Appendices

Additional topics of interest in computer graphics.
These slides are not examinable.

A. *Constructive Solid Geometry*
B. *Antialiasing*
C. *Procedural textures*
D. *Perlin noise*
E. *Voxels*
F. *Particle systems*

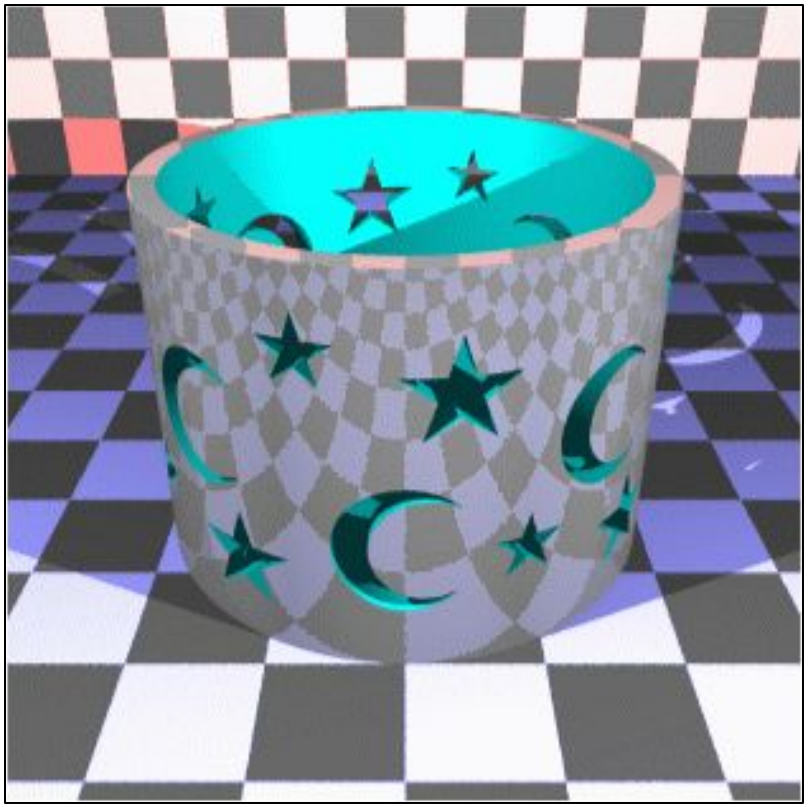
Alex Benton, University of Cambridge – alex@bentonian.com

Supported in part by Google UK, Ltd

Appendix A:
Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a ray-tracing technique which builds complicated forms out of simple primitives, comparable to (and more complicated than, but also more precise than) Signed Distance Fields.

These primitives are combined with the standard boolean operations: *union, intersection, difference*.

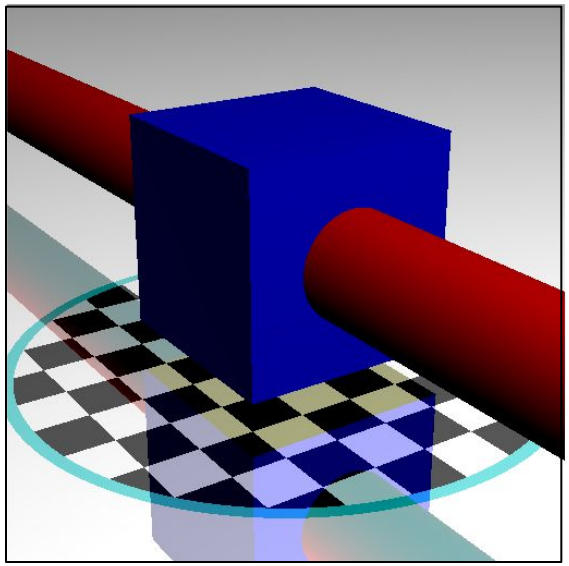


CSG figure by Neil Dodgson

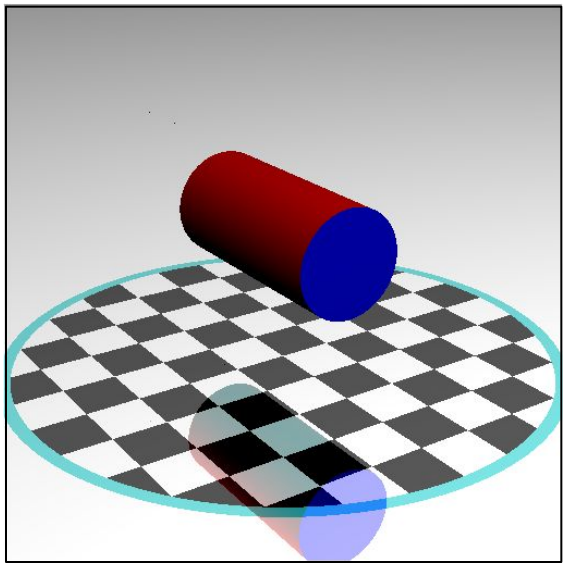
Constructive Solid Geometry

Three operations:

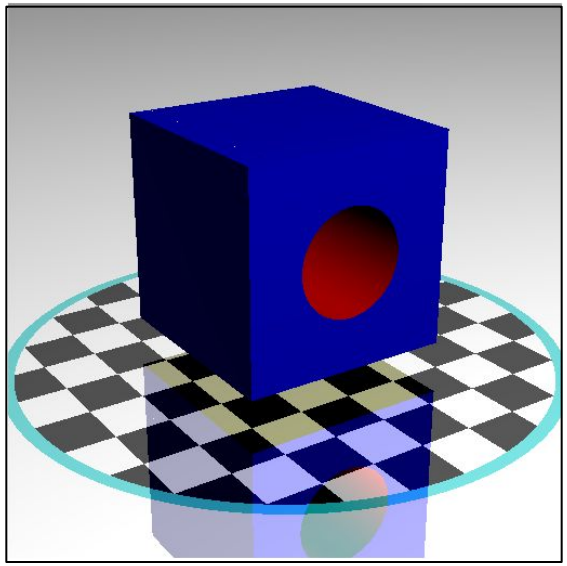
1. *Union*



2. *Intersection*



3. *Difference*



Constructive Solid Geometry

CSG surfaces are described by a binary tree, where each leaf node is a primitive and each non-leaf node is a boolean operation.

(What would the *not* of a surface look like?)

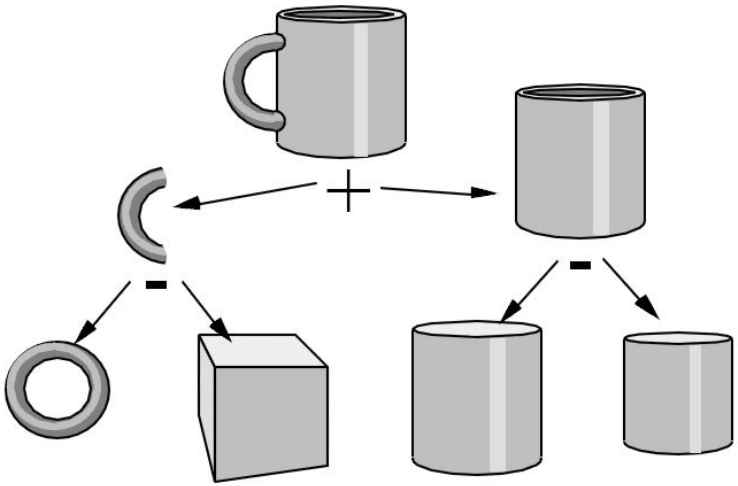
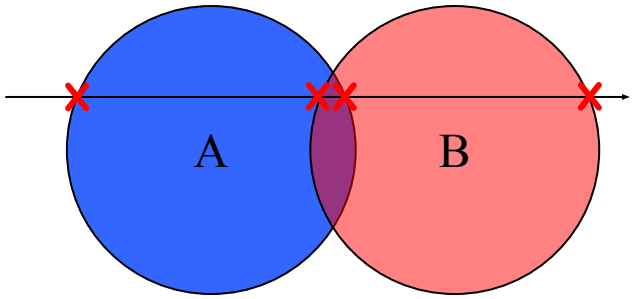


Figure from Wyvill (1995) part two, p. 4

Ray-tracing CSG models

For each node of the binary tree:



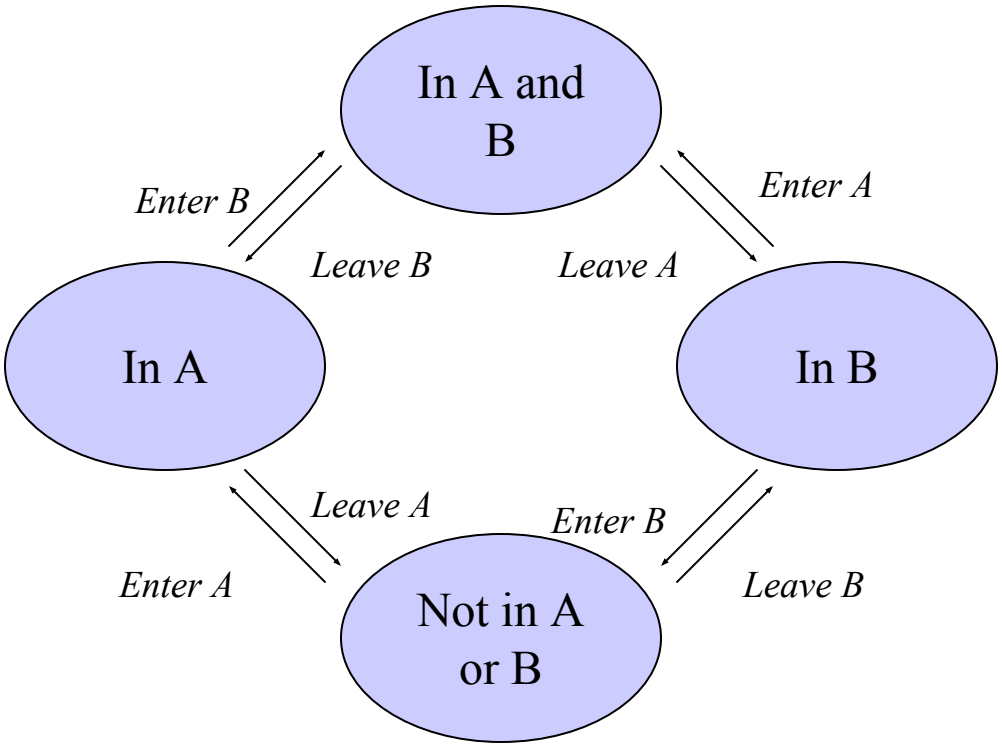
- Fire ray *r* at *A* and *B*.
- List in *t*-order all points where *r* enters of leaves *A* or *B*.
  - You can think of each intersection as a quad of booleans-- (*wasInA*, *isInA, wasInB, isInB*)
- Discard from the list all intersections which don't matter to the current boolean operation.
- Pass the list up to the parent node and recurse.

Ray-tracing CSG models

Each boolean operation can be modeled as a state machine.

For each operation, retain those intersections that transition into or out of the critical state(s).

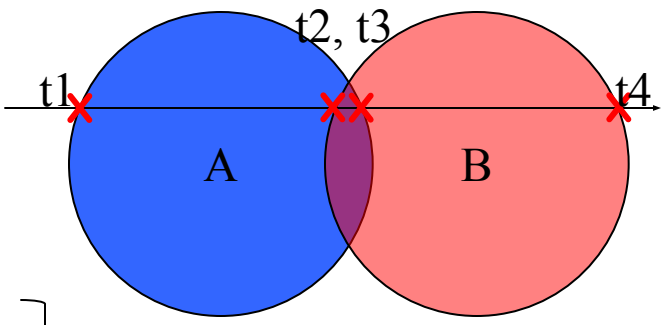
- Union: `{In A | In B | In A and B}`
- Intersection: `{In A and B}`
- Difference: `{In A}`



Ray-tracing CSG models

Example: Difference (A-B)

A-B	Was In A	Is In A	Was In B	Is In B
t1	No	Yes	No	No
t2	Yes	Yes	No	Yes
t3	Yes	No	Yes	Yes
t4	No	No	Yes	No




```
difference =
((wasInA != isInA) &&
 (!isInB)&&(!wasInB))
||
((wasInB != isInB) &&
 (wasInA || isInA))
```

Constructive Solid Geometry - References

- Jules Bloomenthal, *Introduction to Implicit Surfaces* (1997)
- Alan Watt, *3D Computer Graphics*, Addison Wesley (2000)
- MIT lecture notes: http://groups.csail.mit.edu/graphics/classes/6.837/F98/talecture/

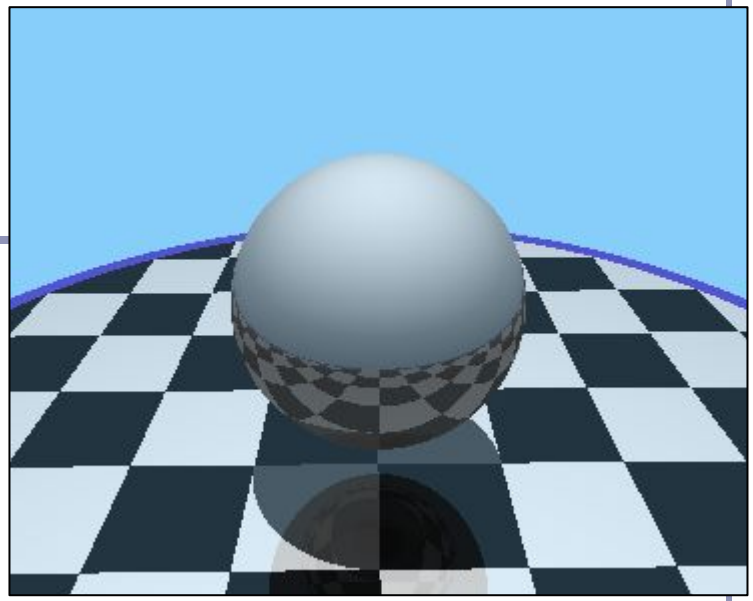
Aliasing

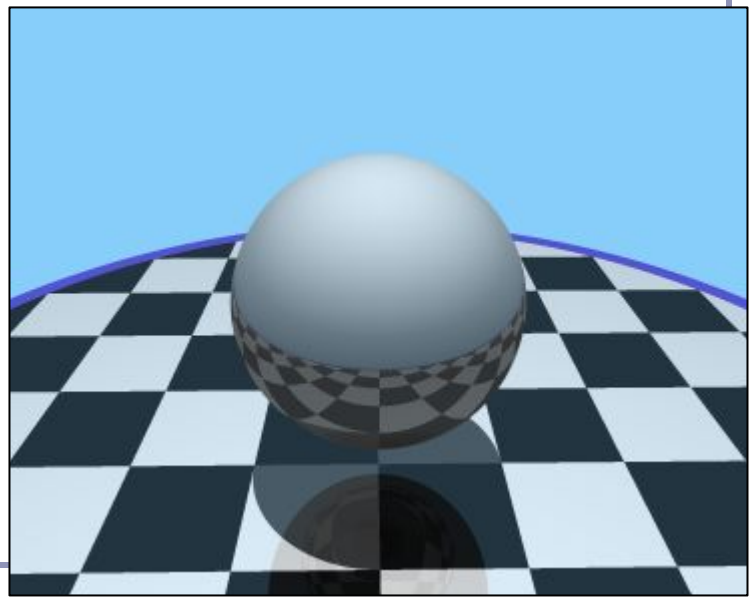
aliasing
/ˈeɪlɪəsɪŋ/
noun: **aliasing**
1. PHYSICS / TELECOMMUNICATIONS

the misidentification of a signal frequency, introducing distortion or error.

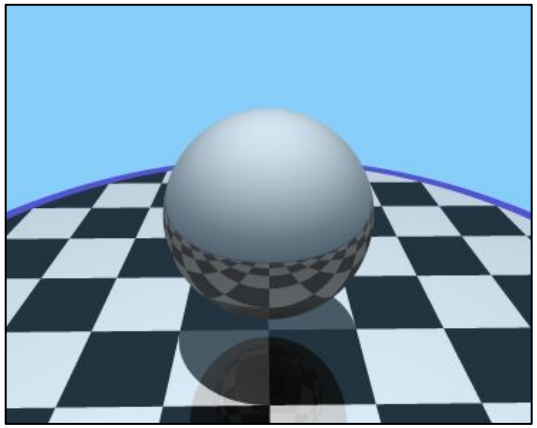
"high-frequency sounds are prone to aliasing"
2. COMPUTING

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.

