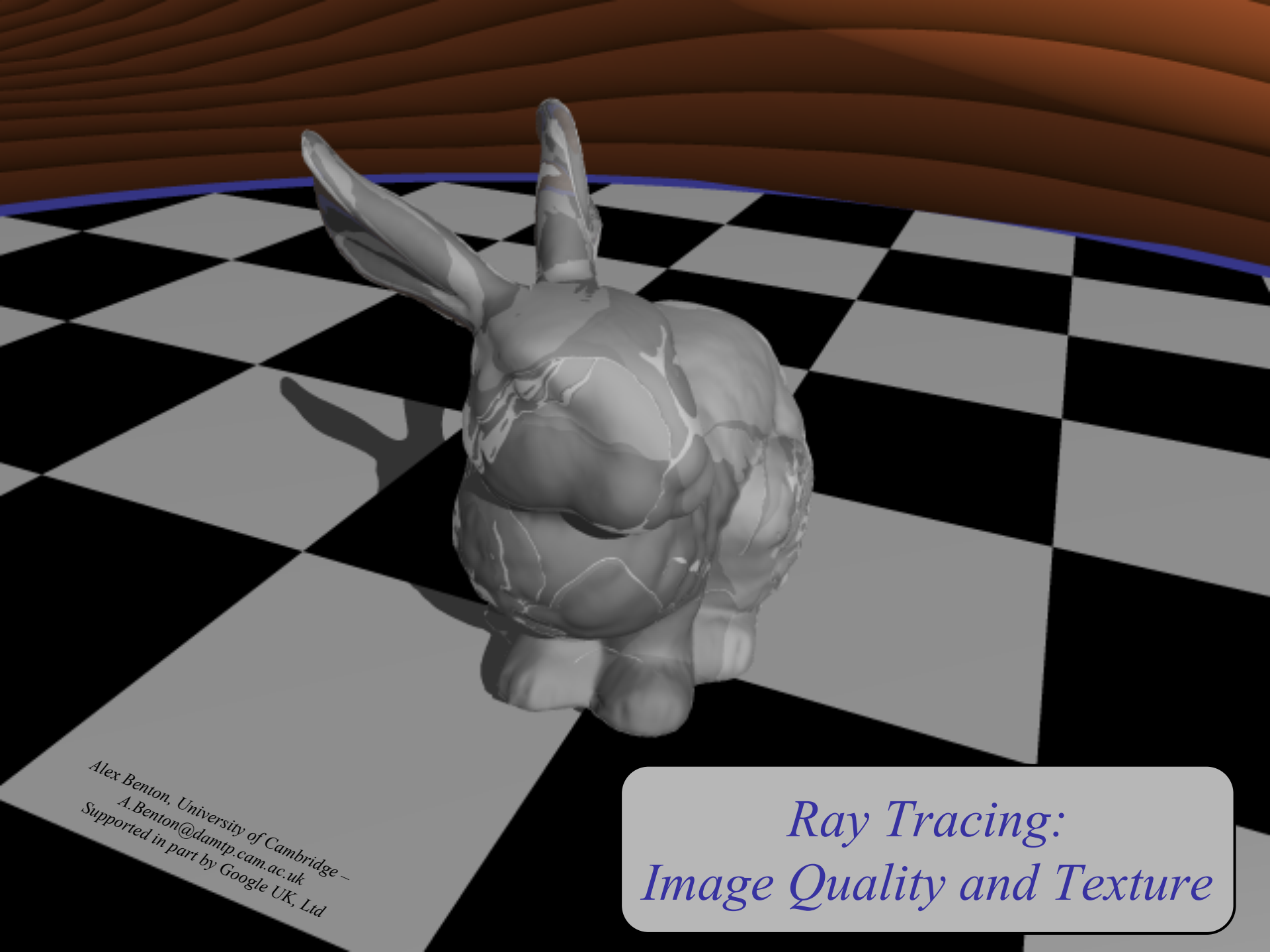


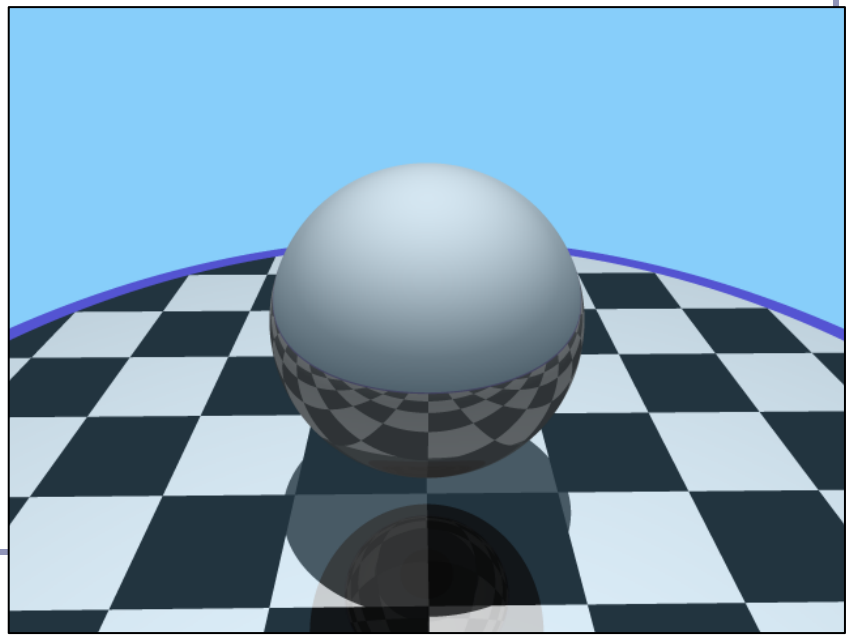
Ray Tracing: Image Quality and Texture

Alex Benton, University of Cambridge –
A.Benton@damtp.cam.ac.uk
Supported in part by Google UK, Ltd

Shadows

To simulate shadows in ray tracing, fire a ray from P towards each light L_i. If the ray hits another object before the light, then discard L_i in the sum.

- This is a boolean removal, so it will give hard-edged shadows.
- Hard-edged shadows suggest a pinpoint light source.



Softer shadows

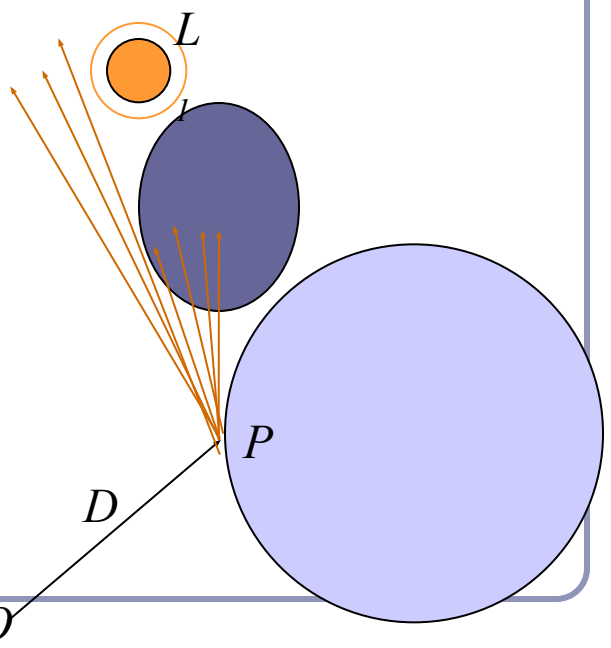
Shadows in nature are not sharp because light sources are not infinitely small.

- Also because light scatters, etc.

For lights with volume, fire many rays, covering the cross-section of your illuminated space.

Illumination is scaled by (the total number of rays that aren't blocked) divided by (the total number of rays fired).

- This is an example of *Monte-Carlo integration*: a coarse simulation of an integral over a space by randomly sampling it with many rays.
- The more rays fired, the smoother the result.

