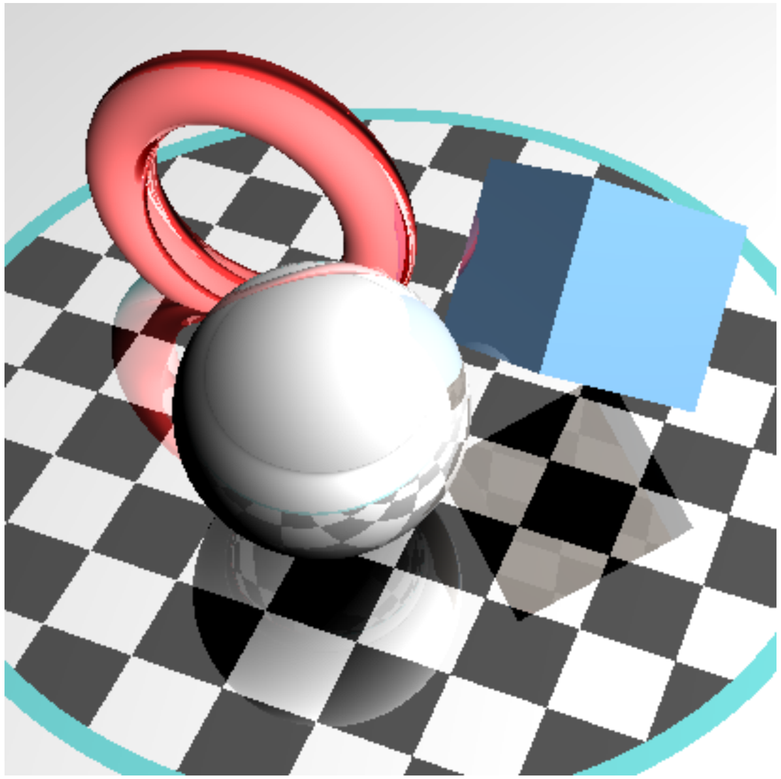
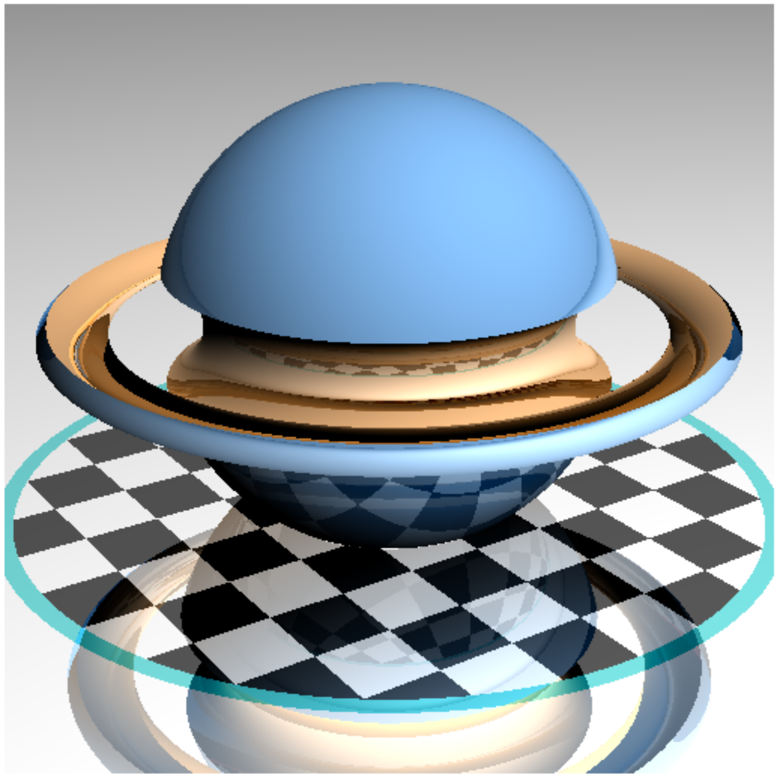
Advanced Graphics

Ray Tracing: Surfaces and Scenes

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

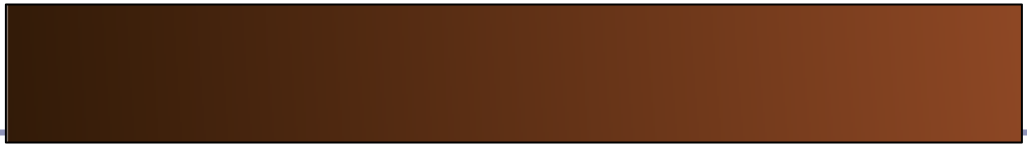
Supported in part by Google UK, Ltd

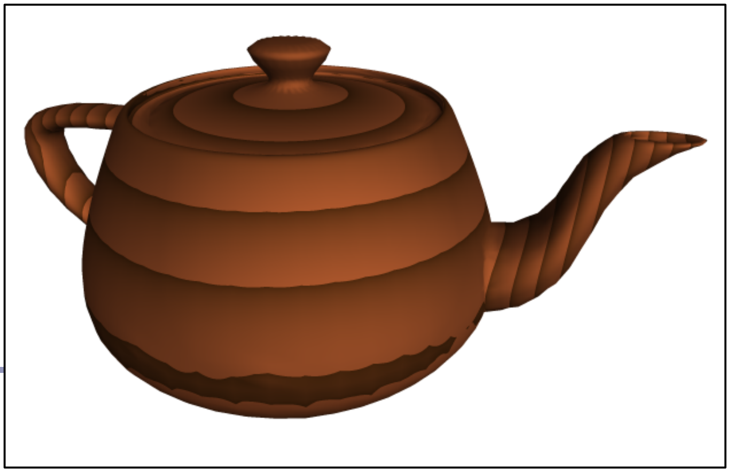
Procedural volumetric texture

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
- $f(P) = (X_P^2+Z_P^2) \bmod 1$
- $color(P) = earlyWood + f(P) * (lateWood - earlyWood)$





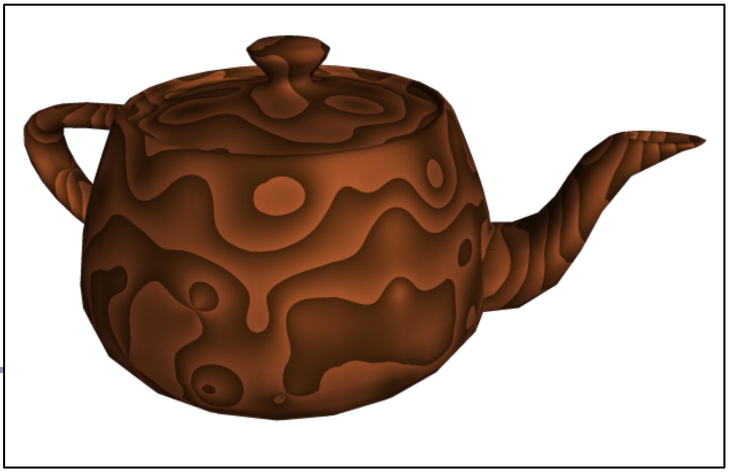
Adding realism

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.

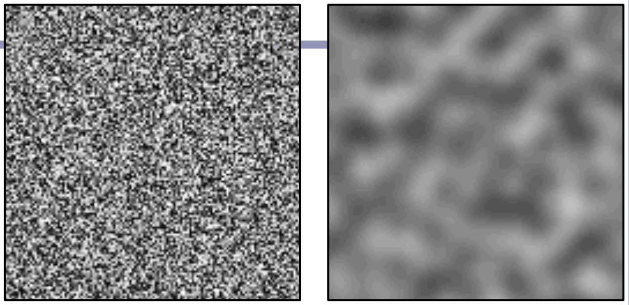
For natural-looking results, use *Perlin noise*, which interpolates smoothly between noise values.