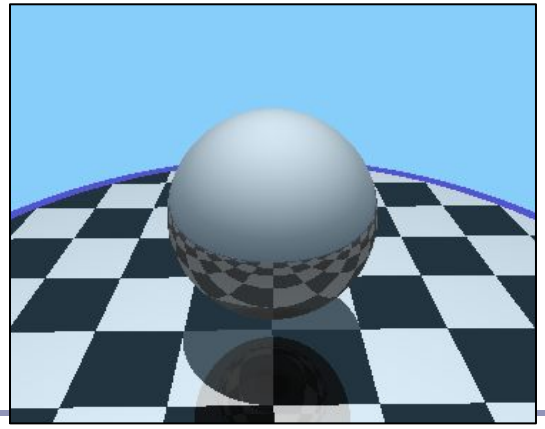




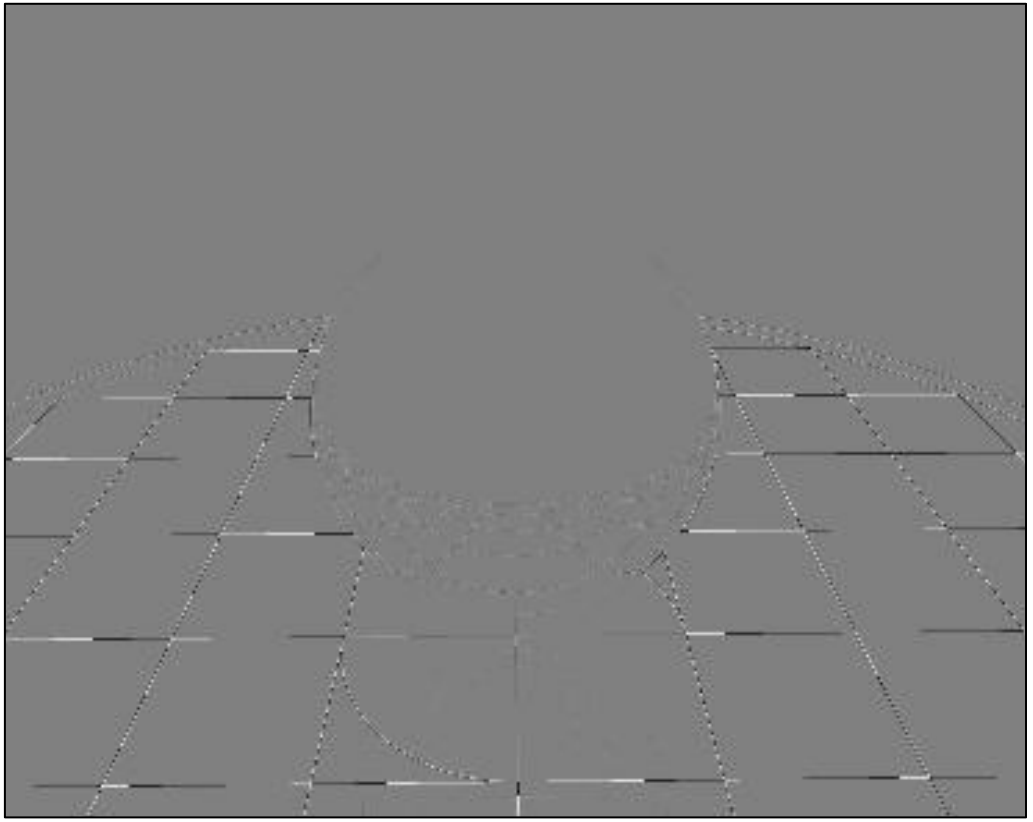
Aliasing



-



=



Antialiasing

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.

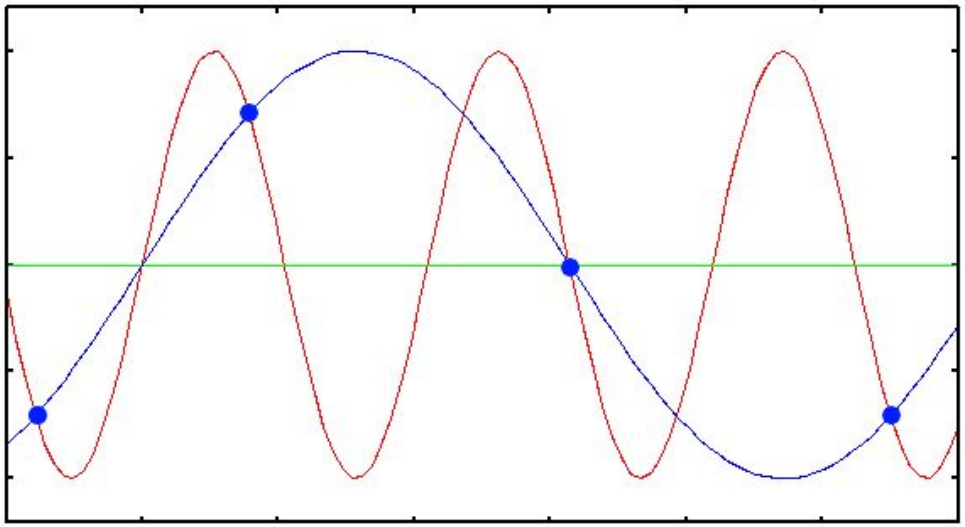


Image source: www.svi.nl

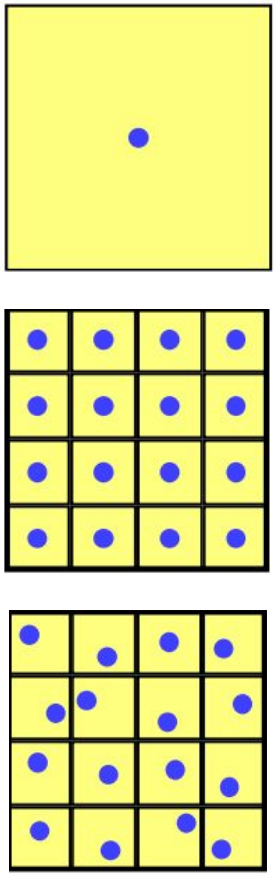
To avoid heavy computational load And also avoid sub-super-sampling artifacts, consider using *jittered super-sampling*.

Lecture note: Four printed slides removed here, reviewing antialiasing from last year's notes.

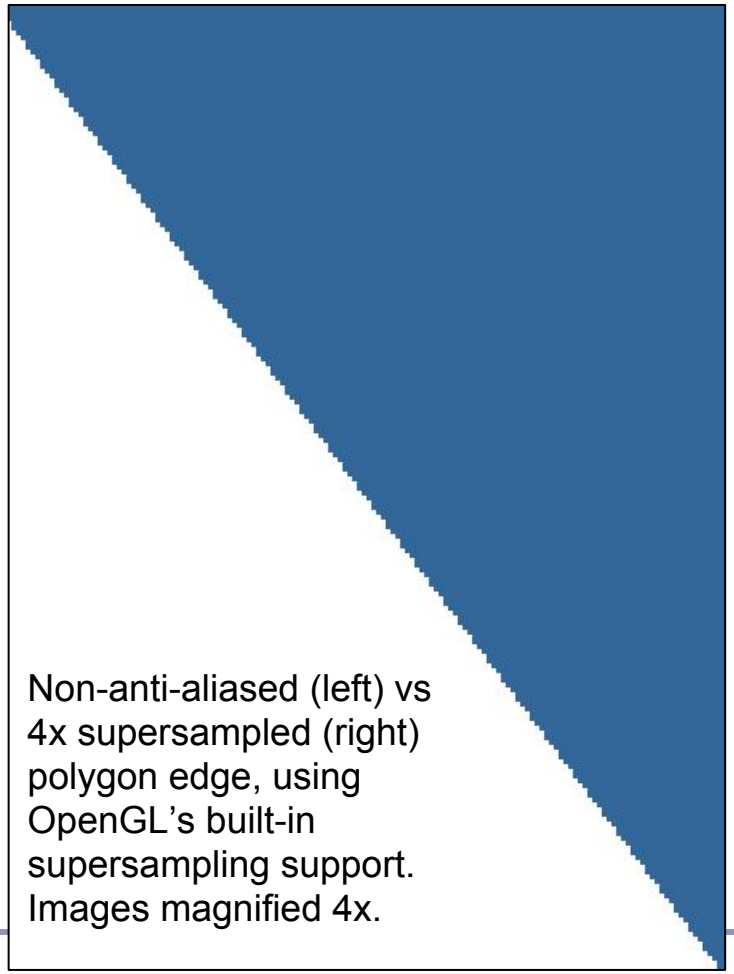
Antialiasing with OpenGL

Antialiasing remains a challenge with hardware-rendered graphics, but image quality can be significantly improved through GPU hardware.

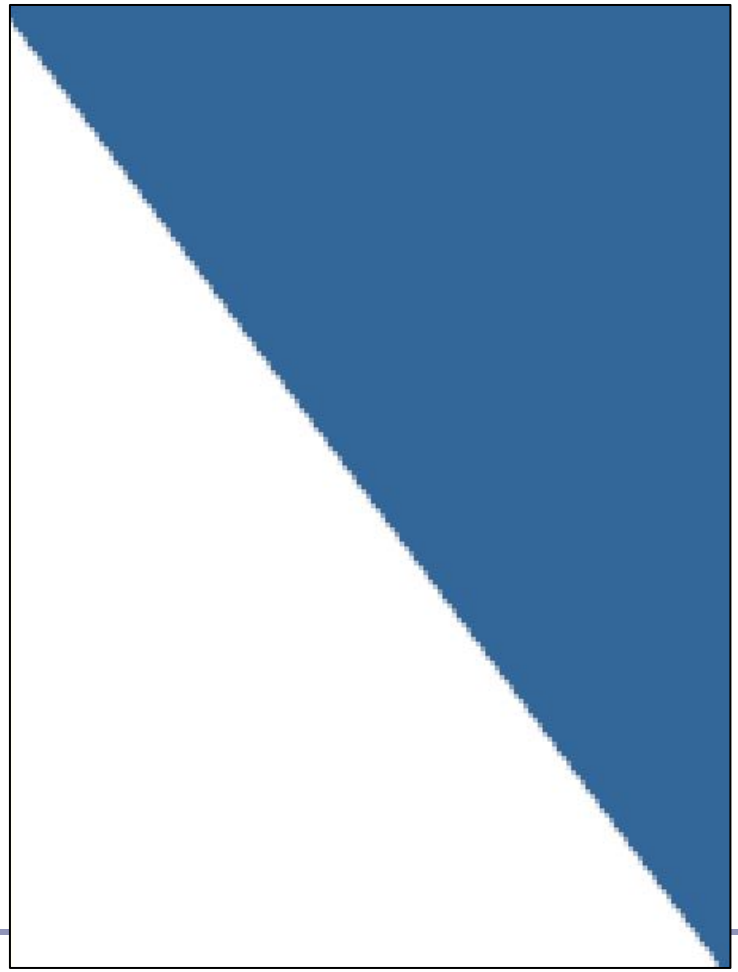
- The simplest form of hardware anti-aliasing is Multi-Sample Anti-Aliasing (*MSAA*).
- "Render everything at higher resolution, then down-sample the image to blur jaggies"
- Enable MSAA in OpenGL with `glfwWindowHint(GLFW_SAMPLES, 4);`