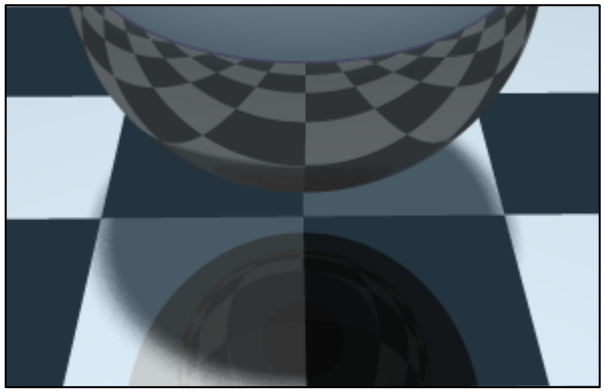


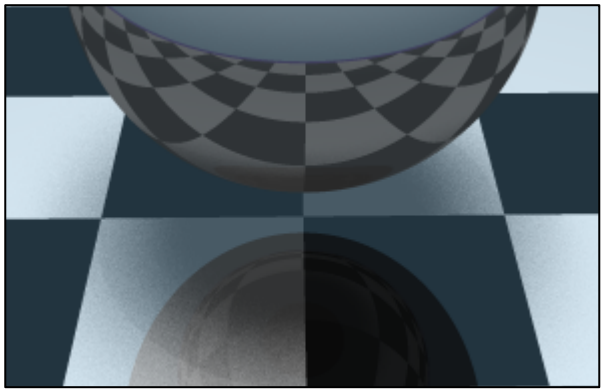
Softer shadows

Light radius: 1

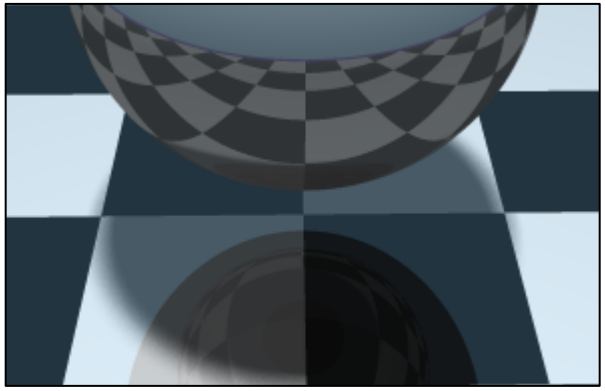
Light radius: 5

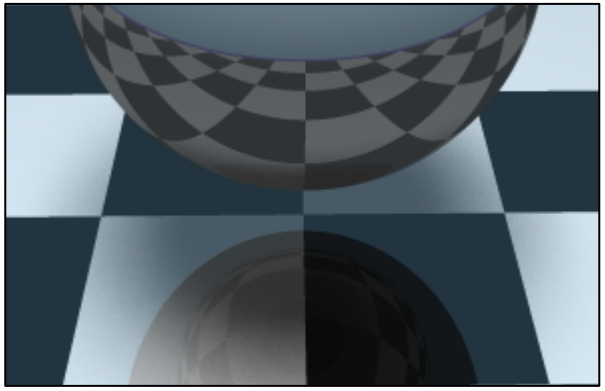
Rays per shadow test: 20





Rays per shadow test: 100



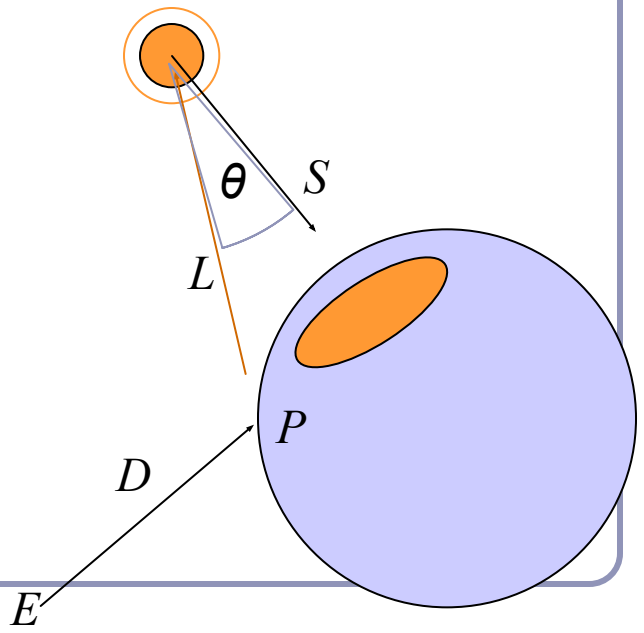


All images anti-aliased with 4x supersampling.
Distance to light in all images: 20 units

Raytraced spotlights

To create a spotlight shining along axis *S*, you can multiply the (diffuse+specular) term by $(\max(L \bullet S, 0))^m$.

- Raising *m* will tighten the spotlight, but leave the edges soft.
- If you'd prefer a hard-edged spotlight of uniform internal intensity, you can use a conditional, e.g. $((L \bullet S > \cos(15^\circ)) \ ? \ 1 : 0)$.



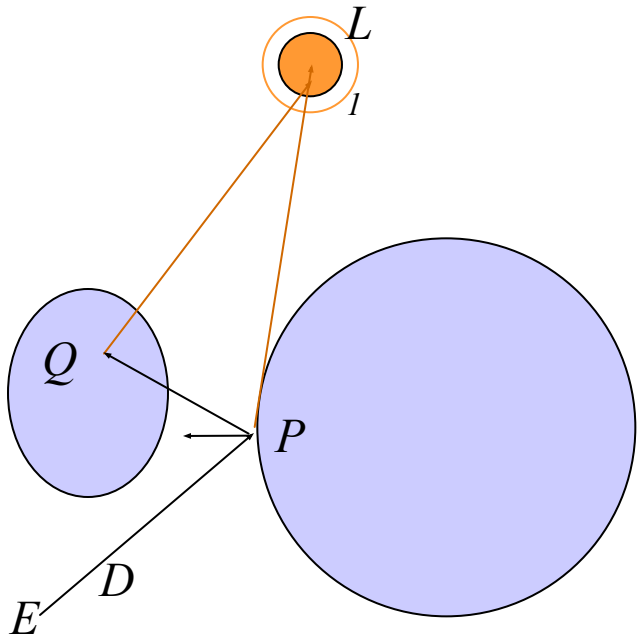
Reflection

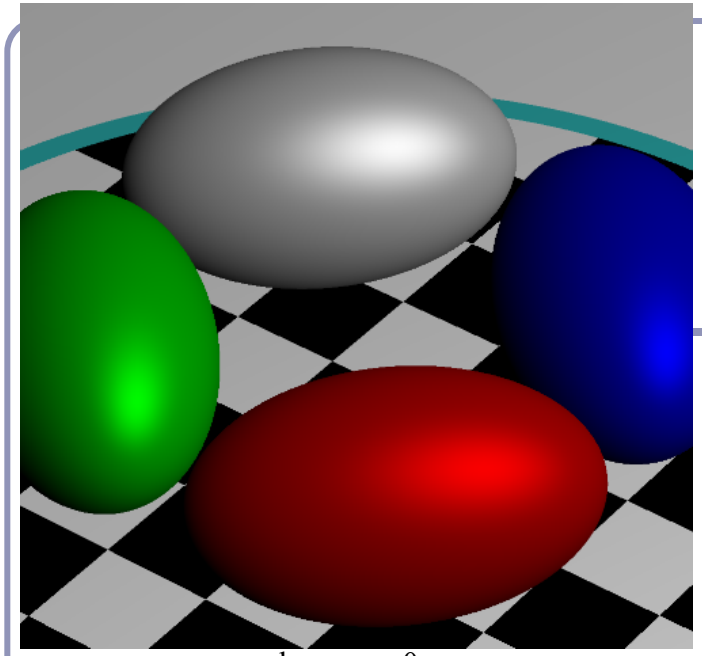
Reflection rays are calculated as:

$$R = 2(-D \bullet N)N + D$$

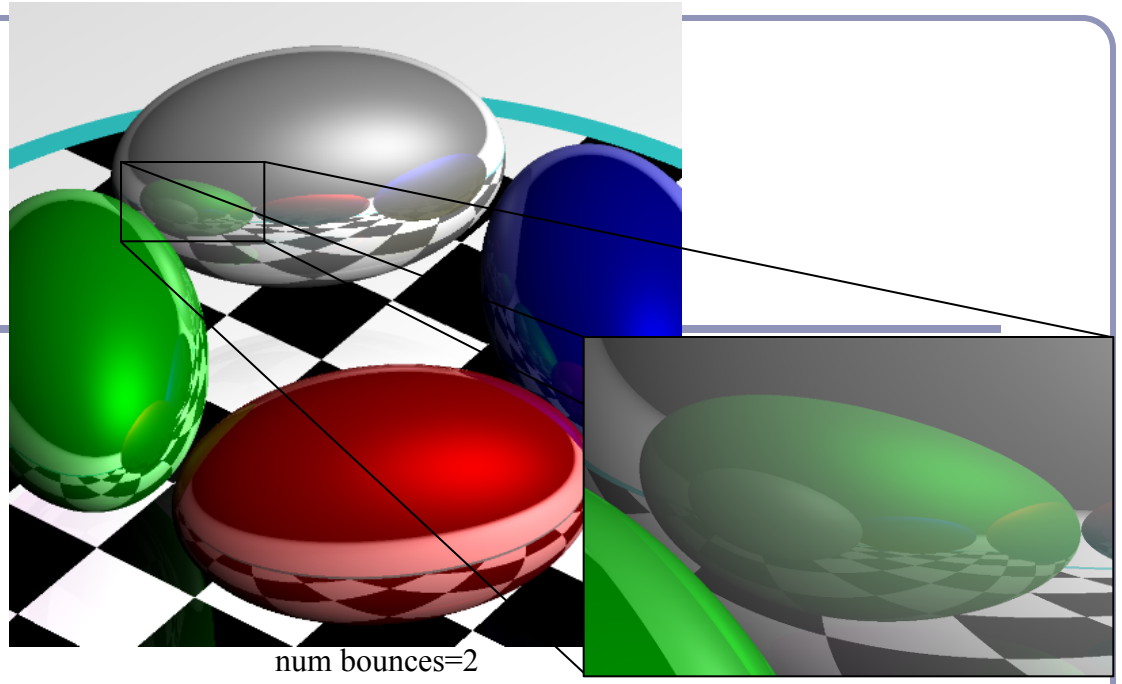
- Finding the reflected color is a recursive raycast.
- Reflection has *scene-dependant* performance impact.

