《Fundamentals of Computer Graphics》

Lecture 8、Ray tracing

Part 3: Anti-aliasing

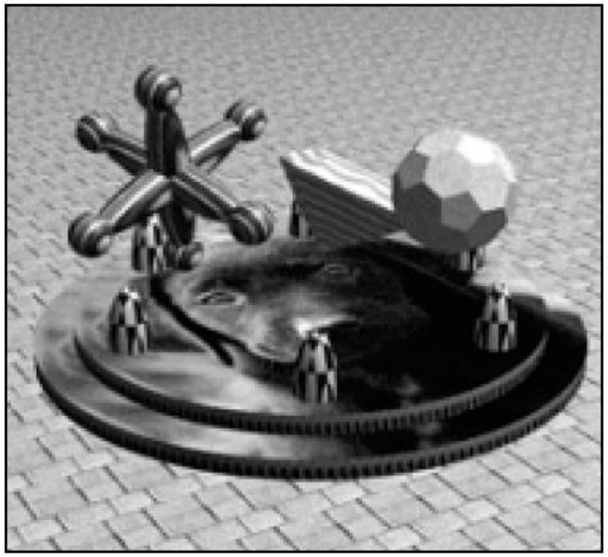
Yong-Jin Liu

liuyongjin@tsinghua.edu.cn

Material by S.M.Lea (UNC)

Adding Surface Texture

- We want to incorporate texturing into a ray tracer. Two principal kinds of texture are used:
  - Image Texture: A 2D image is pasted onto each surface of the object;
  - Solid Texture: The object is considered to be carved out of a block of some material which is textured (e.g., marble). The ray tracer reveals the color of the texture at each surface point on the object.
- Example: a ray-traced scene with several textures.



Adding Solid Surface Texture

- We view an object as carved from some textured material such as marble or wood.
- The texture is represented by a function *texture*(*x*, *y*, *z*) that produces an (*r*, *g*, *b*) color value at every point in space.
- Think of this texture as a color or inkiness that varies with position; if you look (with x-ray vision) at different points (*x*, *y, z*) you see different colors.
- When an object of some shape is defined in this space, and all the material outside of the shape is chipped away to reveal the object’s surface, the point (*x*, *y*, *z*) on the surface is revealed and has the specified texture.

Example: A Solid 3D Checkerboard

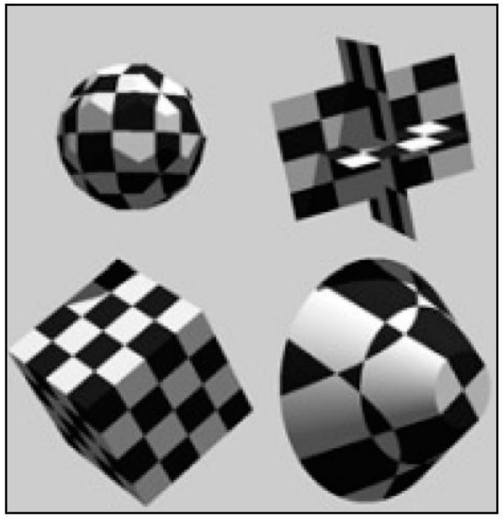
- Imagine a 3D checkerboard made up of alternating red and black cubelets stacked up throughout all of space.
- We position one of the cubelets with a vertex at (0,0,0) and size it so that its diagonally opposite vertex lies at point $S = (S.\mathrm{x}, S.\mathrm{y}, S.\mathrm{z})$.
- All other cubes have this same size (a width of S.x, a height of S.y, etc.) and they are placed adjacent to one another in all three dimensions.

Example: A Solid 3D Checkerboard (2)

- To write an expression for such a checkerboard texture, add together the integer parts of *x/S.x, y/S.y*, and *z/S.z* and reduce the sum modulo 2:
- *jump*(*x*, *y*, *z*) = ((*int*)(*x/S.x*) + (*int*)(*y/S.y*) + (*int*)(*z/S.z*)) % 2
- We then set *texture*(*x*, *y*, *z*) to return black if *jump*() is 0, and red if *jump*() is 1.
- The color of the texture sets the diffuse reflection coefficient of the surface: the color of the material is the color of the texture. The diffuse component varies at different positions relative to the light source, and the Phong specular component is the color of the light source.

Example: A Solid 3D Checkerboard (2)

- Notice that the sphere and cube are clearly made up of solid cubelets.



Ray tracing Objects with Solid Textures

- There are various ways that the texture can alter the light coming from a surface point:
- 1). The light can be set equal to *texture*() itself, as if the object were glowing with that color.
- 2). The texture *tex* can *modulate* the ambient and diffuse reflection coefficients, so that

$$I = tex(x, y, z) \times \left(I_a \rho_a + I_s \rho_d \left(u_s \bullet u_n \right) + I_s \rho_s \left(u_b \bullet u_n \right)^f \right)$$

- where *texture*(*x*, *y*, *z*) is evaluated at the hit point (*x*, *y*, *z*) of the ray (most common use of texture).

Ray tracing Objects with Solid Textures (2)

- The surface mimics the color fluctuations in *texture*().
- Here the specular highlight has the color of the source and is not affected by the texture. This makes the textured object appear to be shiny, as if made of plastic.
- The hit point (*x*, *y*, *z*) used could be either in generic coordinates or in world coordinates.
- Usually it is in **generic** coordinates, in which case the object carries the texture along with it when it is rotated or moved to its final position in the scene.
  - In an animation where the object is rotating or moving from frame to frame, the texture will be solidly attached to the object.

Ray-tracing Objects with Solid Textures (3)

- If, on the other hand, (x, y, z) is the world coordinate version, the texture is fixed in space.
- Now when the object rotates or moves in an animation the texture will sweep over it, making it appear to be carved out of new material at each new position.
- This can produce an interesting visual effect.

