Antialiasing & Texturing

Steve Rotenberg
CSE168: Rendering Algorithms
UCSD, Spring 2017
Texture Minification

- Consider a texture mapped triangle
- Assume that we *point sample* our texture so that we use the nearest texel to the center of the pixel to get our color
- If we are far enough away from the triangle so that individual texels in the texture end up being smaller than a single pixel in the framebuffer, we run into a potential problem
- If the object (or camera) moves a tiny amount, we may see drastic changes in the pixel color, as different texels will rapidly pass in front of the pixel center
- This causes a flickering problem known as *shimmering* or *buzzing*
- Texture buzzing is an example of *aliasing*
Small Triangles

- A similar problem happens with very small triangles
- If we shoot a single ray right through the center of a pixel, then we are essentially *point sampling* the image
- This has the potential to miss small triangles
- If we have small, moving triangles, they may cause pixels to flicker on and off as they cross the pixel centers
- A related problem can be seen when very thin triangles cause pixel gaps
- These are more examples of *aliasing* problems
Stairstepping

• What about the jagged right angle patterns we see at the edges of triangles?
• This is known as the *stairstepping* problem, also affectionately known as "the jaggies"
• These can be visually distracting, especially for high contrast edges near horizontal or vertical
• Stairstepping is another form of *aliasing*
Moiré Patterns

- When we try to render high detail patterns with a lot of regularity (like a grid), we occasionally see strange concentric curve patterns forming.
- These are known as Moiré patterns and are another form of aliasing.
- You can actually see these in real life if you hold two window screens in front of each other.
The Propeller Problem

• Consider an animation of a spinning propeller, that is rendering at 30 frames per second
• If the propeller is spinning at 1 rotation per second, then each image shows the propeller rotated an additional 12 degrees, resulting in the appearance of correct motion
• If the propeller is now spinning at 30 rotations per second, each image shows the propeller rotated an additional 360 degrees from the previous image, resulting in the appearance of the propeller sitting still!
• If it is spinning at 29 rotations per second, it will actually look like it is slowly turning backwards
• These are known as strobing problems and are another form of aliasing
Aliasing

- These examples cover a wide range of problems, but they all result from essentially the same thing
- In each situation, we are starting with a *continuous signal*
- We then *sample* the signal at *discreet points*
- Those samples are then used to *reconstruct* a new signal, that is intended to represent the original signal
- However, the reconstructed signals are a false representation of the original signals
- In the English language, when a person uses a false name, that is known as an *alias*, and so it was adapted in *signal analysis* to apply to falsely represented signals
- Aliasing in computer graphics usually results in visually distracting *artifacts*, and a lot of effort goes into trying to stop it. This is known as *antialiasing*
Signals

• The term *signal* is pretty abstract, and has been borrowed from the science of *signal analysis*
• Signal analysis is very important to several areas of engineering, especially electrical, audio, and communications
• Signal analysis includes a variety of mathematical methods for examining signals such as Fourier analysis, filters, sampling theory, digital signal processing (DSP), and more
• In electronics, a one dimensional signal can refer to a voltage changing over time. In audio, it can refer to the sound pressure changing over time
• In computer graphics, a one dimensional signal could refer to a horizontal or vertical line in our image. Notice that in this case, the signal doesn’t have to change over time, instead it varies over space (the x or y coordinate)
• Often signals are treated as functions of one variable and examples are given in the 1D case, however the concepts of signal analysis extend to multidimensional signals as well, and so we can think of our entire 2D image as a signal
Sampling

- If we think of our image as a bunch of perfect triangles in continuous (floating point) device space, then we are thinking of our image as a continuous signal.
- This continuous signal can have essentially infinite resolution if necessary, as the edges of triangles are perfect straight lines.
- To render this image onto a regular grid of pixels, we must employ some sort of discreet sampling technique.
- In essence, we take our original continuous image and sample it onto a finite resolution grid of pixels.
- If our signal represents the red intensity of our virtual scene along some horizontal line, then the sampled version consists of a row of discreet 8 bit red values.
- This is similar to what happens when a continuous analog sound signal is digitally sampled onto a CD.
Reconstruction