- If you're using the GPU, GLSL supports `reflect()` as a built-in function.

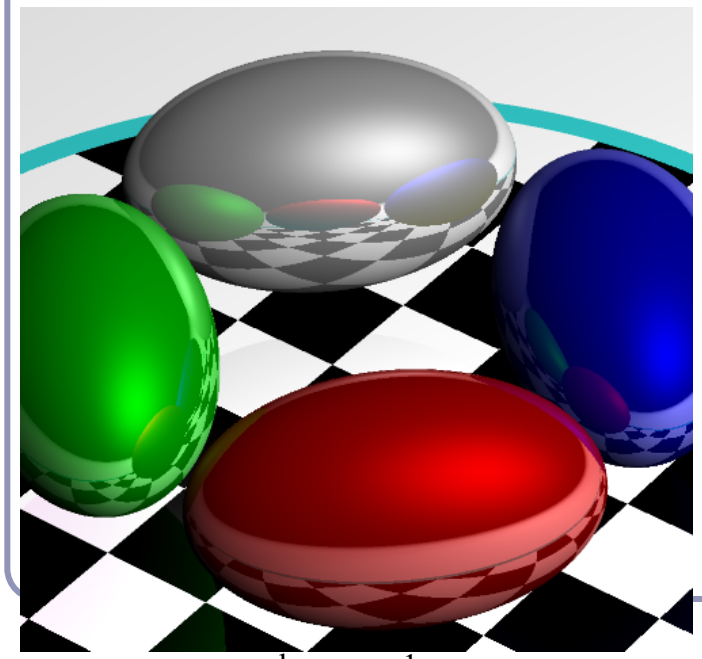




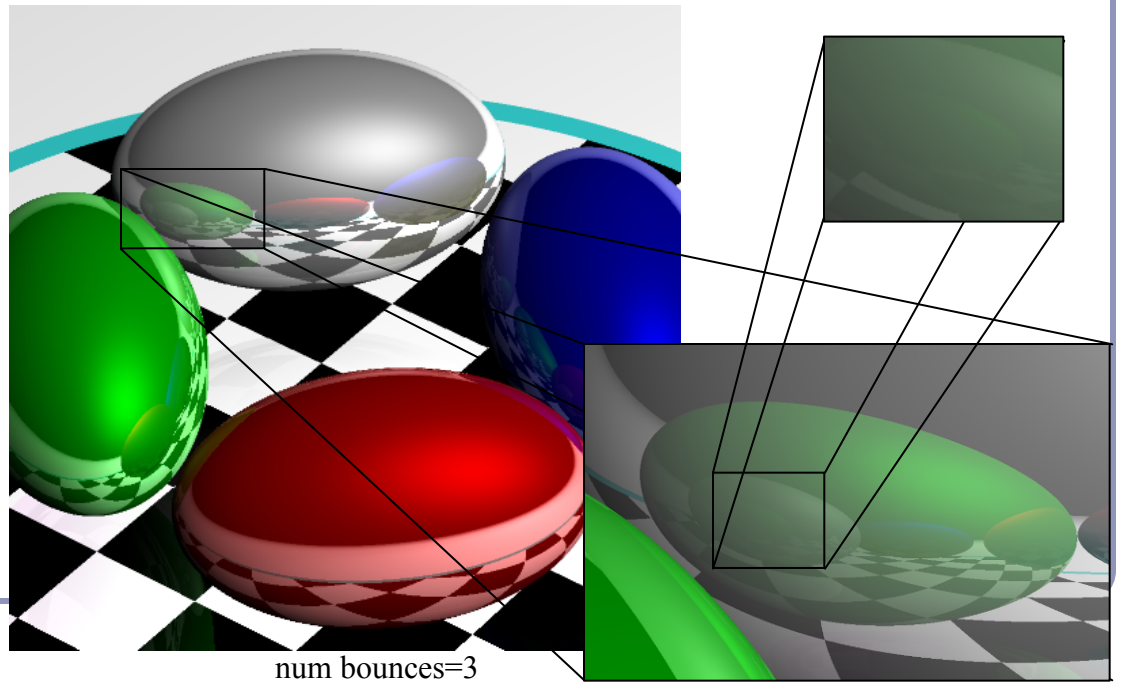
num bounces=0



num bounces=2



num bounces=1

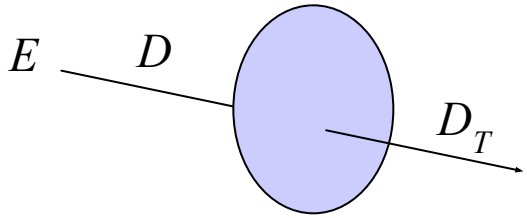


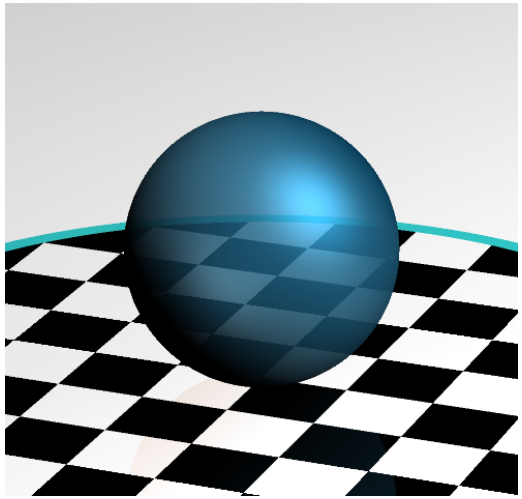
num bounces=3

Transparency

To add transparency, generate and trace a new *transparency ray* with $E_T=P$, $D_T=D$.

To support this in software, make color a 1x4 vector where the fourth component, ‘alpha’, determines the weight of the recursed transparency ray.





Refraction

The *angle of incidence* of a ray of light where it strikes a surface is the acute angle between the ray and the surface normal.

The *refractive index* of a material is a measure of how much the speed of light[1] is reduced inside the material.

- The refractive index of air is about 1.003.
- The refractive index of water is about 1.33.

[1] Or sound waves or other waves

Refraction

Snell's Law:

$$\frac{sin\,\theta_1}{sin\,\theta_2} = \frac{n_2}{n_1} = \frac{v_1}{v_2}$$

"The ratio of the sines of the *angles of incidence* of a ray of light at the interface between two materials is equal to the inverse ratio of the *refractive indices* of the materials is equal to the ratio of the speeds of light in the materials."

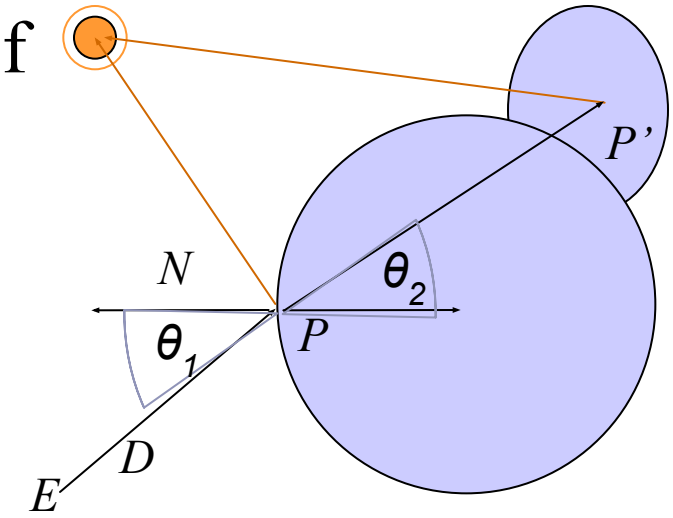
Historical note: this formula has been attributed to Willebrord Snell (1591-1626) and René Descartes (1596-1650) but first discovery goes to Ibn Sahl (940-1000) of Baghdad.

Refraction in ray tracing

$$\theta_1 = cos^{-1}(N \bullet D)$$

$$\frac{sin\,\theta_1}{sin\,\theta_2} = \frac{n_2}{n_1} \rightarrow \theta_2 = sin^{-1}\left(\frac{n_1}{n_2} sin\,\theta_1\right)$$

Using Snell's Law and the angle of incidence of the incoming ray, we can calculate the angle from the negative normal to the outbound ray.