Wood Texture

- The grain in wood is due to concentric cylinders of varying color, corresponding to the rings seen where a log is cut. As the distance of points from some axis varies, the function jumps back and forth between two values. This can be simulated with a modulo function:
- rings(r) = ((int)r) % 2; The value of *rings* jumps between 0 and 1 as r increases from 0. The texture can be made to jump between two preset values, say *D* and *D* + *A*, using: simple_wood(x, y, z) = D + A * rings(r/M));
- producing rings of thickness *M* concentric about the *z*-axis*.*

$$r = \sqrt{x^2 + y^2}$$

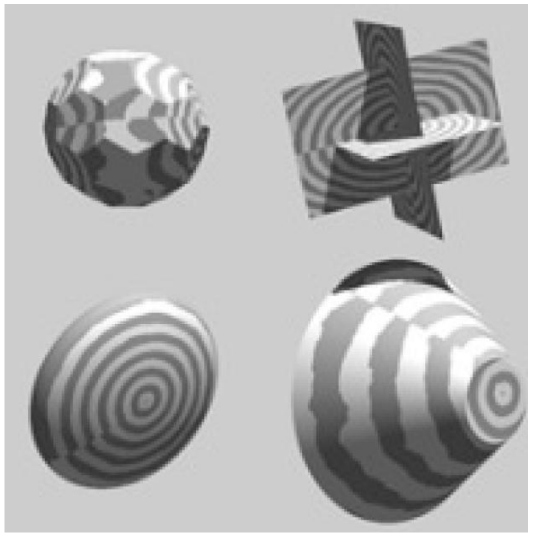
Wood Texture (2)

- Things get more interesting if we wobble, skew, and rotate the rings.
- To wobble the rings, add a component that varies with azimuth θ about the *z*-axis: rings(r/M + Ksin(θ/N)).
- To add a twist, use rings(r/M + Ksin(θ/N + Bz)).
- To tilt the grain to be concentric about some axis other than the *z*-axis, apply a rotation before evaluating *r* and θ. T(x, y, z) = (x', y', z') [a rotational transformation] and use

$$r = \sqrt{x'^2 + y'^2}$$

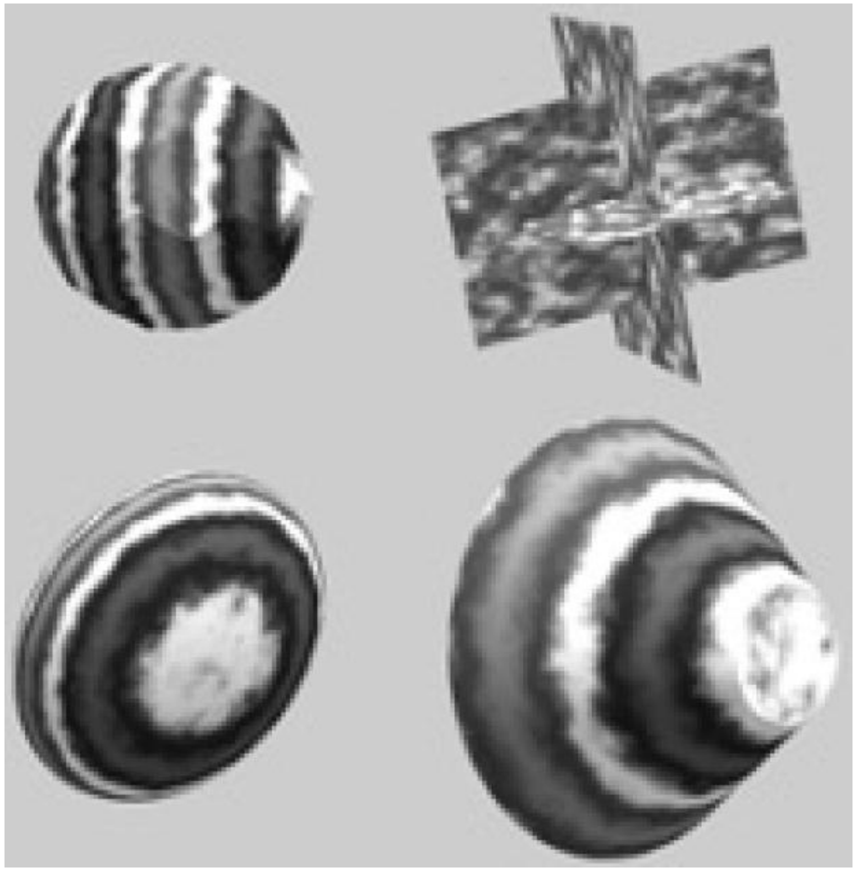
Wood Texture (3)

- Examples: the figure shows some objects that appear to be carved out of wood, having wood grain defined in these ways.



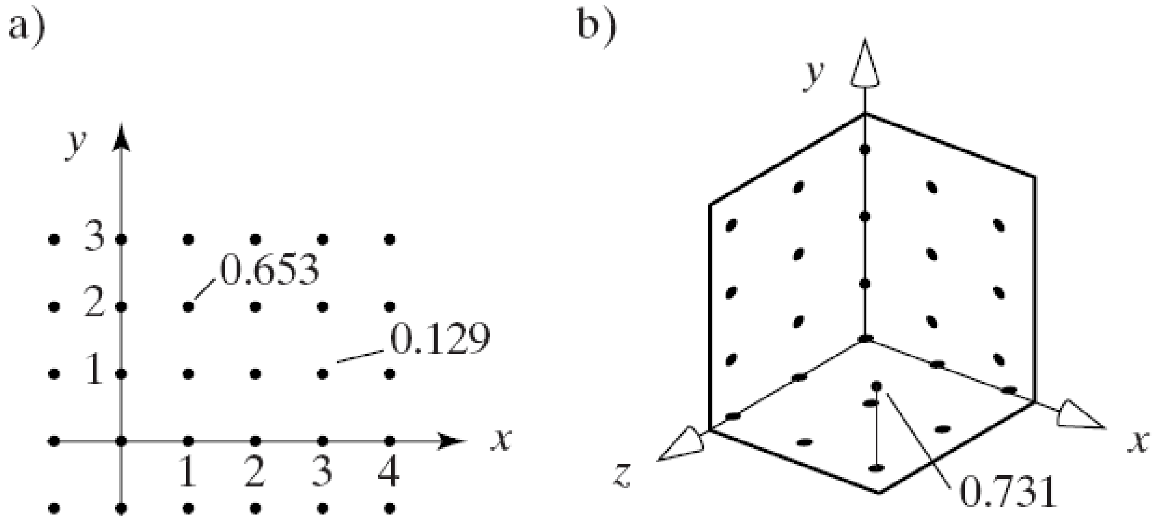
3D Noise and Marble Texture

- The grain in materials such as marble is quite chaotic. There are turbulent rivulets of dark material coursing through the stone, with random whirls and blotches, as if the stone was formed out of some violently stirred molten material.



