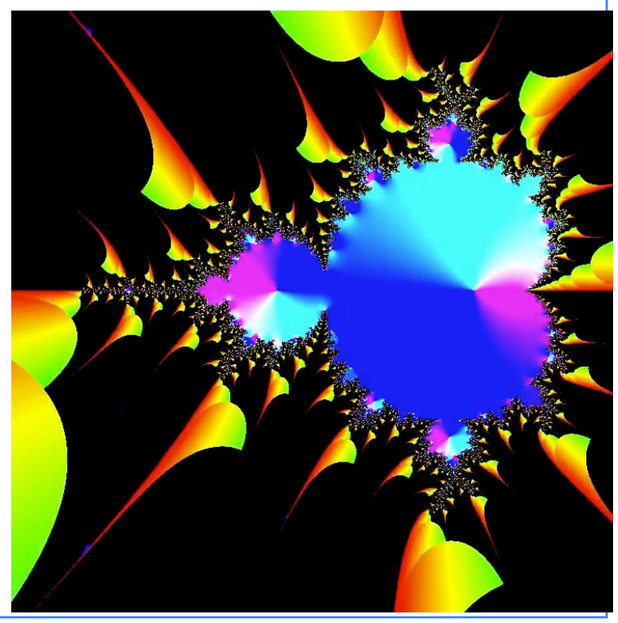
TSBK07/TSBK11

Computer Graphics

Ingemar Ragnemalm, ISY



Lecture 13

Fractals: Terrain generation

Anti-aliasing

Ray-tracing

Global illumination

Lecture questions

1. How does the frequency contents vary in typical images?

2. Which terrain generation method has the lowest computational complexity?

3. What are the advantages of gradient noise?

4. What is the highest magnitude of errors for 2x supersampling?

5. How do you get shadows in ray-tracing?

6. How can you improve anti-aliasing by supersampling even more in ray-tracing?

Fractal terrain generation

Statisticaly self-similar fractal ("Fractal Brownian Motion")

but also

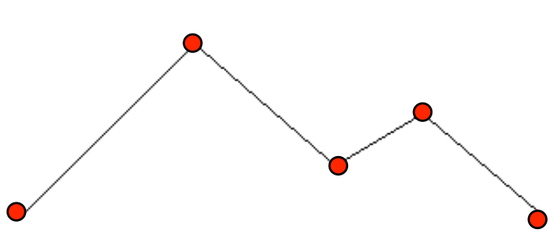
Application of noise functions

Three approaches:

- Geometric fractals: Midpoint displacement/ heightfield refinement
- Signal processing: Frequency space filtered noise
- Noise functions: Multiple bands of band-limited noise functions (Perlin/gradient noise)

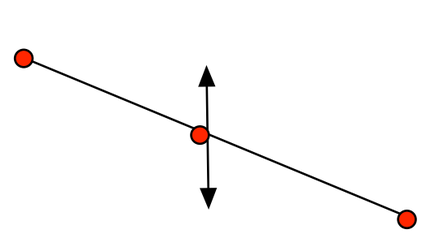
Random midpoint-displacement

Good for fractal terrain generation



Initiator

Desired rough overall shape



Generator

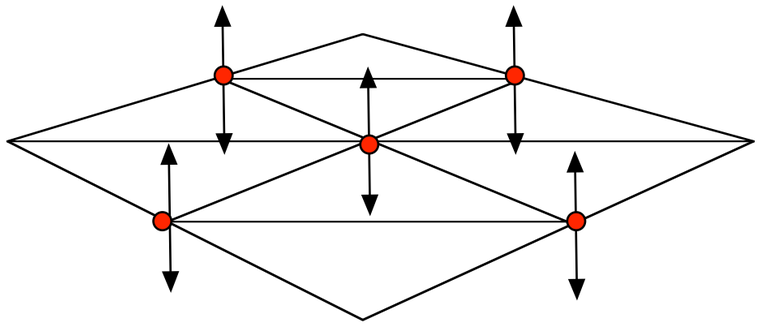
Find midpoint, displace along y only

7 iterations

Fractal terrain generation for 2D heightmap

Split a square to four

Displace midpoints of each side and middle



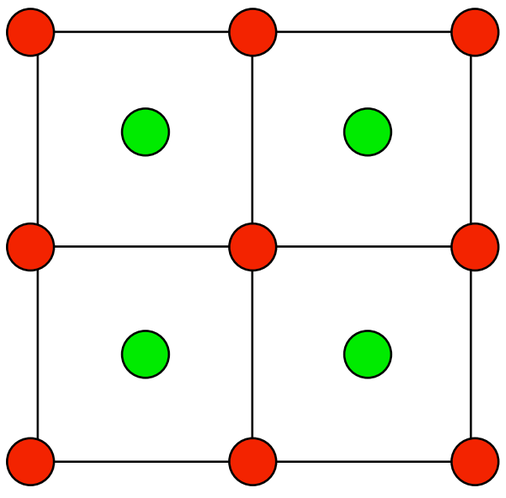
But: What dependencies? Any outside the square?

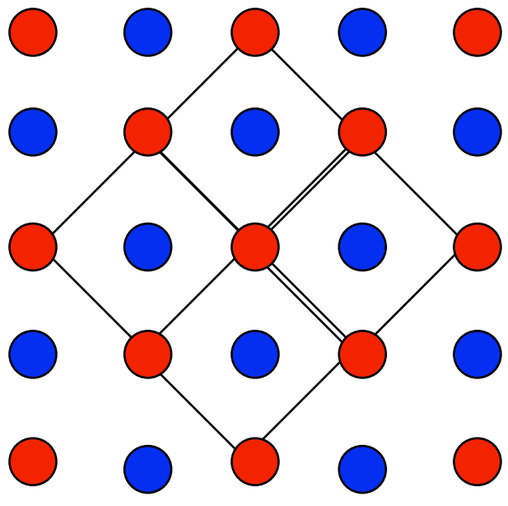
Edge points must match neighbor patches

Diamond-square algorithm

1) Midpoint from corners

2) Midpoint from resulting "diamonds"





Repeat to desired resolution

Diamond-square algorithm

Random offset at each stage

Proportional to length of the side of the square

=> Scale down by sqrt(2) for each phase!

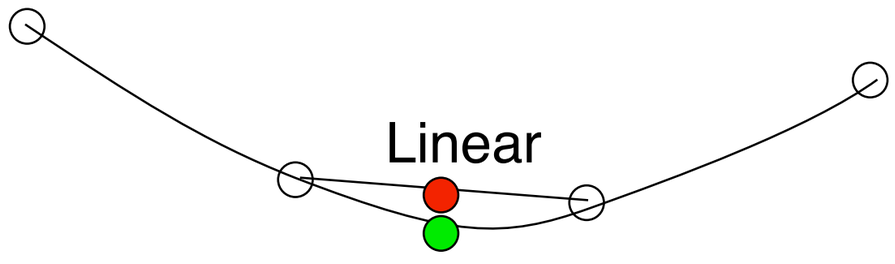
(Not by 2 for every two phases! Popular misconception!)

Diamond-square algorithm filtering

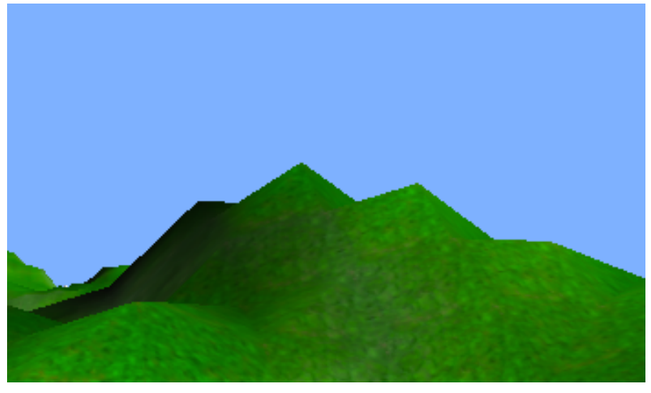
Important feature! We are reconstructing a signal from samples! (And then add HF detail.)

Simple and fast: Averaging (linear interpolation)

Better: Higher precision filter from larger neighborhood. Usual signal processing rules apply! Use a 4x4 neighborhood.



Better, e.g. cubic spline



Jagged peaks caused by bad filter

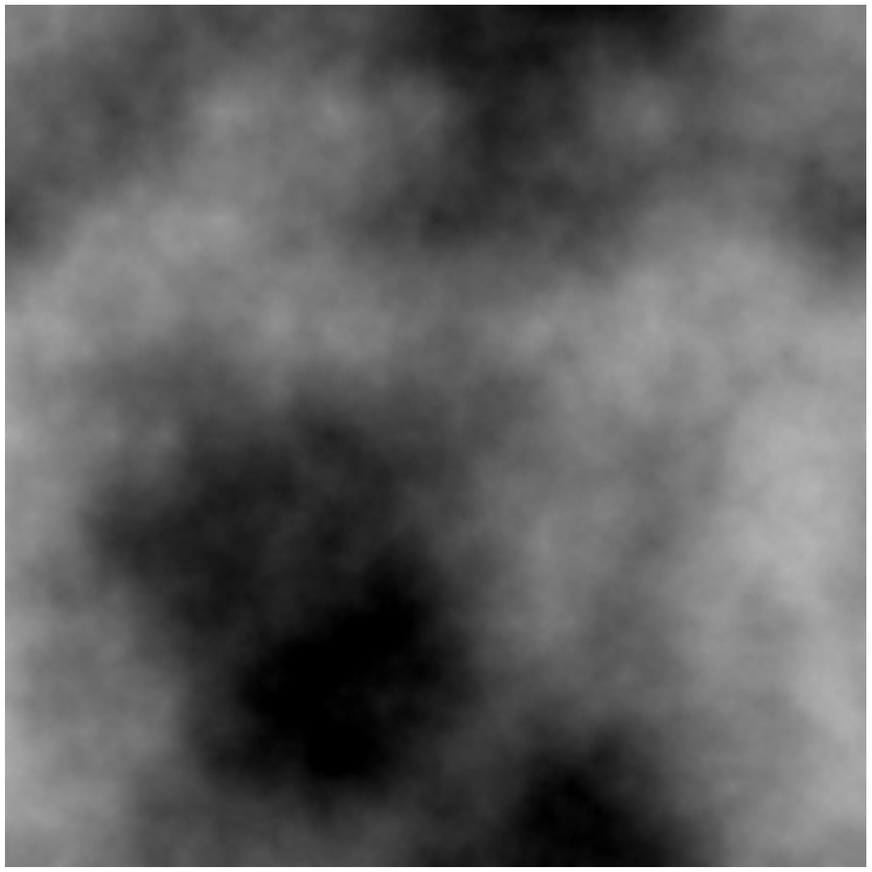
Computational complexity of Diamond-square

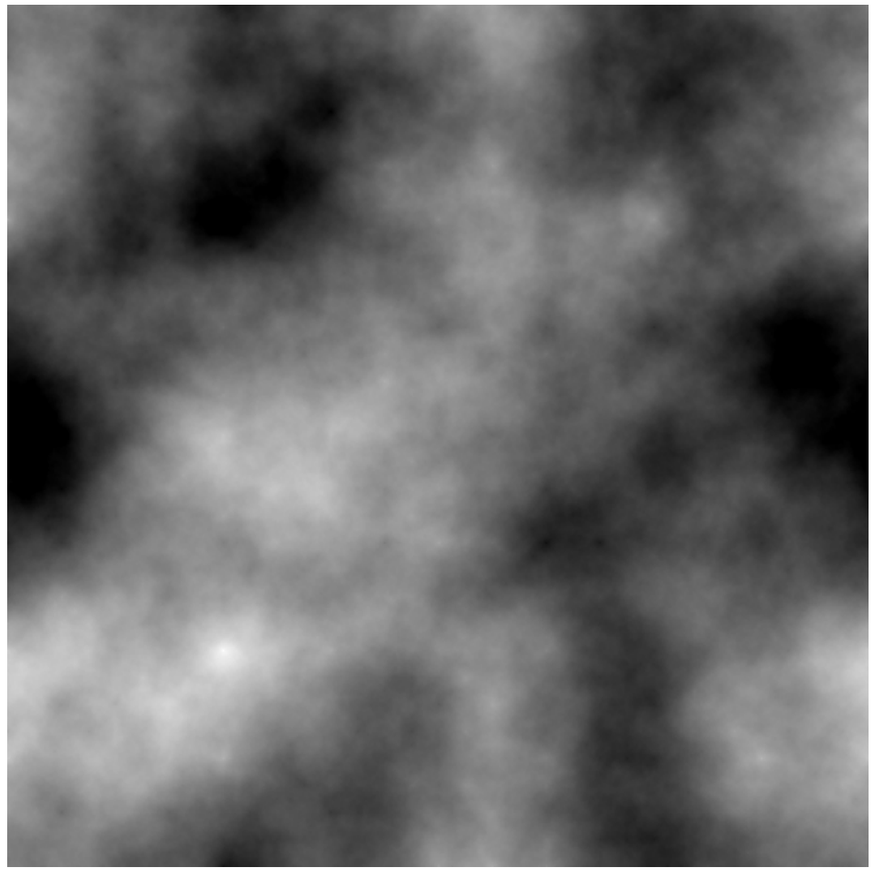
Every point is visited only once!

Hence, $O(N^2)$ for an N^2 image

Exceptionally fast in its simplest forms

A constant factor gets higher for better filtering





Diamond-square results

Visually pleasing but has artifacts if not properly filtered!