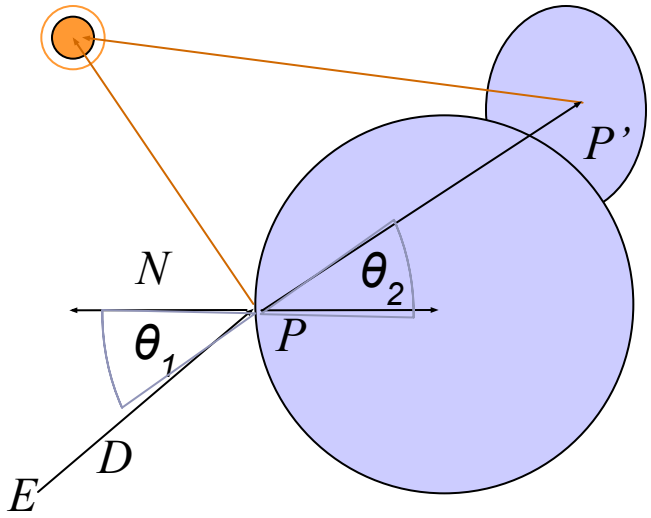
Refraction in ray tracing

What if the arcsin parameter is > 1?

- Remember, arcsin is defined in [-1,1].
- We call this the *angle of total internal reflection*: light is trapped completely inside the surface.

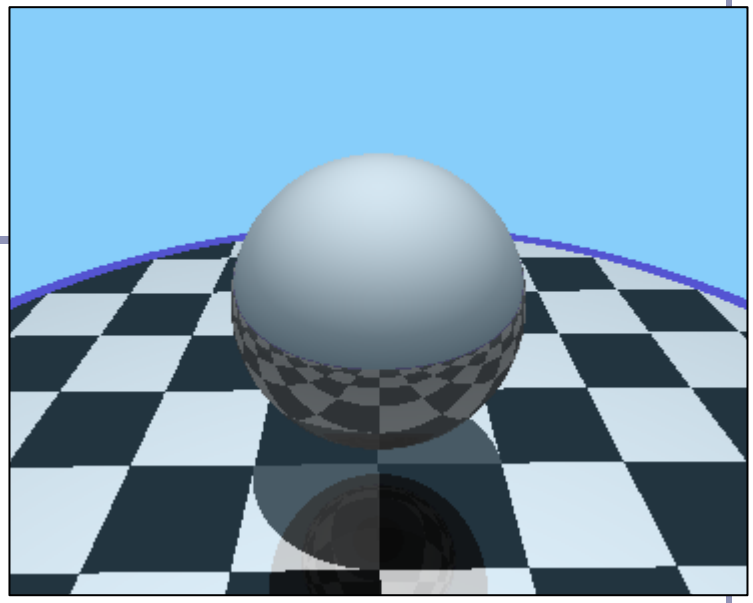
$$\theta_2 = sin^{-1}\left(\frac{n_1}{n_2} sin\,\theta_1\right)$$

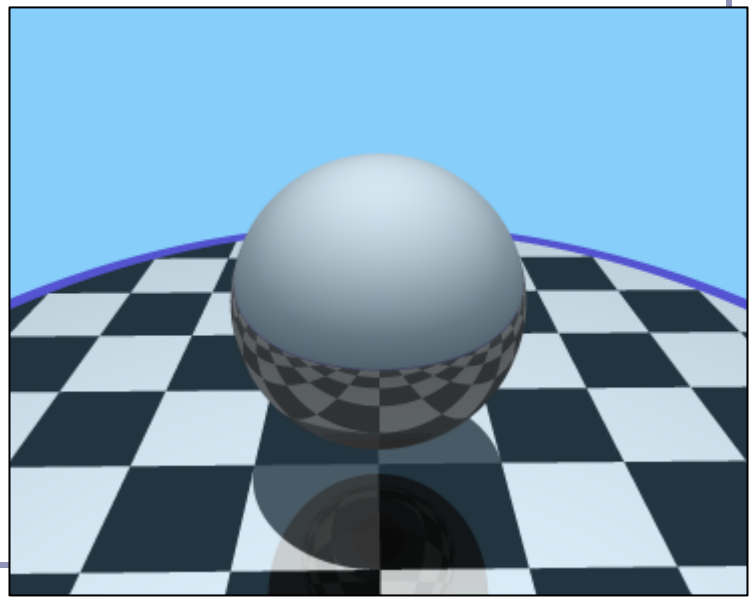
Total internal reflection



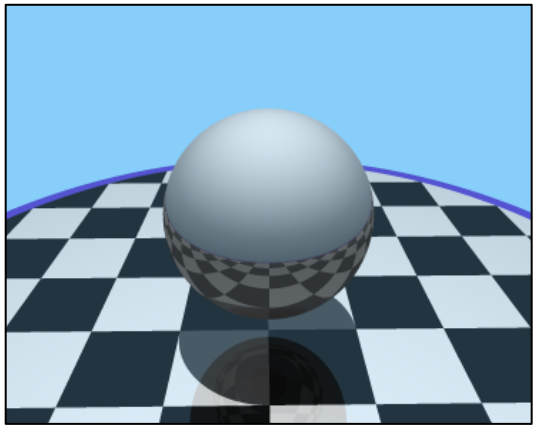
Aliasing

aliasing
/ˈeɪlɪəsɪŋ/
noun: **aliasing**
1. PHYSICS / TELECOMMUNICATIONS
the misidentification of a signal frequency, introducing distortion or error.
"high-frequency sounds are prone to aliasing"
2. COMPUTING
the distortion of a reproduced image so that curved or inclined lines appear inappropriately jagged, caused by the mapping of a number of points to the same pixel.

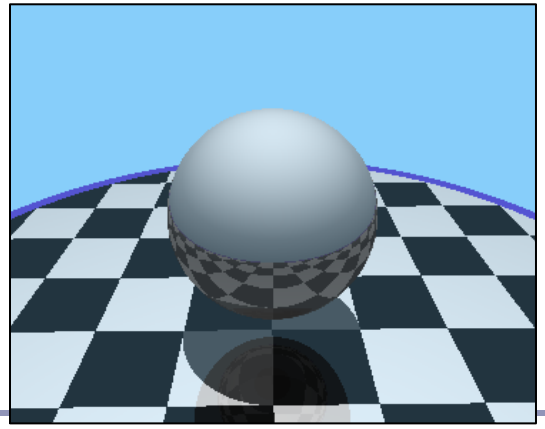




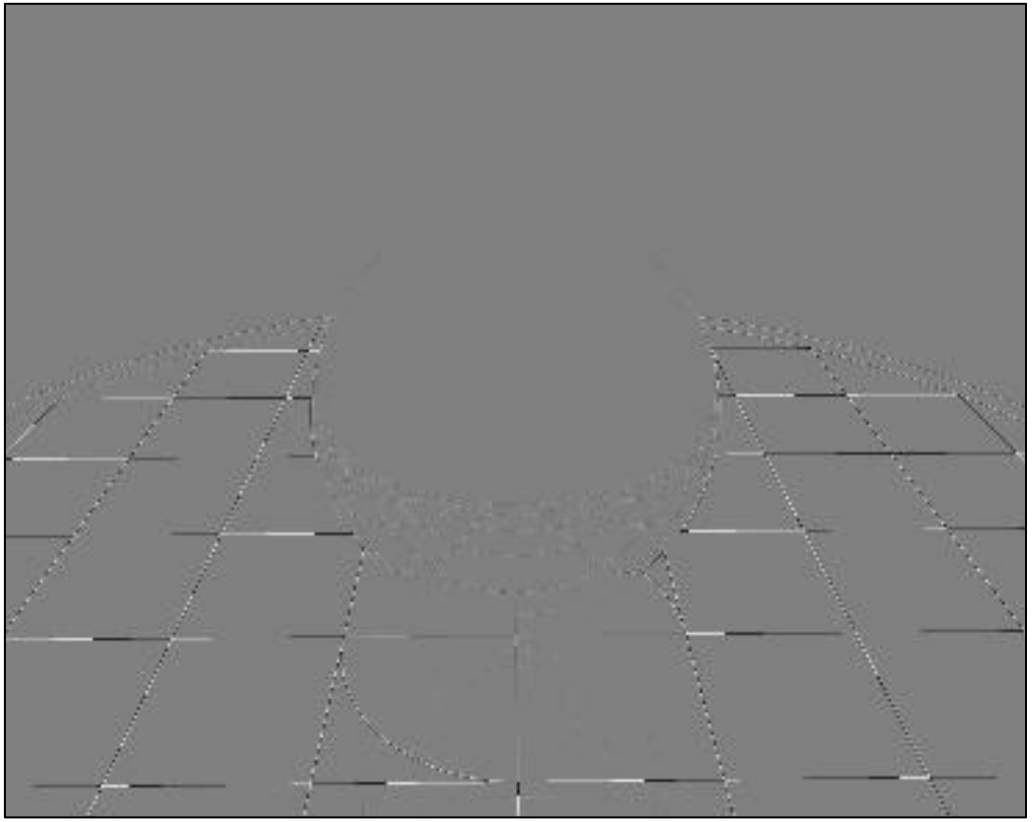
Aliasing



-



=



Anti-aliasing

Fundamentally, the problem with aliasing is that we're sampling an infinitely continuous function (the color of the scene) with a finite, discrete function (the pixels of the image).