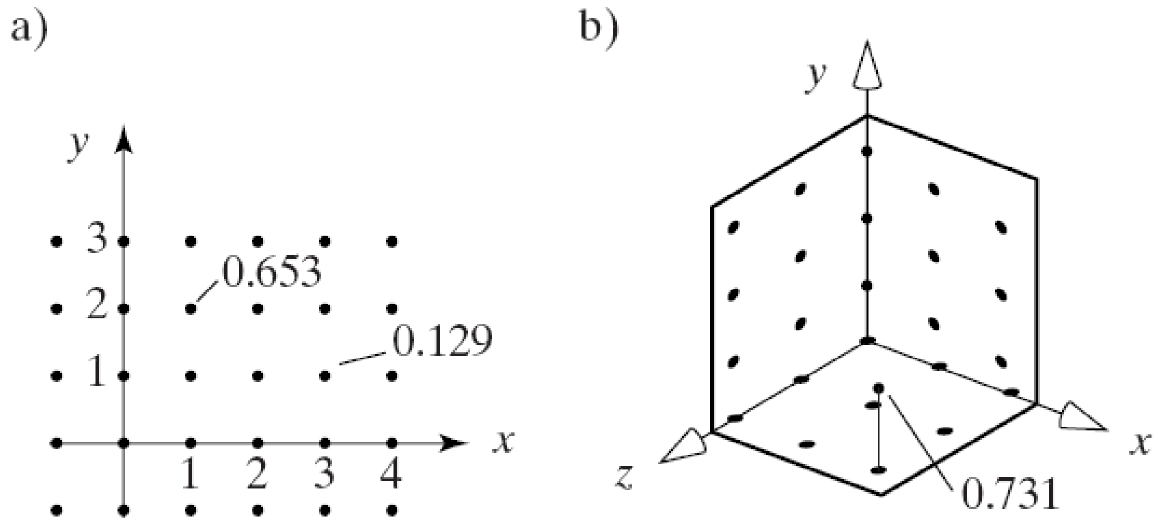
Marble Texture (2)

- We can simulate turbulence by a noise function that produces an apparently random value at each point (*x*, *y*, *z*) in space and then stirring it up in a well-controlled way to give the appearance of turbulence.
- The noise field itself is easy to program. Imagine defining a random value at each *integer* position in space, that is, at (*x*, *y*, *z*) = (*i*, *j*, *k*) for every combination of integers *i*, *j*, and *k*. Such an arrangement of points is called an **integer lattice**.



Marble Texture (3)

- The figure shows a 2D version, where points in a 2D integer lattice are labeled with noise values between 0 and 1.
- For instance, point (1, 2) has noise value 0.653 and point (3, 1) has value 0.129.
- Part b shows a 3D integer lattice. Visualize every integer point having some fixed noise value, such as 0.7341 at (2, 2, 1).



Marble Texture (4)

- It is simplest to generate each noise value each time it is needed.
- For this approach, we need a function, say float latticeNoise(int i,int j,int k), that returns an apparently random value given the integers *i*, *j*, and *k* that specify the position in the lattice.
- The function must be efficient and completely repeatable: it always returns the same noise value for a given (*i*, *j*, *k*).

Marble Texture (5)

- We set up a fixed array, say noiseTable[ ], of pseudorandom noise values in an initialization step. Arrays of length 256 have been found to be quite adequate, so we use this length.
- The main function, latticeNoise(i, j, k) simply indexes into noiseTable[ ] in a repeatable way.
- To ensure that there is little or no pattern in the noise values as i, j, or k vary, the indexing function effectively scrambles or hashes the (*i*, *j*, *k*) combination into a value between 0 and 255. This is easy to accomplish using a second array, index[ ], that contains the values 0 through 255 randomly permuted.

Marble Texture (6)

- We define two macros
- #define PERM(x) index[(x) & 255]
- #define INDEX(ix, iy, iz) PERM((ix) + PERM((iy) + PERM(iz)))
- The PERM macro takes an integer value of *x* and performs a bitwise AND operation on it with 255, effectively retaining only its low order 8 bits, so it is hashed into a value between 0 and 255. The value of PERM is therefore one of the values selected from the index array.
- The INDEX macro uses PERM to dip into the index array three times, in each case choosing an element of index[ ] based on one of the values ix, iy, or iz. Note that this is a repeatable and efficient operation, and that there is plenty of scrambling taking place.

Marble Texture (7)

- The latticeNoise() function then is simply:

```
float latticeNoise(int i, int j, int k)
{    return noiseTable[INDEX(i,j,k)];  }
```

- It is convenient to encapsulate the noise functions and data into a Noise class.
- The class declaration is given in Fig. 12.41.

Marble Texture (8)

- marble() returns a position-dependent value of brightness between 0 and 1 that mimics the rivulets of dark and light stone in marble.
- It would be used to generate a greenish marble by constructing a noise object at the start of the ray tracing with:

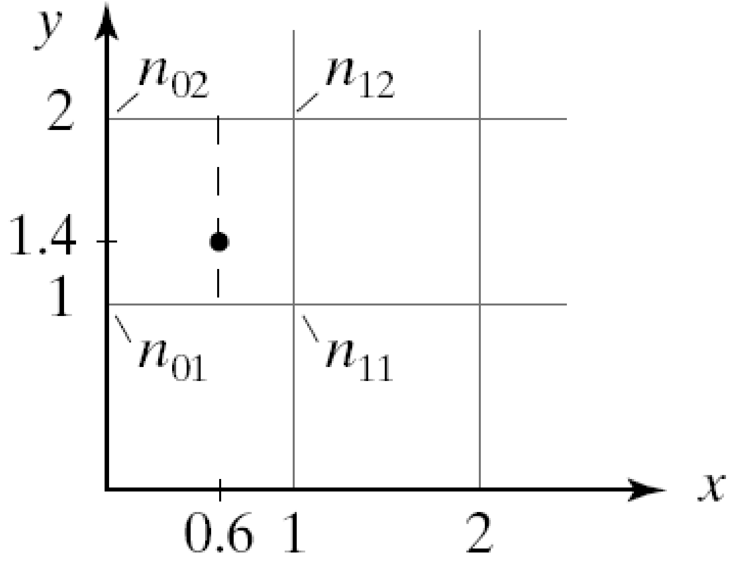
  Noise n; // create and construct a noise object
- and thereafter obtaining the *texture*(*x*, *y*, *z*) value at each point (*x*, *y*, *z*) desired as simply n.marble(x, y, z);
- The method marble() uses noise() and turbulence(), also developed shortly, as well as the helper function latticeNoise().
- The class constructor creates and fills the arrays noiseTable[ ] and index[ ].

