# Texture Mapping

Steve Rotenberg

CSE168: Rendering Algorithms

UCSD, Spring 2014

# 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*

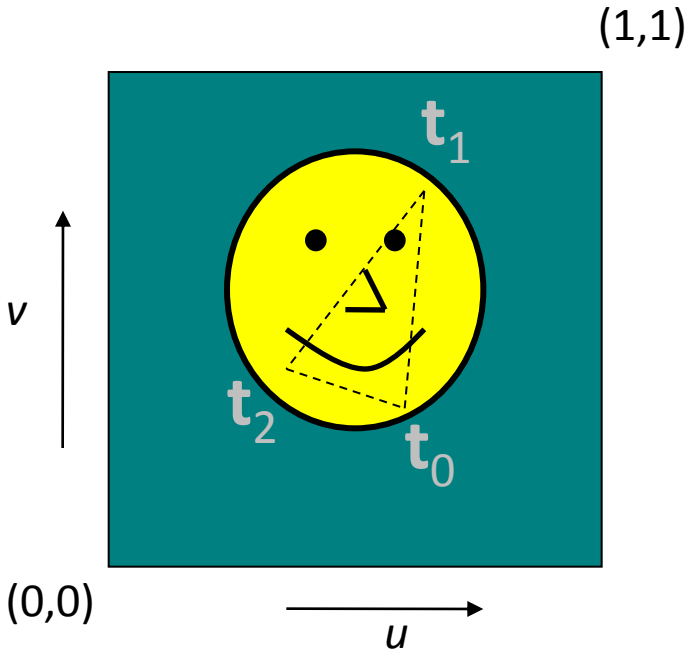
# Texture Space

- 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

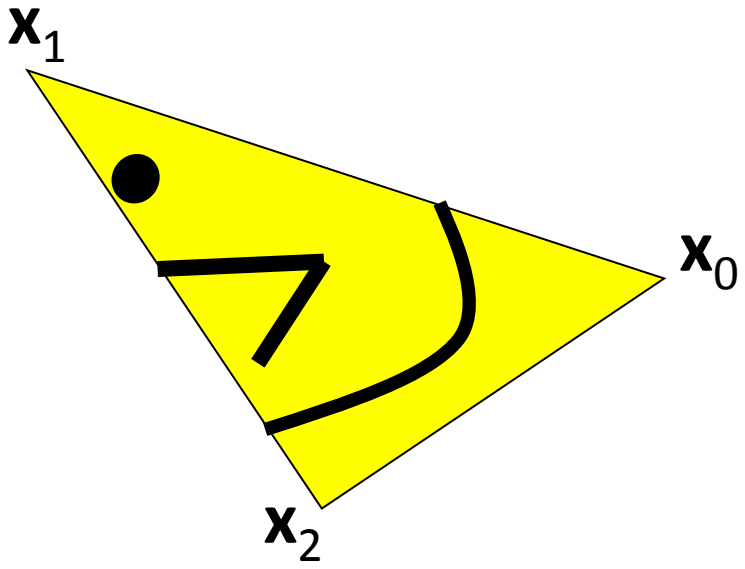
# Texture Coordinates

- 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

$$u=(1-\alpha-\beta)u_0 + \alpha u_1 + \beta u_2$$
$$v=(1-\alpha-\beta)v_0 + \alpha v_1 + \beta v_2$$

- 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

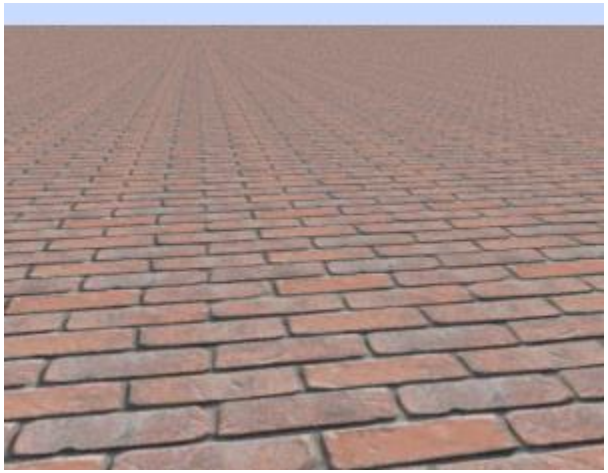
# Texture Wrapping

# 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 *tile* or *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 *wrapping modes* for each texture
- In fact, it is useful to be able to control the wrapping behavior independently in the *u* and *v* directions