One solution to this is *super-sampling*. If we fire multiple rays through each pixel, we can average the colors computed for every ray together to a single blended color.

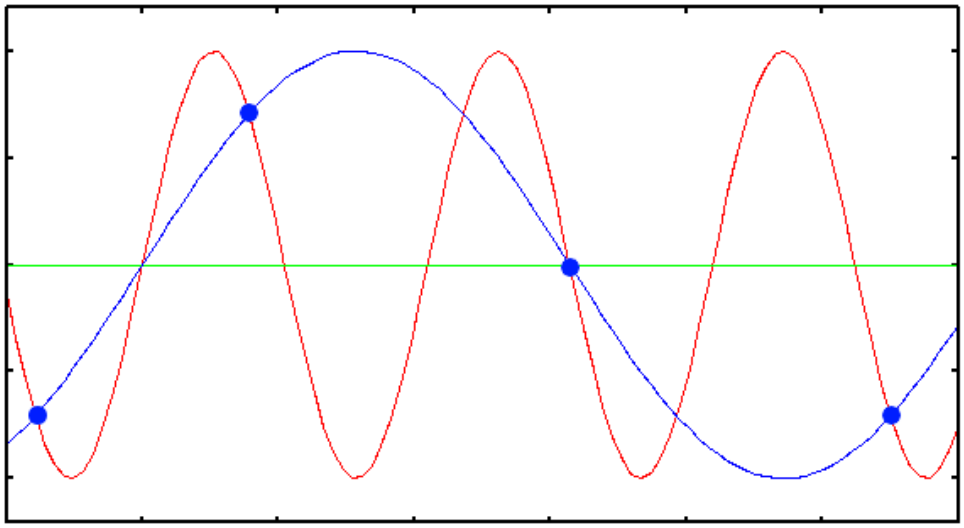
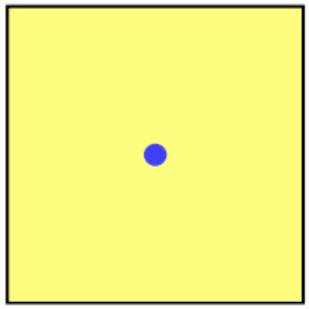


Image source: www.svi.nl

Anti-aliasing

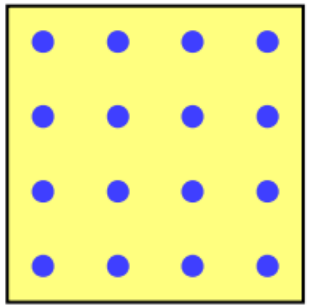
Single point

- Fire a single ray through the pixel's center



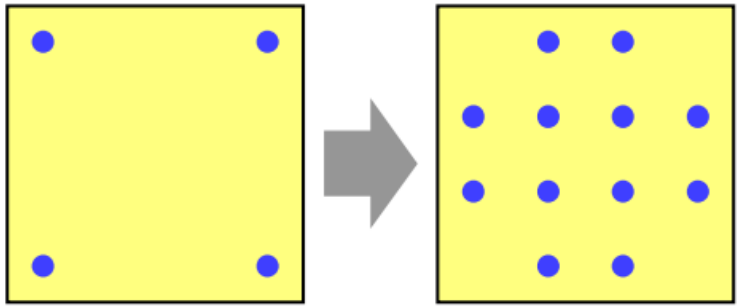
Super-sampling

- Fire multiple rays through the pixel and average the result
- Regular grid, random, jittered, Poisson disks



Adaptive super-sampling

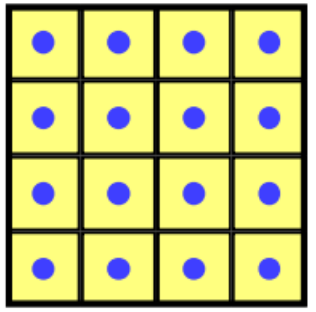
- Fire a few rays through the pixel, check the variance of the resulting values, if similar enough then stop else fire more rays



Types of super-sampling

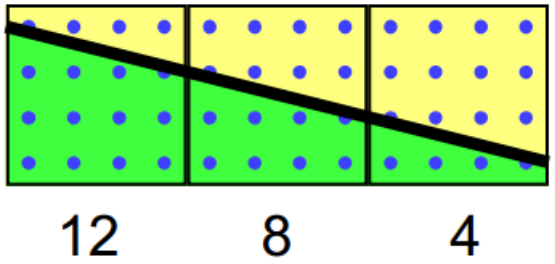
Regular grid

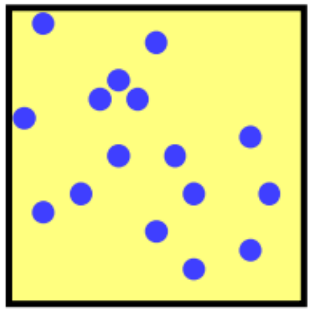
- Divide the pixel into a number of sub-pixels and fire a ray through the center of each
- This can still lead to noticeable aliasing unless a very high resolution of sub-pixel grid is used



Random

- Fire *N* rays at random points in the pixel
- Replaces aliasing artifacts with noise artifacts
  - But the human eye is much less sensitive to noise than to aliasing
- Requires special treatment for animation

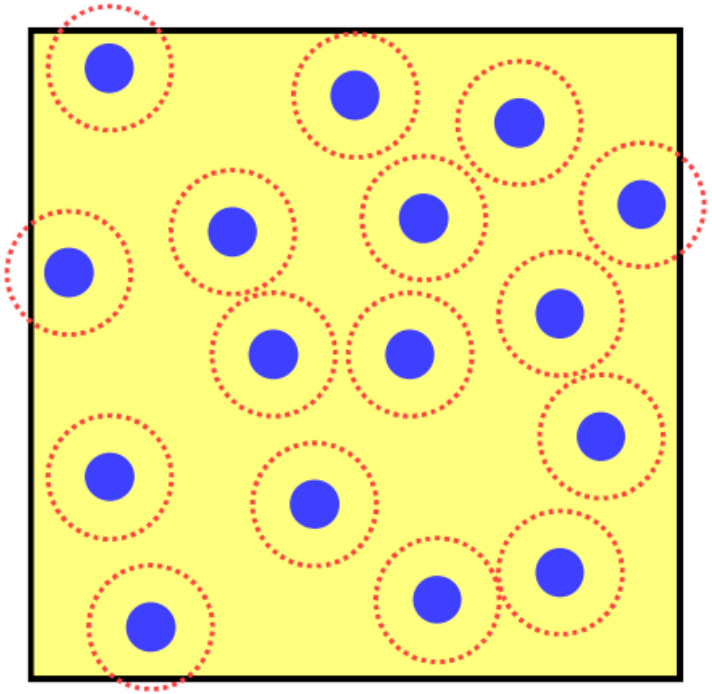


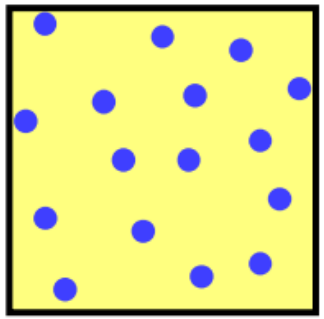


Types of super-sampling

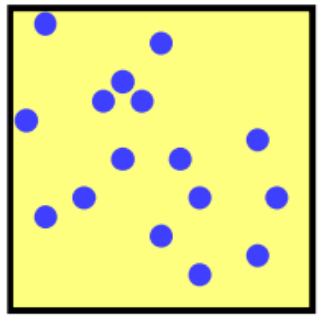
Poisson disk

- Fire *N* rays at random points in the pixel, with the proviso that no two rays shall pass through the pixel closer than ε to one another
- For *N* rays this produces a better looking image than pure random sampling
- However, can be very hard to implement correctly / quickly





