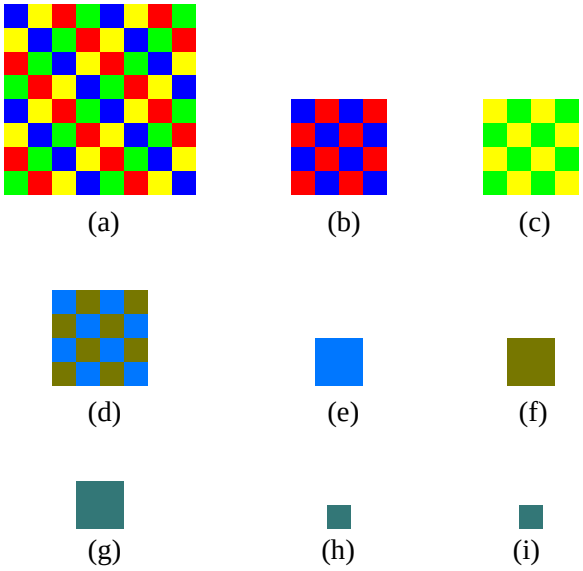


Figure 11: *Anti-aliasing for texture sampling. Continue from the previous case shown in Figure 10, but now we have a more complicated pattern. In Figure 10, two mipmap levels are enough, but here we need three mipmap levels.*

For more complicated textures, two mipmap levels might not be enough, as demonstrated in Figure 11. In general, for a texture with size $2^N \times 2^N$, we need to build a mipmap with $N + 1$ levels to avoid aliasing for all possible sampling rates. The middle image shown in Figure 8 demonstrates the result after mipmapping. Note that the aliasing artifacts are all gone.

4.2.1 Anisotropic Filtering

However, one problem with mipmapping is that it can sometimes over-blur, as shown in Figure 8. When we build a mipmap, we always filter a higher resolution level isotropically before down-sampling for a lower resolution level. This can cause over-blurring when the desired filter footprint is anisotropic, which can happen when a polygon is viewed perspectively, as shown in Figure 8. When the footprint is anisotropic, we will have to enclose it with a big isotropic footprint in order to avoid aliasing (Figure 12). However, this would also cause over blurring in the short axis of the anisotropic footprint.

One possible solution to reduce over-blurring is to use multiple isotropic footprints to approximate one anisotropic footprint, as shown in Figure 12. This was initially proposed by [McCormack

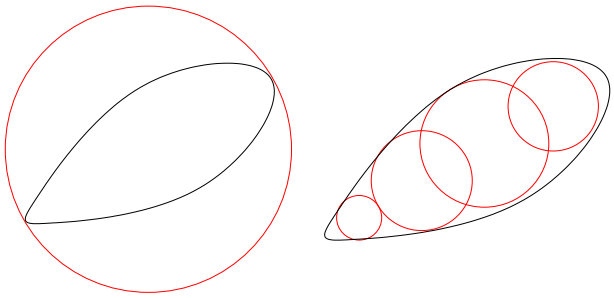


Figure 12: *Anisotropic filtering via isotropic footprints. Left: mipmap filtering, causing blurring. Right: anisotropic filtering by multiple isotropic footprints.*

et al. 1999], and is currently the prevailing solution for anisotropic filtering on graphics hardware. The disadvantage of this approach is that the filtering would become slower due to the use of multiple footprints.

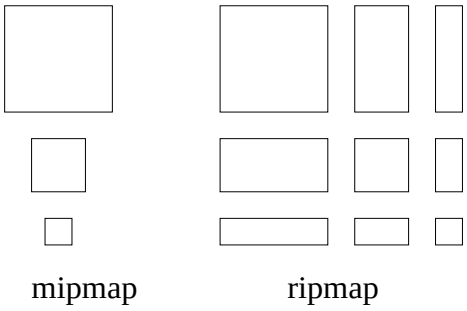


Figure 13: *Mipmap versus Ripmap.*

An alternative solution is to build a ripmap instead of mipmap by pre-computing the anisotropic filtering. This is illustrated in Figure 13. Compared to [McCormack et al. 1999], this approach is faster because only one footprint is required. However, ripmap takes more texture memory. In addition, it only supports anisotropy in a limited set of directions. In Figure 13, only vertical and horizontal directions are supported. In contrast, [McCormack et al. 1999] can support anisotropic footprint in any direction while consuming memory of a standard mipmap. This is probably why graphics hardware vendors prefer [McCormack et al. 1999] over ripmap.

Exercise Render a perceptively viewed checker-board pattern (as in Figure 8) on your graphics chip. Try to tune the anisotropic texture filtering option to see the impact on visual quality and computation speed. Can you design a good quality test for anisotropic texture filtering on graphics hardware?

5 Texturing on Graphics Hardware

We have now introduced the basic ideas about texturing, including texture generation, texture mapping, and texture sampling. We now move on to look at some applications and implementation issues for real-time texturing on graphics hardware. For offline applications such as ray tracing, these are relatively non-issues because we dont care too much about computation speed. In fact, in some applications the entire scene is represented as micro-polygons and no texture is used. However, for real-time applications running on graphics hardware, we have to carefully consider these issues in order to achieve good performance.

5.1 Modeling Geometry as Texture

The performance of a real-time application often depends on the geometry complexity of the scene. One method to simplify the geometry is to represent detailed surface bumps by texture maps. Depending on the desired visual effect, surface geometry can be approximated as bump maps [Blinn 1978], which uses textures to modulate surface normals but ignoring self-occlusion and silhouettes, or displacement maps [Cook 1984], which handles occlusion, shadowing and silhouettes at the expense of more geometry processing.