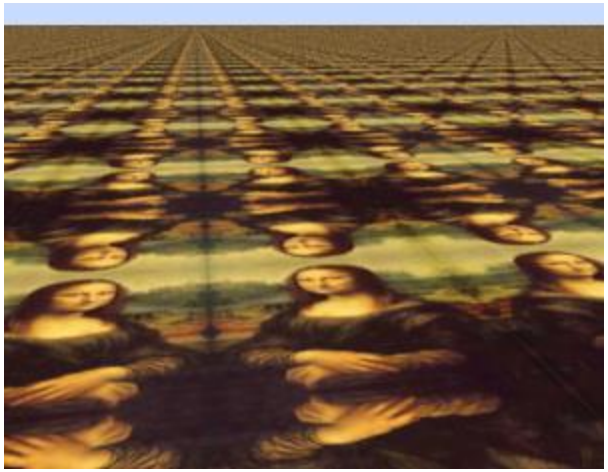


# Wrapping Modes

- 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

# Wrapping

```
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;
}
```

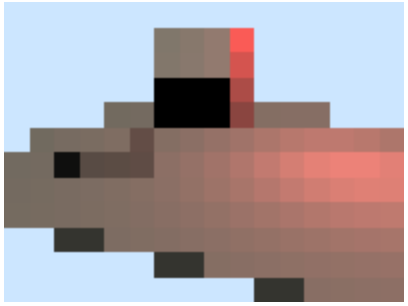
# Texture Sampling

# 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

# TextureMap Class

```
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 mipmapping* and is the preferred method. It requires 8 texels to be sampled per pixel

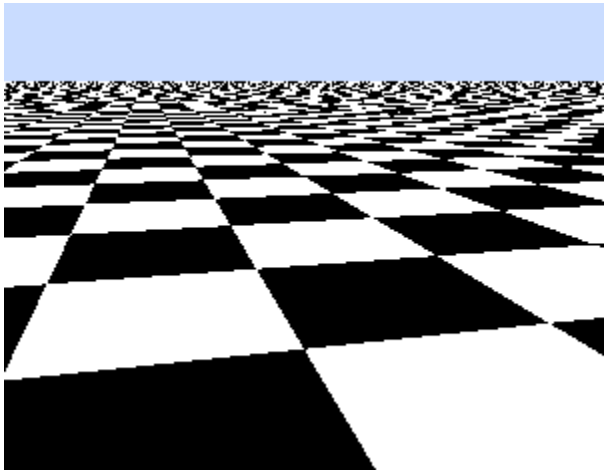