Antialiasing with OpenGL: MSAA



Non-anti-aliased (left) vs 4x supersampled (right) polygon edge, using OpenGL's built-in supersampling support. Images magnified 4x.



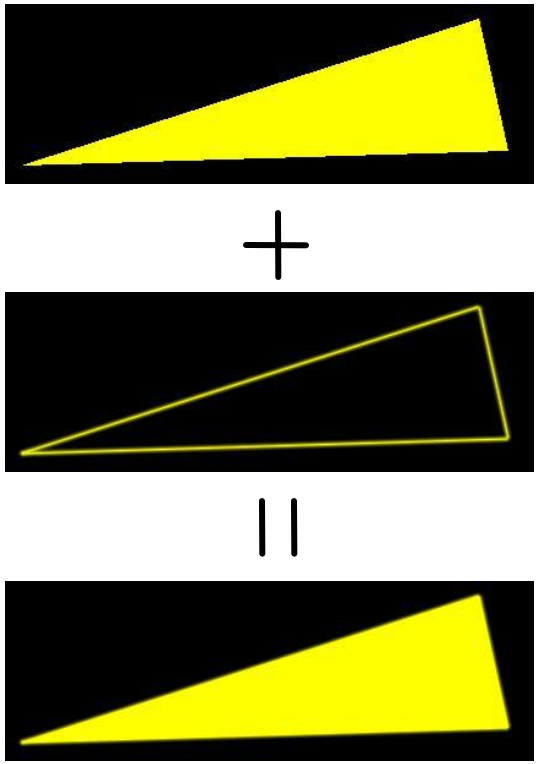
Antialiasing on the GPU

MSAA suffers from high memory constraints, and can be very limiting in high-resolution scenarios (high demand for time and texture access bandwidth.)

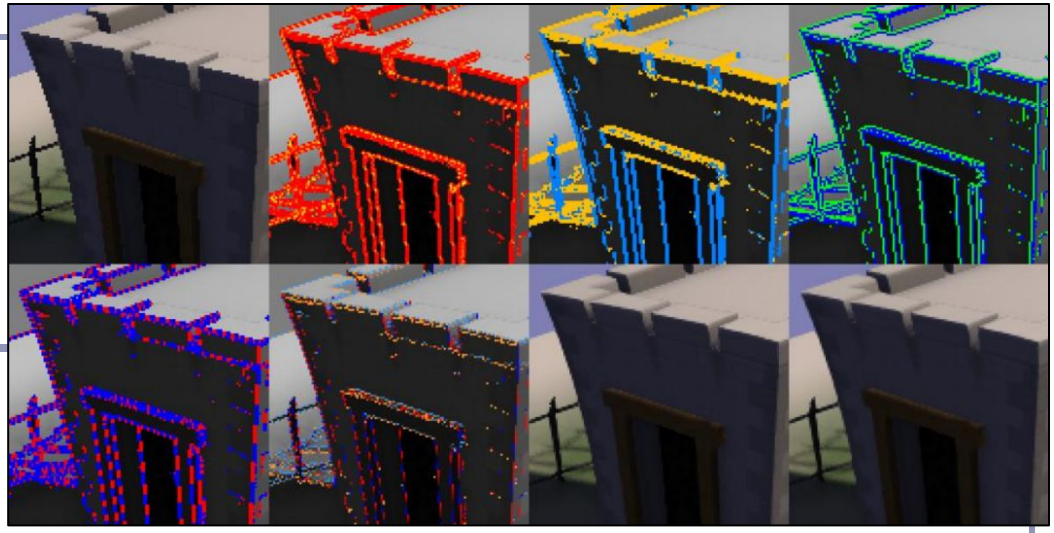
Eric Chan at MIT described an optimized hardware-based anti-aliasing method in 2004:

1. Draw the scene normally
2. Draw wide lines at the objects' silhouettes
   a. Use blurring filters and precomputed luminance tables to blur the lines' width
3. Composite the filtered lines into the framebuffer using alpha blending

This approach is great for polygonal models, tougher for effects-heavy visual scenes like video games



Antialiasing on the GPU



More recently, NVIDIA's *Fast Approximate Anti-Aliasing* ("FXAA") has become popular because it optimizes MSAA's limitations.

Abstract:

1. Use local contrast (pixel-vs-pixel) to find edges (red), especially those subject to aliasing.
2. Map these to horizontal (gold) or vertical (blue) edges.
3. Given edge orientation, the highest contrast pixel pair 90 degrees to the edge is selected (blue/green)
4. Identify edge ends (red/blue)
5. Re-sample at higher resolution along identified edges, using sub-pixel offsets of edge orientations
6. Apply a slight blurring filter based on amount of detected sub-pixel aliasing

Image from https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

Preventing aliasing in texture reads

Antialiasing technique: *adaptive analytic prefiltering*.

- The precision with which an edge is rendered to the screen is dynamically refined based on the rate at which the function defining the edge is changing with respect to the surrounding pixels on the screen.

This is supported in GLSL by the methods `dFdx(F)` and `dFdy(F)`.

- These methods return the derivative with respect to X and Y, *in screen space,* of some variable F.
- These are commonly used in choosing the filter width for antialiasing procedural textures.

(A) Jagged lines visible in the box function of the procedural stripe texture
(B) Fixed-width averaging blends adjacent samples in texture space; aliasing still occurs at the top, where adjacency in texture space does not align with adjacency in pixel space.
(C) Adaptive analytic prefiltering smoothly samples both areas.
Image source: Figure 17.4, p. 440, *OpenGL Shading Language, Second Edition*, Randi Rost, Addison Wesley, 2006. Digital image scanned by Google Books.
Original image by Bert Freudenberg, University of Magdeburg, 2002.

Antialiasing texture reads with Signed Distance Fields

Conventional anti-aliasing in texture reads can only smooth pixels immediately adjacent to the source values.

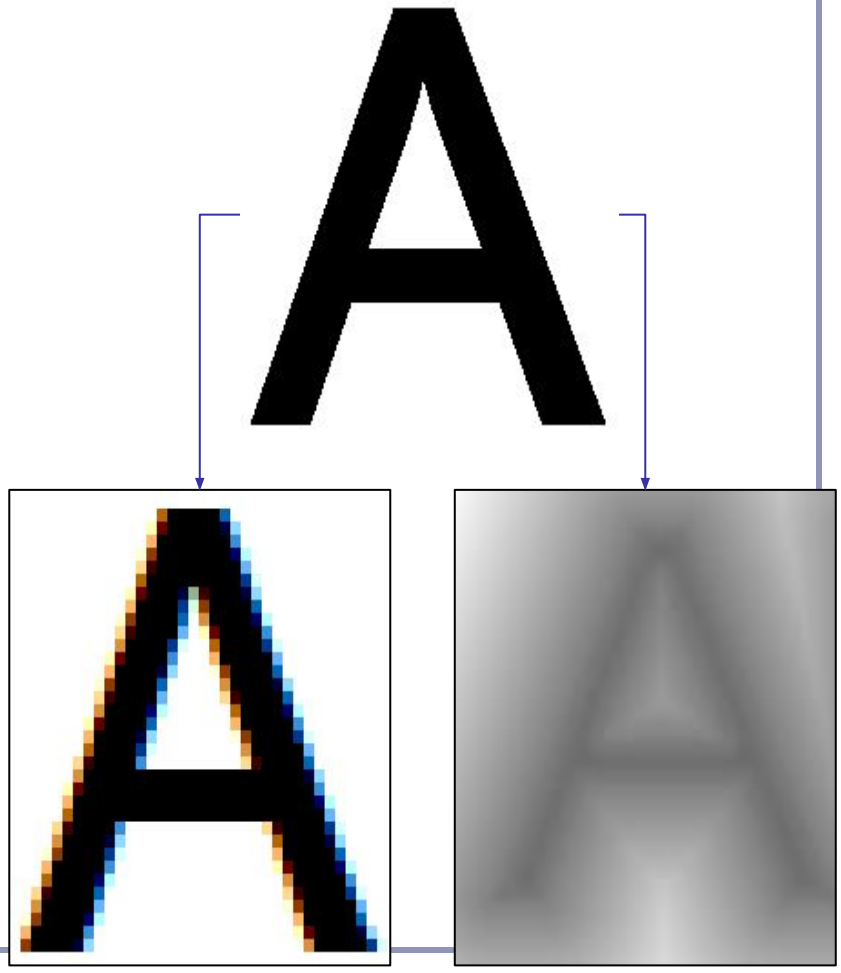
Signed distance fields represent monochrome texture data as a distance map instead of as pixels. This allows per-pixel smoothing at multiple distances.