”Square-square” algorithm

Resolution pyramid, top-to-bottom

Terrain level k is array of resolution 2^k x 2^k

The next level has 4x the resolution

Generate new 2x2 block from one, or (better) filter over a small neighborhood

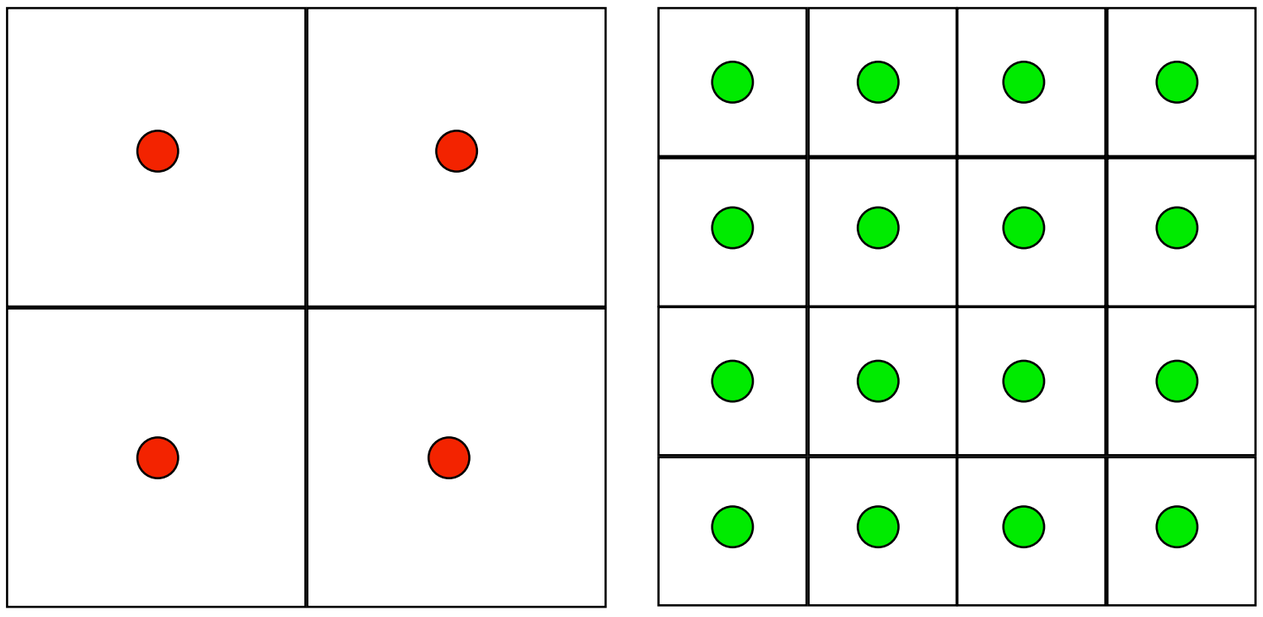
Add random offset to all values

Offset should be smaller for higher k

=> magnitude of frequency components inverse proportional to frequency!

Square square

Image upsampling + add noise



Again: Linear interpolation is simple and fast, better filter gives better result

Essentially: Create a resolution pyramid level by level!

Computational complexity of Square-square

Image size + 1/4 + 1/16... = 1 1/3!

Hence, $O(N^2)$ but with a higher constant than Diamond-Square

Additional benefit

Produces a resolution pyramid as a side effect! Can be saved and used for level-of-detail

This is possible with other methods but automatic here.

Noise functions

Fractals and noise functions are closely related

Noise can look natural... but when?

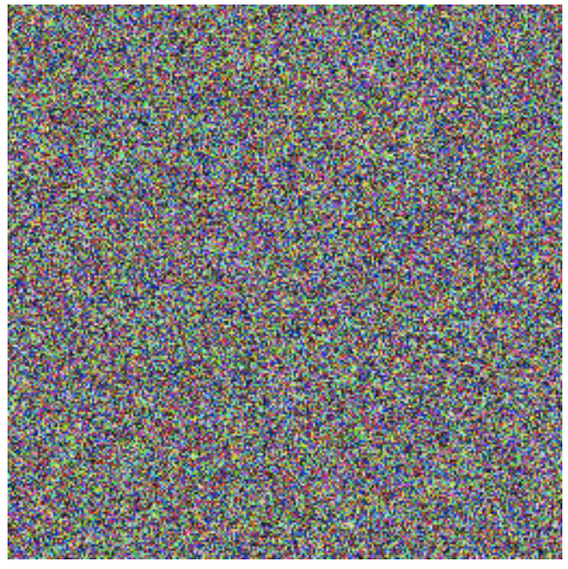
- white noise
- colored noise
- value noise
- gradient noise

White noise

Same amplitude in all frequencies

Useless as it is?

Can be processed to something better.

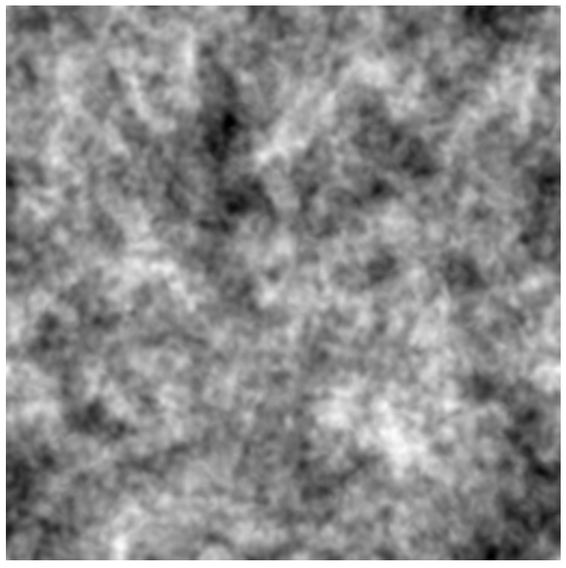


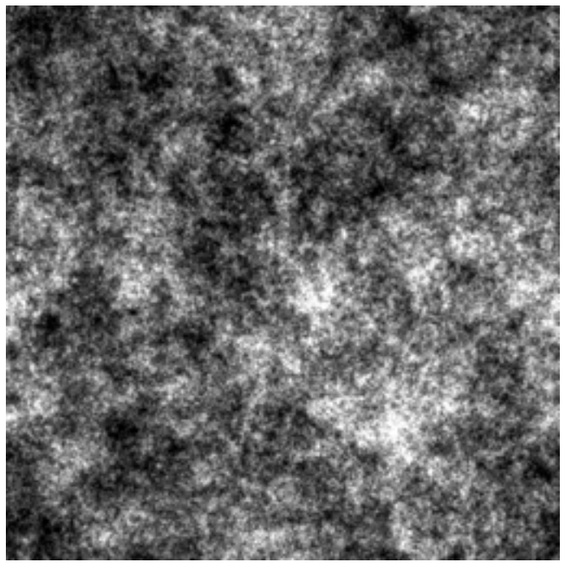
Colored noise

Amplitude varies with frequencies

With the right variation, it can look nice - natural!

The 1/f rule!





Colored noise

Can be processed with filters, e.g. frequency plane functions

Considered too computationally heavy. (Questionable!)

Therefore other methods became popular: Simplex noise, Perlin noise.

Colored noise by filtering in the frequency plane

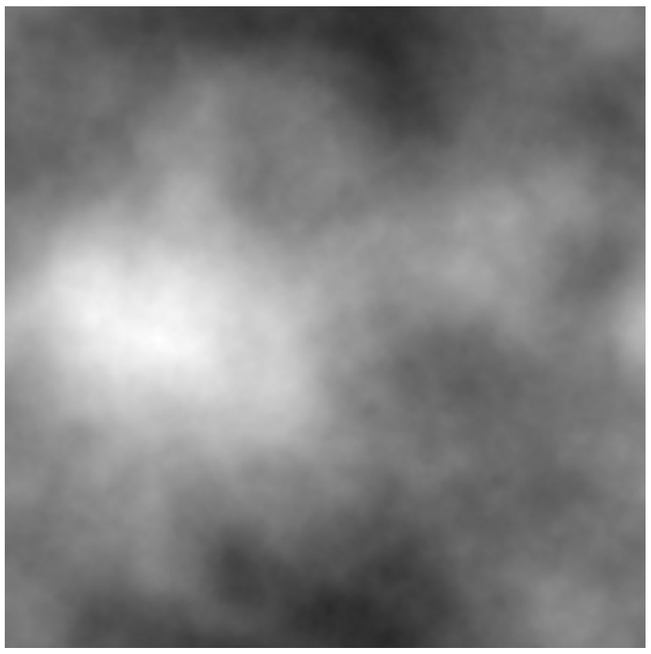
Fill frequency space (2D) with random numbers
(white noise)

Filter by G(f) = F(f) * 1/|f|

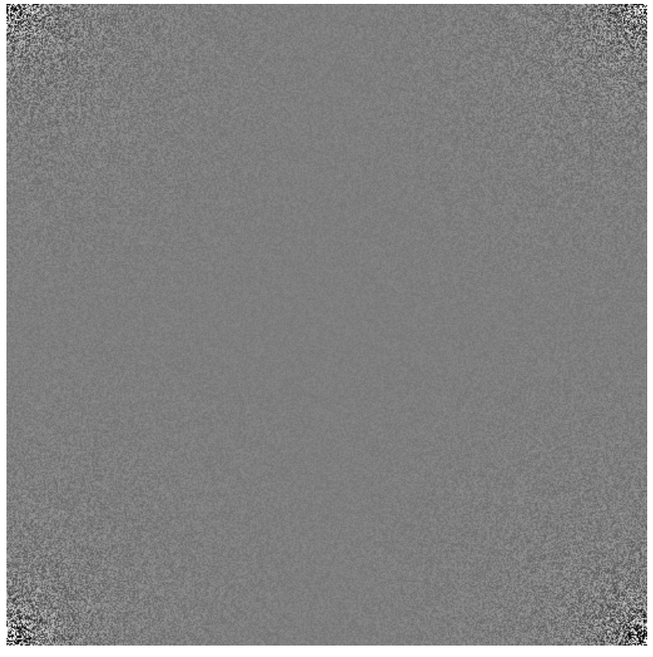
Convert to spatial image with FFT
(FHT, actually, Fast Hartley Transform)

Filter white noise by 1/f

Example

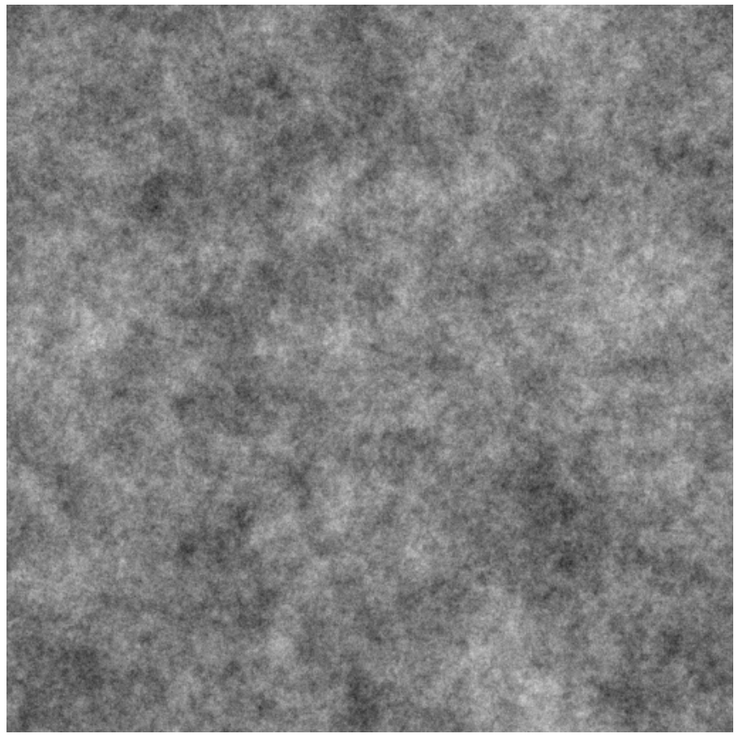


Frequency space:



Other falloffs than 1/|f|

Too much ramp



Too little ramp

Advantages of frequency plane filtered noise

Extremely good precision for the frequency behavior

Repeating patterns, good for textures

FFT/FHT well researched, highly optimized implementations = fast

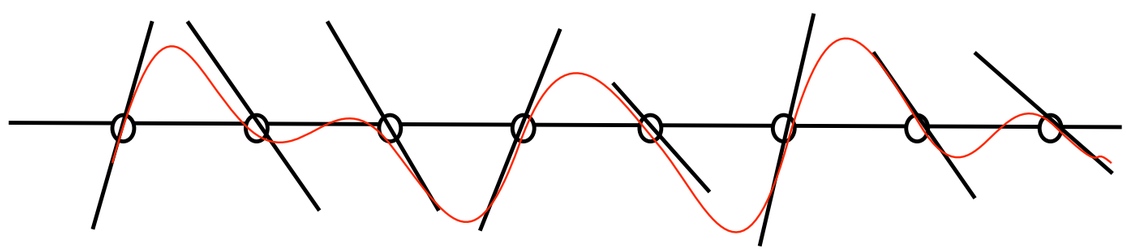
Gradient noise

Band limited noise using random gradients in a grid.

Interpolates between grid points

Most famous: Perlin noise.

Newer: Simplex noise, PSRD noise



Band limited noise

Not very useful... but combine several frequency bands!

A frequency band is often called an *octave*.