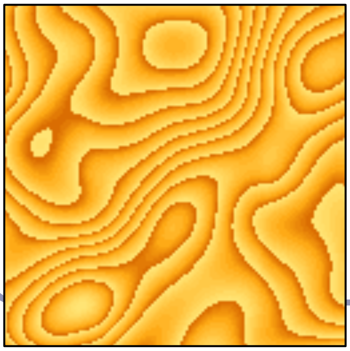
Perlin noise

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

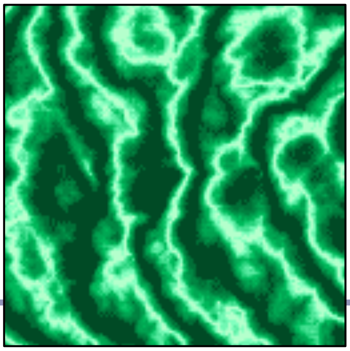
- It is a *band-limited repeatable pseudorandom* function (in the words of its author, Ken Perlin)
- It is bounded within a range close [-1, 1]
- It varies continuously, without discontinuity
- It has regions of relative stability
- It can be initialized with random values, extended arbitrarily in space,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



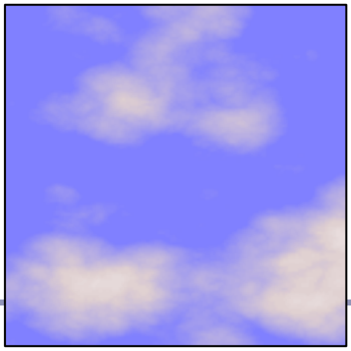
Non-coherent noise (left) and Perlin noise (right)
Image credit: Matt Zucker



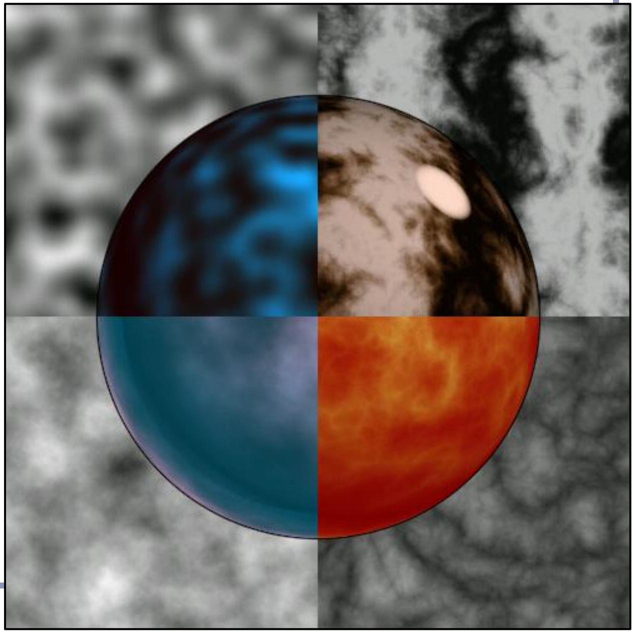
Matt Zucker



Matt Zucker



Matt Zucker



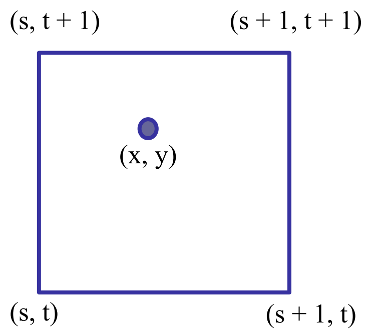
Ken Perlin

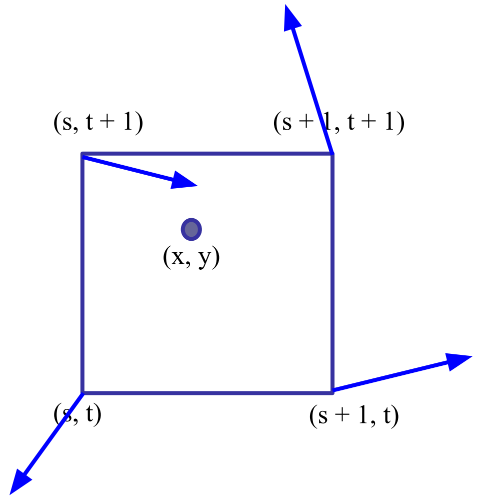
Perlin noise 1

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.

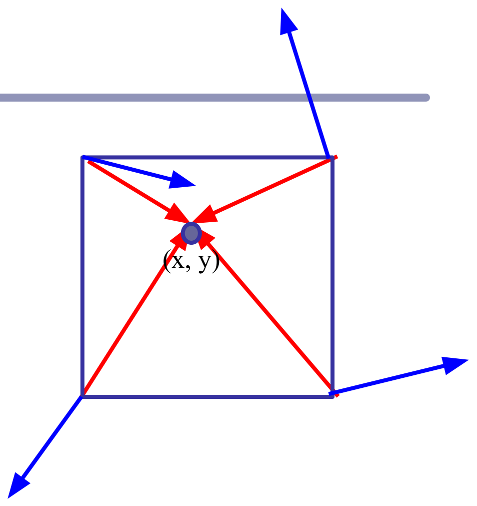


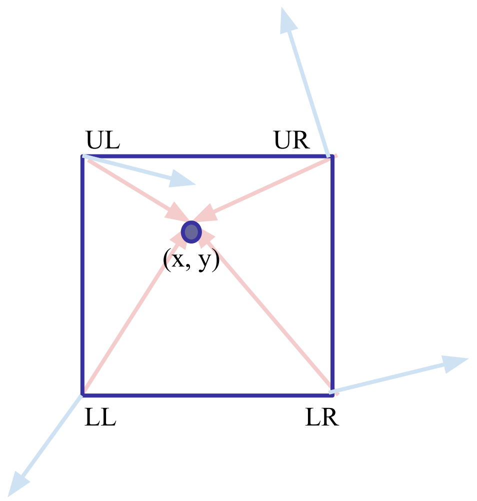


Perlin noise 2

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.

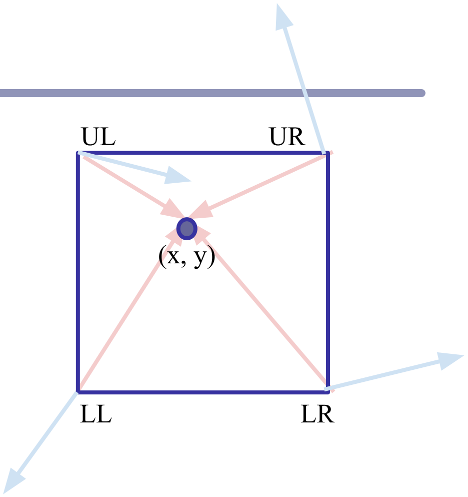
- As *(x, y)* moves across this cell of the grid, the values of the dot products will change smoothly, with no discontinuity.
- As *(x, y)* approaches a grid point, the contribution from that point will approach zero.
- The values of *LL, LR, UL, UR* are clamped to a range close to [-1, 1].