# Mipmapping Limitations

- 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

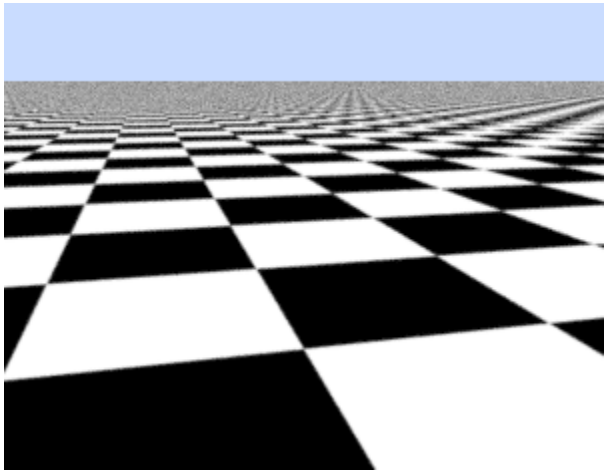
# Elliptical Weighted Average

- 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

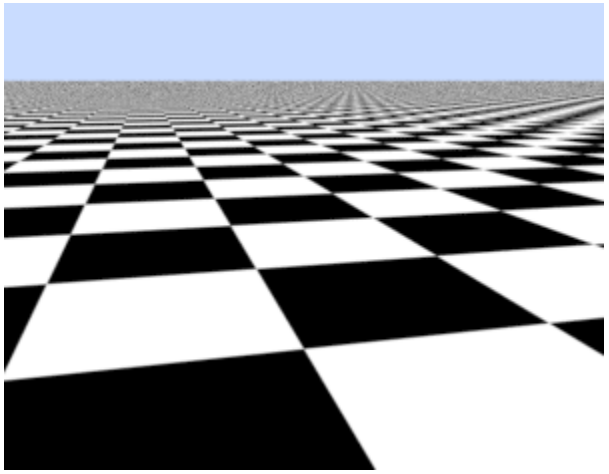
# Elliptical Weighted Average



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*

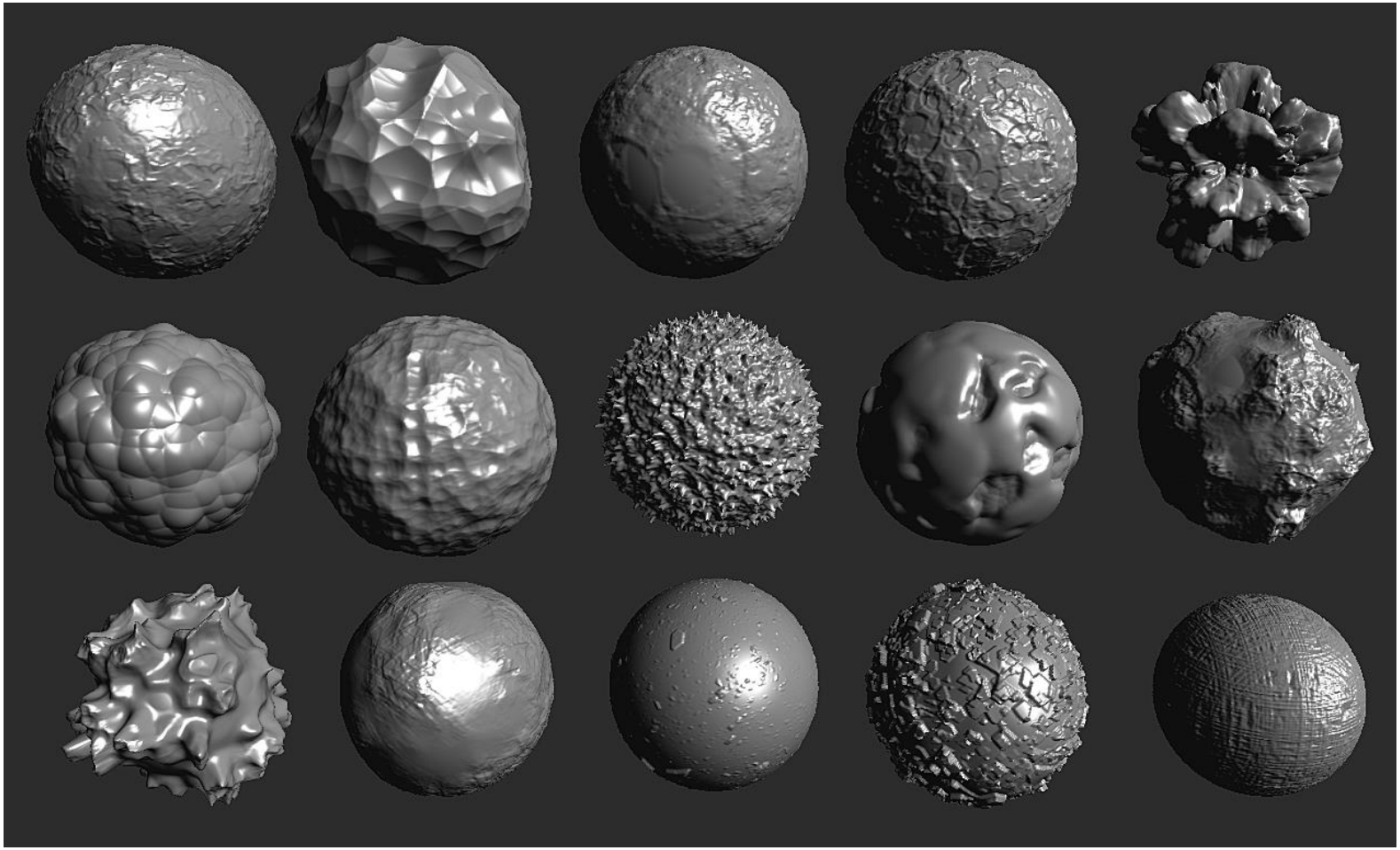
# Bump & Normal Mapping

- 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

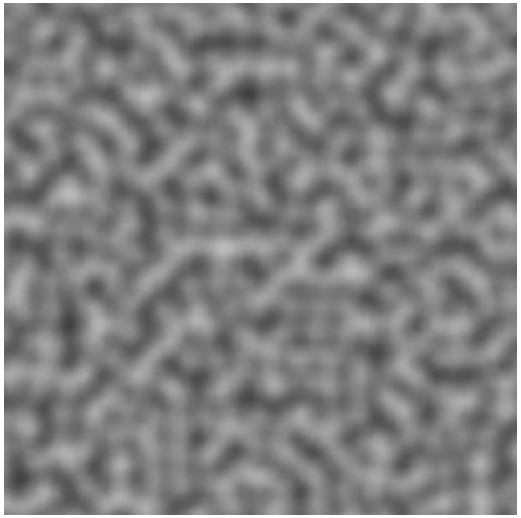