Marble Texture (9)

- The values in noiseTable[ ] are created using the standard C function rand(), scaling the values to lie between 0.0 and 1.0.
- The array index[ ] is first loaded with values 0 to 255 in order, and then this array is shuffled by swapping each of its elements in turn with some randomly-selected element.
- With the function latticeNoise() in hand that produces random values at integer lattice points, we want a function *noise*(*x*, *y*, *z*) that produces random-like values at points in between, in fact at *any* point in space. We also want the noise to vary smoothly as *x*, *y*, and *z* vary.

Marble Texture (10)

- Simple linear interpolation between the lattice values gives acceptable results. The figure shows interpolation in 2D; the 3D case is similar. Here we wish to evaluate noise at (x, y) = (0.6, 1.4), given the noise values on the four surrounding corners of the lattice.
- First interpolate in x along $y = 1$ and $y = 2$ to form the values $n(0.6, 1) = lerp(0.6, n_{01}, n_{11})$ and $n(0.6, 2) = lerp(0.6, n_{02}, n_{12})$.

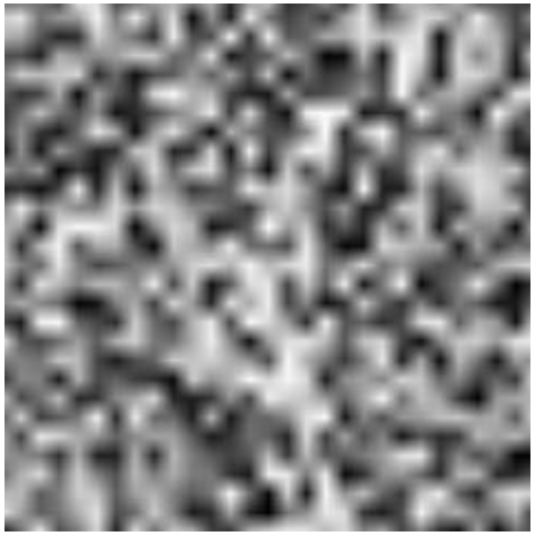


Marble Texture (11)

- Then interpolate these in *y* to form
  n(0.6, 1.4) = *lerp*(0.4, *n*(0.6, 1), *n*(0.6, 2))
- Figure 12.43 shows a possible implementation of the function noise() for the 3D case. It has an extra parameter scale that scales the given 3D point (*x*, *y*, *z*); this will be useful when we are creating turbulence.
- The scaled point is first offset by 1000 in *x*, *y*, and *z* so that all components will be positive. (Since the lattice values are random anyway, this shift doesn't change the statistical nature of the noise generated.)
- Then noise values are generated at the eight lattice vertices that surround the point. Finally seven *lerp*'s are used to find the interpolated noise value.

Marble Texture (12)

- The figure shows a plot of the function *noise*(20, *x*, *y*, 0), using black for 0.0 and white for 1.0.
- In the figure, both *x* and *y* range from –1 to 1. Some structure is apparent in the noise field due to the vagaries of the random number generation process, but it is not excessive.



Marble Texture (13)

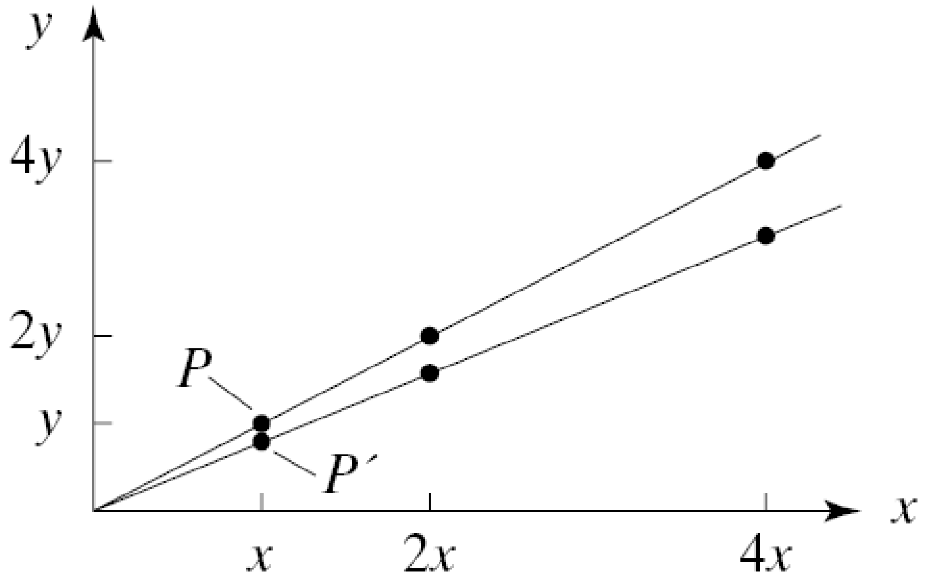
- Perlin described a method for generating more interesting noise. The idea is to mix together several noise components: one that fluctuates slowly as you move slightly through space, one that fluctuates twice as rapidly, one that fluctuates four times as rapidly, etc. The more rapidly varying components are given progressively smaller strengths. The function *turb*():

$$turb(s,x,y,z) = \tfrac{1}{2}noise(s,x,y,z) + \tfrac{1}{4}noise(2s,x,y,z) + \tfrac{1}{8}noise(4s,x,y,z)$$

- adds three such components: each is half as strong, and varies twice as rapidly, as its predecessor. Parameter *s* scales distances just as it does in noise().