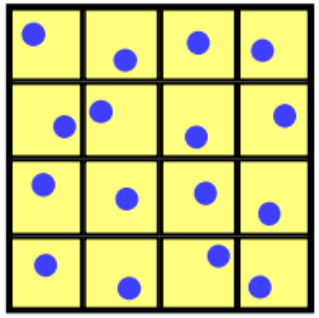
Poisson disk



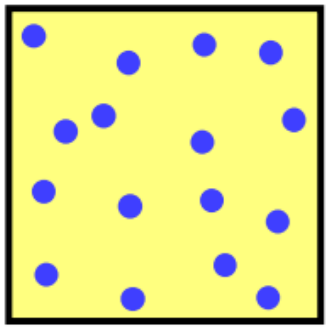
pure random

Types of super-sampling

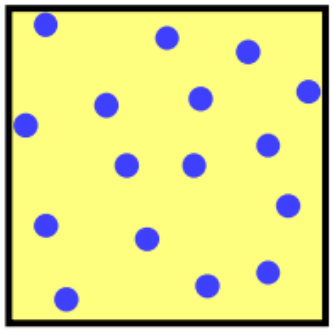
Jittered



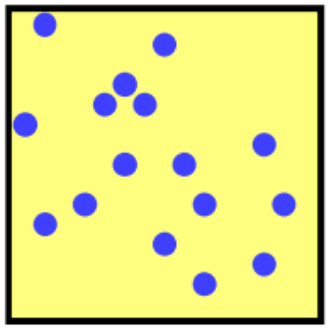
- Divide the pixel into *N* sub-pixels and fire one ray at a random point in each sub-pixel
- Approximates the Poisson disk behavior
- Better than pure random sampling, easier (and significantly faster) to implement than Poisson



jittered



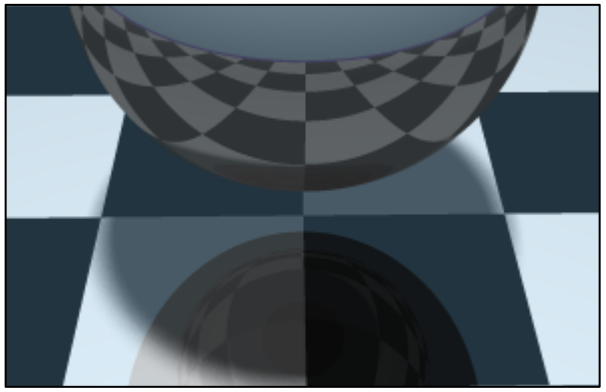
Poisson disk

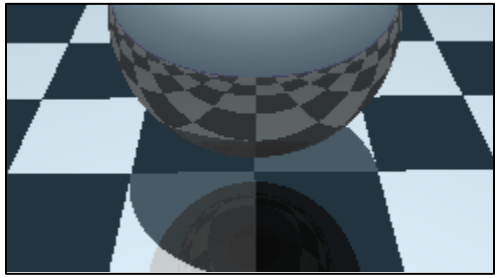


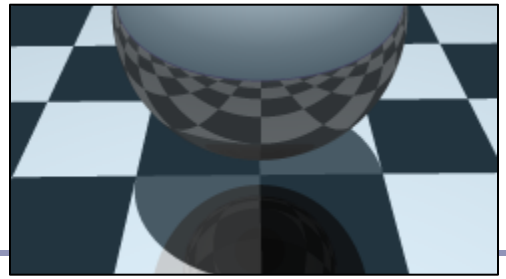
pure random

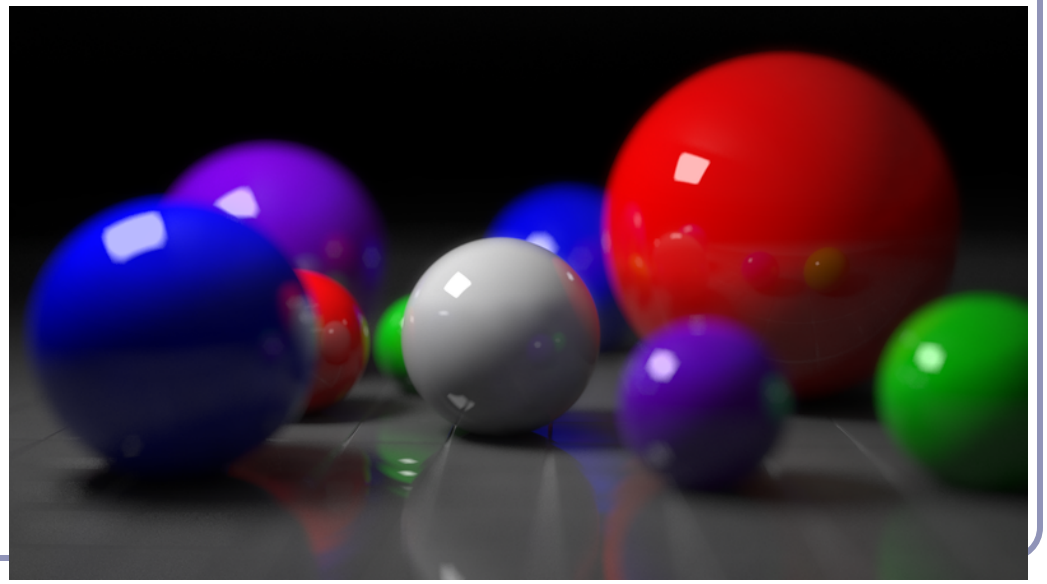
Applications of super-sampling

- Anti-aliasing
- Soft shadows
- Depth-of-field camera effects
  (fixed focal depth, finite aperture)









http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Anisotropic shading

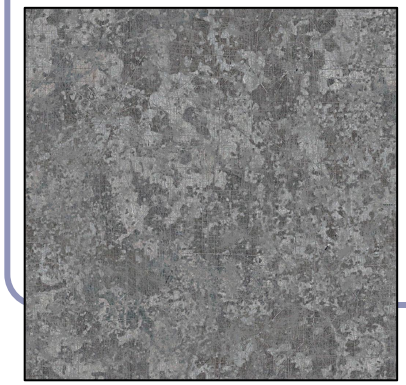
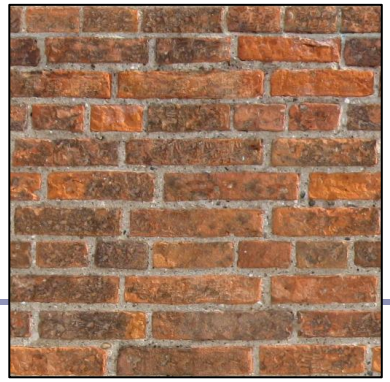
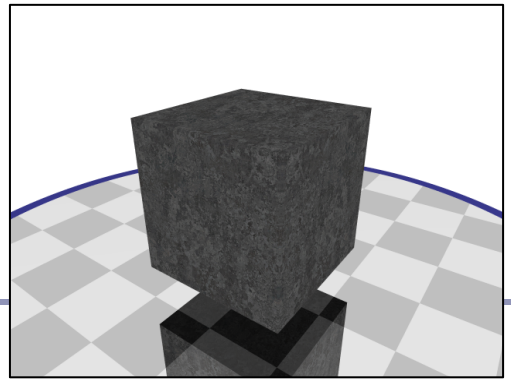
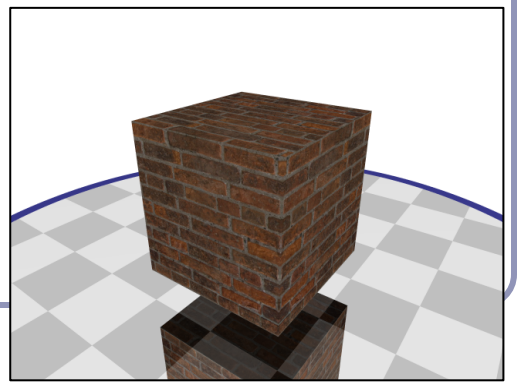
Anisotropic shading occurs in nature when light reflects off a surface differently in one direction from another, as a function of the surface itself. The specular component is modified by the direction of the light.