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:

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

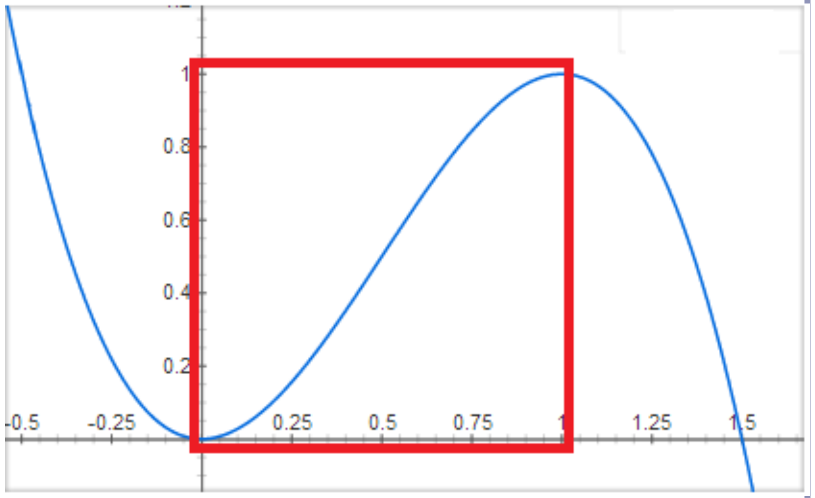
$$U(x, y) = UL + S(x - floor(x))(UR-UL)$$

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

$$noise(x, y) =$$

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

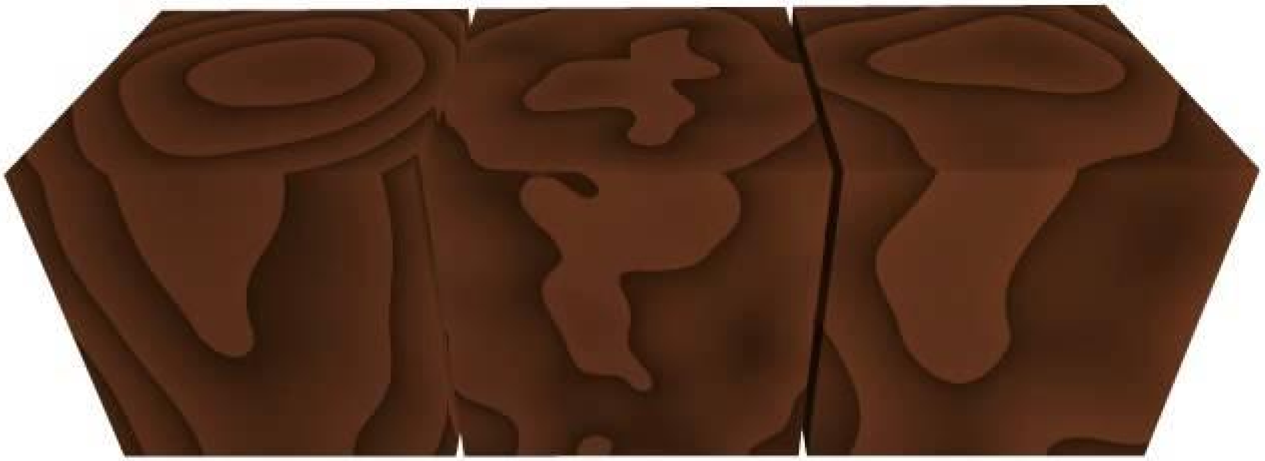
Voila!



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



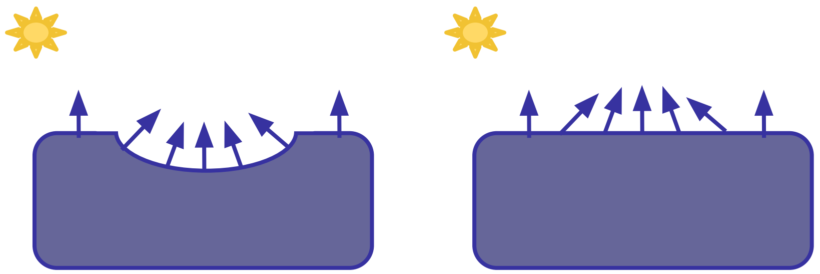
Texture frequency 1 → 3

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.

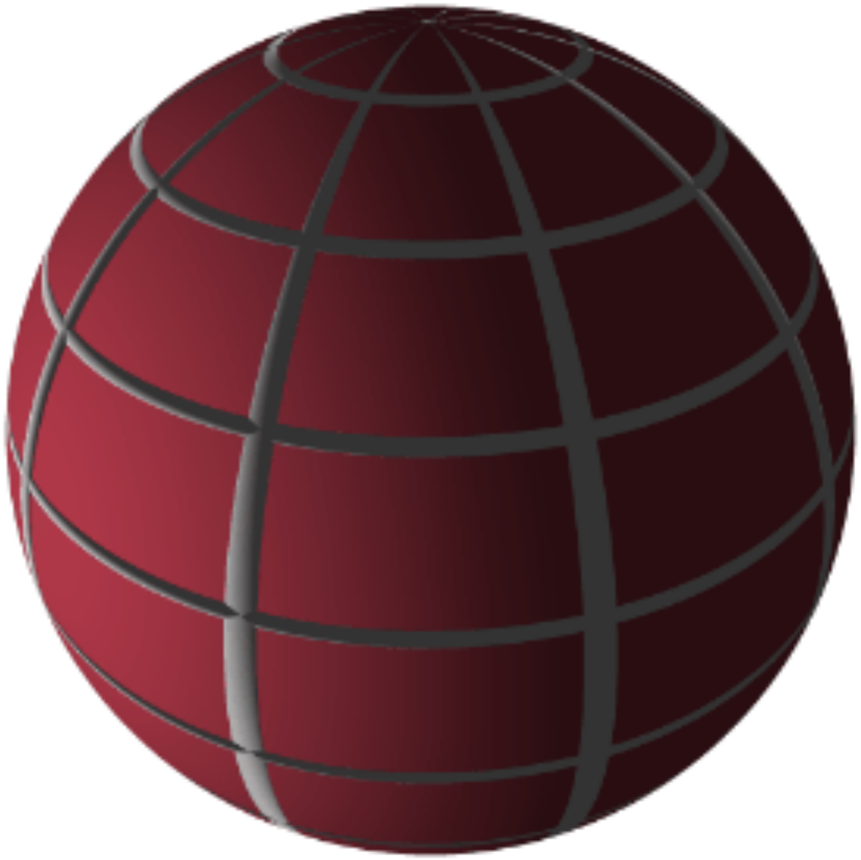


The specular and diffuse shading of the surface varies with the normals in a dent on the surface.

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.