- Once we have our sampled signal, we then reconstruct it.
- In the case of computer graphics, this reconstruction takes place as a bunch of colored pixels on a monitor.
- In the case of CD audio, the reconstruction happens in a DAC (digital to analog converter) and then finally in the physical movements of the speaker itself.
Reconstruction Filters

- Normally, there is some sort of additional *filtration* that happens at the reconstruction phase.
- In other words, the actual pixels on the monitor are not perfect squares of uniform color. Instead they will have some sort of color distribution.
- Additional filtration happens in the human eye so that the grid of pixels appears to be a continuous image.
- In audio, the perfect digital signal is filtered first by the analog electronic circuitry and then by the physical limitations of the speaker movement.
Low Frequency Signals

- Original signal

- Point sampled at relatively high frequency

- Reconstructed signal
High Frequency Signals

- Original signal
- Point sampled at relatively low frequency
- Reconstructed signal
Regular Signals

- Original repeating signal
- Point sampled at relatively low frequency
- Reconstructed signal repeating at incorrect frequency
Nyquist Frequency

• Theoretically, in order to adequately reconstruct a signal of frequency $x$, the original signal must be sampled with a frequency of greater than $2x$
• This is known as the *Nyquist frequency* or *Nyquist limit*
• However, this is assuming that we are doing a somewhat idealized sampling and reconstruction
• In practice, it’s probably a better idea to sample signals at a minimum of $4x$
Aliasing Problems

- **Shimmering / Buzzing:**
  Rapid pixel color changes (flickering) caused by high detail textures or high detail geometry. Ultimately due to point sampling of high frequency color changes at low frequency pixel intervals.

- **Stairstepping / Jaggies:**
  Noticeable stairstep edges on high contrast edges that are nearly horizontal or vertical. Due to point sampling of effectively infinite frequency color changes (step gradient at edge of triangle).

- **Moiré patterns:**
  Strange concentric curve features that show up on regular patterns. Due to sampling of regular patterns on a regular pixel grid.

- **Strobing:**
  Incorrect or discontinuous motion in fast moving animated objects. Due to low frequency sampling of regular motion in regular time intervals.
Spatial / Temporal Aliasing

- Aliasing shows up in a variety of forms, but usually those can be separated into either \textit{spatial} or \textit{temporal} aliasing.
- Spatial aliasing refers to aliasing problems based on regular sampling in space. This usually implies device space, but we see other forms of spatial aliasing as well.
- Temporal aliasing refers to aliasing problems based on regular sampling in time.
- The antialiasing techniques used to fix these two things tend to be very different, although they are based on the same fundamental principles.
Point Sampling

• The aliasing problems we’ve seen are due to low frequency \textit{point sampling} of high frequency information
• With point sampling, we sample the original signal at precise points (pixel centers, etc.)
• Is there a better way to sample continuous signals?
Box Sampling

- We could also do a hypothetical *box sampling* (or *box filter*) of our image.
- In this method, each triangle contributes to the pixel color based on the area of the triangle within the pixel.
- The area is equally weighted across the pixel.
Pyramid Sampling

- Alternately, we could use a weighted sampling filter such as a pyramid filter
- The pyramid filter considers the area of triangles in the pixel, but weights them according to how close they are to the center of the pixel
Sampling Filters

- We could potentially use any one of several different sampling filters
- Common options include the point, box, pyramid, cone, and Gaussian filters
- Different filters will perform differently in different situations, but the best all around sampling filters tend to be Gaussian in shape
- The filters aren’t necessarily limited to cover only pixel. It is possible, and not uncommon to use filters that extend slightly outside of the pixel, thus overlapping with the neighboring pixels. Filters that cover less than the square pixel, however, tend to suffer from similar problems as point sampling
Edge Antialiasing
Supersampling

- The easiest way to improve the antialiasing in a ray tracer is to trace more than one ray per pixel.
- Instead of just point sampling the center of the pixel, we can trace several.
- We refer to this process as *supersampling*.
- For high quality edge-antialiasing, it is not uncommon to use 16 or more samples per pixel.
Uniform Sampling

• With *uniform sampling*, the pixel is divided into a uniform grid of *subpixels*

• Uniform supersampling should certainly generate better quality images than single point sampling

• It will filter out some high frequency information, but may still suffer from Moiré problems with highly repetitive signals
Random Sampling

- With *random sampling*, the pixel is supersampled at several randomly located points.
- Random sampling has the advantage of breaking up repeating signals, and so can completely eliminate Moiré patterns. It does, however, trade the regular patterns with random *noise* in the image, which tends to be less annoying to the viewer.
- It also suffers from potential clustering and gaps of the samples.

