Ray Tracing II

SOCS Teaching Assistant Excellence Award

- Had a great experience with one of your TAs?
- Nominate them for a TA Award!

Nomination Information:

https://www.cs.mcgill.ca/academic/ta/taawards

Criteria: organized, knowledgeable, effective, accessible, inspiring, role model...

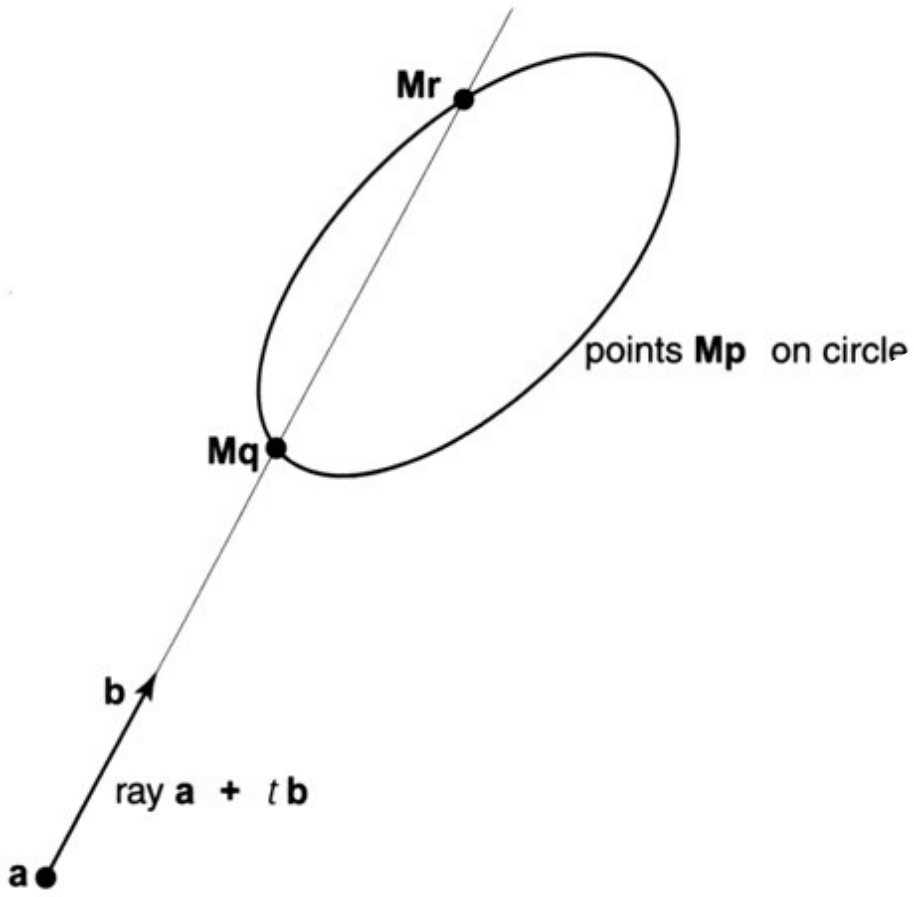
Topics

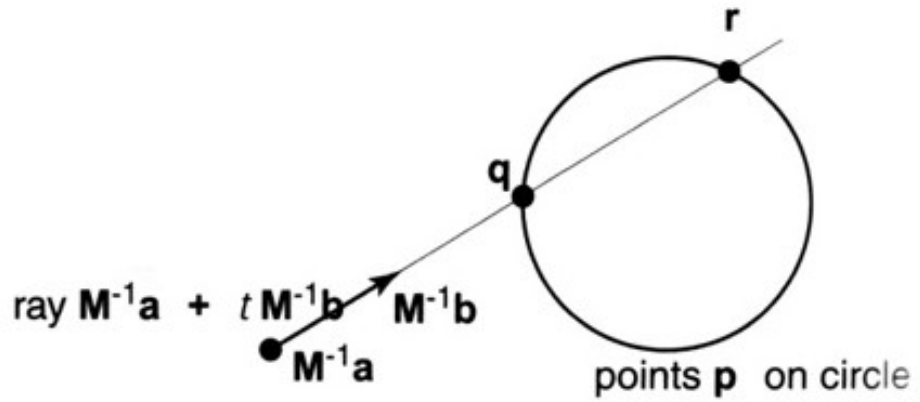
- Transformations in ray tracing
  - Transforming objects
  - Transformation hierarchies
- Ray tracing acceleration structures
  - Bounding volumes
  - Bounding volume hierarchies
  - Uniform spatial subdivision
  - Adaptive spatial subdivision

Transforming objects

- In modeling, we've seen the usefulness of transformations
  - How to do the same in RT?
- Take spheres as an example: want to support transformed spheres
  - Need a new Surface subclass
- Option 1: transform sphere into world coordinates
  - Write code to intersect arbitrary ellipsoids
- Option 2: transform ray into sphere's coordinates
  - Then just use existing sphere intersection routine