Antialiasing texture reads with Signed Distance Fields

The bitmap becomes a height map.

Each pixel stores the distance to the closest black pixel (if white) or white pixel (if black). Distance from white is negative.

3.6	2.8	2	1	-1
3.1	2.2	1.4	1	-1
2.8	2	1	-1	-1.4
2.2	1.4	1	-1	-2
2	1	-1	-1.4	-2.2
2	1	-1	-2	-2.8



Conventional antialiasing

Signed distance field

Antialiasing texture reads with Signed Distance Fields

Conventional bilinear filtering computes a weighted average of color, but an SDF computes a weighted average of distances.

This means that a small step away from the original values we find smoother, straighter lines where the slope of the isocline is perpendicular to the slope of the source data.

By smoothing the isocline of the distance threshold, we achieve smoother edges and nifty edge effects.

```
low = 0.02;    high = 0.035;
double dist =
  bilinearSample(tex coords);
double t =
  (dist - low) / (high - low);
return (dist < low) ? BLACK
  : (dist > high) ? WHITE
  : BLACK*(1 - t) + WHITE*(t);
```

Adding a second isocline enables colored borders.

Antialiasing - Interesting further reading

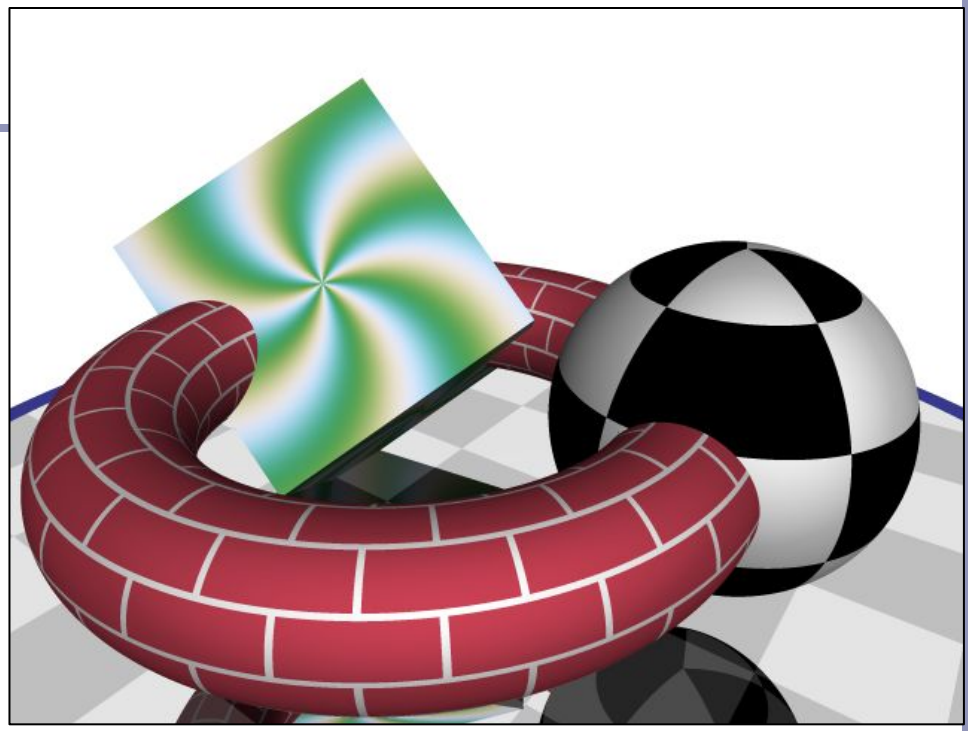
- https://people.csail.mit.edu/ericchan/articles/prefilter/
- https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf
- http://iryoku.com/aacourse/downloads/09-FXAA-3.11-in-15-Slides.pdf

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):

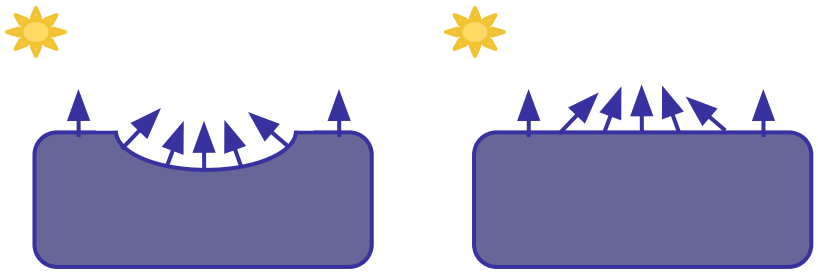


I've cheated slightly and multiplied the u coordinate by 4 to repeat the brick texture four times around the torus.

```
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
```

Non-color textures: 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.

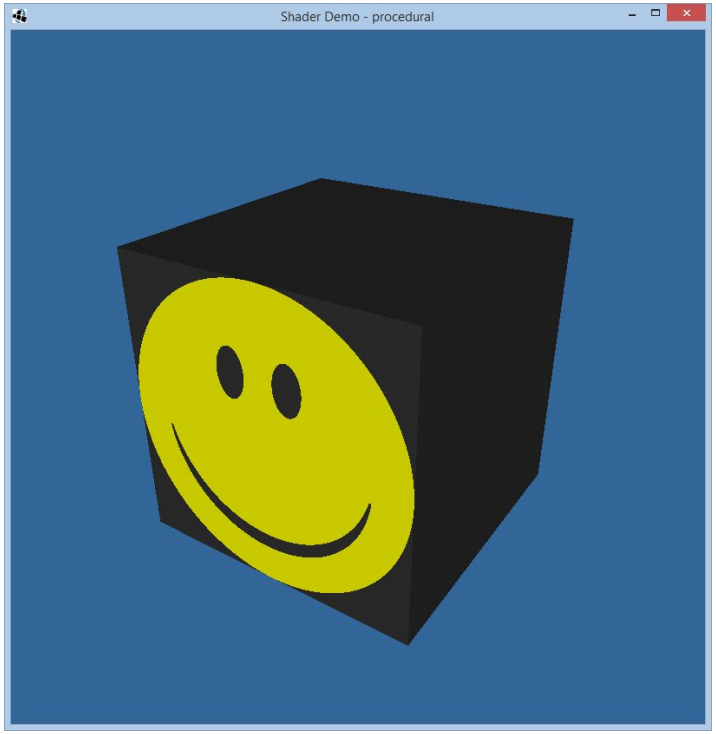
If we duplicate the normals, we don't have to duplicate the dent.

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

Non-color textures: normal mapping

Procedural texturing in the fragment shader

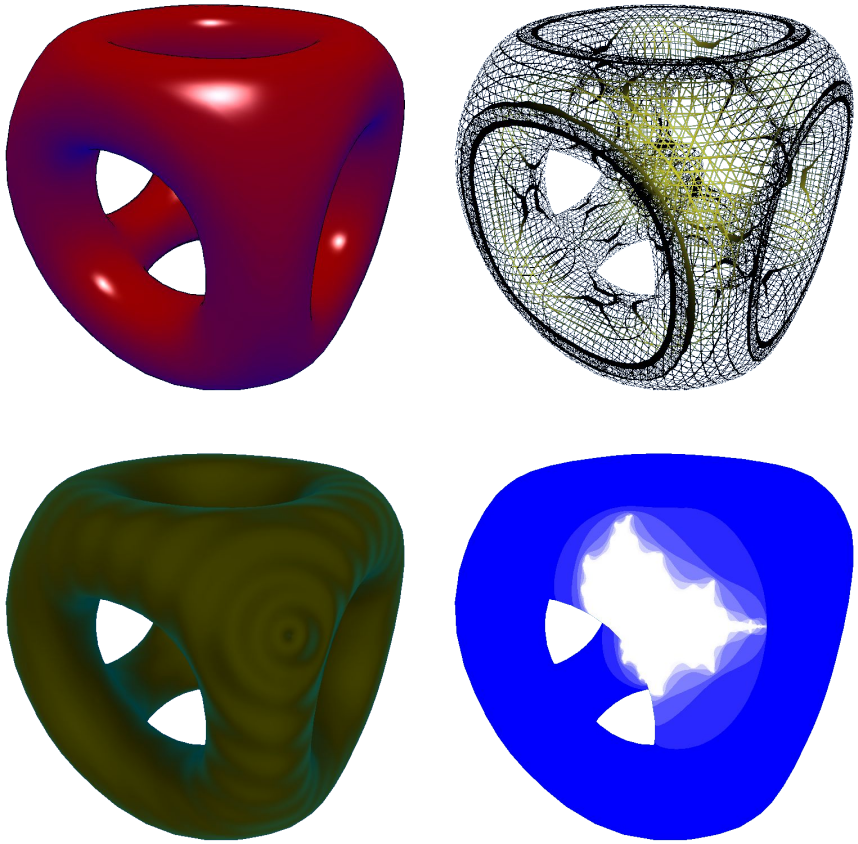
```
// ...
const vec3 CENTER = vec3(0, 0, 1);
const vec3 LEFT_EYE = vec3(-0.2, 0.25, 0);
const vec3 RIGHT_EYE = vec3(0.2, 0.25, 0);
// ...

void main() {
  bool isOutsideFace = (length(position - CENTER) > 1);
  bool isEye = (length(position - LEFT_EYE) < 0.1)
      || (length(position - RIGHT_EYE) < 0.1);
  bool isMouth = (length(position - CENTER) < 0.75)
      && (position.y <= -0.1);

  vec3 color = (isMouth || isEye || isOutsideFace)
      ? BLACK : YELLOW;
  fragmentColor = vec4(color, 1.0);
}
```