![Random Sampling Example](image-url)
**Jittered Sampling**

- With *jittered* or *stratified sampling*, the pixel is divided into a grid of *subpixels*, but the subpixels themselves are sampled at a random location within the subpixel.
- This combines the advantages of both uniform and random sampling.
Jittered vs. Non-Jittered

1 sample
Non-jittered
Jittered

16 samples
Non-jittered
Jittered
Weighted Sampling

• If we average all of the samples equally to get the final pixel color, we are essentially performing a box filter on the samples.

• We can also perform a weighted average of the samples to achieve other shaped filters.

• For example, we can weight the samples according to a box, cone, pyramid, or Gaussian shape if desired.

• We can apply weighting to uniform, random, or jittered supersamples with little additional work.
Another option to apply a weight (such as Gaussian) to a pixel is to modify the *distribution* of the samples.

In the case of a Gaussian weight, we could either take a uniform distribution and weight the individual samples or we could apply the weighing to the spacing of the samples and weight the samples themselves equally.

Which approach is better?
Weighted Samples vs. Distributions

- If we look at the 16 samples in the left image, we see that some are much more important than others, yet they all have the same computational cost.
- In other words, the 4 samples in the center of the grid might have more total weight than the other 12 samples around the perimeter.
- By adjusting our distribution so there are more samples in the higher valued areas, we can achieve the benefits of jittered and weighted sampling while maintaining efficiency by treating all samples equally.
- This is an important concept that we will come back to again and again...
Weighted Distributions

• Let’s say we start with two random numbers $s$ and $t$ that are uniformly distributed in the $[0...1]$ interval

• We want to generate two new random numbers $s’$ and $t’$ in the same interval but with some type of weighting towards the center

• Gaussian:
  
  
  $a = \mu \sqrt{-2 \log s}$  \hspace{1cm} (\mu \approx 0.4)
  
  $b = 2\pi t$
  
  $s’ = 0.5 + a \cdot \sin b$
  
  $t’ = 0.5 + a \cdot \cos b$

• Shirley:
  
  $s’ = \begin{cases} 
  -0.5 + \sqrt{2s} & \text{if } s < 0.5 \\
  1.5 - \sqrt{2 - 2s} & \text{if } s \geq 0.5
  \end{cases}$

  $t’ = \begin{cases} 
  -0.5 + \sqrt{2t} & \text{if } t < 0.5 \\
  1.5 - \sqrt{2 - 2t} & \text{if } t \geq 0.5
  \end{cases}$
Weighted Distributions
Pixel Sampling

Random

Jittered

Uniform

Gaussian

Shirley
Adaptive Sampling

• Another approach to the problem is to perform *adaptive sampling*
• With this scheme, we start with a small number of samples and analyze their statistical variation
• If the colors are all similar, we accept that we have an accurate sampling
• If we find that the colors have a large variation, we continue to take further samples until we have reduced the statistical error to an acceptable tolerance
• Adaptive sampling schemes can be very efficient compared to brute force supersampling, but can also suffer from an important problem called *bias*
Sampling & Bias

• When we do random sampling for antialiasing (and for other things like area lights), we are essentially estimating the value of a function.

• The more random samples we take, the better our estimate of the result.

• We say that as we take more random samples, our estimate *converges* to the actual value.

• By taking a fixed number of supersamples, we generate an *unbiased* estimate that will converge to the correct value.

• However, when we add decisions based on the results from previous samples (such as with the adaptive sampling methods), we can add a statistical *bias* to the estimate, resulting in a tendency towards either over or underestimating the resulting value.

• Many rendering optimizations can add bias to the resulting image but its usually small enough to not be a problem, and the resulting performance improvements typically justify it.