http://www.blenderguru.com/videos/introduction-to-anisotropic-shading/

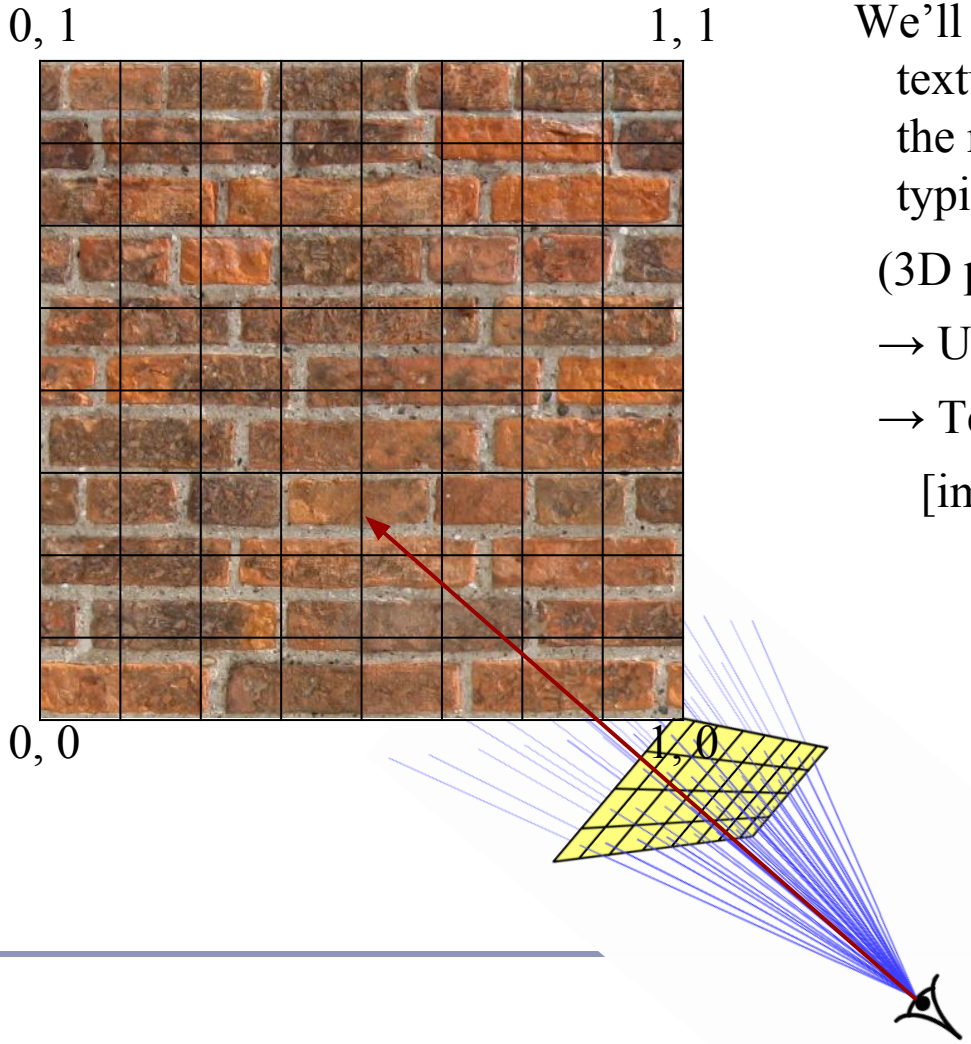
Texture mapping

As observed in last year's course, real-life objects rarely consist of perfectly smooth, uniformly colored surfaces.

Texture mapping is the art of applying an image to a surface, like a decal. Coordinates on the surface are mapped to coordinates in the texture.

Texture mapping



We'll need to query the color of the texture at the point in 3D space where the ray hits our surface. This is typically done by mapping

(3D point in local coordinates)

→ U,V coordinates bounded [0-1, 0-1]

→ Texture coordinates bounded by [image width, image height]

UV mapping the primitives

UV mapping of a unit cube

if $|x| == 1$*:*

$u = (z + 1) / 2$

$v = (y + 1) / 2$

elif $|y| == 1$*:*

$u = (x + 1) / 2$

$v = (z + 1) / 2$

else:

$u = (x + 1) / 2$

$v = (y + 1) / 2$

UV mapping of a unit sphere

$u = 0.5 + atan2(z, x) / 2\pi$

$v = 0.5 - asin(y) / \pi$

UV mapping of a torus of major radius *R*

$u = 0.5 + atan2(z, x) / 2\pi$

$v = 0.5 + atan2(y, ((x^2 + z^2)^{1/2} - R) / 2\pi$

UV mapping is easy for primitives but can be very difficult for arbitrary shapes.

Texture mapping

One constraint on using images for texture is that images have a finite resolution, and a virtual (ray-traced) camera can get quite near to the surface of an object.

This can lead to a single image pixel covering multiple ray-traced pixels (or vice-versa), leading to blurry or aliased pixels in your texture.

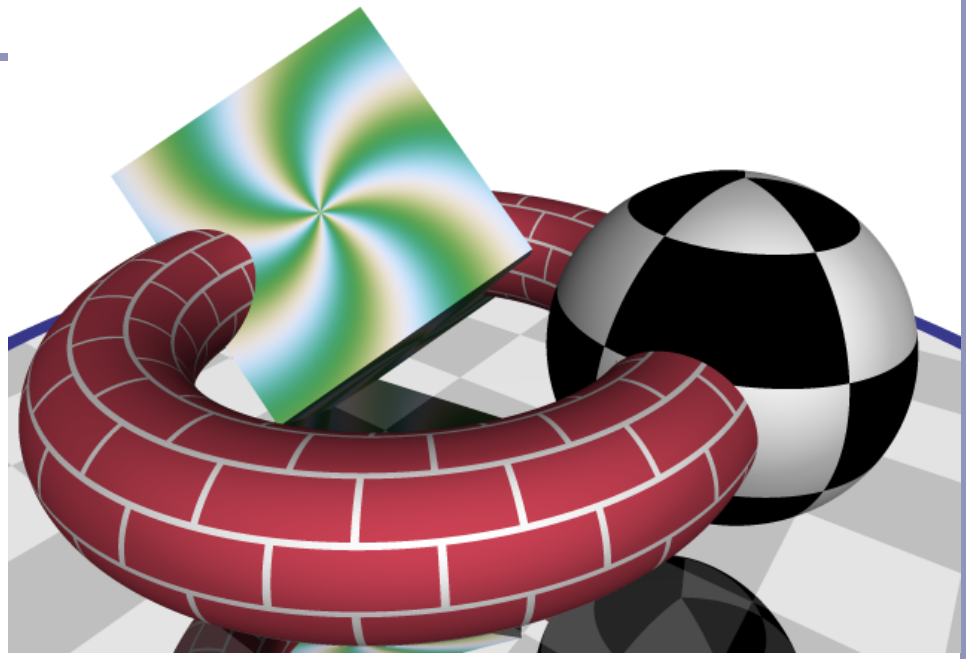
Procedural texture

Instead of relying on discrete pixels, you can get infinitely more precise results with procedurally generated textures.

Procedural textures compute the color directly from the U,V coordinate without an image lookup.

For example, here's the code for the torus' brick pattern (right):

```
tx = (int) 10 * u
ty = (int) 10 * v
oddity = (tx & 0x01) == (ty & 0x01)
edge = ((10 * u - tx < 0.1) && oddity) || (10 * v - ty < 0.1)
return edge ? WHITE : RED
```



Confession: I cheated slightly and multiplied the u coordinate by 4 to repeat the brick texture four times around the torus.

Procedural volumetric texture

By mapping 3D coordinates to colors, we can create *volumetric texture*. The input to the texture is local model coordinates; the output is color and surface characteristics.

For example, to produce wood-grain texture, trees grow rings, with darker wood from earlier in the year and lighter wood from later in the year.

- Choose shades of early and late wood
- $f(P) = (X_P^2 + Z_P^2) \bmod 1$
- $color(P) = earlyWood + f(P) * (lateWood - earlyWood)$

$f(P)=0$ $f(P)=1$

Adding realism

The teapot on the previous slide doesn't look very wooden, because it's perfectly uniform. One way to make the surface look more natural is to add a randomized noise field to f(P):

$$f(P) = (X_P^2 + Z_P^2 + noise(P)) \bmod 1$$

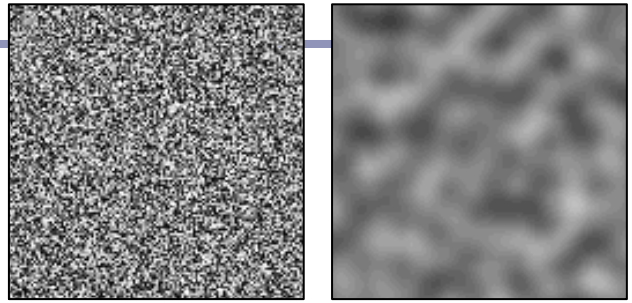
where *noise(P)* is a function that maps 3D coordinates in space to scalar values chosen at random.

For natural-looking results, use *Perlin noise*, which interpolates smoothly between noise values.

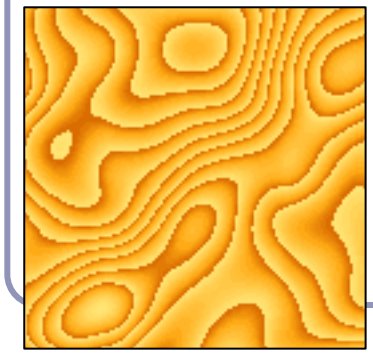
Perlin noise

Perlin noise (invented by Ken Perlin) is a method for generating noise which has some useful traits:

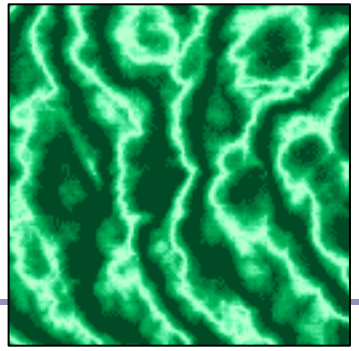
- It is a *band-limited repeatable pseudorandom* function (in the words of its author, Ken Perlin)
- It is bounded within a range close [-1, 1]
- It varies continuously, without discontinuity
- It has regions of relative stability
- It can be initialized with random values, extended arbitrarily in space, yet cached deterministically
  - Perlin's talk: http://www.noisemachine.com/talk1/



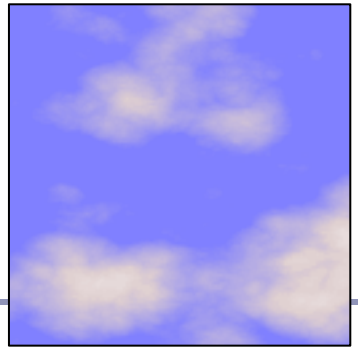
Non-coherent noise (left) and Perlin noise (right)
Image credit: Matt Zucker