(Code truncated for brevity--check out the source on github for how I did the curved mouth and oval eyes.)

Advanced surface effects

- Ray-tracing, ray-marching!
- Specular highlights
- Non-photorealistic illumination
- Volumetric textures
- Bump-mapping
- Interactive surface effects
- Ray-casting in the shader
- Higher-order math in the shader
- ...much, much more!



Perlin Noise

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):

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

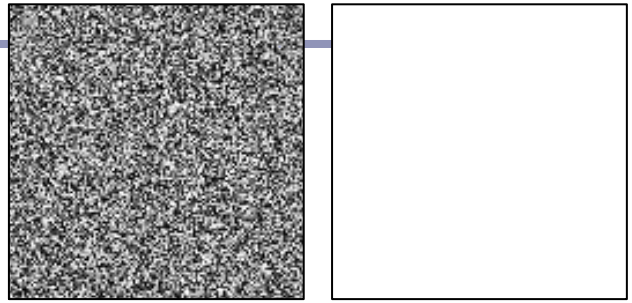
where *noise(P)* is a function that maps 3D coordinates in space to scalar values chosen at random.

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

Perlin noise

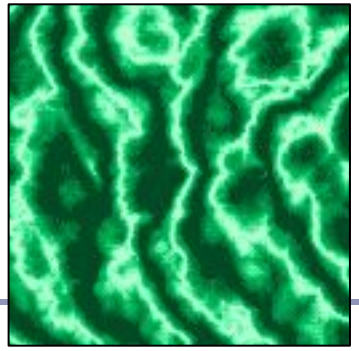
Perlin noise (invented by Ken Perlin) is a method for generating noise which has some useful traits:

- 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/

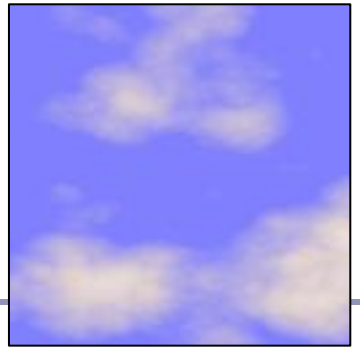


Non-coherent noise (left) and Perlin noise (right)
Image credit: Matt Zucker

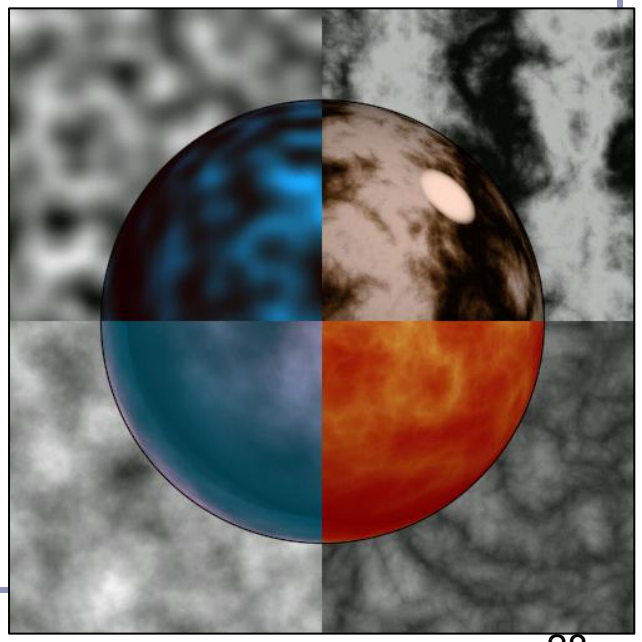
Matt Zucker



Matt Zucker



Matt Zucker



Ken Perlin

Perlin noise 1

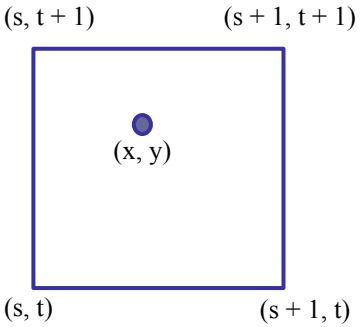
Perlin noise caches 'seed' random values on a grid at integer intervals. You'll look up noise values at arbitrary points in the plane, and they'll be determined by the four nearest seed randoms on the grid.

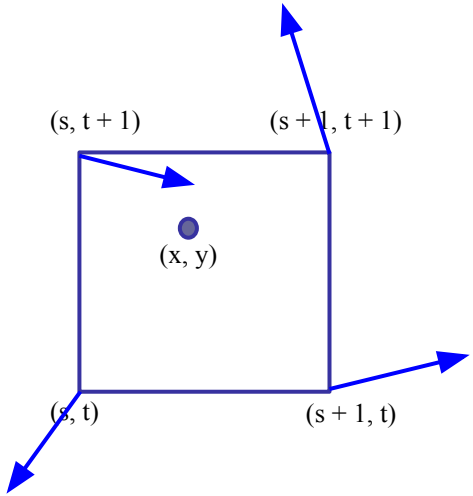
Given point *(x, y)*, let *(s, t) = (floor(x), floor(y))*.

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.



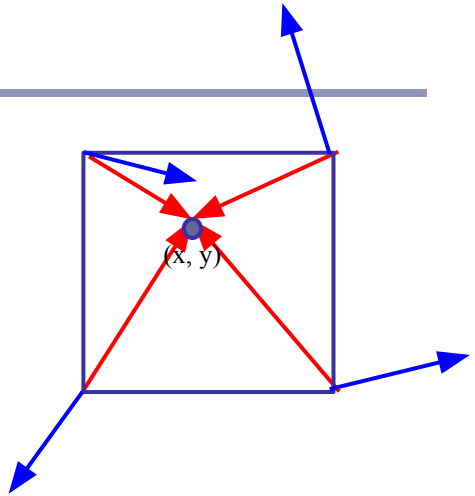


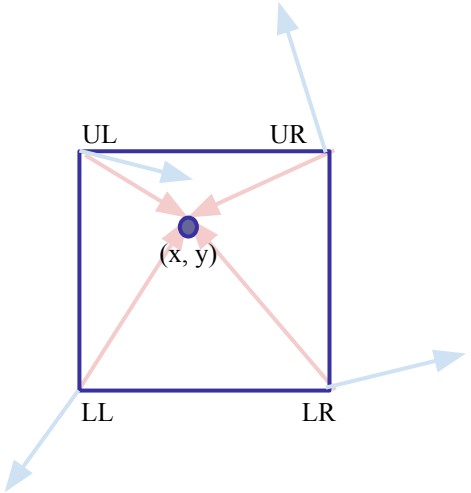
These slides borrow heavily from Mark Zucker's excellent page on Perlin noise at http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

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:

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

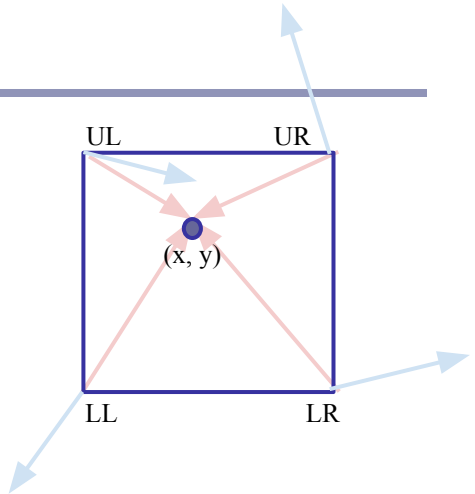
$U(x, y) = UL + S(x - floor(x))(UR-UL)$

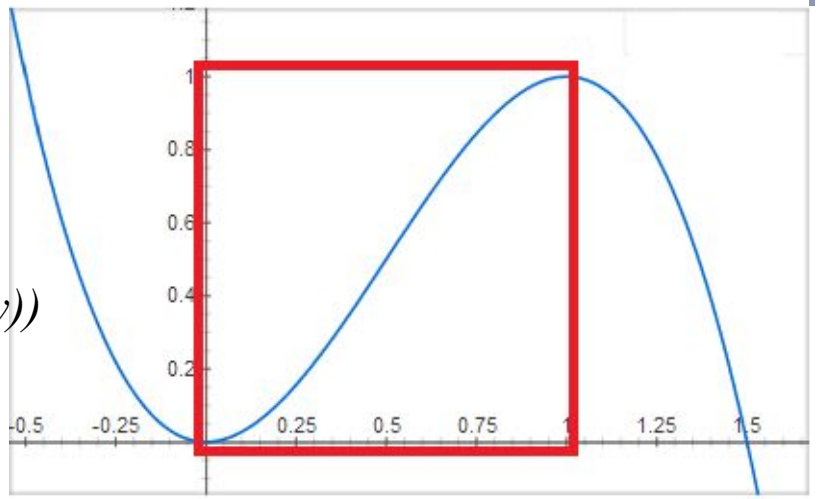
Then we interpolate again to merge the two upper and lower functions:

$noise(x, y) =$

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

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

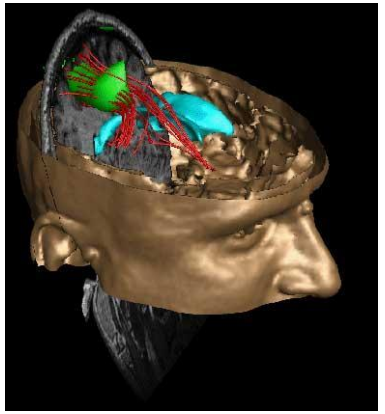
Perlin Noise - References

- https://web.archive.org/web/20160303232627/http://www.noisemachine.com/talk1/
- http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

Voxels and volume rendering

A *voxel* ("volume pixel") is a cube in space with a given color; like a 3D pixel.

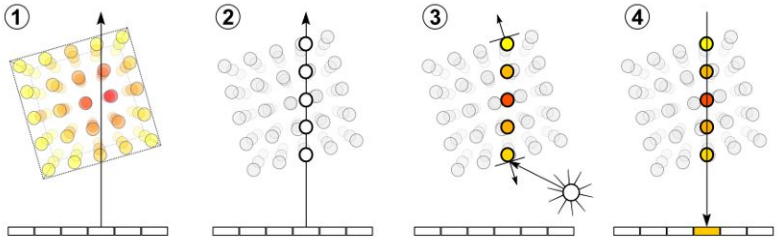
- Voxels are often used for medical imaging, terrain, scanning and model reconstruction, and other very large datasets.
- Voxels usually contain color but could contain other data as well—flow rates (in medical imaging), density functions (analogous to implicit surface modeling), lighting data, surface normals, 3D texture coordinates, etc.
- Often the goal is to render the voxel data directly, not to polygonalize it.



Volume ray casting

If speed can be sacrificed for accuracy, render voxels with *volume ray casting*:

- Fire a ray through each pixel;
- Sample the voxel data along the ray, computing the weighted average (*trili* filter) of the contributions to the ray o each voxel it passes through or near;
- Compute surface gradient from of each voxel from local sampling; generate surface normals; compute lighting with the standard lighting equation;
- 'Paint' the ray from back to front, occluding more distant voxels with nearer voxels; this gives hidden-surface removal and easy support for transparency.

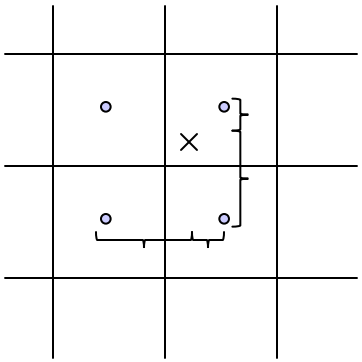


The steps of volume rendering; a volume ray-cast skull. Images from wikipedia.

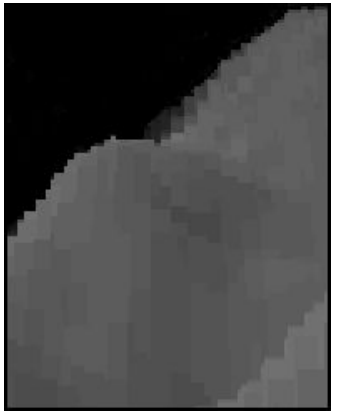
Sampling in voxel rendering

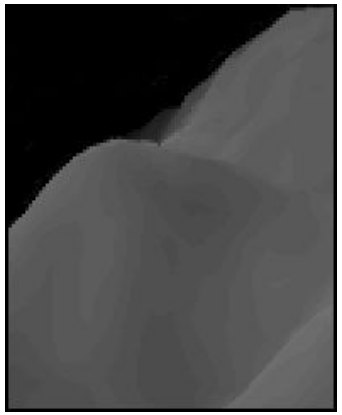
Why trilinear filtering?

- If we just show the color of the voxel we hit, we'll see the exact edges of every cube.
- Instead, choose the weighted average between adjacent voxels.
  - Trilinear: averaging across X, Y, and Z



Your sample will fall somewhere between eight (in 3d) voxel centers. Weight the color of the sample by the inverse of its distance from the center of each voxel.





Reasonably fast voxels

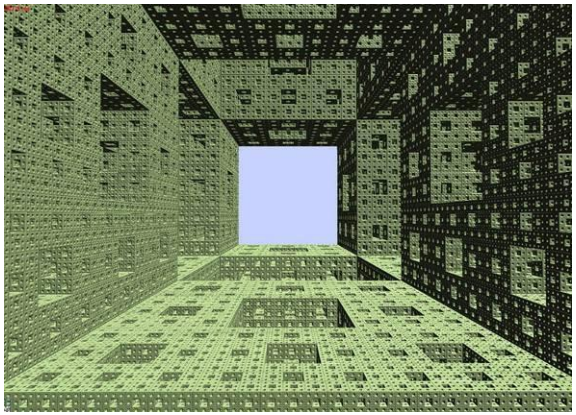
If speed is of the essence, cast your rays but stop at the first opaque voxel.

- Store precomputed lighting directly in the voxel
- Works for diffuse and ambient but not specular
- Popular technique for video games (e.g. Comanche →)

Another clever trick: store voxels in a *sparse voxel octree*.

- Watch for it in id's next-generation engine…

Comanche Gold, NovaLogic Inc (1998)

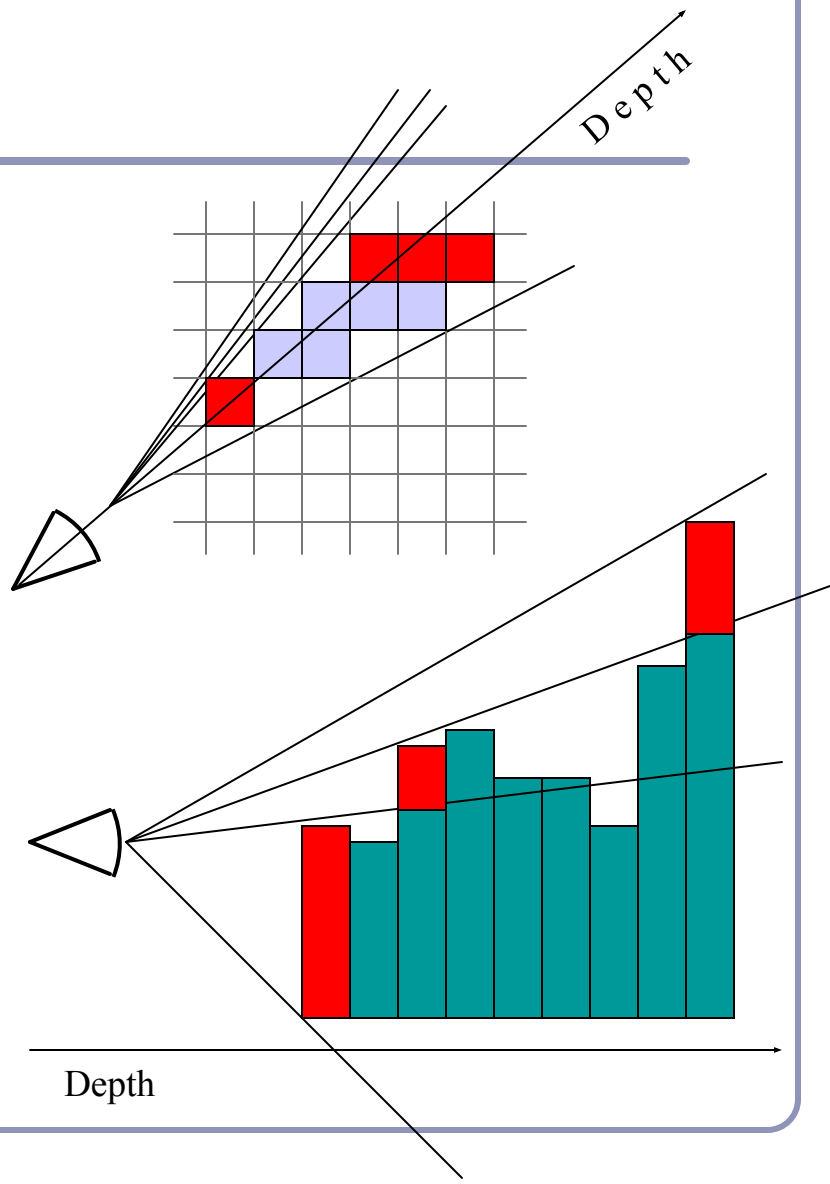


Sparse Voxel Octree Ray-Casting, Cyril Crassin

Ludicrously fast voxels

If speed is essential (like if, say, you're writing a video game in 1992) and you know that your terrain can be represented as a height-map (ie., without overhangs), replace ray-casting with 'column'-casting and use a "*Y-buffer*":