• There is however a tendency towards more brute force unbiased renderers these days as GPUs get faster and faster.
Temporal Aliasing

• Properly tuned supersampling techniques address the spatial aliasing problems pretty well
• We still may run into temporal aliasing or strobing problems when we are generating animations
• Just as the spatial antialiasing techniques apply a certain blurring at the pixel level, temporal antialiasing techniques apply blurring at the frame level
• In other words, the approach to temporal antialiasing is to add *morion blur* to the image
Motion Blur

- Motion blur can be a tricky subject, and several different approaches exist to address the issue.
- One way to incorporate it into a ray tracer is to randomly distribute the rays in time!
- Let’s assume that every object in the scene has two matrices— one for its position/orientation at the beginning of the frame and one for the end of the frame.
- The camera has an initial and final matrix as well.
- Each ray shot from the camera is randomly distributed in time and computes the appropriate interpolated matrix when it traverses through the scene.
- This can actually be done pretty efficiently by modifying the Instance and Camera classes.
Combining Antialiasing Techniques

• We can even combine the techniques of pixel antialiasing and motion blur without exponentially increasing the work
• This can be done by rendering a fixed number of supersamples total, each one spread in time and jittered at the pixel level
• We can also combine this with the area lighting samples from an earlier lecture. Previously, we needed hundreds of shadow rays per camera ray, but as we are now generating dozens of camera rays, we can reduce this to just a few shadow rays per camera ray
• This overall approach offers a powerful foundation for other blurry effects such as: soft shadows (penumbrae), lens focus (depth of field), color separation (dispersion), glossy reflections, diffuse interreflections, etc.
Texture Mapping
Texture Mapping

- Texture mapping is the process of mapping an image onto a triangle (or other surface) in order to increase the detail of the rendering.
- This allows us to get fine scale details without resorting to rendering tons of tiny triangles.
- The image that gets mapped onto the triangle is called a texture map or texture.
- Textures are usually a regular 24-bit color image, but can often be 8-bit greyscale or various other image types.
- The pixels of the texture map are often called texels.
Texture Space

- We define our texture map as existing in *texture space*, which is a normal 2D space.
- The lower left corner of the image is the coordinate (0,0) and the upper right of the image is the coordinate (1,1).
- The actual texture map might be 512 x 256 pixels for example, with a 24 bit color stored per pixel.
Texture Coordinates

• To render a textured triangle, we must start by assigning a texture coordinate to each vertex

• A texture coordinate is a 2D point \((u,v)\) in texture space that is the coordinate of the image that will get mapped to a particular vertex
Texture Mapping

Texture Space

Triangle (in 3D space)
Texture Mapping

• When the ray intersects the triangle, we need to find the interpolated texture coordinate at the location of the intersection
• We’ve already looked at interpolating the normals across a triangle by using the barycentric coordinates
• We can do the exact same thing for texture coordinates

\[
\begin{align*}
    u &= (1-\alpha-\beta)u_0 + \alpha u_1 + \beta u_2 \\
    v &= (1-\alpha-\beta)v_0 + \alpha v_1 + \beta v_2 
\end{align*}
\]

• We just need to store texture coordinates with the Vertex class and make sure that they are interpolated and stored in the Intersection class within the Triangle::Intersect() function
• Then it is up to the shading system to evaluate the actual texture mapping
Texture Maps

• Adding texture mapping to a ray tracer is pretty straightforward

• The hardest part typically is dealing with loading various image file formats. There are some standard libraries and open source packages that attempt to ease some of this

• Once an image is loaded, we really just need to associate the texture map to a particular property of a material (for example- to the diffuse color property of the Lambert material)
Material Evaluation

• We set up the Material::ComputeReflectance() virtual function to take a const Intersection as input

• This allows it access to the texture coordinates, so the Material can evaluate whatever texture mapping it requires

• For example, this allows more complex materials to use separate texture maps for diffuse and specular colors
Attribute Mapping

• Most general purpose rendering systems allow arbitrary properties (or attributes) to be mapped across the triangles (or other surfaces)

• For example, one can map texture coordinates, colors, arbitrary shading properties such as material smoothness, or any application-specific requirements

• This implies that the Vertex and Intersection classes would need to support a more general array of attributes that are typically matched to the specific requirements of a derived Material
Texture Wrapping
Wrapping