Octave

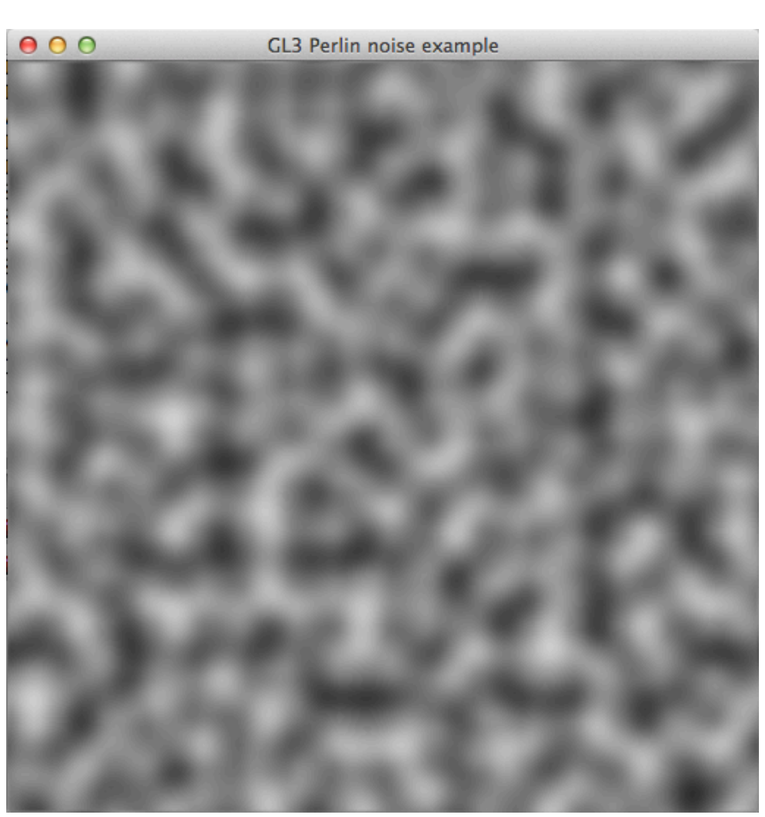
Usually a music term.

One octave = 2x frequency!

One octave is exactly at
half of the string length!

Gradient/Perlin noise

Single octave



Single octave Perlin

```
void makeperlintexture()
{
   int x, y;
   float i[3];
   char val;

   for (x = 0; x < 128; x++)
   for (y = 0; y < 128; y++)
   {
      i[0] = x / 8.0;
      i[1] = y / 8.0;
      val = (char)(clamp(noise2(i) * 200.0, -127, 127))+128;

      ptex[x][y][0] = val;
      ptex[x][y][1] = val;
      ptex[x][y][2] = val;
   }
}
```

Implementation of Perlin noise

Very elegant minimal implementation!

Random numbers in shaders. Not the best possible!

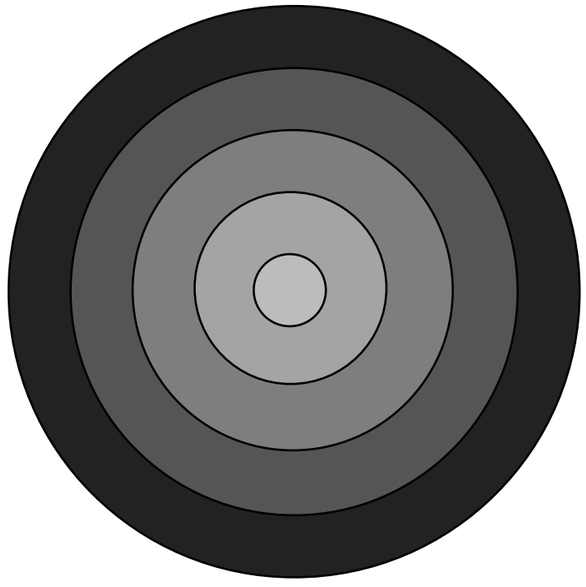
```
vec2 random2(vec2 st)
{
    st = vec2( dot(st,vec2(127.1,311.7)),
              dot(st,vec2(269.5,183.3)) );
    return -1.0 + 2.0*fract(sin(st) *
                    43758.5453123);
}

// Gradient Noise by Inigo Quilez - iq/2013
// https://www.shadertoy.com/view/XdXGW8
// This is a 2D gradient noise. Input your texture
coordinates as argument, scaled properly.
```

```
float noise(vec2 st)
{
    vec2 i = floor(st);
    vec2 f = fract(st);

    vec2 u = f*f*(3.0-2.0*f);

    return mix( mix( dot( random2(i + vec2(0.0,0.0) ),
                                    f - vec2(0.0,0.0) ),
                     dot( random2(i + vec2(1.0,0.0) ),
                                    f - vec2(1.0,0.0) ), u.x),
                mix( dot( random2(i + vec2(0.0,1.0) ),
                                    f - vec2(0.0,1.0) ),
                     dot( random2(i + vec2(1.0,1.0) ),
                                    f - vec2(1.0,1.0) ), u.x), u.y);
}
```

Multi-octave Perlin

Multiple "rings" in frequency space

Each ring scaled after the 1/f rule!

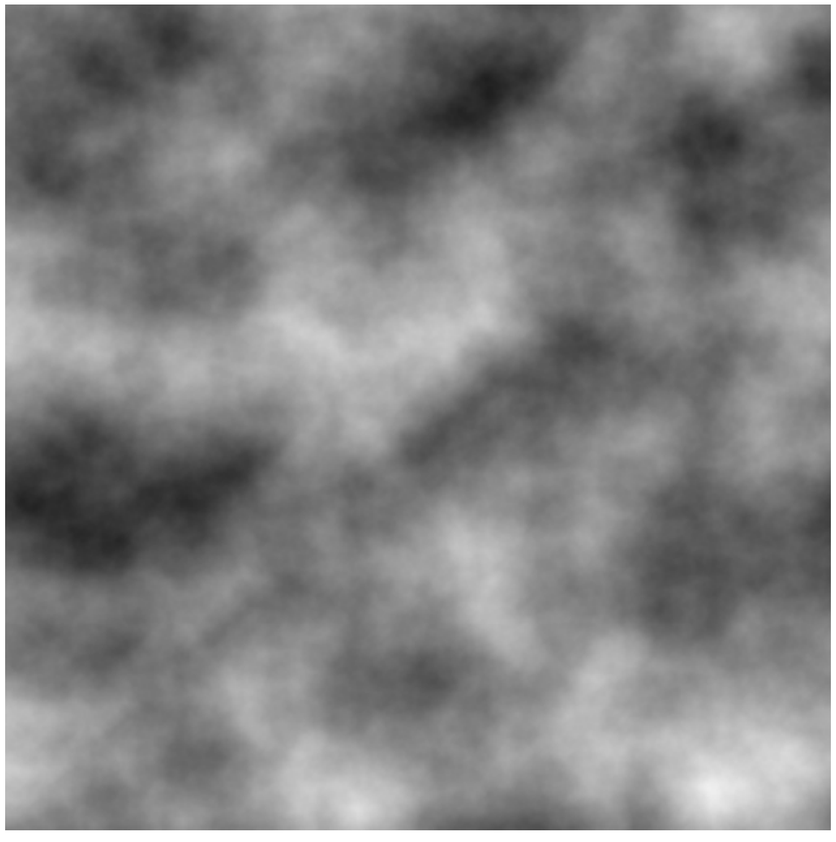


```
void makeperlintexture()
{
   int x, y;
   float i[3];
   char val;

   for (x = 0; x < 128; x++)
   for (y = 0; y < 128; y++)
   {
// 5 octaves:

      i[0] = x / 32.0;
      i[1] = y / 32.0;
      i[2] = 0.0;

      val = (char)(noise3(i) * 128.0)+128;

      i[0] = x / 16.0;
      i[1] = y / 16.0;
      i[2] = 2.0;

      val += (char)(noise3(i) * 64.0);

      i[0] = x / 8.0;
      i[1] = y / 8.0;
      i[2] = 3.0;

      val += (char)(noise3(i) * 32.0);

      i[0] = x / 4.0;
      i[1] = y / 4.0;
      i[2] = 4.0;

      val += (char)(noise3(i) * 16.0);

      i[0] = x / 2.0;
      i[1] = y / 2.0;
      i[2] = 5.0;

      val += (char)(noise3(i) * 8.0);

      ptex[x][y][0] = val;
      ptex[x][y][1] = val;
      ptex[x][y][2] = val;
   }
}
```

Twice the frequency - half the amplitude!

Result

Similar to the other methods



Gradient noise vs FFT (FHT)

Some say FFT is slow. Questionable today!

Gradient noise claimed to be very fast. (Compared to what?)

Frequency space processing much simpler algorithm (simple weighting curve, based on 1/f, FFT) and great control, but requires O(NlogN) operations.

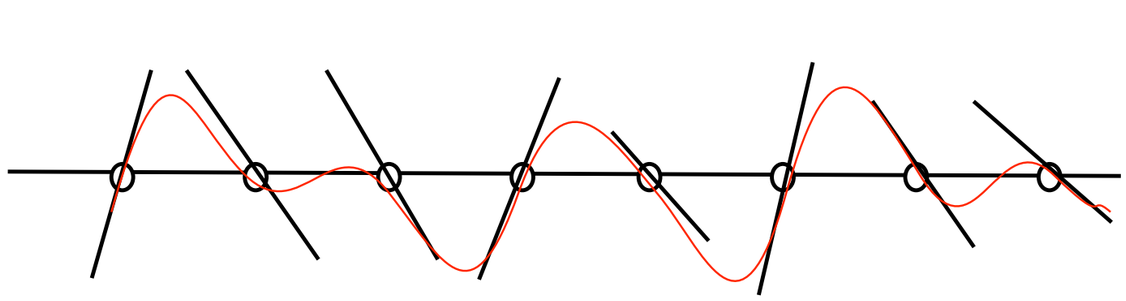
One pass Gradient noise faster... but don’t we need many?

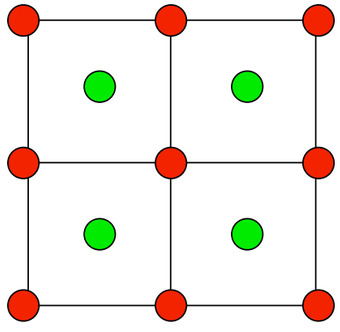
Artifacts

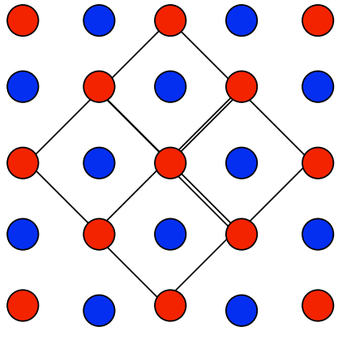
Diamond square and Perlin noise are incomplete! They both "lock" in certain points, only producing certain phases of the signal.

I.e. produce only the cosine part of a signal and skipping the sin!

This can be corrected by generating two sets of the signal, with a proper offset!







Applications of FBM

Terrains

Textures, texture detail

Smoke

Etc...

Feature comparison

Scalability: Diamond square and Square square very easy to scale to more detail but hard to expand to new patches.

Perlin/gradient easy to scale if you add additional octaves - which degrades performance. Strongest point: *Easy to expand to new patches* + easy to parallellize.

Frequency plane filtering can not be expanded.

Control: Frequency plane filtering has extreme control. The others depend on weights on octaves.

Parallelization: Gradient noise independent of neighbors - efficient to parallelize!

Your application needs may decide.

Repeating textures

Frequency plane filtering creates repeating textures automatically!

Diamond square can, trivially

Some variants of gradient noise can, but not the original Perlin noise

Good feature for texture generation.

Adapt frequency behavior for gradient noise to other needs

Gradient noise can be adjusted by changing the frequency variation (lacunarity) as well as the amplitude (gain).

General advice: Stay close to 2 and 1/2!

Performance comparison

Produce an NxN image

Diamond square: $O(N^2)$

Square square: $O(N^2)$

Single octave gradient: $O(N^2)$

Multi octave gradient: $O(N^2 \log N)$

Frequency plane filtered: $O(N^2 \log N)$

All produce similar results except single octave gradient noise.

Square square and FP filter improves quality but raises the constant!

Parallelism

Parallel implementations of Diamond square, Square square and Frequency plane filtering all require multiple passes!

Perlin noise is calculated in one pass = *fragment shader friendly!*

This is where Perlin and similar shines! Locality!

What more can we do with the terrain?

- Add water, calculate rivers and lakes
- Erosion effects (esp along rivers)
- Roads
- Support caves, steep cliffs and chasms
- Vegetation and buildings
- Expand into new patches
- Multitexturing for different kinds of locations (slopes, height)
- Different generation for different climates/biomes (mountains, deserts...?)

Multi-patch terrains

Terrains are never drawn one triangle at a time.

But very large terrains should not be drawn as a single model!

- Too much geometry out of view
- Too much data loaded in VRAM

Split to patches!

- Use frustum culling
- Re-generate or swap to CPU/disk as needed
- LOD on the patches is a good idea

Conclusions of terrain generation

The backbone of procedural environment generation!

Fractal or noise? Same thing!