Marble Texture (14)

- The figure suggests a way in 2D to see how *turb*() fluctuates. Think of the *xy*-plane covered with fixed values of *noise*(1, *x*, *y*, 0). For each point *P* = (*x*, *y*), *turb*(1, *x*, *y*, 0) sums together three noise values, at the points (*x*, *y*), (2*x*, 2*y*), and (4*x*, 4*y*) shown.

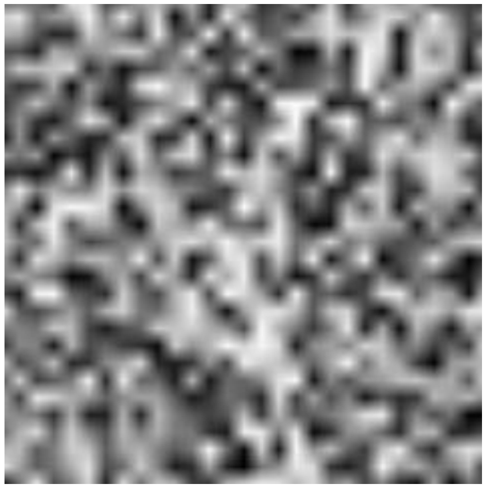


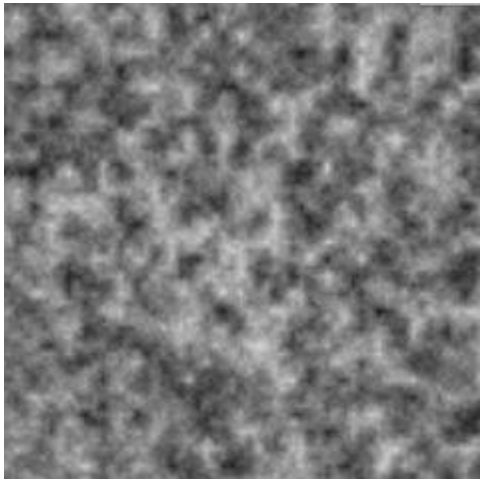
Marble Texture (15)

- At nearby *P*’ the value *noise*(1, *x*’, *y*’, 0) is very similar to *noise*(1, *x*, *y*, 0), but *noise*(2, *x*’, *y*, 0’) will be quite different from *noise*(2, *x*, *y*, 0), and *noise*(4, *x*’, *y*’, 0) will be still more different.
- Features in the first noise component will appear at half size in the next component, and at quarter size in the next.

Marble Texture (16)

- The figure shows a plot of *turb*() (bottom) generated from the noise() field (top), when *M* = 3. The greater level of detail is apparent, and the fluctuations seem softer and more cloud-like.
- The *turb*() values can be used to perturb some attribute of a shape or texture to give it a more realistic appearance.





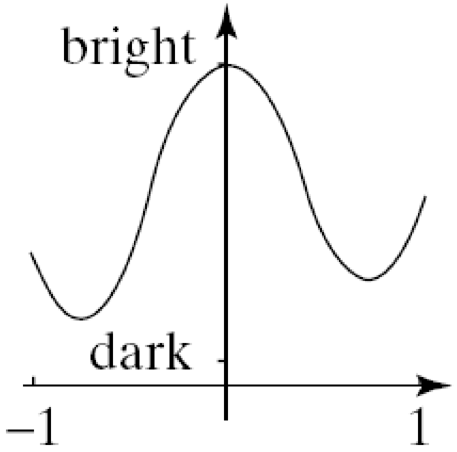
Creating a Marble Texture

- Marble shows veins of dark and light material that have some regularity, but the veins also exhibit some chaotic irregularities.
- We can build up a marble-like 3D texture by giving the veins a smoothly fluctuating behavior in, say, the *z*-direction, and then perturbing it chaotically using *turb*().
- We start with a texture that is constant in *x* and *y* and smoothly varying in *z*: *marble(x, y, z) = undulate(sin(z));*

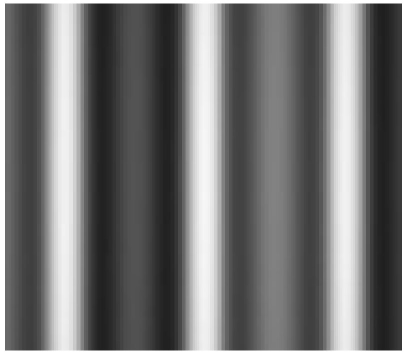
Creating a Marble Texture (2)

- *undulate*() is the spline-shaped function shown in the Figure that varies between some dark and some light value as its argument varies from -1 to 1.
- Using *sin*(*z*) for this argument produces a periodic ripple in *z* that moves back and forth across the spline-curve, once each period, producing the fluctuation in intensity shown.

a)



b)



Creating a Marble Texture (3)

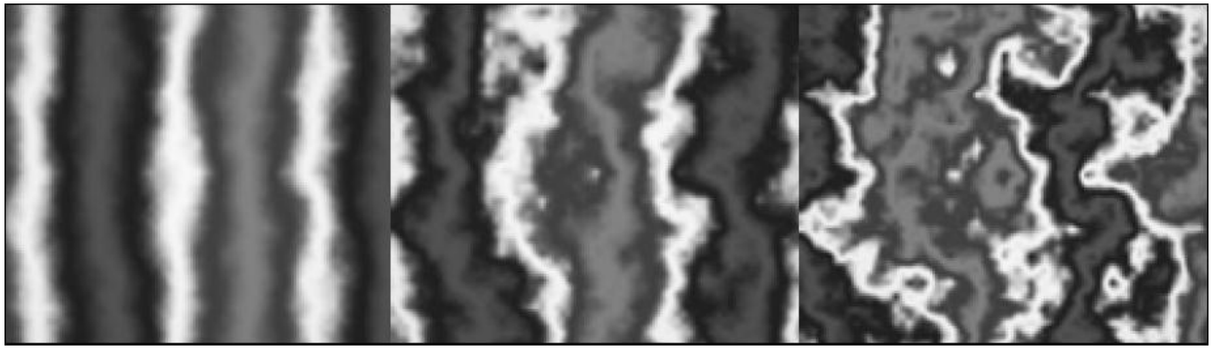
- The vertical veins of color in the marble are of course much too regular. So the argument of *sin*() is modulated with some turbulence: *marble*(*x*, *y*, *z*) = *undulate*(*sin*(*z* + A *turb*(s, *x*, *y*, *z*)));
- The phase of the *sin*() is offset different amounts at different positions in the marble. This produces much more realistic veins.
- Parameter *s* makes the turbulence vary more or less rapidly at different points; parameter *A* changes the amount of the perturbation.