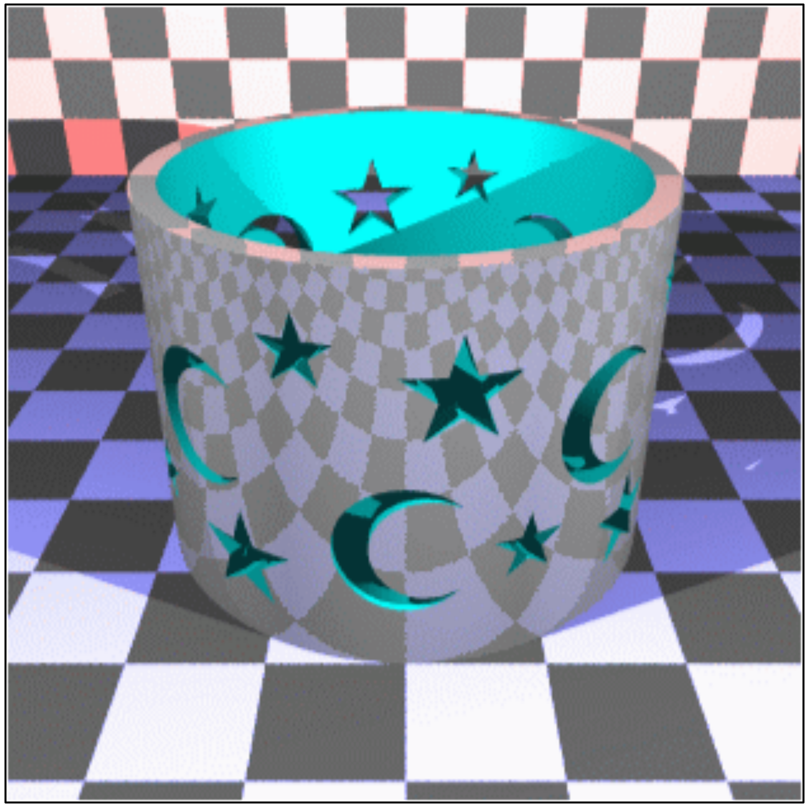
Normal mapping



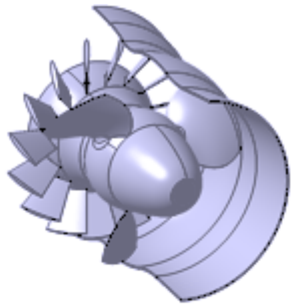
Constructive Solid Geometry

Constructive Solid Geometry (CSG) builds complicated forms out of simple primitives.

These primitives are combined with basic boolean operations: add, subtract, intersect.


CSG figure by Neil Dodgson

Constructive Solid Geometry



CSG models are easy to ray-trace but difficult to polygonalize

- Issues include choosing polygon boundaries at edges; converting adequately from pure smooth primitives to discrete (flat) faces; handling ‘infinitely thin’ sheet surfaces; and others.
- This is an ongoing research topic.

CSG models are well-suited to machine milling, automated manufacture, etc

- Great for 3D printers!

Constructive Solid Geometry

CSG surfaces can be 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?)

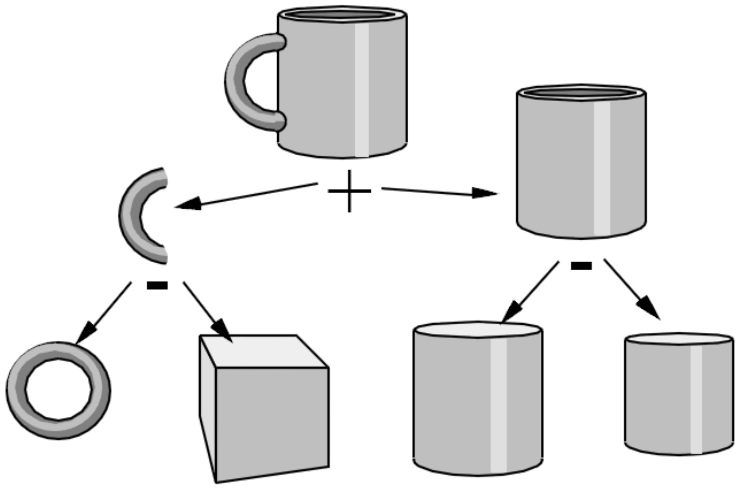
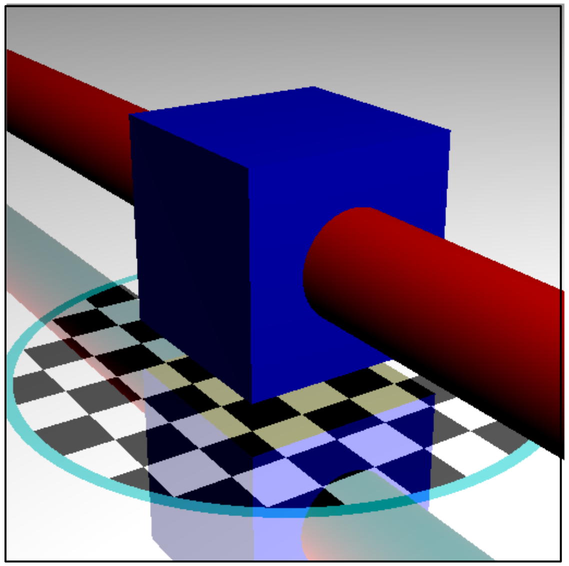


Figure from Wyvill (1995) part two, p. 4

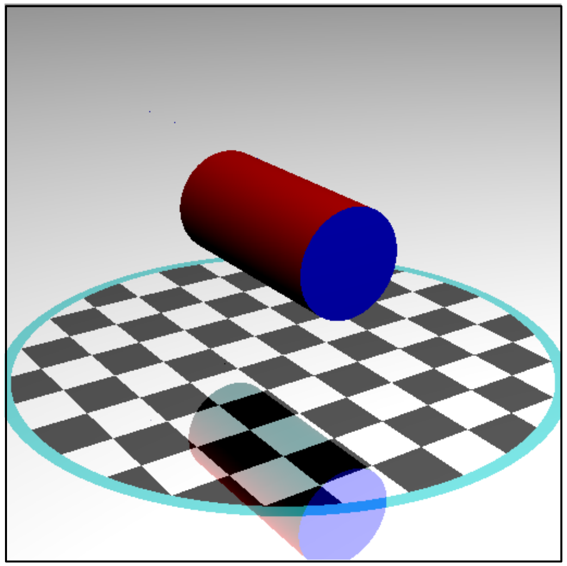
Constructive Solid Geometry

Three operations:

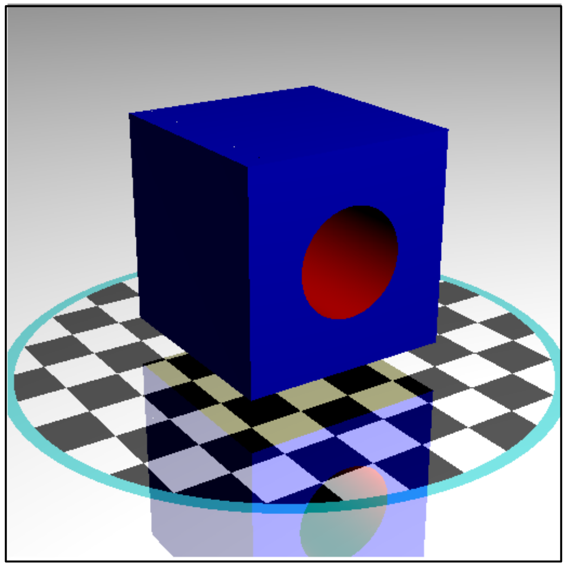
1. *Union*



2. *Intersection*

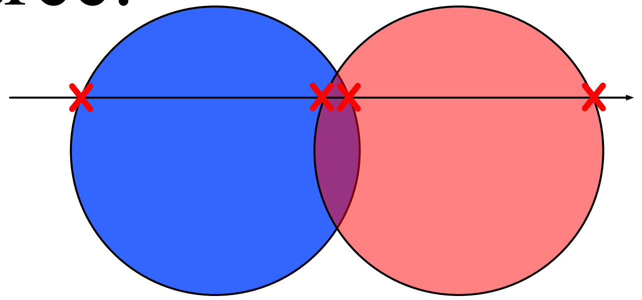


3. *Difference*



Constructive Solid Geometry

For each node of the binary tree:



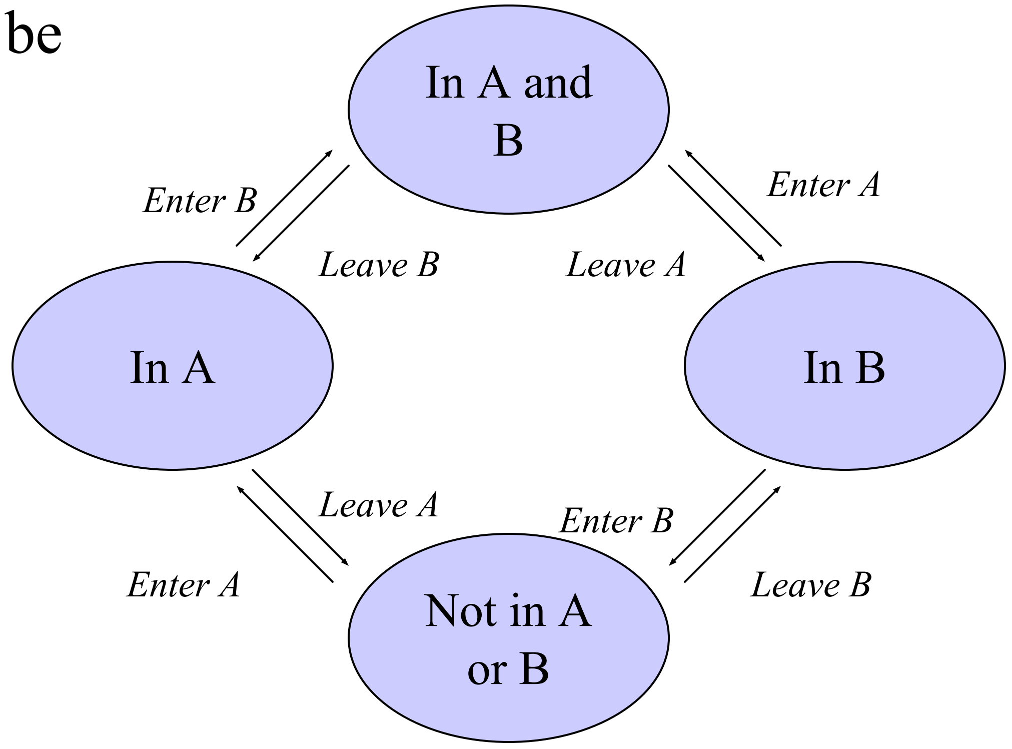
- Fire ray *r* at *A* and *B*.
- List in *t*-order all points where *r* enters of leaves *A* or *B*.
  - You can think of each intersection as a quad of booleans-- (*wasInA, isInA, wasInB, isInB*)
- Discard from the list all intersections which don't matter to the current boolean operation.
- Pass the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

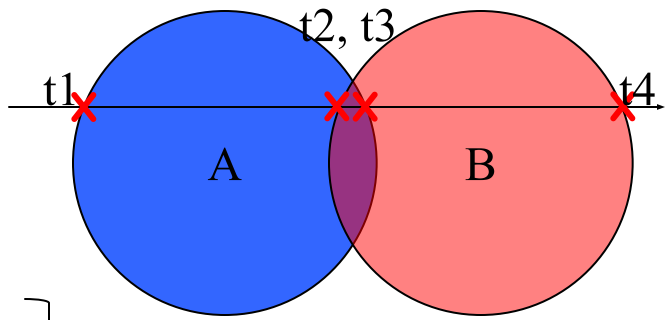
- Union: `{In A | In B | In A and B}`
- Intersection: `{In A and B}`
- Difference: `{In A}`



Ray-tracing CSG models

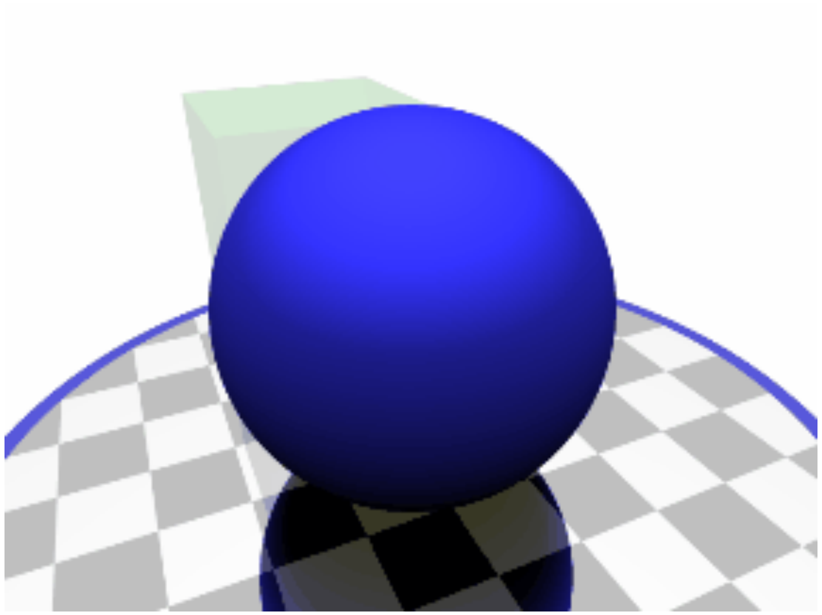
- Example: Difference (A-B)

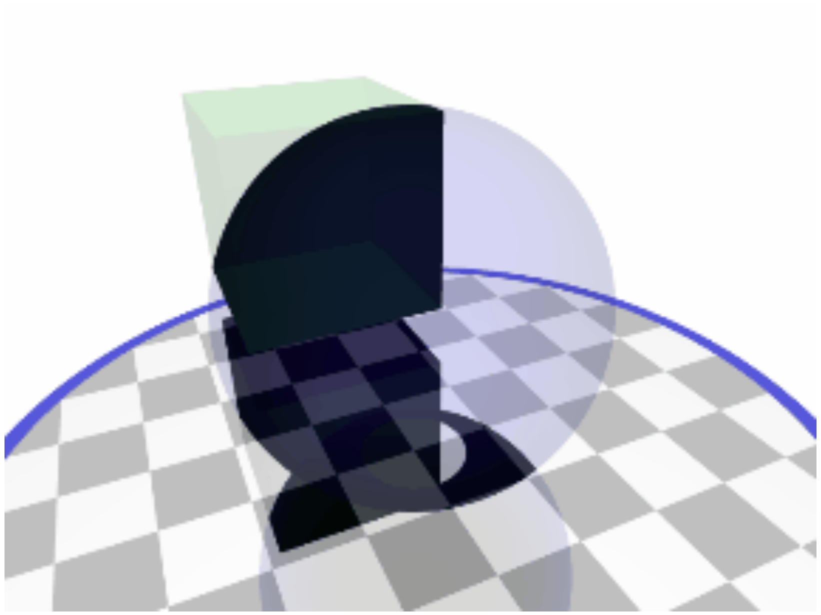
A-B	Was In A	Is In A	Was In B	Is In B
t1	No	Yes	No	No
t2	Yes	Yes	No	Yes
t3	Yes	No	Yes	Yes
t4	No	No	Yes	No



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

CSG in action


Difference

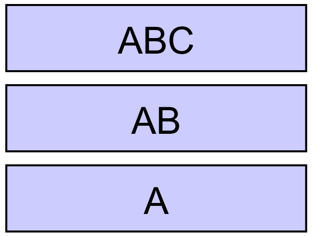

Intersection

The *matrix stack* design pattern

A common design pattern in 3D graphics, especially when objects can contain other objects, is to use *matrix stacks* to store stacks of matrices. The topmost matrix is the product of all matrices below.