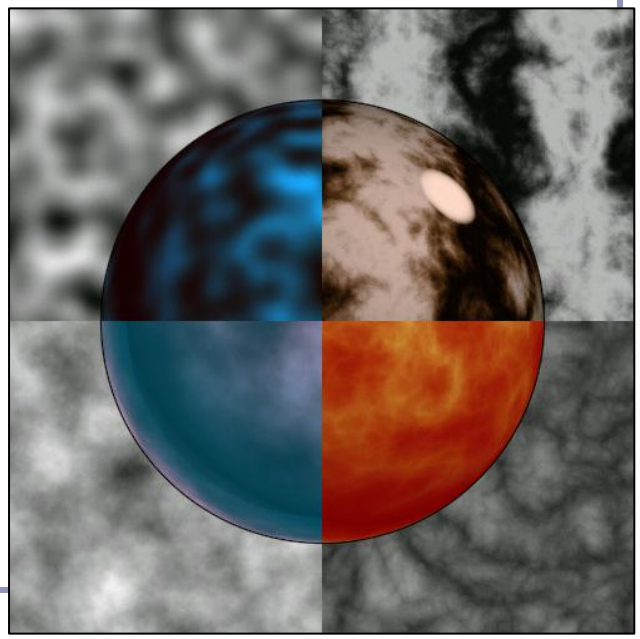
Matt Zucker



Matt Zucker



Matt Zucker



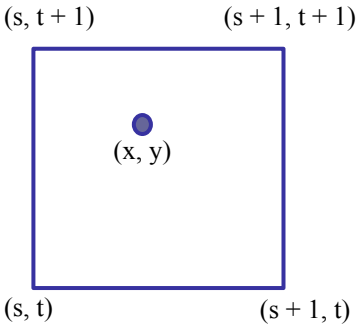
Ken Perlin

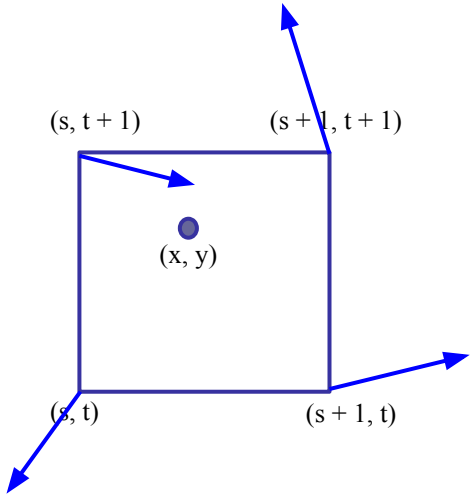
Perlin noise 1

Perlin noise caches ‘seed’ random values on a grid at integer intervals. You’ll look up noise values at arbitrary points in the plane, and they’ll be determined by the four nearest seed randoms on the grid.

Given point *(x, y)*, let *(s, t) = (floor(x), floor(y))*.

For each grid vertex in
{(s, t), (s+1, t), (s+1, t+1), (s, t+1)}
choose and cache a random vector of length one.



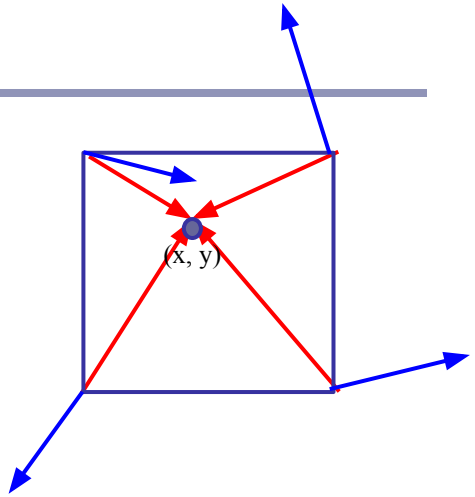


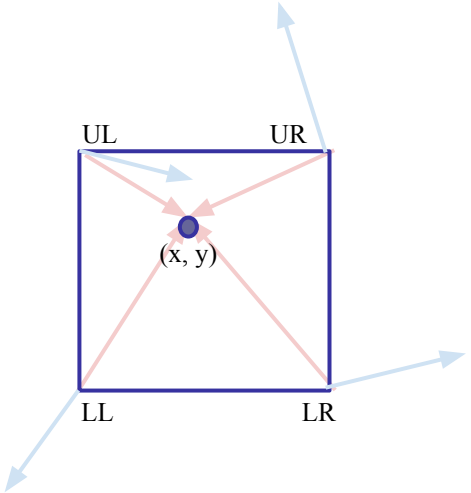
These slides borrow heavily from Mark Zucker’s excellent page on Perlin noise at
http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Perlin noise 2

For each of the four corners, take the dot product of the random seed vector with the vector from that corner to *(x, y)*. This gives you a unique scalar value per corner.

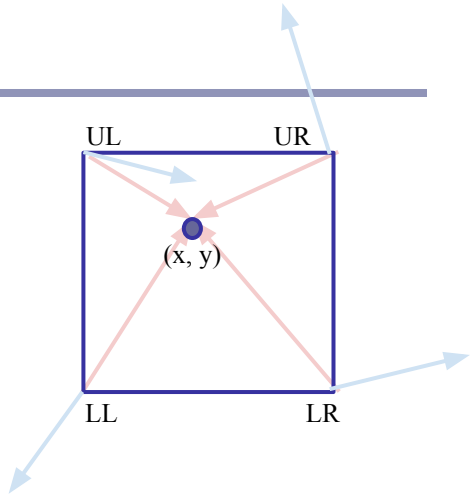
- As *(x, y)* moves across this cell of the grid, the values of the dot products will change smoothly, with no discontinuity.
- As *(x, y)* approaches a grid point, the contribution from that point will approach zero.
- The values of *LL, LR, UL, UR* are clamped to a range close to [-1, 1].





These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at
http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Perlin noise 3



Now we take a weighted average of *LL, LR, UL, UR*.

Perlin noise uses a weighted averaging function chosen such that values close to zero and one are moved closer to zero and one, called the *ease curve*:

$S(t) = 3t^2-2t^3$

We interpolate along one axis first:

$L(x, y) = LL + S(x - floor(x))(LR-LL)$

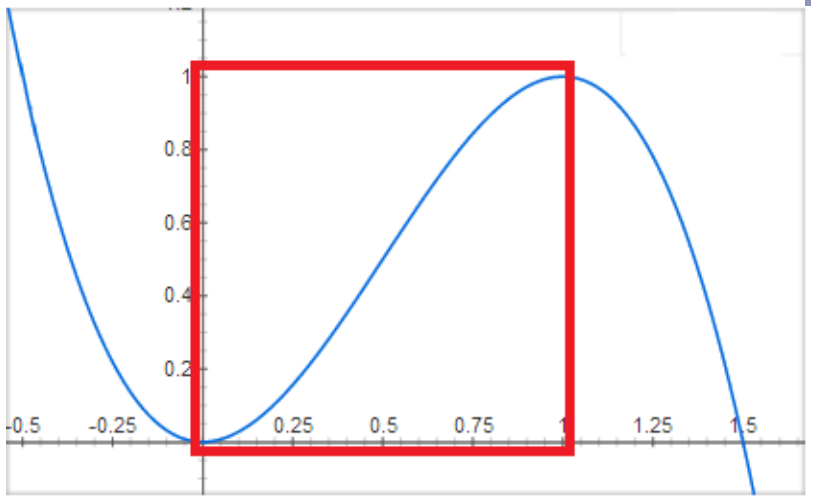
$U(x, y) = UL + S(x - floor(x))(UR-UL)$

Then we interpolate again to merge the two upper and lower functions:

$noise(x, y) =$

$L(x, y) + S(y - floor(y))(U(x, y) - L(x, y))$

Voila!



The 'ease curve'

These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at
http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Tuning noise

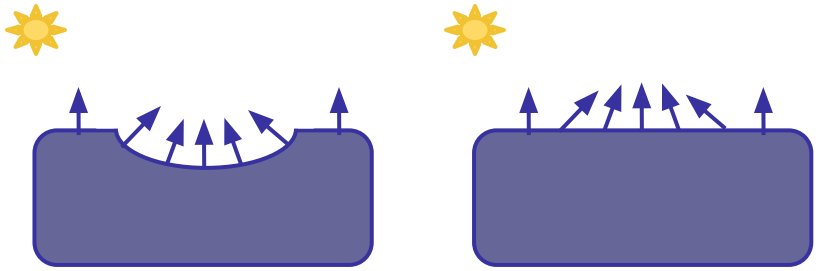
Texture frequency
1 → 3

Noise frequency
1 → 3

Noise amplitude
1 → 3

Normal mapping

Normal mapping applies the principles of texture mapping to the surface normal instead of surface color.



The specular and diffuse shading of the surface varies with the normals in a dent on the surface.

If we duplicate the normals, we don't have to duplicate the dent.

In a sense, the ray tracer computes a trompe-l'oeuil image on the fly and 'paints' the surface with more detail than is actually present in the geometry.

Normal mapping