Creating a Marble Texture (4)

- The figure shows the marble texture seen on the face of a cube. The function plotted is

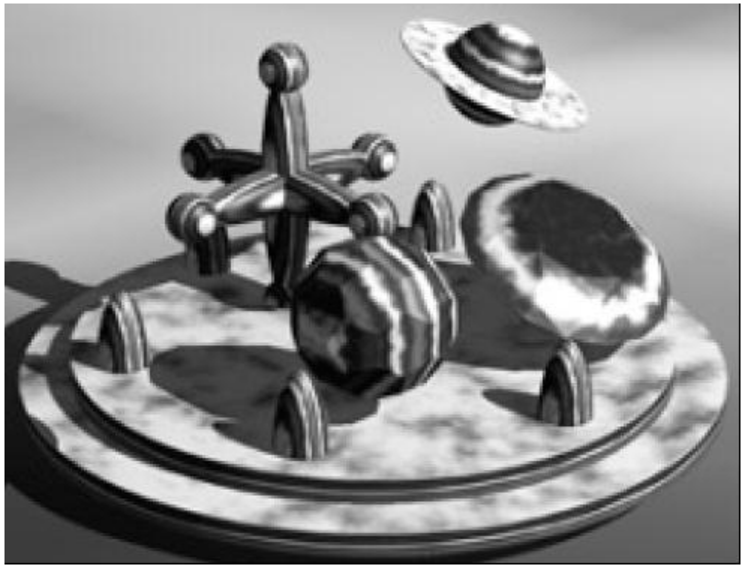
$$g = spline(\sin(2\pi z + A \times turb(5, x, y, z)))$$

- where *z* moves from 0 at the right to 1 at the left, and *y* points upward. The turbulence amplitude A = 1, 3, 6 left to right in the figure.



Creating a Marble Texture (5)

- The value of *marble*() would be used as a reflection coefficient to modulate the amount of light returning from different points in the object hit by different rays.
- It is straightforward to extend *marble*() to return red, green, and blue components for full-color ray tracers.
- The figure shows a ray traced scene containing a number of marble objects. The effect is quite convincing.



Pasting Images on Surfaces

- We examine how to paste images onto arbitrary curved surfaces in a ray tracer.
- The routines for doing so are simple, and results can be excellent, but more execution time is usually required than with OpenGL.
  - Each pixel is computed individually, and no scan line coherence can be exploited.

Pasting Images on Surfaces (2)

- We assume that a 2D texture function *texture*(*u*, *v*) has been defined, as *u* and *v* vary from 0 to 1, that produces an intensity or color at each point (*u*, *v*).
- *texture*(*u*, *v*) might be a procedural texture such as the checkerboard or a Mandelbrot set, or it might be an image texture stored in a pixmap. Suppose the pixmap is arranged as an *N* by *M* array of pixel values txtr[ ][ ].
- Then given values for *u* and *v* between 0 and 1, we can index into the appropriate pixel of txtr simply using txtr[(int)(u/N)] [(int)(v/M)].

Pasting Images on Surfaces (3)

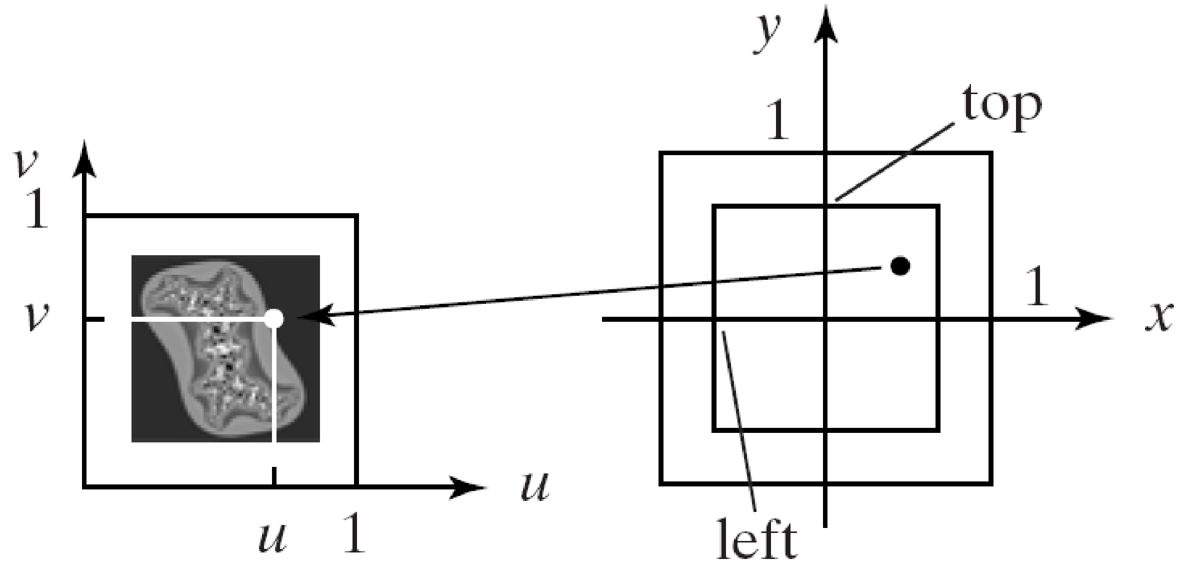
- It is simplest to paste a texture to a generic object rather than the transformed version in scene coordinates.
- The designer associates texture coordinates with coordinates on the generic object in such a way that when the object is transformed into the scene the texture appears correctly and with the proper aspect ratio on the transformed object.
- We need a way to associate points (x, y, z) on a generic object's surface to texture coordinates (u, v). Different mappings are needed for different generic shapes.

Pasting Images on Surfaces (4)

- **Example: Textures for the square and plane.**
- The generic plane is the *xy*-plane, and the generic square lies in this plane.
- There is a natural association between the image plane of *texture*(*u*, *v*) and the generic square or plane.
- The designer simply chooses a window on the plane (*left*, *top*, *right*, *bottom*), and if the ray hits within this window it is easy to compute which point (*u*, *v*) in the texture is to be used.