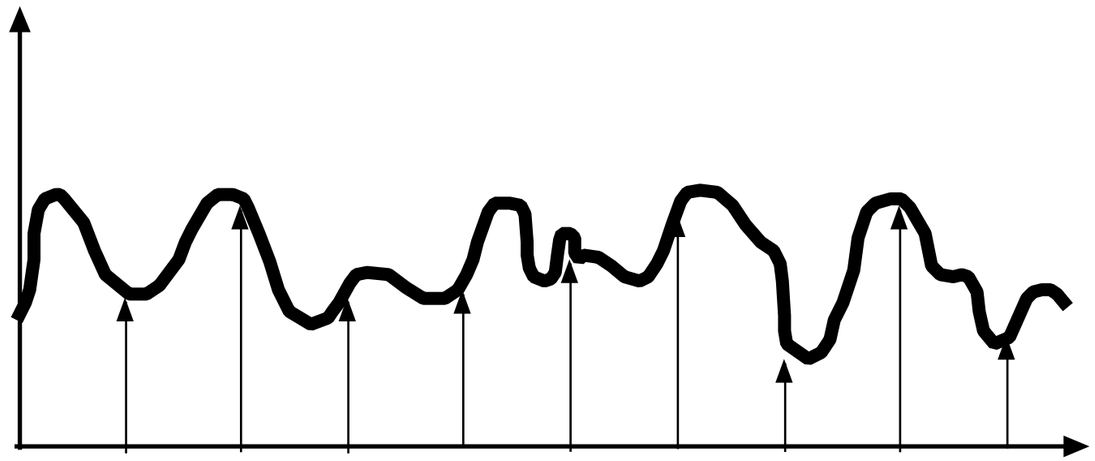
Higher frequencies - lower amplitudes. (Typical for natural images as well as a rule in fractals.)

Perlin and similar good for parallelism, but not the lowest computational complexity!

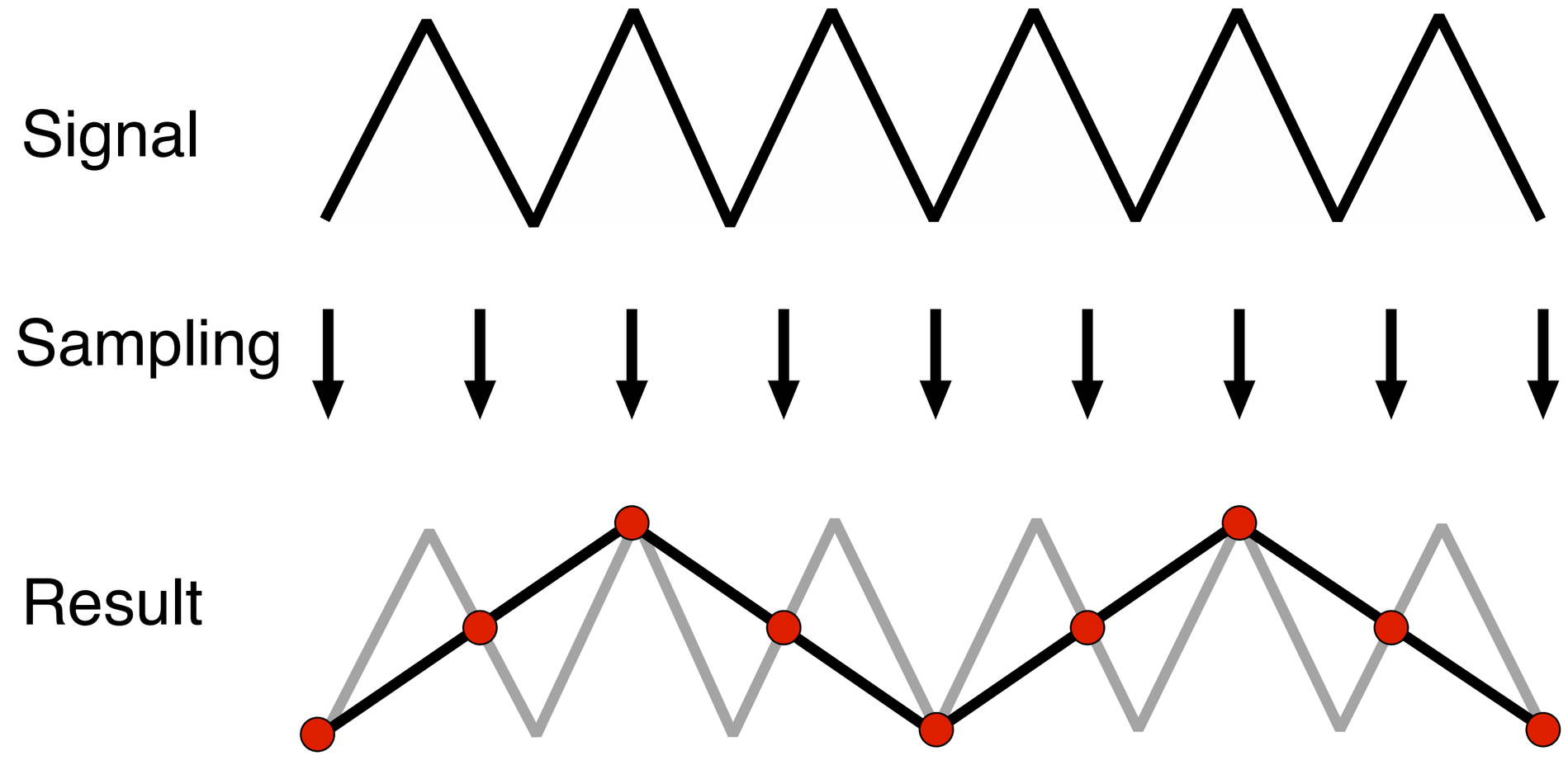
Other methods are better for repeating textures.

Anti-aliasing

Sampling



A digital image is a sampled version of an underlying analog 2D signal.


Signal
Sampling
Result

Frequency space

Any signal can be decomposed into sinus waves

The amplitudes of all sinus waves form a new space - frequency space

This space exists in any dimension. An image is a 2D signal and has a 2D frequency space.

Sampling + reconstruction

When presenting the digital signal, it is reconstructed to an analog signal.

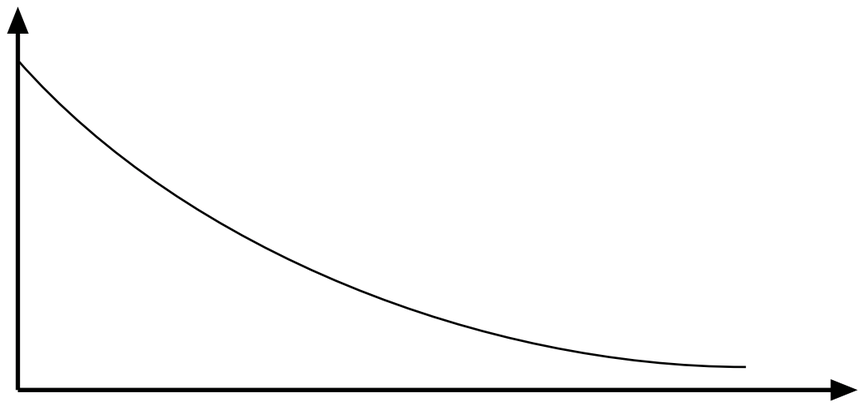
A signal can be decomposed into different frequency bands.

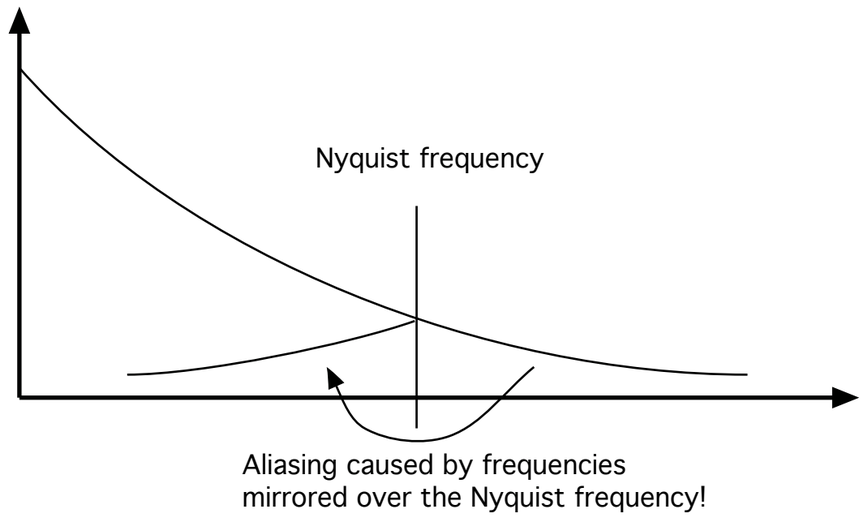
What parts of the original signal that are accurately reconstructed depend on the frequencies.

Aliasing in frequency space!

Which frequencies are preserved and which cause problems?

Amplitudes usually lower for higher frequencies!

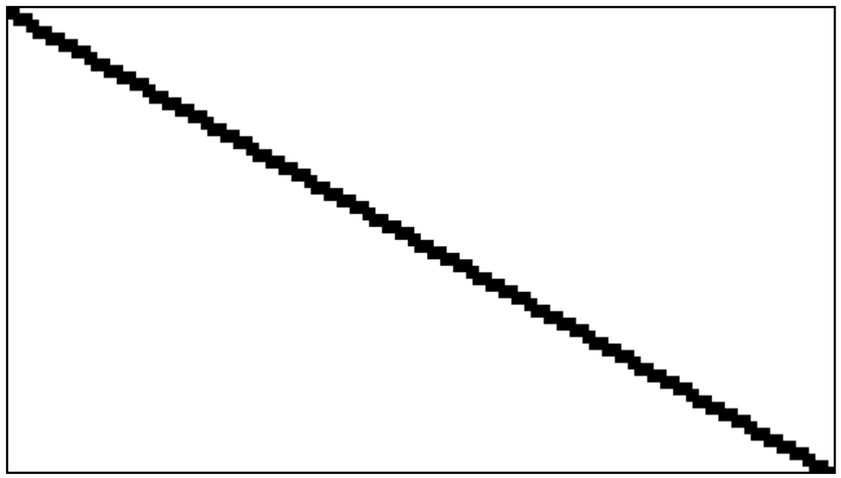




Anti-aliasing methods

- Linear post-filtering. Bad.
- Super-sampling: Split pixels in sub-pixels. Check how many sub-pixels that hit one pixel.
- Area sampling: Calculate covered areas of each pixel.
- Non-linear post-filtering.

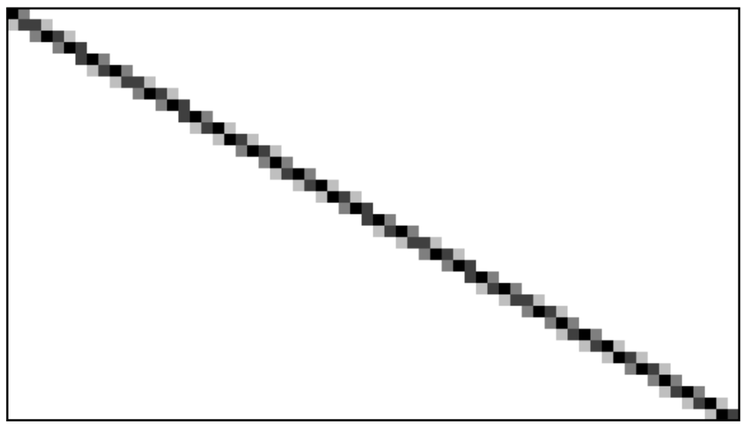
Supersampling



Draw in a high-res image buffer

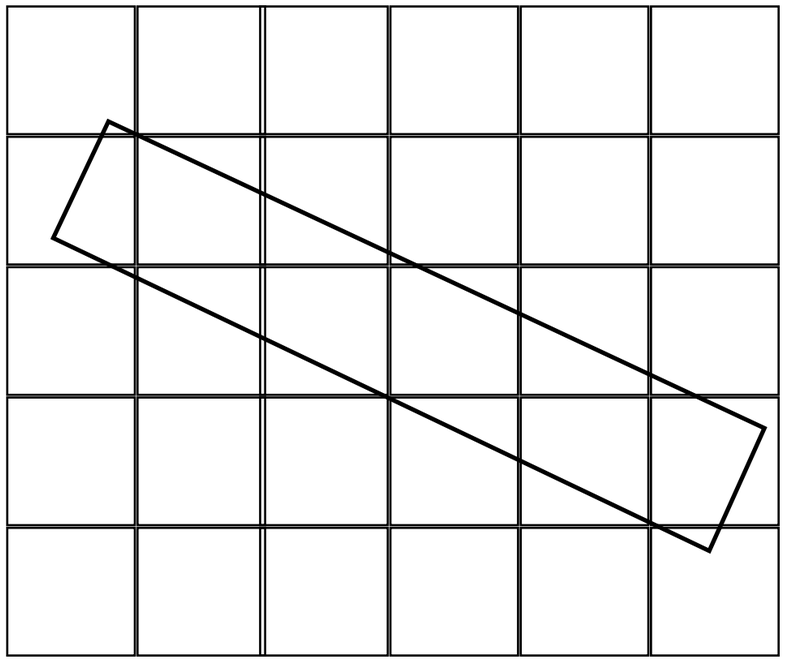
Simple but slow and memory-demanding

Sample down to destination buffer



Area sampling

Determine how much of each pixel that is covered by the shape



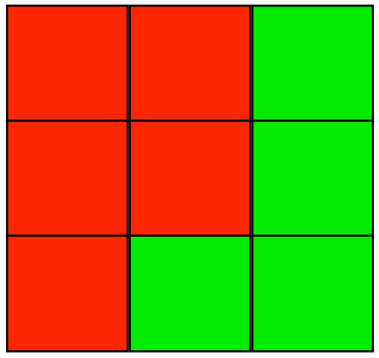
Multisampling

Variant of supersampling

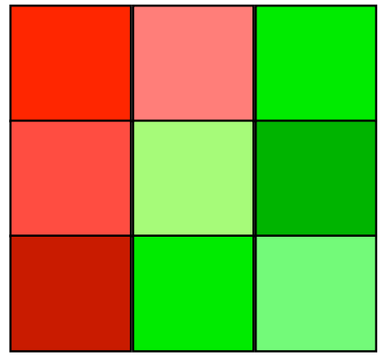
More efficient

Only improves edges

Multisampling



Multisampling: Only one execution of fragment shader for all samples from the same geometry