- This allows you to build a local frame of reference—local space—and apply transforms within that space.
- Remember: matrix multiplication is associative but not commutative.
  - $ABC = A(BC) = (AB)C \neq ACB \neq BCA$

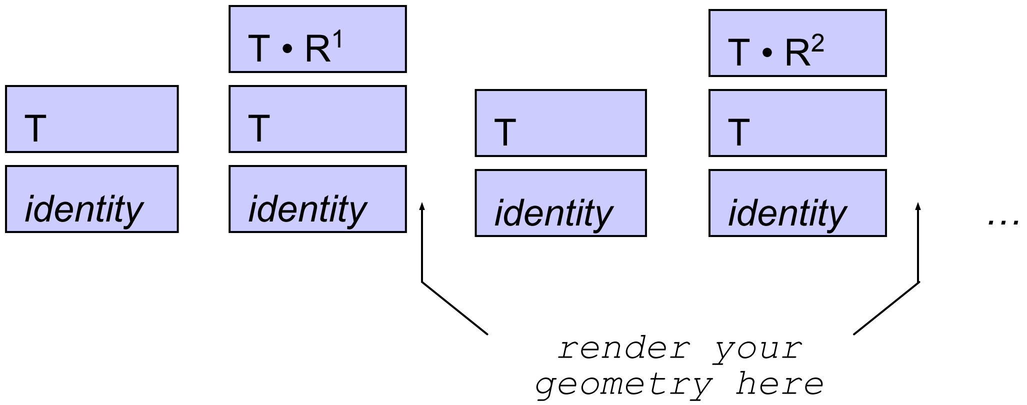
Pre-multiplying matrices that will be used more than once is faster than multiplying many matrices every time you render a primitive.



Matrix stacks and scene graphs

Matrix stacks are designed for nested relative transforms.

```
pushMatrix();
  translate(0,0,-5);
  pushMatrix();
    rotate(45,0,1,0);
    render();
  popMatrix();
  pushMatrix();
    rotate(-45,0,1,0);
    render();
  popMatrix();
popMatrix();
```

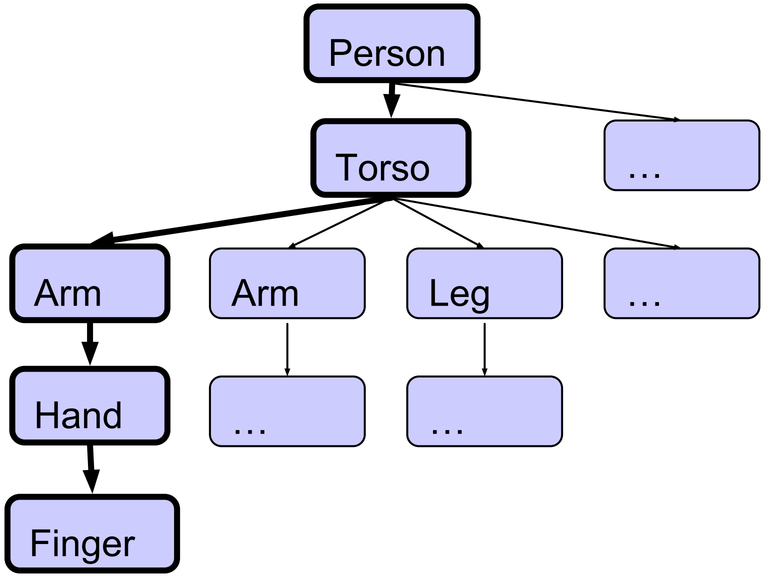


Scene graphs

A *scene graph* is a tree of scene elements where a child's transform is relative to its parent.

The final transform of the child is the ordered product of all of its ancestors in the tree.

Matrix stacks and depth-first traversal of your scene graph: two great tastes that go great together!



$$M_{\text{fingerToWorld}} = (M_{\text{person}} \bullet M_{\text{torso}} \bullet M_{\text{arm}} \bullet M_{\text{hand}} \bullet M_{\text{finger}})$$

Your scene graph and you

Many 2D GUIs today favor an event model in which events ‘bubble up’ from child windows to parents. This is sometimes mirrored in a scene graph.

- Ex: a child changes size, changing the size of the parent’s bounding box
- Ex: the user drags a movable control in the scene, triggering an update event

If you do choose this approach, consider using the *Model View Controller* or *Model View Presenter* design pattern. 3D geometry objects are good for displaying data but they are not the proper place for control logic.

- For example, the class that stores the geometry of the rocket should not be the same class that stores the logic that moves the rocket.
- Always separate logic from representation.

Your scene graph and you

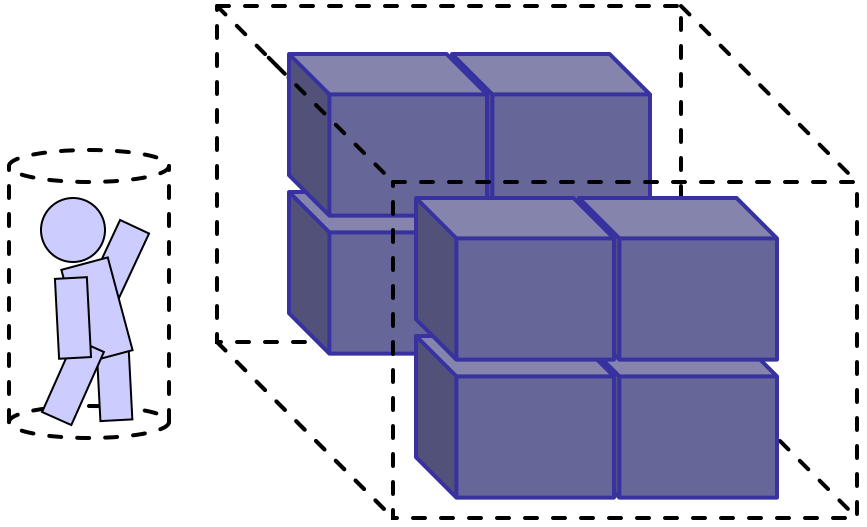
A common optimization derived from the scene graph is the propagation of *bounding volumes*.

Nested bounding volumes allow the rapid culling of large portions of geometry

- Test against the bounding volume of the top of the scene graph and then work down.

Great for…

- Collision detection between scene elements
- Culling before rendering
- Accelerating ray-tracing



Speed up ray-tracing with *bounding volumes*

Bounding volumes help to quickly accelerate volumetric tests, such as "does the ray hit the cow?"

- choose fast hit testing over accuracy
- 'bboxes' don't have to be tight

Axis-aligned bounding boxes

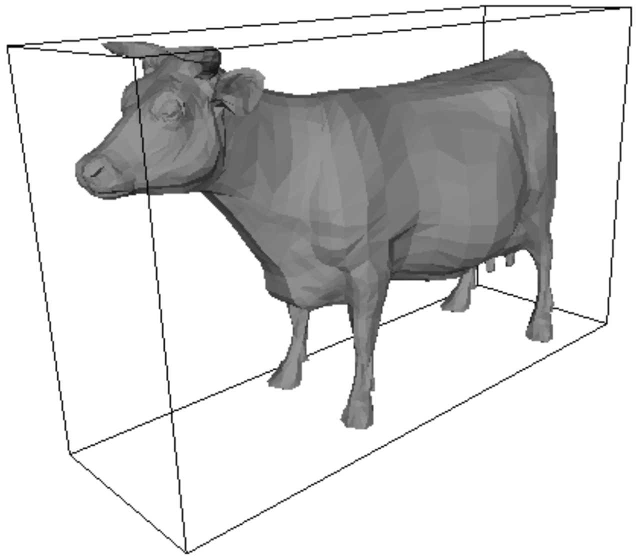
- max and min of x/y/z.