- Procedural texturing is an entire topic itself and we will possibly do an entire lecture on it later in the quarter if the schedule permits
- For today, we will just cover a couple basic concepts

# Noise

- 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

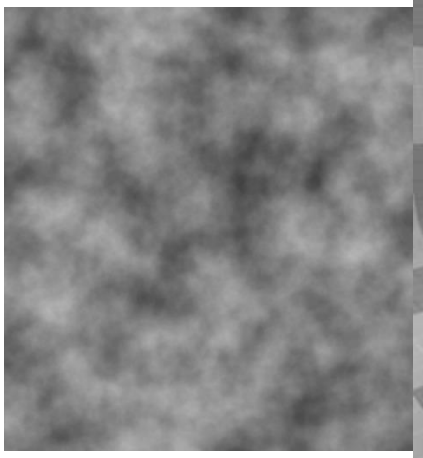
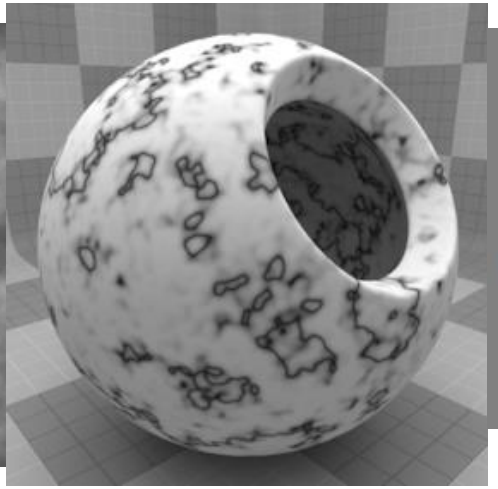
# Perlin Noise

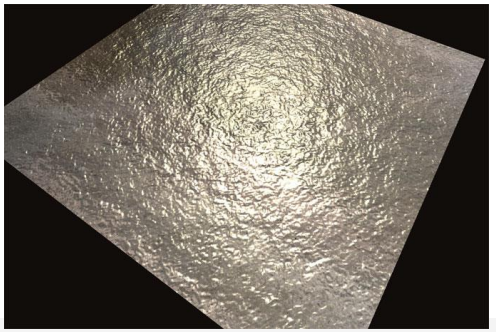
- Ken Perlin introduced the concept of procedural noise to the computer graphics community in 1985, and showed a wide variety of applications