- Draw from *front to back*, drawing columns of pixels from the bottom of the screen up. For each pixel in receding order, track the current max *y* height painted and only draw new pixels above that *y*. Anything shorter must be behind something that's nearer, and it's shorter; so don't draw it.



References

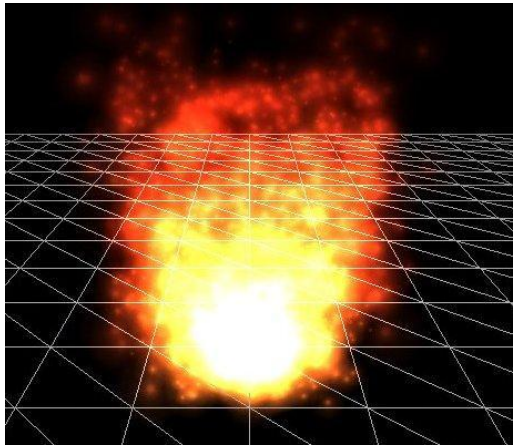
Voxels:
J. Wilhelms and A. Van Gelder, *A Coherent Projection Approach for Direct Volume Rendering*, Computer Graphics, 35(4):275-284,July 1991.
http://en.wikipedia.org/wiki/Volume_ray_casting

Particle systems

Particle systems are a monte-carlo style technique which uses thousands (or millions) or tiny graphical artefacts to create large-scale visual effects.

Particle systems are used for hair, fire, smoke, water, clouds, explosions, energy glows, in-game special effects and much more.

The basic idea:
"If lots of little dots all do something the same way, our brains will see the thing they do and not the dots doing it."



A particle system created with 3dengfx, from wikipedia.

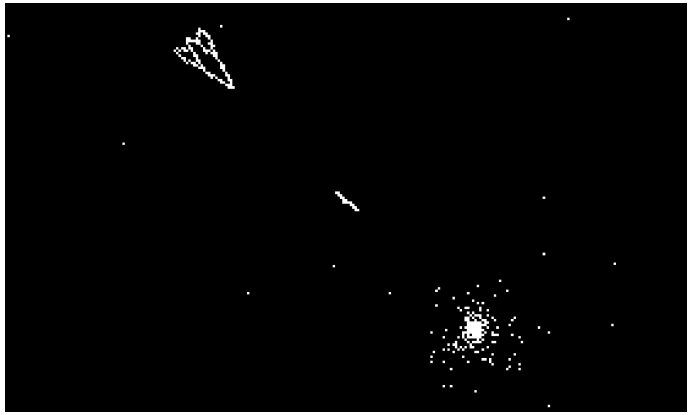
Screenshot from the game *Command and Conquer 3* (2007) by Electronic Arts; the "lasers" are particle effects.

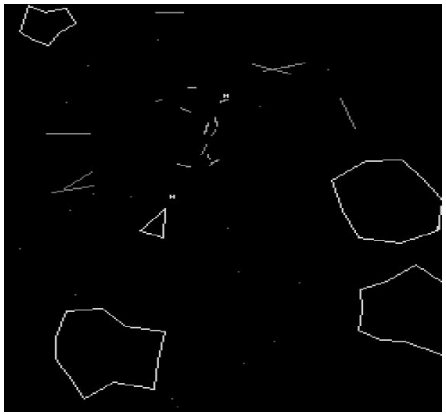
History of particle systems

1962: Ships explode into pixel clouds in "Spacewar!", the 2nd video game *ever*.

1978: Ships explode into broken lines in "Asteroid".

1982: The Genesis Effect in "*Star Trek II: The Wrath of Khan*".





Fanboy note: OMG. You can play the original Spacewar! at http://spacewar.oversigma.com/ -- the actual original game, running in *a PDP-1 emulator inside a Java applet*.

“The Genesis Effect” – William Reeves
Star Trek II: The Wrath of Khan (1982)

Particle systems

How it works:

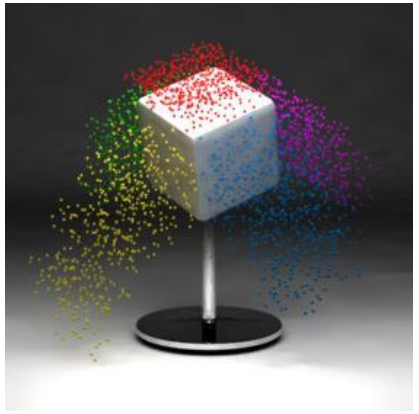
- Particles are generated from an *emitter*.
  - Emitter position and orientation are specified discretely;
  - Emitter rate, direction, flow, etc are often specified as a bounded random range (monte carlo)
- Time ticks; at each tick, particles move.
  - New particles are generated; expired particles are deleted
  - Forces (gravity, wind, etc) accelerate each particle
  - Acceleration changes velocity
  - Velocity changes position
- Particles are rendered.

Particle systems — emission

Each frame, your emitter will generate new particles.

Here you have two choices:

- Constrain the average number of particles generated per frame:
  - # new particles = average # particles per frame + rand() * variance
- Constrain the average number of particles per screen area:
  - # new particles = average # particles per area + rand() * variance * screen area



Transient vs persistent particles emitted to create a ‘hair’ effect (source: Wikipedia)

Particle systems — integration

Each new particle will have at least the following attributes:

- initial position
- initial velocity (speed and direction)

You now have a choice of integration technique:

- Evaluate the particles at arbitrary time *t* as a closed-form equation for a stateless system.
- Or, use iterative (numerical) integration:
  - Euler integration
  - Verlet integration
  - Runge-Kutta integration

Particle systems — two integration shortcuts:

Closed-form function:

- Represent every particle as a parametric equation; store only the initial position p_0, initial velocity v_0, and some fixed acceleration (such as gravity g.)
- $p(t)=p_0+v_0t+\frac{1}{2}gt^2$

No storage of state

- *Very* limited possibility of interaction
- Best for water, projectiles, etc—non-responsive particles.

Discrete integration:

- Remember your physics—integrate acceleration to get velocity:
  - $v'=v + a \bullet \Delta t$
- Integrate velocity to get position:
  - $p'=p + v \bullet \Delta t$
- Collapse the two, integrate acceleration to position:
  - $p''=2p'-p + a \bullet \Delta t^2$

Timestep must be nigh-constant; collisions are hard.

Particle systems—rendering

Can render particles as points, textured polys, or primitive geometry

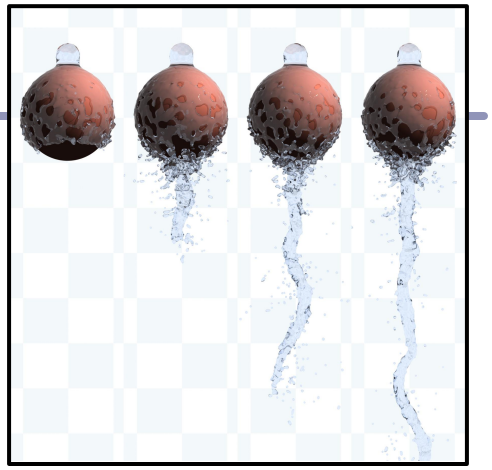
- Minimize the data sent down the pipe!
- Polygons with alpha-blended images make pretty good fire, smoke, etc

Transitioning one particle type to another creates realistic interactive effects

- Ex: a ‘rain’ particle becomes an emitter for ‘splash’ particles on impact

Particles can be the force sources for a blobby model implicit surface

- This is sometimes an effective way to simulate liquids



Hagit Schechter
http://www.cs.ubc.ca/~hagitsch/Research/

nvidia

References

Particle Systems:
William T. Reeves, "*Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*", Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83).
Lutz Latta, *Building a Million Particle System*, http://www.2ld.de/gdc2004/MegaParticlesPaper.pdf , 2004
http://en.wikipedia.org/wiki/Particle_system