• The image exists from (0,0) to (1,1) in texture space, but that doesn’t mean that texture coordinates have to be limited to that range

• We can define various rules to determine what happens when we need to render surfaces with texture coordinates that go outside of the [0...1] range
Wrapping Modes

• Some texture maps are intended to \textit{tile} or \textit{wrap} indefinitely
• For example, we could make a brick wall by tiling a texture of a small section of brick
• There are other times when we don’t want a texture to tile
• It is useful to be able to define \textit{wrapping modes} for each texture
• In fact, it is useful to be able to control the wrapping behavior independently in the \textit{u} and \textit{v} directions
Wrapping Modes

• It is common to allow three basic wrapping modes: wrap, clamp, and mirror

• *Wrap* implies the texture will tile indefinitely

• *Clamp* implies the texture coordinates will clamp to the [0…1] range

• *Mirror* implies the texture will tile indefinitely, but flipping directions on alternate repeats

• Remember that it is important to be able to control these modes independently for the $u$ and $v$ directions on the texture
Wrapping Modes

- Wrap U & V
- Mirror U & V
- Clamp U & V
- Mirror U, Clamp V
float WrapCoordinate(float f , WrapMode mode) {
    if(mode==WRAP) {
        if (f < 0.0)  f=1.0 – mod(-f , 1.0);
        else  f=mod(f , 1.0);
    }
    else if(mode==MIRROR) {
        if (f < 0.0)  f=2.0 – mod(-f , 2.0);
        else  f=mod(f , 2.0);
        if (f > 1.0)  f=2.0-f;
    }
    else if(mode==CLAMP) {
        if(f<0.0) f=0.0;
        else if(f>1.0) f=1.0;
    }
    return f;
}
Texture Sampling
Magnification

• What happens when we get too close to the textured surface, so that we can see the individual texels up close?
• In texture mapping terminology, this is called *magnification*.
• When the texture is magnified, we can see how the choice of *texture sampling* modes affects the appearance.
• Some of the common texture sampling modes are:
  – Point sampling
  – Bilinear sampling
  – Bicubic sampling
Texture Sampling

- With *point sampling*, each rendered pixel just samples the texture at a single texel, nearest to the texture coordinate. This causes the individual texels to appear as solid colored rectangles.
- With *bilinear sampling*, each rendered pixel performs a bilinear blend between the nearest 4 texel centers. This causes the texture to appear smoother when viewed up close, however, when viewed too close, the bilinear nature of the blending can become noticeable.
- *Bicubic sampling* is an enhancement to bilinear sampling that actually samples a small 4x4 grid of texels and performs a smoother bicubic blend. This may improve image quality for up close situations, but adds some memory access costs.
Texture Sampling

• The choice of texture sampling mode is typically associated with the individual textures
• This allows different textures to use different modes
• Organic material textures would probably want to use one of the smoothing modes such as bilinear or bicubic sampling
• Some man-made textures however (like checker patterns) work best with point sampling
class TextureMap {
    public:
        TextureMap(char *filename);
        ~TextureMap();

        void Evaluate(Color &col, float u, float v);

        enum WrapMode {WRAP, CLAMP, MIRROR};
        void SetWrapMode(WrapMode u, WrapMode v);

        enum SampleMode {POINT, BILINEAR, BICUBIC};
        void SetSampleMode(SampleMode s);

    private:
        float WrapCoordinate(float f, WrapMode mode);

        Bitmap *BMP;
        WrapMode WrapU, WrapV;
        SampleMode Sample;
};
Filtering
Minification

- *Minification* is the opposite of magnification and refers to how we handle the texture when we view it from far away such that the texels are considerably smaller than the pixels in the final image.
- If we are just point sampling our pixels, we can get serious aliasing problems in these situations.
- We saw how to address these problems by pixel antialiasing and supersampling in a previous lecture, but these methods are expensive as they involve many additional rays.
- *Texture filtering* is a process to address the aliasing issues.
- It mainly applies to minification, but can also be used for antialiasing in magnification situations as well.
Minification Filters

• Ideally, we would look at all of the texels that fall within a single pixel and blend them somehow to get our final color
• This would be expensive, mainly due to memory access cost, and would get worse the farther we are from the texture
• A variety of minification techniques have been proposed over the years (and new ones still show up)
• One of the most popular methods is known as mipmapping
Mipmapping