- The classic Perlin noise is essentially a grid of random intensities that is smoothly interpolated (bicubic)



# Perlin Turbulence

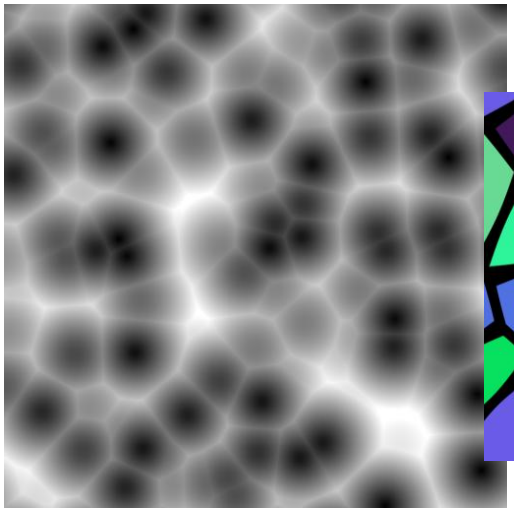
- By scaling and adding several different sized copies of the noise function, you can create more complex *turbulence* functions
- These can be combined in any number of ways to generate a final color or displacement

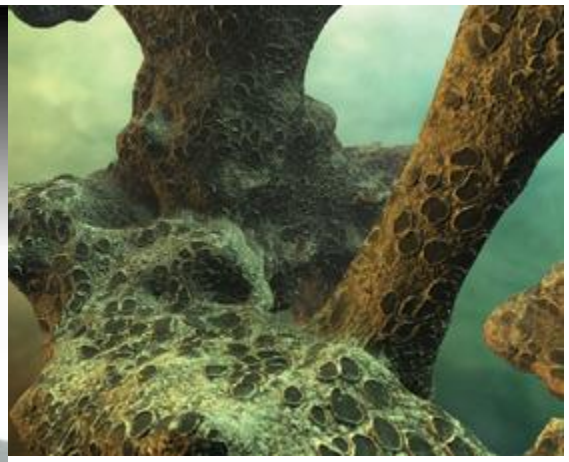





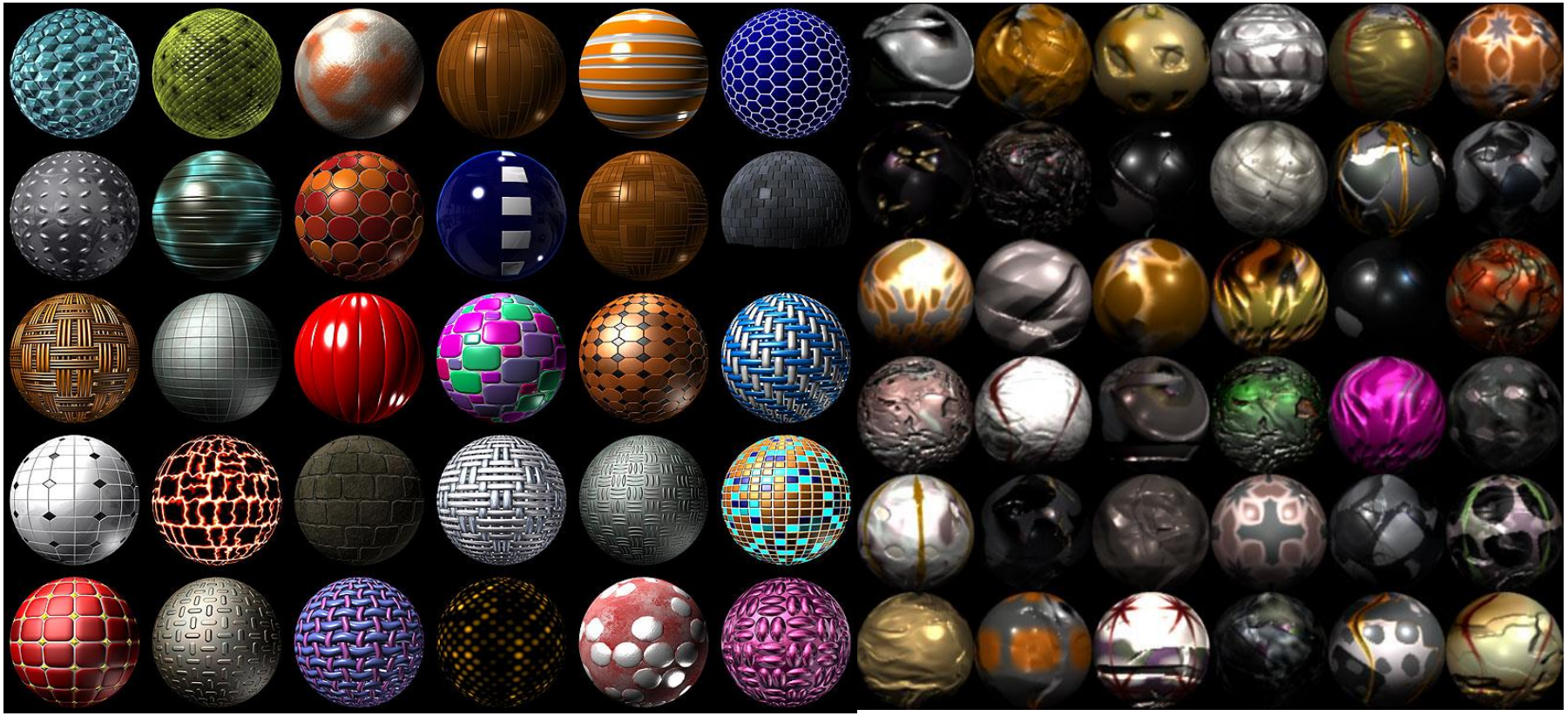

# Worley Noise

- 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