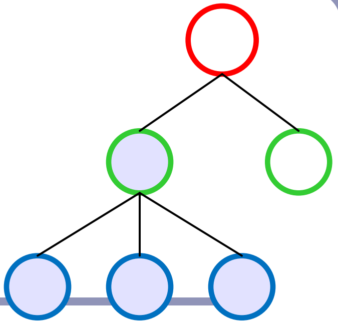
Bounding spheres

- max of radius from some rough center

Bounding cylinders

- common in early FPS games

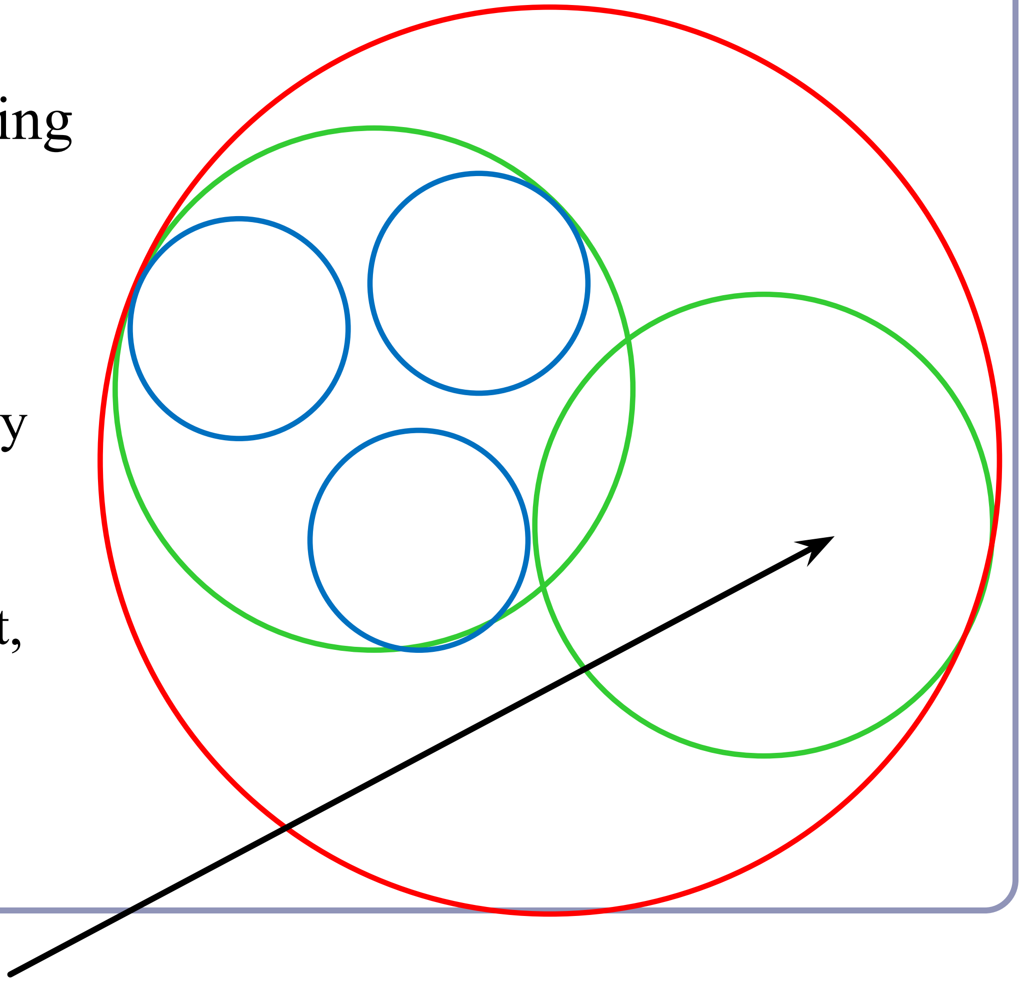




Bounding volumes in hierarchy

Hierarchies of bounding volumes allow early discarding of rays that won't hit large parts of the scene.

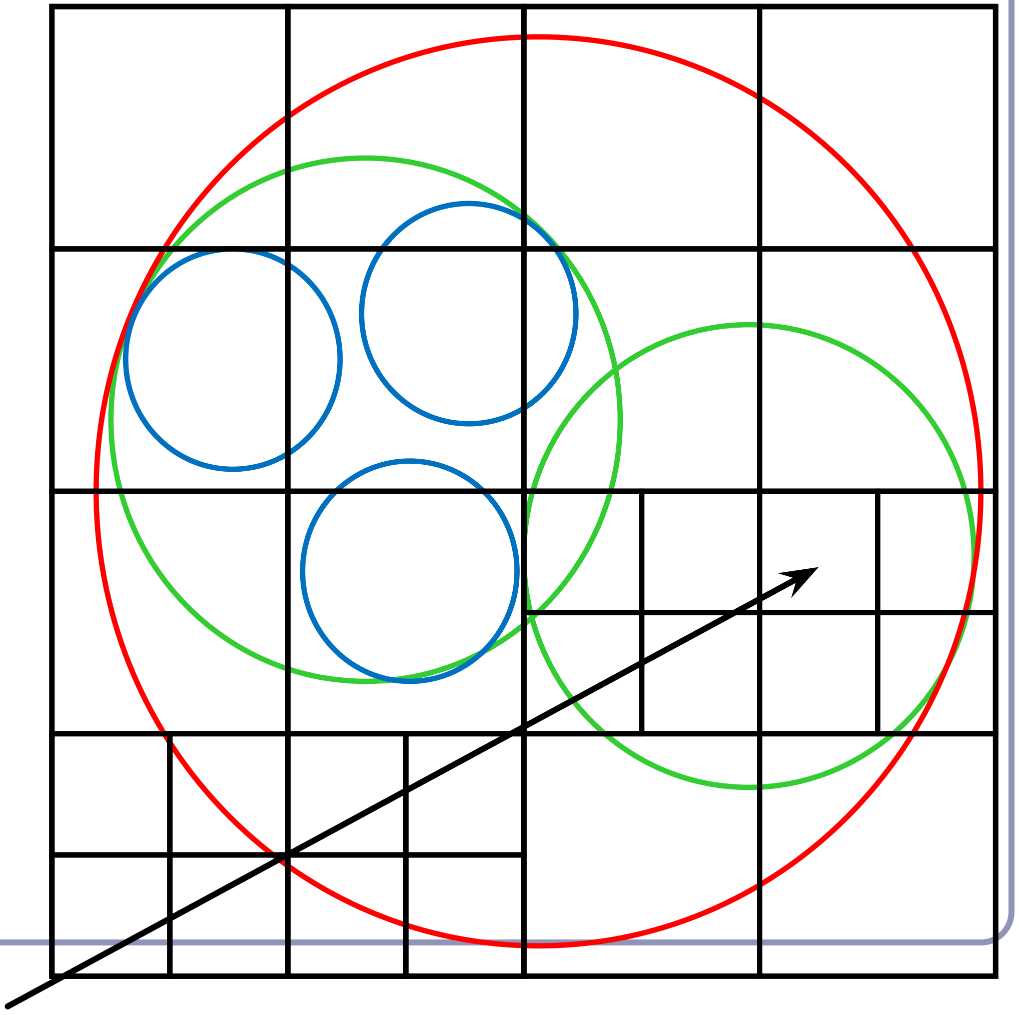
- Pro: Rays can skip subsections of the hierarchy
- Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



Subdivision of space

Split space into cells and list in each cell every object in the scene that overlaps that cell.