- Mipmapping is a popular technique for texture filtering in realtime and hardware rendering applications.
- It’s quality, however, is limited and generally not preferred for high quality rendering situations, but it can still be used in combination with other techniques.
- In addition to storing the texture image itself, several *mipmaps* are precomputed and stored.
- Each mipmap is a scaled down version of the original image, computed with a medium to high quality scaling algorithm.
- Usually, each mipmap is half the resolution of the previous image in both $u$ and $v$.
- For example, if we have a 512x512 texture, we would store up to 8 mipmaps from 256x256, 128x128, 64x64, down to 1x1.
- Altogether, this adds $1/3$ extra memory per texture ($1/4 + 1/16 + 1/64... = 1/3$).
- Usually, texture have resolutions in powers of 2. If they don’t, the first mipmap (the highest res) is usually the original resolution rounded down to the nearest power of 2 instead of being half the original res. This causes a slight memory penalty.
- Non-square textures are no problem, especially if they have power of 2 resolutions in both x and y. For example, an 16x4 texture would have mipmaps of 8x2, 4x1, 2x1, and 1x1.
Mipmapping

• When rendering a mipmapped texture, we still compute an interpolated $u,v$ per pixel, as before
• Instead of just using this to do a point sampled (or bilinear...) lookup of the texture, we first determine the appropriate mipmap to use
• Once we have the right mipmap, we can either point sample or bilinear sample it
• If we want to get fancy, we can also perform a blend between the 2 nearest mipmaps instead of just picking the 1 nearest. With this method, we perform a bilinear sample of both mips and then do a linear blend of those. This is called trilinear mipmapming and is the preferred method. It requires 8 texels to be sampled per pixel
**Mipmapping Limitations**

- Mipmapping is a reasonable compromise for realtime rendering in terms of performance and quality.
- However, it tends to have visual quality problems in certain situations.
- Overall, it tends to blur things a little much, especially as the textured surfaces appear more edge-on to the camera.
- The main reason for its problems comes from the fact that the mipmaps themselves are generated using a uniform aspect filter.
- This means that if a roughly 10x10 region of texels maps to a single pixel, we should be fine, as we would blend between the 16x16 and 8x8 mipmaps.
- However, if a roughly 10x3 region maps to a single pixel, the mip selector will generally choose the worst of the two dimensions to be safe, and so will blend between the 4x4 and 2x2 mipmaps.
- There are some extensions to the standard mipmapping algorithm to support *anisotropic mipmapping*, which attempts to improve the handling of situations with non-uniform stretching.
Elliptical Weighted Average

• Perhaps the best quality texture filtering technique is the elliptical weighted average
• Think of each camera ray actually representing a cone with a slight divergence angle to fit around the pixel
• When this cone hits a flat surface, it projects to an ellipse on the surface
• We can essentially find this ellipse back in the texture space and then use it to perform a weighted average of the pixels in the ellipse
• The average is weighted according to a Gaussian type distribution, with a high weight on texels at the center of the ellipse dropping to a low weight for texels on the perimeter
• It mimics the effect of antialiasing the pixels with a Gaussian distribution, however can be done with a single ray per pixel
Elliptical Weighted Average

1 ray, point sample

1 ray, EWA sample

100 rays, jittered Gaussian distribution
High Quality Texture Filters

- The previous slide demonstrates the power of using proper texture filtering
- However, one has to remember that the previous slide only showed texture filtering and no other type of antialiasing
- In other words, we would still have to shoot multiple rays per pixel to avoid jagged edges and aliasing problems with small triangles
- If we are going to shoot many camera rays to achieve antialiasing and other effects, then why bother to use high quality texture filtering?
- It is still helpful to be able to reduce the number of rays whenever possible. There may be situations where high quality texture filtering doesn’t improve anything, but these are generally situations when you are shooting so many rays that it is going to be slow no matter what you do
Normal & Displacement Mapping
Mapping