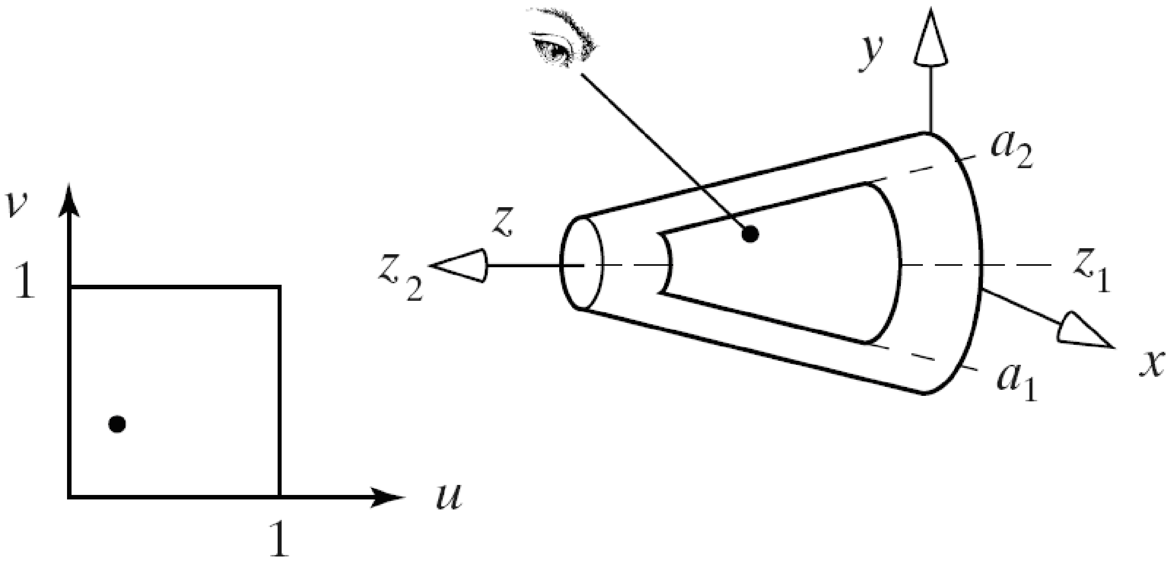
Pasting Images on Surfaces (4)

- Given hit point (*x*, *y*, 0) in generic coordinates, the corresponding texture coordinates are
- u = (x – left)/(right – left)
- v = (y-bottom)/(top-bottom).
- Hit points outside the window are usually set to some fixed value.



Pasting Images on Surfaces (5)

- Wrapping a texture around a generic cylinder is almost as easy.
- The generic tapered cylinder is shown with a window specified on its surface. The window extends in azimuth from a_1 to a_2, and in z from z_1 to z_2.



Pasting Images on Surfaces (6)

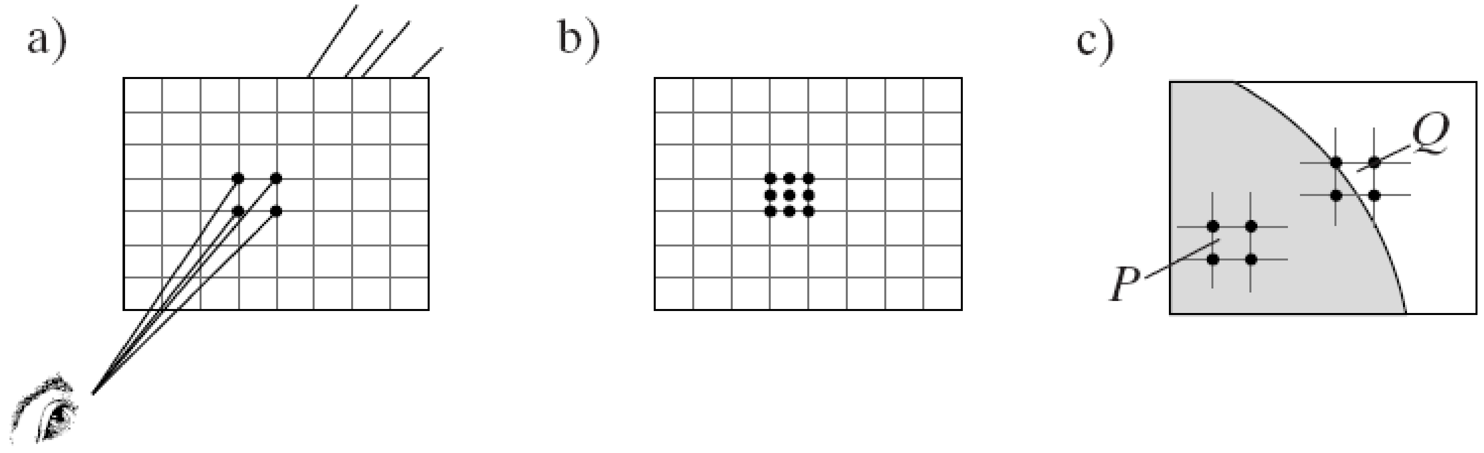
- When a ray hits a cylinder at (*x*, *y*, *z*), we simply compute the azimuth as *θ* = *arctan*(*y*, *x*) and compute the texture coordinates (*u*, *v*) using $u = \frac{\theta - a_1}{a_2 - a_1}, \quad v = \frac{z - z_1}{z_2 - z_1}$

Anti-Aliasing Ray Tracings

- Ray tracing is inherently a point sampling process - taking discrete looks at a scene along individual rays.
- Aliasing effects often degrade the quality of ray traced images and can be reduced by sampling a scene at more points, often called **supersampling**.
- Several rays per pixel are traced into the scene, and the intensities that are returned along the ray are averaged. This is, of course, costly in execution time.