- Pro: The ray can skip empty cells
- Con: Depending on cell size, objects may overlap many filled cells or you may waste memory on many empty cells



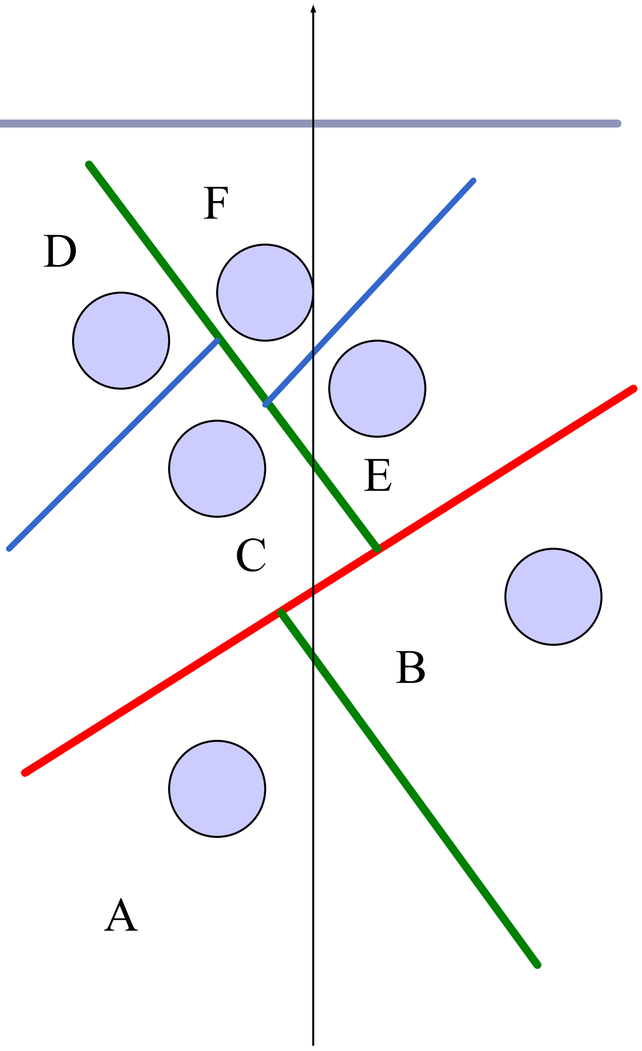
Popular acceleration structures: *BSP Trees*

The *BSP tree* partitions the scene into objects in front of, on, and behind a tree of planes.

- When you fire a ray into the scene, you test all near-side objects before testing far-side objects.

Problems:

- choice of planes is not obvious
- computation is slow
- plane intersection tests are heavy on floating-point math.



Popular acceleration structures: *kd-trees*

The *kd-tree* is a simplification of the BSP Tree data structure

- Space is recursively subdivided by axis-aligned planes and points on either side of each plane are separated in the tree.
- The *k*d-tree has O(*n* log *n*) insertion time (but this is very optimizable by domain knowledge) and O($n^{2/3}$) search time.
- *k*d-trees don't suffer from the mathematical slowdowns of BSPs because their planes are always axis-aligned.

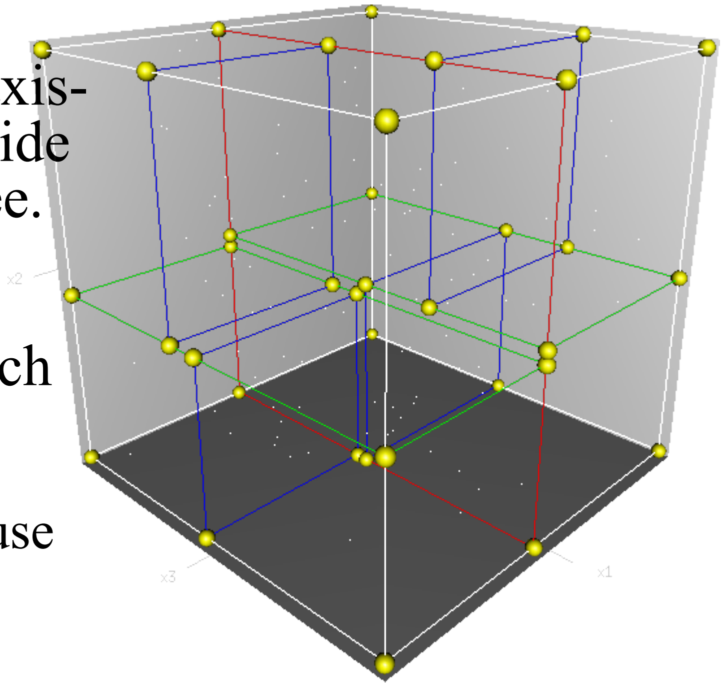


Image from Wikipedia, bless their hearts.

Popular acceleration structures: *Bounding Interval Hierarchies*

The *Bounding Interval Hierarchy* subdivides space around the volumes of objects and shrinks each volume to remove unused space.

- Think of this as a "best-fit" *k*d-tree
- Can be built dynamically as each ray is fired into the scene

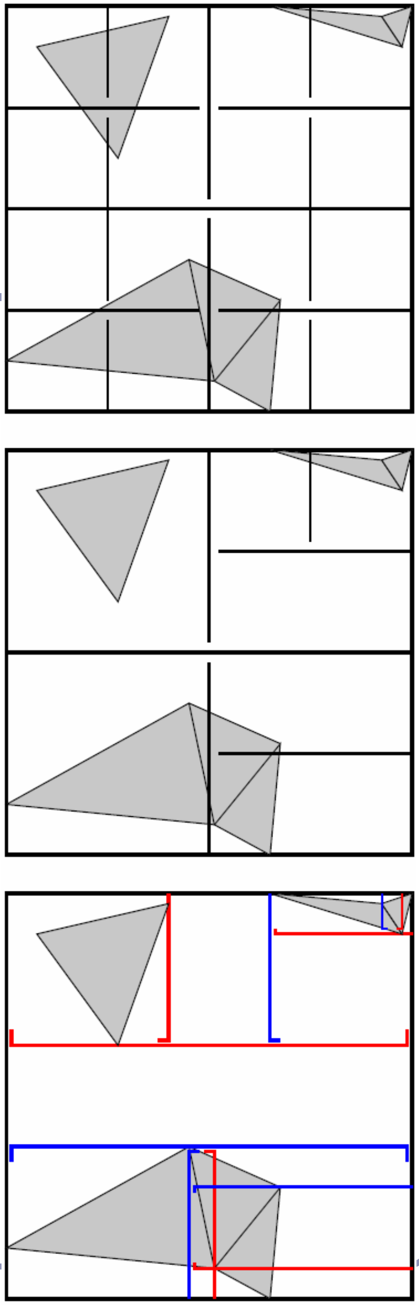


Image from Wächter and Keller's paper, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics (2006)

Hierarchical modeling in action

```
void renderLevel(GL gl, int level, float t) {
  pushMatrix();
  rotate(t, 0, 1, 0);
  renderSphere(gl);
  if (level > 0) {
    scale(0.75f, 0.75f, 0.75f);
    pushMatrix();
      translate(1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    popMatrix();
    pushMatrix();
      translate(-1, -0.75f, 0);
      renderLevel(gl, level-1, t);
    popMatrix();
  }
  popMatrix();
}
```

Hierarchical modeling in action