• Texture mapping usually refers specifically to mapping the colors of an image onto a surface
• However, there are various other related forms of mappings that sometimes fall into the category of texture mapping
• Some popular options are *bump mapping*, *normal mapping*, and *displacement mapping*
• Plus, we should remember that we can map any material property we want (such as roughness, transparency, shininess, anisotropy, etc.) and refer to this as more general *attribute mapping*
Bump & Normal Mapping

- The concepts of *bump mapping* and *normal mapping* are very closely related and are really just two variations on the same idea.
- The idea is to render a triangle as a flat surface, but to vary the normal across it, giving the appearance of a bumpy surface.
- It is related to the Phong smooth shading technique of interpolating normals across a triangle to fake the appearance of a smooth surface.
- With bump mapping, the map itself is a 2D image of height values representing a hypothetical height variance on the surface. The Phong interpolated normals further adjusted by the gradient of the height data.
- With normal mapping, the map is a 2D image of 3D normals. This can be encoded into a 24-bit RGB image (where each color component is gives +/- 7 bits of precision, which is usually sufficient for a normal map).
- Normal mapping is more common in modern use, as it is more flexible and should be just as fast. Plus normal maps can easily be derived from a bump map, but the opposite is not true. This means that a normal mapping implementation can easily accommodate both bump and normal maps.
Displacement Mapping

- Normal and bump mapping involve tweaking the normals without modifying the underlying triangle geometry and are really just hacks
- A better (but more expensive) approach is to modify the actual geometry by using displacement mapping
- With displacement mapping, one uses a 2D image of either height values or full 3D displacement values
- The height offsets or displacements are used to displace the actual triangle vertices
- This involves tessellating the original surface down to a whole bunch of tiny triangles, typically around the size of a single pixel or smaller
Displacement Mapping
Displacement Mapping

- To use displacement mapping on a simple cube (12 triangles), we might first have to dice it up into a million small triangles.
- As we do this, we apply the displacements and recompute the normals based on the actual geometry.
- This entire process would probably be done before the actual rendering begins, so that the new geometry can be placed into a spatial data structure.
- This is the *tessellation* phase of a renderer, that also includes the triangulation of curved surfaces like NURBS and subdivision surfaces.
- Tessellation is usually done adaptively so that surfaces are triangulated down to the size of pixels or smaller. This is therefore a view-dependent process. It could also be triangulated based on world-space tolerances and thus lead to a view-independent tessellation.
- We will discuss tessellation and displacement mapping in more detail in a future lecture.
Procedural Texturing
Procedural Texturing

- Procedural texturing is the process of generating texture data algorithmically at render time, rather than getting texture data from a previously stored image.
- This allows texture maps to be defined as procedural functions.
- This is nice for things like organic, bumpy, irregular surfaces that can be defined by non-repeating functions.
- This reduces visual tiling artifacts that can be visible when texture maps are repeated over and over.
- Procedural texturing also allows for effectively unlimited resolution, as details are described by mathematical functions rather than pixels of a fixed size.
Procedural Texturing

• Procedural texturing is an entire topic itself and we will possibly do an entire lecture on it later in the quarter if the schedule permits
• For today, we will just cover a couple basic concepts
Many procedural textures are built up from *noise* functions

A *noise* function is an n-dimensional function that generates a random pattern with some adjustable properties

There are many popular noise functions that generate different patterns
Perlin Noise

• Ken Perlin introduced the concept of procedural noise to the computer graphics community in 1985, and showed a wide variety of applications.

• The classic Perlin noise is essentially a grid of random intensities that is smoothly interpolated (bicubic).
Perlin Turbulence

• By scaling and adding several different sized copies of the noise function, you can create more complex turbulence functions

• These can be combined in any number of ways to generate a final color or displacement
Worley Noise

• Another popular choice is the Worley noise function

• This one generates a cellular pattern that can be applied to a variety of applications
Procedural Shaders
Solid Textures

• It is also possible to use volumetric or *solid textures*
• Instead of a 2D image representing a colored surface, a solid texture represents the coloring of a 3D space
• Solid textures are often procedural, because they would require a lot of memory to explicitly store a 3D bitmap
• A common example of solid texture is marble or granite. These can be described by various mathematical functions that take a position in space and return a color
• Texturing a model with a marble solid texture can achieve a similar effect to having carved the model out of a solid piece of marble