Supersampling: One execution of fragment shader for each sample

Multisampling

Less fragment processing

Fewer texture accesses

Same number of memory writes and same post-processing

FXAA = Fast approXimative AA

Post-processing

No higher resolution image

Non-linear filter

Don’t filter patterns

Several recent methods of this kind

Anti-aliasing in OpenGL

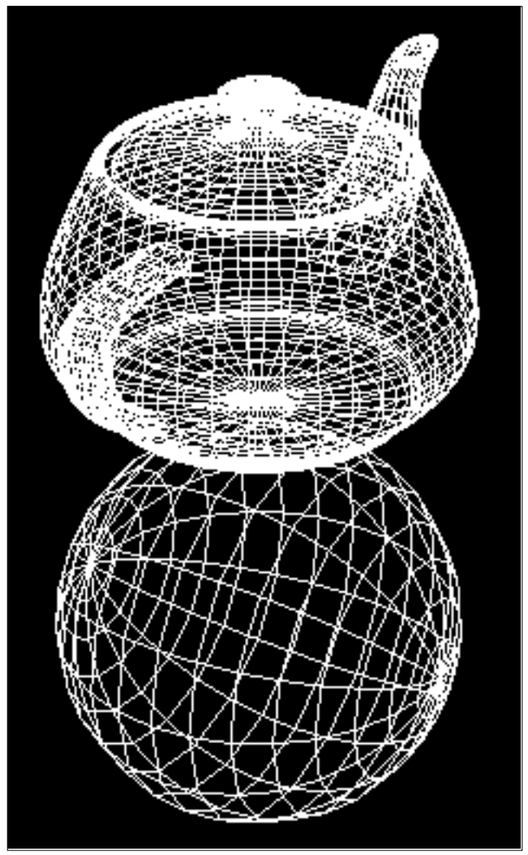
GL_POLYGON_SMOOTH - Old built-in, avoid

Accumulation buffer tricks: Obsolete, avoid

glEnable(GL_MULTISAMPLE); Preferred!

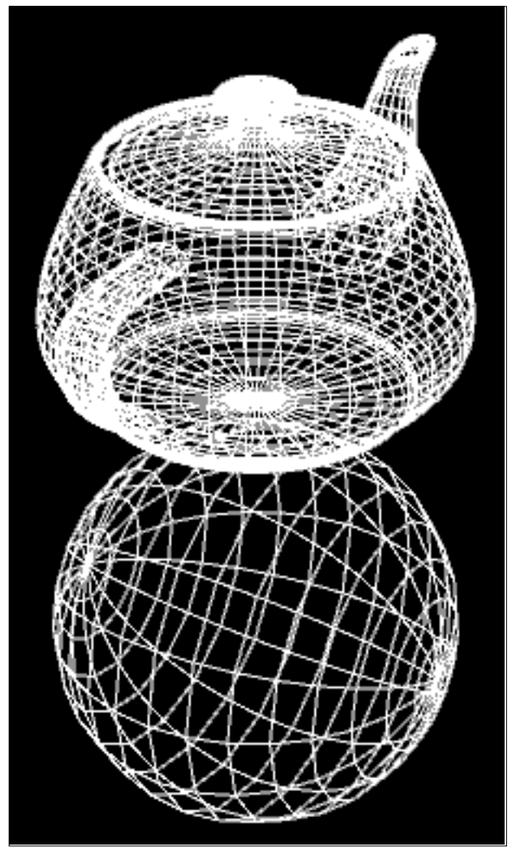
Can also be done by shaders. Usually unnecessary.

FSAA example



No AA

AA



Anti-aliasing

We will return to anti-aliasing in the ray-tracing part (with even more elaborate supersamling)

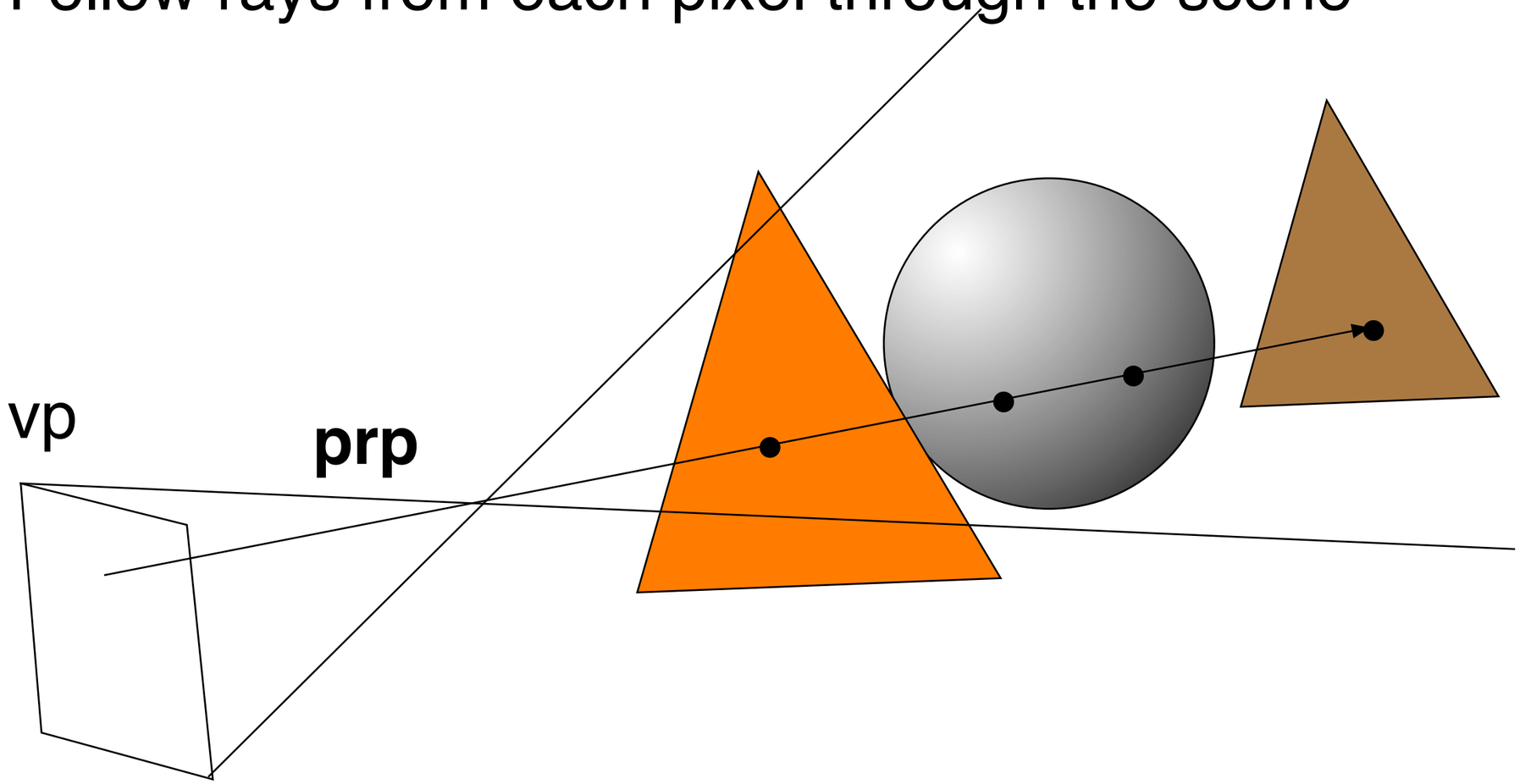
Off-line (?) rendering and global illumination

Performance demanding methods that give better lighting

Ray-casting
Ray-tracing
Radiosity
Photon mapping
Path tracing

Ray-casting

Follow rays from each pixel through the scene



Full 3D raycasting

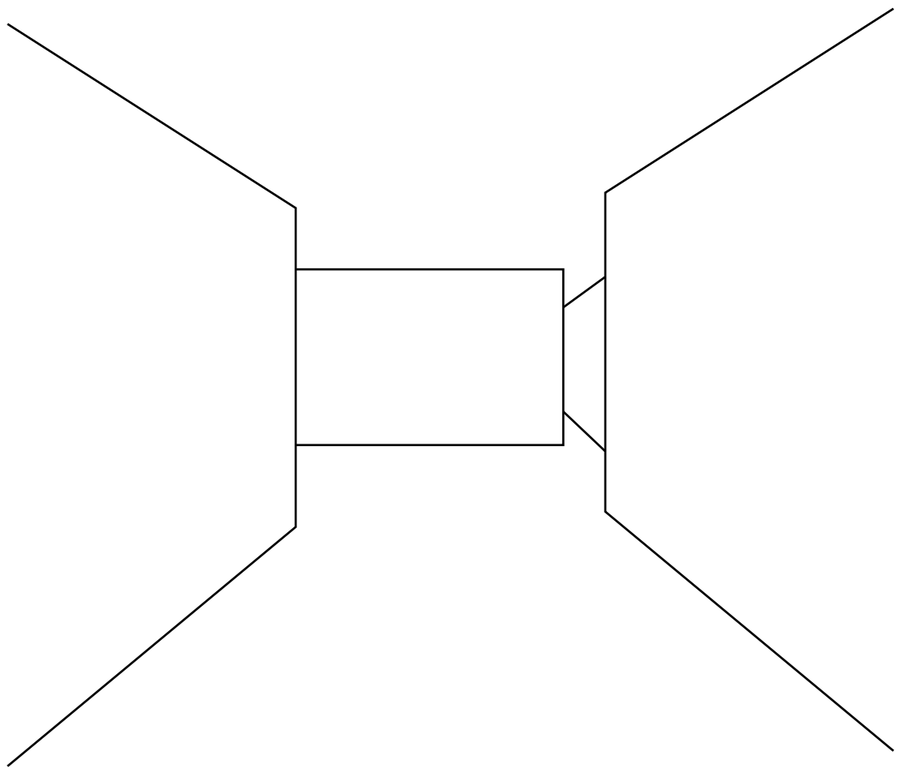
for every pixel (x,y) in the image

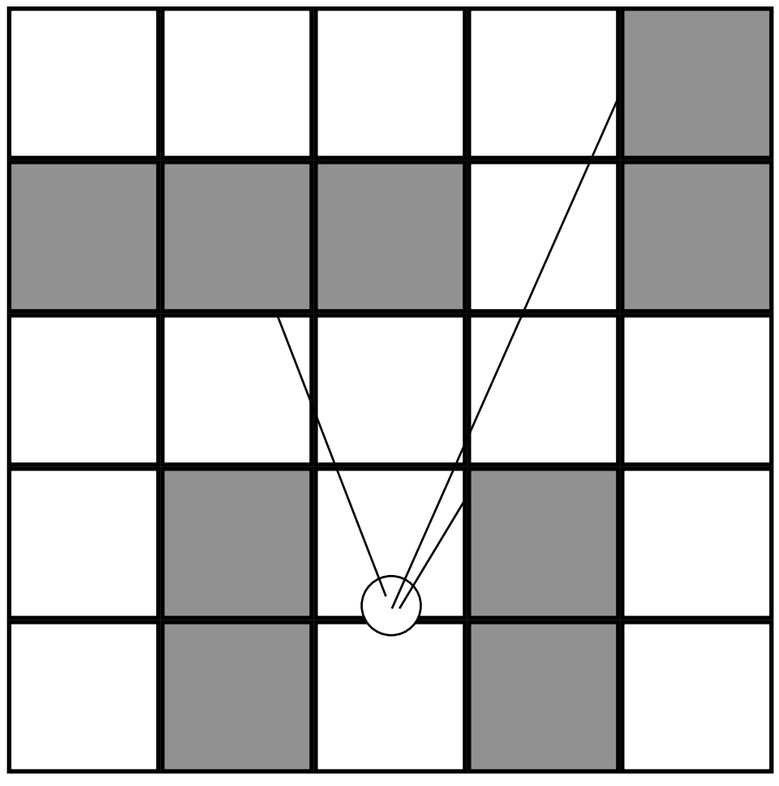
calculate a ray from the pixel through the camera (cop) and through the scene

calculate intersecions with all objects in the scene

the pixel value is calculated from the closest intersection found

Raycasting in 2D or 3D grid ("Ray-marching")