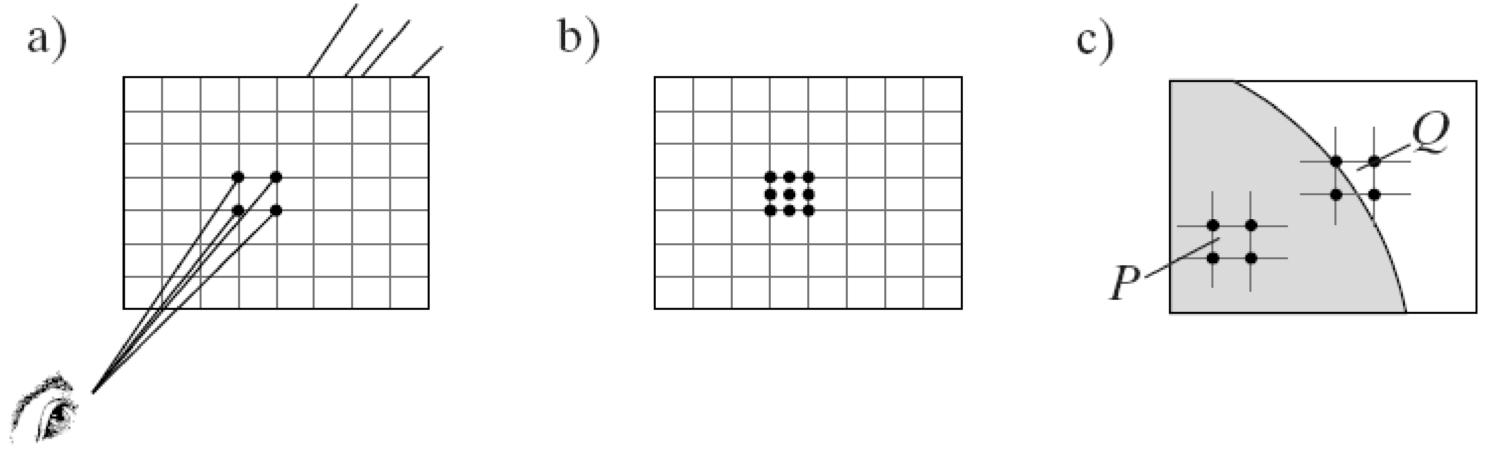
Anti-Aliasing Ray Tracings (2)

- The figure (a) shows a sampling pattern where rays are shot through the corners of the pixels. The final color given to each pixel is the average of the colors found at its four corners.
  - This level of anti-aliasing is easy to do and only costs a little in time.



Anti-Aliasing Ray Tracings (3)

- Supersampling can involve many more rays per pixel. An example of shooting nine rays through parts of a pixel is shown in the figure (b). The light returned along all nine rays is averaged to form the final pixel value.

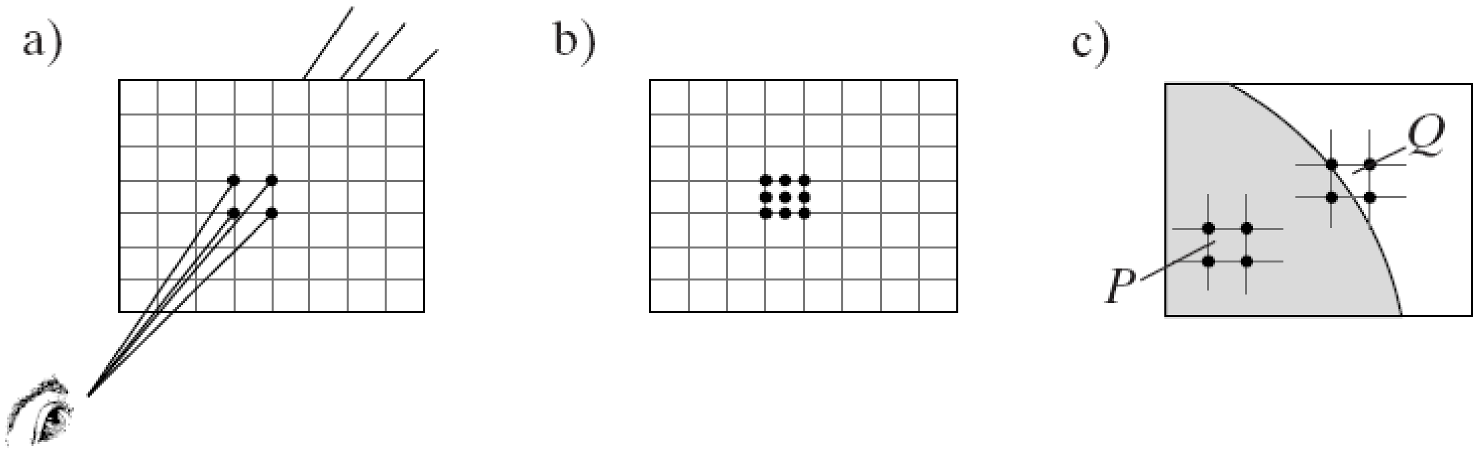


Anti-Aliasing Ray Tracings (4)

- An adaptive procedure shoots more rays into regions where anti-aliasing is needed more (where there are abrupt changes in the image) and does a better job.
- Rays are shot through the four corners of each pixel and the average intensity is formed; the average is compared with the four individual intensities.

Anti-Aliasing Ray Tracings (5)

- If a corner intensity differs too much from the average, the pixel is subdivided into quadrants, and additional rays are sent through the corners of the quadrant (c).



Anti-Aliasing Ray Tracings (6)

- The four rays for pixel *P* return nearly the same intensity because the scene is not changing in that region, but one of the rays for pixel *Q* sends back as intensity very different from the others.
- Thus three new rays are shot through the corners of the lower left-hand quadrant of *Q*, and again the intensity from each is compared with the average.

Anti-Aliasing Ray Tracings (7)

- Subdivision is performed recursively until either a prefixed recursion level has been reached, or the four intensities are sufficiently close to the average to accept the intensity as close enough.
- When this has been done to the four quadrants of a pixel as needed, the final pixel value is formed as a weighted average of the quadrant averages.

Anti-Aliasing Ray Tracings (8)

- A distributed sampling technique uses a form of stochastic sampling.
- A random pattern of rays is shot into the scene for each pixel, and the resulting intensities are averaged.
- For instance, a pixel can be subdivided into a regular 4 by 4 grid.
- But instead of shooting rays exactly through these grid points, a ray is shot through displaced or jittered grid points.
- Jittering the sample points adds a measure of noise to the image but this noise can be less intrusive to the eye than aliasing errors. A smaller grid of samples can be used with jittering than without it.

Anti-Aliasing Ray Tracings (9)

- The figure shows the effect of jittering on a billiard ball.