Recently, several techniques have been proposed to render displacement maps on programmable graphics hardware via texture maps without increasing geometry complexity. [Wang et al. 2003] pre-computes displacement of a small surface patch from a dense set of viewing angles. This 4D information is stored and compressed as a set of texture maps, and are looked-up during run time for rendering. [Policarpo et al. 2005] reduces the data size to 2D by performing binary search to find the correct displacement. This binary search is heuristic and does not guarantee a completely accurate result. [Donnelly 2005] improves the search robustness via sphere tracing, with 3D texture storage.

Texturing can be utilized to accelerate many other aspects of rendering, such as reflections, refractions, shading, global illumination, animation, and motion blur. A good reference for these techniques is [Akenine-Moller and Haines 2002], as well papers on Symposium of Interactive 3D Graphics and Games.

Exercise Implement one of the displacement texture map techniques as described above. Since they are all approximations of real displacements, they must all break under certain situations. Try to figure out when they will break, by varying the parameters of the algorithm, like texture resolution, displacement field frequency, or number of search iterations.

5.2 Texture Compression

The computation power of recent graphics hardware has been advancing in an amazing speed. Unfortunately, this is not the case for memory latency and bandwidth. As a result, a texture read instruction is becoming more and more expensive compared to arithmetic instructions. For good performance, we have to either reduce the amount of texture read, and/or improve the texture cache coherence.

We can achieve these goals by texture compression. Texture compression has several advantages. First, it reduces texture memory usage. So that given the same amount of memory, an application can use more textures if compressed. Second, a compressed texture often improves cache coherence, since a single texture datum covers more texels than an uncompressed texture.

There are a variety of methods to compress textures. In theory, a texture can be compressed by ordinary image compression algorithms, but for hardware implementation, the algorithm must allow random access and real-time decompression. The current standard for texture compression is DXT [DXT 1998], which compresses 4×4 pixel blocks independently. DXT is simple, fast, and works surprisingly well for a variety of natural images, but the compression ratio is limited (8:1). For textures containing repeating patterns, much high compression ratio can be achieved via techniques such as texture tiling via shader programming [Wei 2004].

In general, improving texturing performance on graphics hardware is a hot on-going research, both in terms of GPU architecting or shader programming tricks.

Exercise Experiment with DXT compression on a graphics chip. Try to observe the compression artifacts, and come up with a criteria on which textures can be compressed well and which cannot, in terms of visual quality.

6 Conclusion

We have introduced the three basic components of texturing: texture generation, texture map, and texture sampling, as well described the issues on graphics hardware. Texturing is still pretty much research in progress; pay attention to latest advances in conferences!

References

AKENINE-MOLLER, T., AND HAINES, E. 2002. *Real-Time Rendering*. A.K. Peters Ltd.

BENSON, D., AND DAVIS, J. 2002. Octree textures. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 785–790.

BLINN, J. F. 1978. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, 286–292.

COOK, R. L. 1984. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 223–231.

DONNELLY, W. 2005. Per-pixel displacement mapping with distance functions. In *GPU Gems II*, 123.

DXT, 1998. DXT Texture Compression Standard. `http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dx8_vb/directx_vb/graphics_using_1ir7.asp`.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1998. *Texturing & Modeling, A Procedural Approach*. AP Professional.

EFROS, A. A., AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, 1033–1038.

ELIAS, H., 2003. Perlin noise. `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`.

(GRUE) DEBRY, D., GIBBS, J., PETTY, D. D., AND ROBINS, N. 2002. Painting and rendering textures on unparameterized models. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 763–768.

HEEGER, D. J., AND BERGEN, J. R. 1995. Pyramid-based texture analysis/synthesis. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 229–238.

KWATRA, V., SCHODL, A., ESSA, I., TURK, G., AND BOBICK, A. 2003. Graphcut textures: image and video synthesis using graph cuts. *ACM Trans. Graph. 22*, 3, 277–286.

KWATRA, V., ESSA, I., BOBICK, A., AND KWATRA, N. 2005. Texture optimization for example-based synthesis. *ACM Trans. Graph. 24*, 3, 795–802.

LEE, A. W. F., SWELDENS, W., SCHRöDER, P., COWSAR, L., AND DOBKIN, D. 1998. Maps: multiresolution adaptive parameterization of surfaces. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 95–104.

MAILLOT, J., YAHIA, H., AND VERROUST, A. 1993. Interactive texture mapping. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, 27–34.

MCCORMACK, J., PERRY, R., FARKAS, K. I., AND JOUPPI, N. P. 1999. Feline: fast elliptical lines for anisotropic texture mapping. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 243–250.

PERLIN, K. 1985. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, 287–296.

PERLIN, K. 2002. Improving noise. *ACM Transactions on Graphics 21*, 3 (July), 681–682.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 155–162.

SOLER, C., CANI, M.-P., AND ANGELIDIS, A. 2002. Hierarchical pattern mapping. *ACM Transactions on Graphics 21*, 3 (July), 673–680.

TARINI, M., HORMANN, K., CIGNONI, P., AND MONTANI, C. 2004. Polycube-maps. *ACM Transactions on Graphics 23*, 3 (Aug.), 853–860.

TURK, G. 2001. Texture synthesis on surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 347–354.

UVATLAS, 2005. Uvatlas. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/ProgrammingGuide/AdvancedTopics/UsingUVAtlas.asp`.

WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2003. View-dependent displacement mapping. *ACM Trans. Graph. 22*, 3, 334–339.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 479–488.

WEI, L.-Y., AND LEVOY, M. 2001. Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 355–360.

WEI, L.-Y. 2004. Tile-based texture mapping on graphics hardware. In *Graphics Hardware 2004*, 55–64.

WILLIAMS, L. 1983. Pyramidal parametrics. In *Proceedings of SIGGRAPH 1983*, 1–11.