Ray-tracing

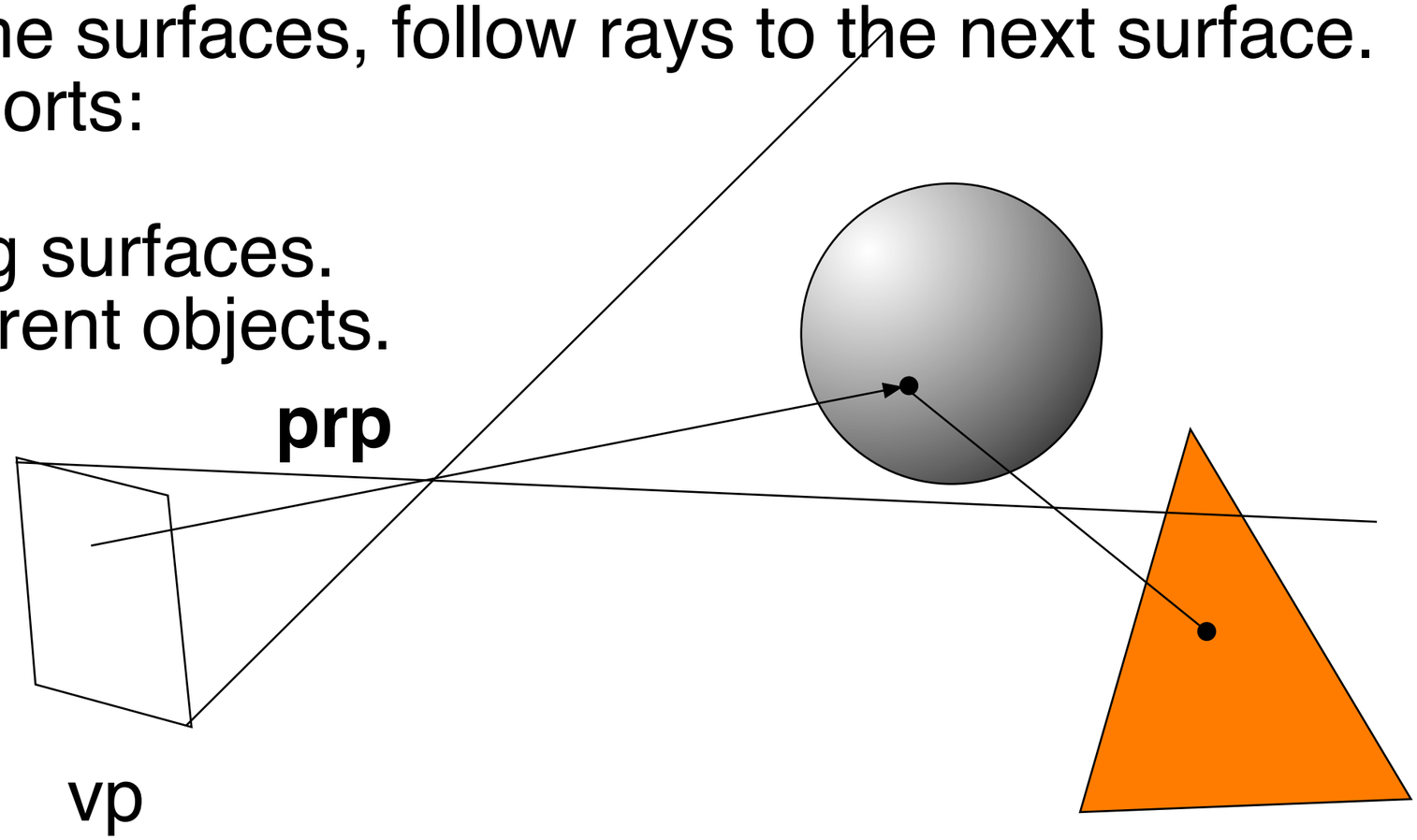
The classic method for rendering realistic images of shiny and transparent objects with shadows.

IMAGE REMOVED

Ray-tracing basics

Cast rays from the camera as before.
From some surfaces, follow rays to the next surface.
This supports:

- Mirroring surfaces.
- Transparent objects.



At the intersection

Three things can happen when a ray intersects an object

1) Lighting, non-mirroring reflection

2) Reflection

3) Refraction

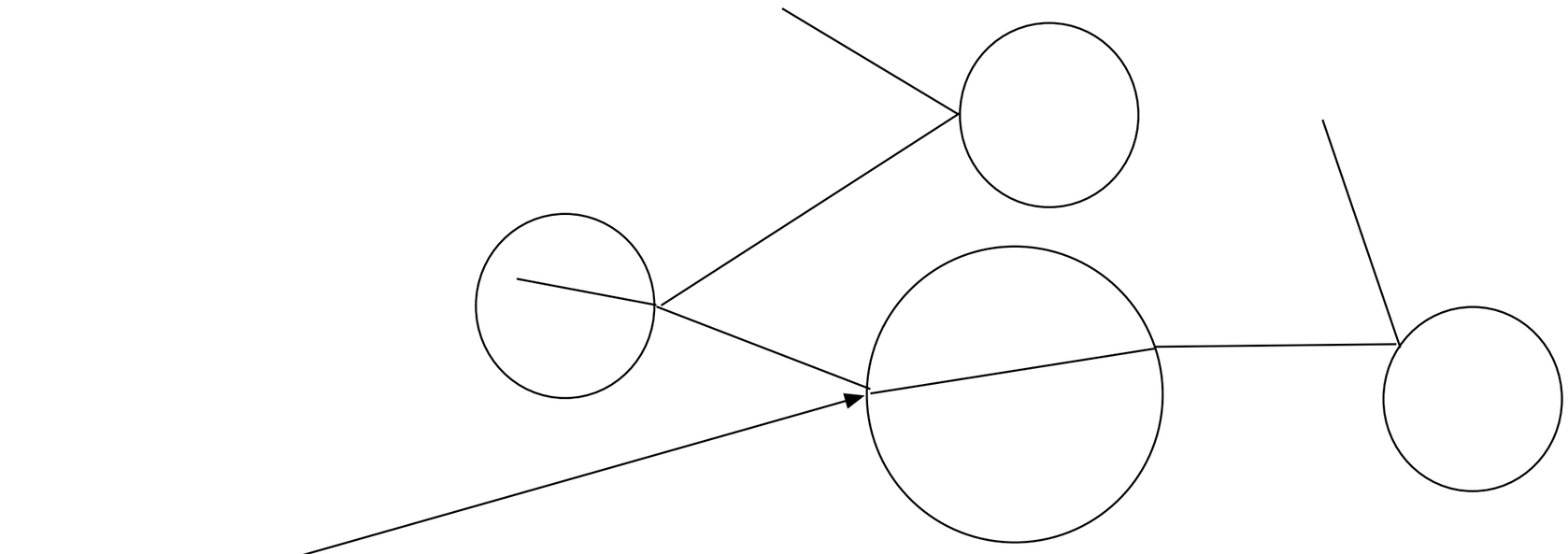
Non-mirroring reflection

Apply e.g. the three-component light model

Ambient light
Diffuse reflection
Specular reflection

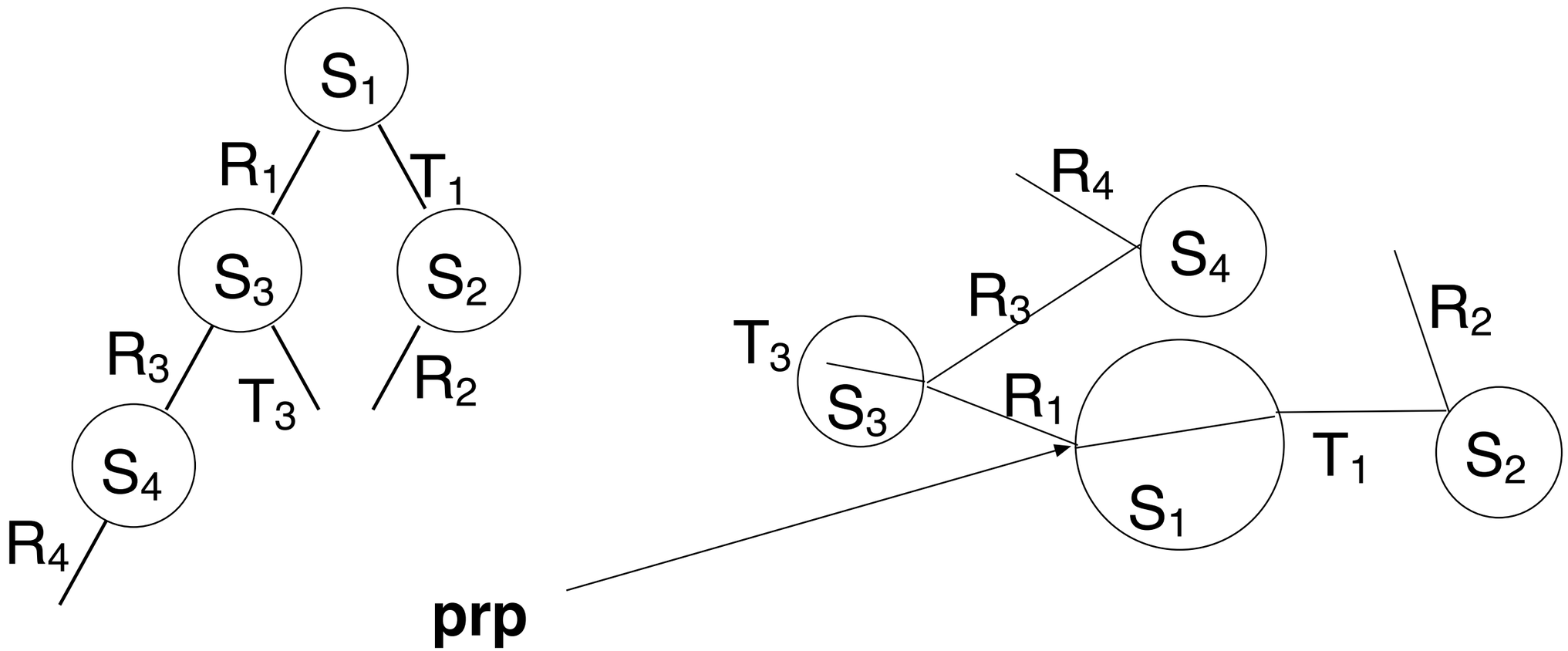
(or other model)

Reflection and refraction



Recursive process!

The ray-tracing tree



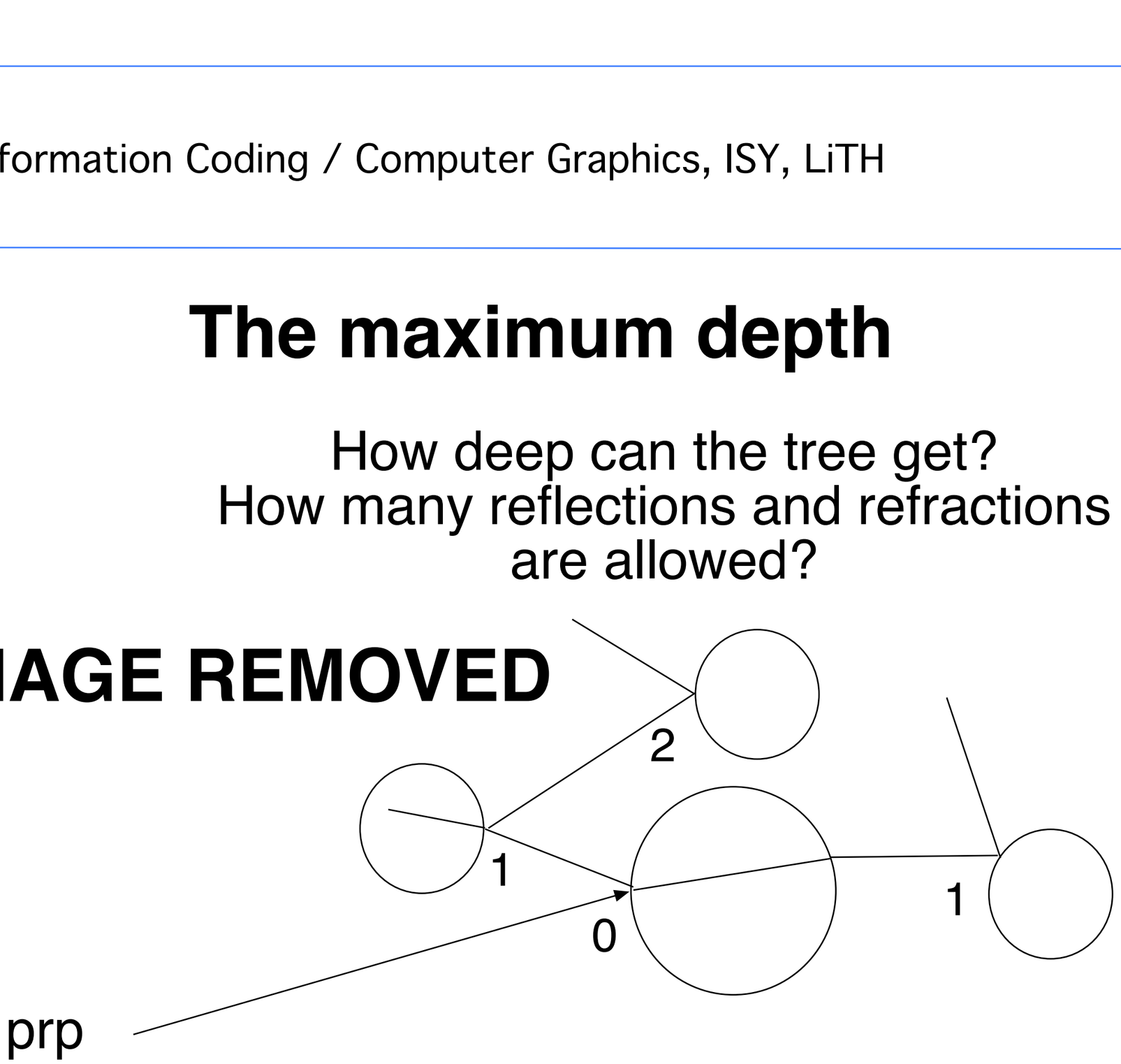
The maximum depth

0

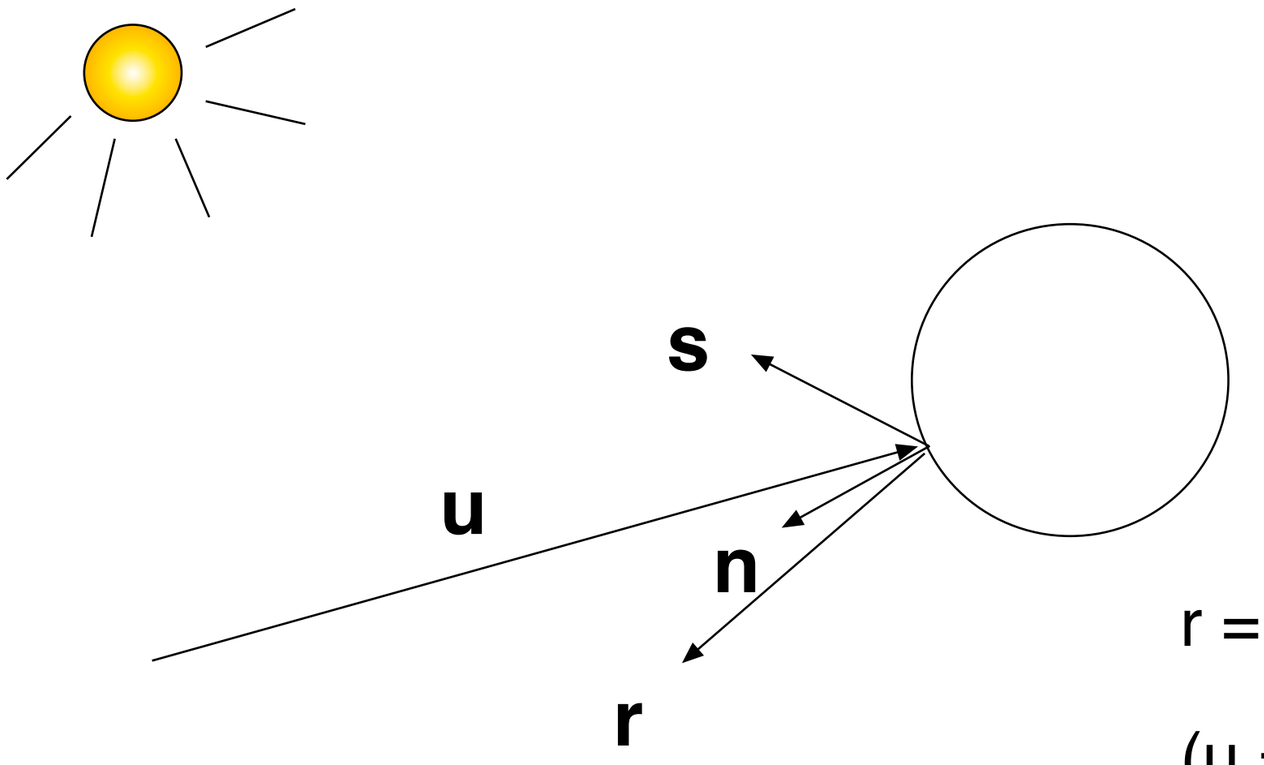
How deep can the tree get?
How many reflections and refractions are allowed?

1 **IMAGE REMOVED**

2



Reflections



Look for more contributions here

$r = u - (2u \cdot n)n$

(u = direction vector of the incoming ray)

Refractions

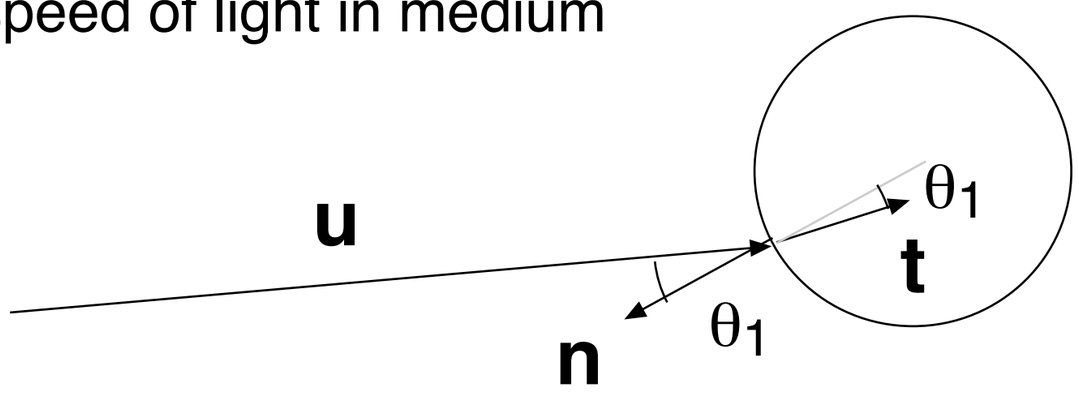
Snell's law:

$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$

Refraction index:
$\eta = c / v$
c: speed of light in vacuum
v: speed of light in medium

Transparent object

Look for more contributions here



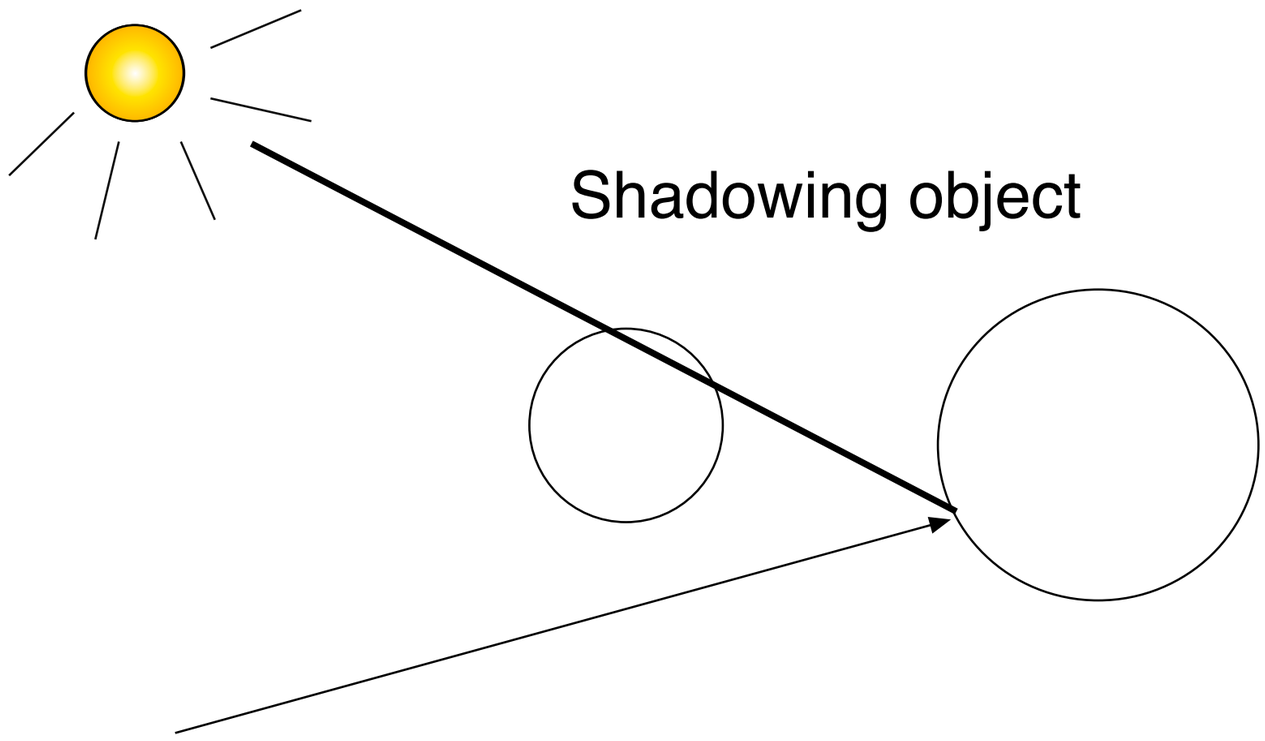
Outgoing angle is given by incoming angles and the density of each material.

Summing up

The total intensity is the sum of

- ambient light
- diffuse reflections from each light source
- specular reflections from each light source
- mirroring reflections
- refractions

The shadow ray



Object that can have illumination (shading) by diffuse or non-perfect specular reflections

The shadow ray tells if a light source can contribute to illumination

Ray-surface intersections

Ray equation:

$\mathbf{p} = \mathbf{p_0} + \mu\mathbf{u}$, $\mu > 0$

Combine with the equation of the surface.

Easiest surface: Sphere!

$x^2 + y^2 + z^2 = r^2$

Not quite as easy as for ray casting, since $\mathbf{p_0}$ can now be any point.

function RayTrace(p_0, u, depth)

if depth > max then return BLACK

μ := FindIntersection(p_0, u) // Returns more data, see below

if μ <= 0 then return BACKGROUND_COLOR

I_{local} :=0
I_R := 0
I_T := 0

if $k_a \neq 0$ and $k_d \neq 0$ and $k_s \neq 0$ then
    I_{local} := ka*Ia + $\sum$ (diffuse shading + specular shading)
// Sum is for all visible light sources

if $k_R \neq 0$ then
    R := CalculateReflection(u, N)
    I_R := RayTrace(p_0 + μ*u, R, depth+1)

if kT ≠ 0 then
    T := CalculateRefraction(u, N, h_1, h_2)
    I_T := RayTrace(p_0 + μ*u, T, depth+1)

return I_{local} + I_R + I_T

Problems in ray-tracing

- Computational load
- Aliasing
- Lack of realism due to extreme sharpness
- No indirect light

Reducing object-intersection calculations

The same large world problems showing up again!

Don’t test everything, use some large world structure!

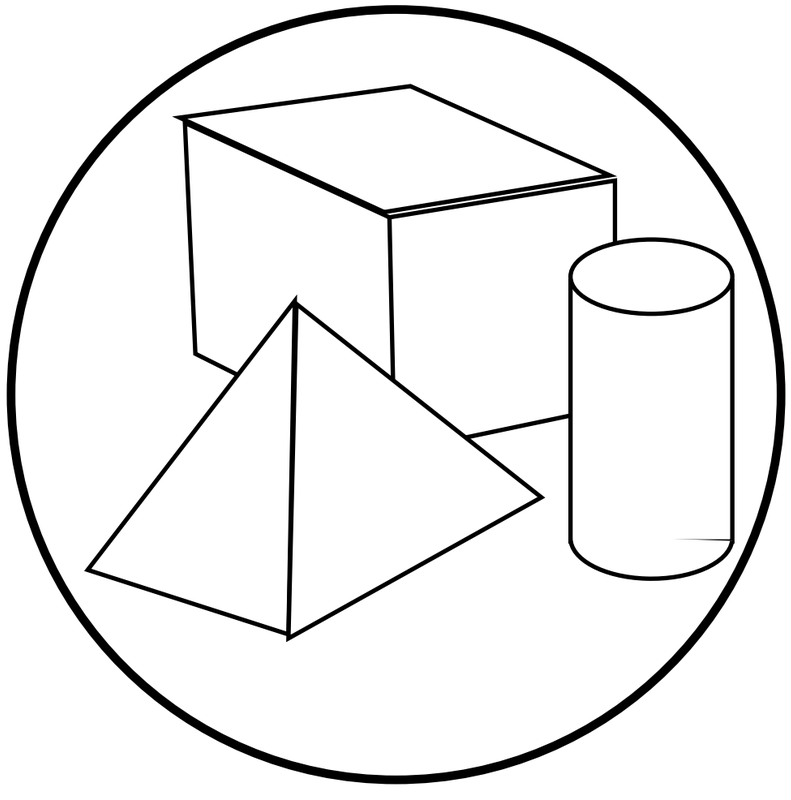
Reducing object-intersection calculations

Make sure that each ray doesn't have to be tested against all objects!

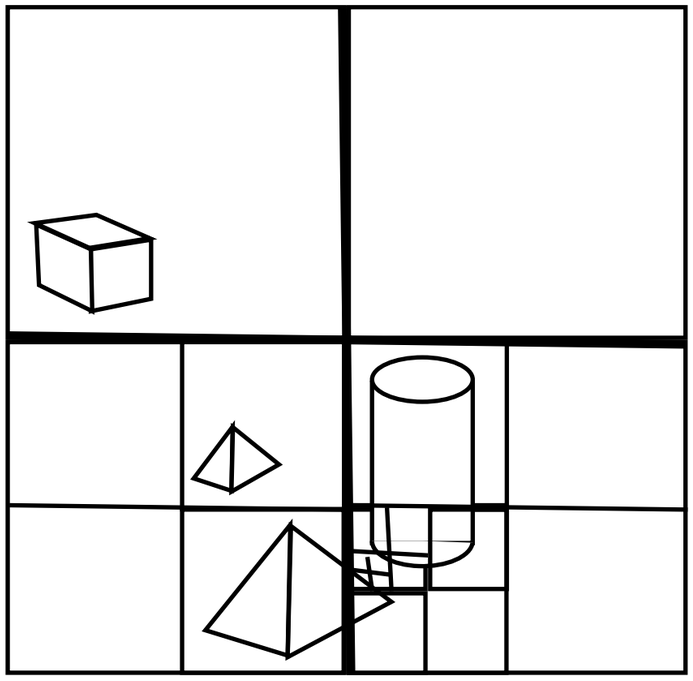
- Group objects within simple grouping surfaces (e.g. spheres or boxes)

- Divide space into cells
  - uniform or adaptive subdivision

Group objects into simple bounding objects



Divide space into cells



(Quad-tree)

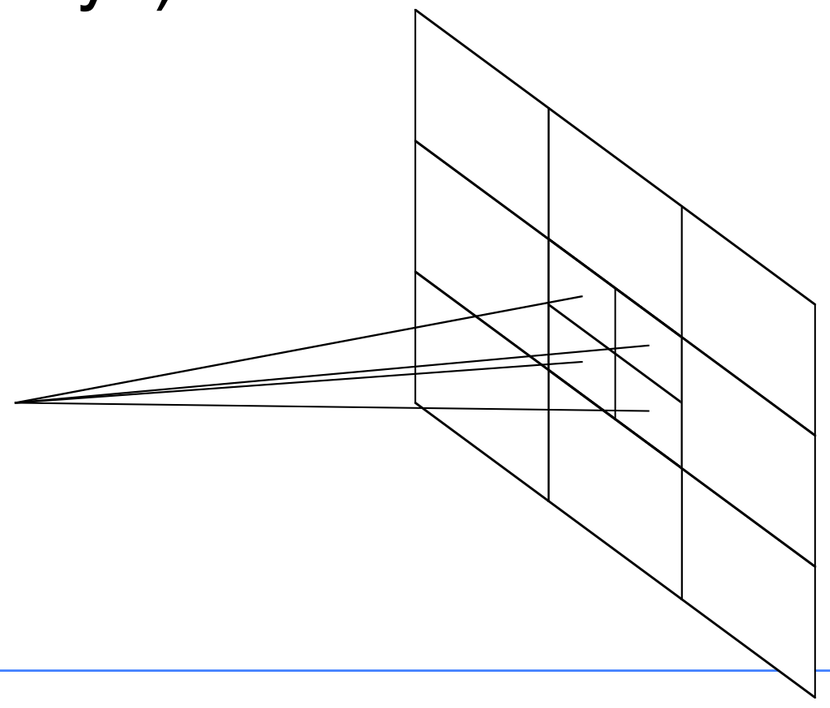
Same principle as in VSD and collision detection

Anti-aliasing in ray-tracing

- Supersampling
- Distributed ray tracing

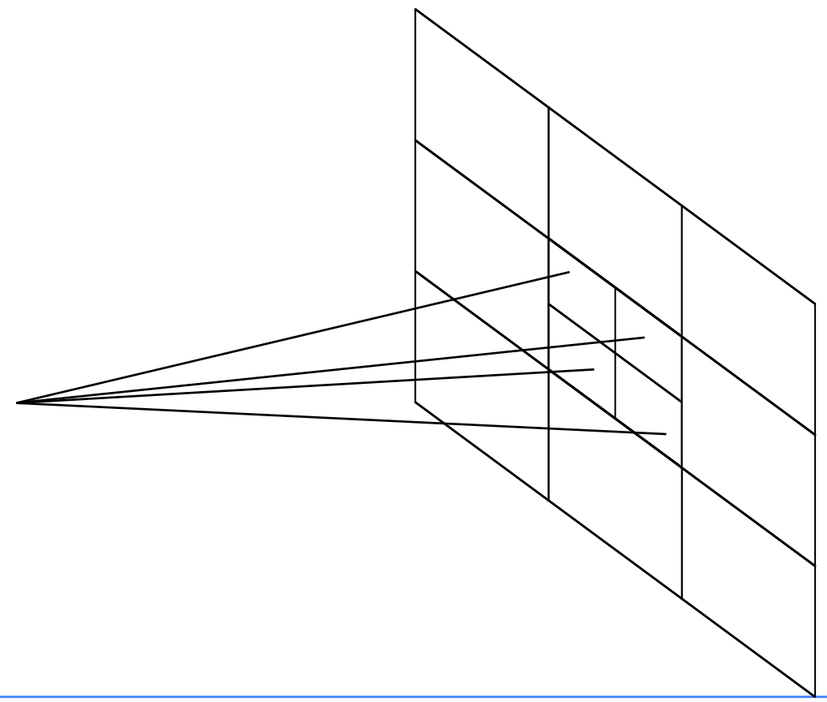
Supersampling
in ray-tracing

Send several rays for each pixel. (Typical for fast rendering: 4 rays)



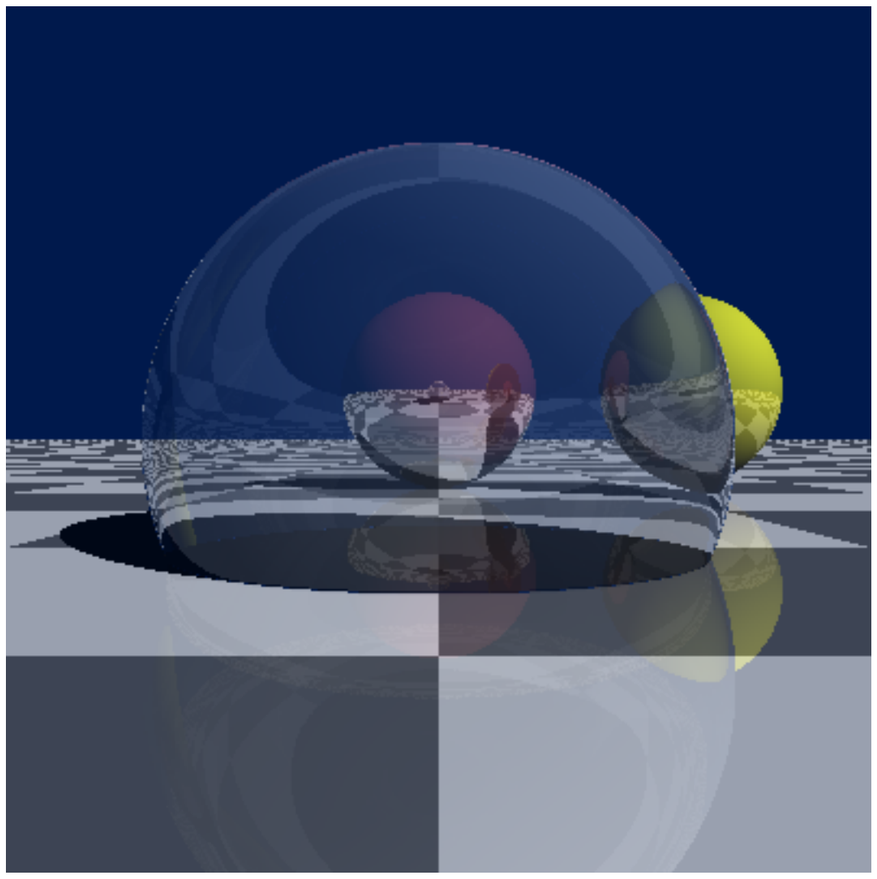
Distributed ray-tracing

Rays (multiple per pixel) are sent in a randomized pattern. Aliasing is replaced by noise!



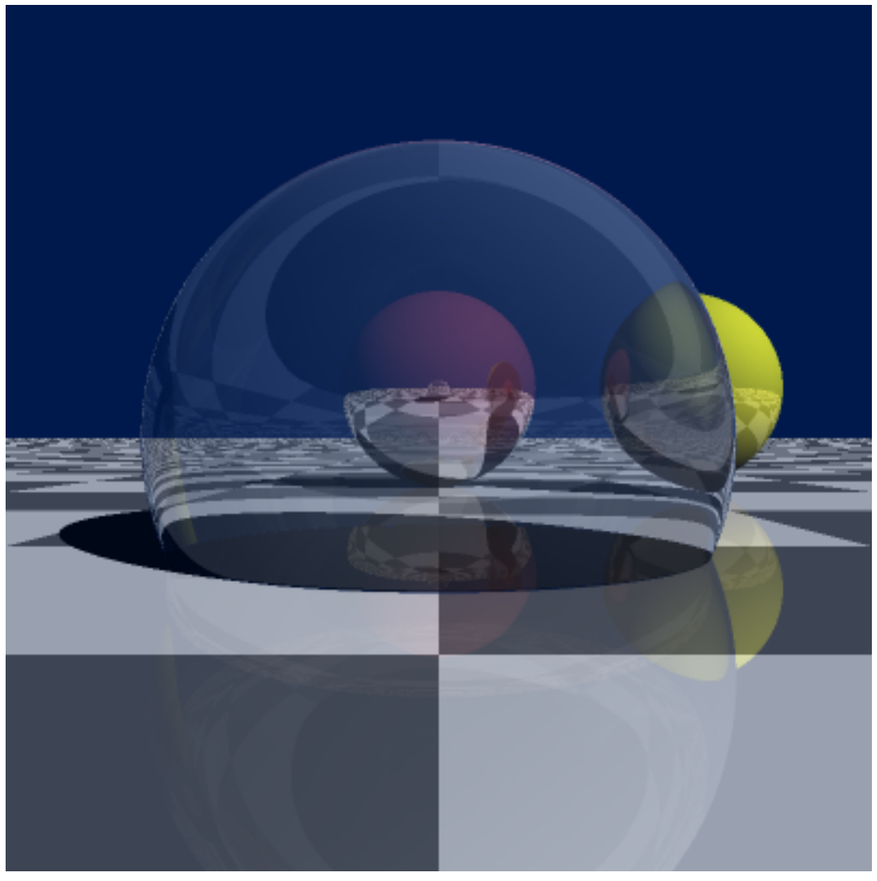
Anti-aliasing in ray-tracing

Example:
Without
anti-aliasing



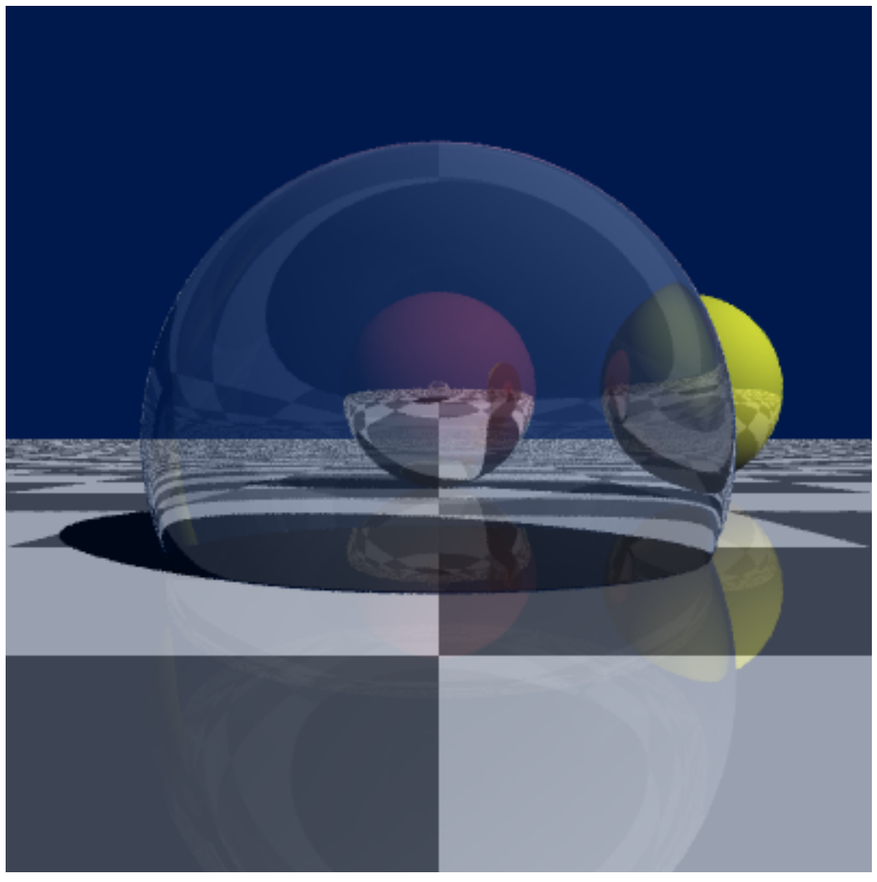
Anti-aliasing in ray-tracing

Example: Super-sampling (2x2)



Anti-aliasing in ray-tracing

Example:
Distributed
raytracing
(2x2)



Anti-aliasing in ray-tracing

Zoomed-in detail

Without anti-alias

Super-sampling

Distributed raytracing

Distributed ray-tracing

Anti-aliasing through stochastic sampling.

Also used for:
- gloss (fuzzy reflections)
- fuzzy translucency
- soft shadows
- depth of field (out-of-focus effect)
- motion blur

Gloss (fuzzy reflections)

IMAGE REMOVED

Jittering reflection ray

Fuzzy translucency

IMAGE REMOVED

Jittering refraction ray

Soft shadows

IMAGE REMOVED

Jittering shadow ray

Depth of field (out-of-focus effect)

IMAGE REMOVED

Jittering main ray (simulates the size of the lens)

Motion blur

IMAGE REMOVED

Jittering time

Ray-tracing in the fragment shader

Does not fit the GPU rendering model!

- Recursion not allowed! Use lists instead!

- Limited amount of data available to a shader - needs to use textures or other buffers for input

Not impossible but we may need to use tricks, multi-pass rendering etc.

Ray-tracing on GPU often done in ”GPU computing languages” (CUDA, OpenCL) rather than shaders.

The Turing GPU and later RTX boards

IMAGE REMOVED

RT cores = Raytracing cores

Special-purpose hardware in RTX GPUs

Available through Vulkan, CUDA etc.

RT cores

Accelerates ray-box and ray-triangle calculations

IMAGE REMOVED

Conclusions on ray-tracing

No longer an offline method!

Moving into real-time now!

Variations of ray-tracing a hot topic.

But... how about global illumination?

- Radiosity
- Photon mapping
- Path tracing