Intersecting transformed objects





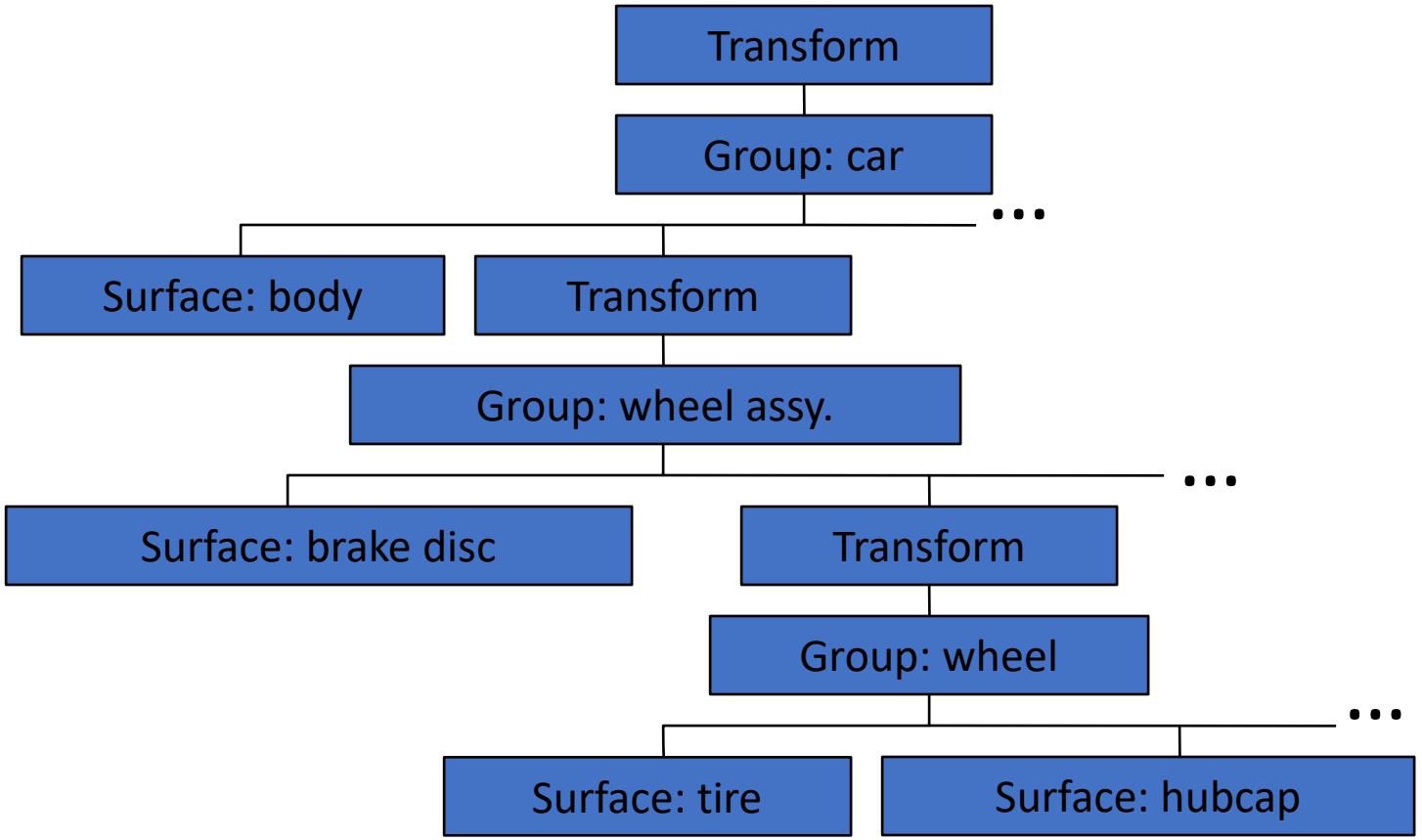
Implementing RT transforms

- Create wrapper object “TrasformedSurface”
  - Has a transform T and a reference to a surface S
- To intersect with a transformed surface:
  - Transform ray to local coords (by inverse of T)
  - Call surface.intersect
  - Transform hit data back to global coords (by T)
    - Intersection point
    - Surface normal
    - Any other relevant data (maybe none)

Groups, transforms, hierarchies

- Often it's useful to transform several objects at once
  - Add "SurfaceGroup" as a subclass of Surface
    - Has a list of surfaces
    - Returns closest intersection
      - Opportunity to move ray intersection code here to avoid duplication
- With TransformedSurface and SurfaceGroup you can put transforms below transforms
  - This gives us a transformation hierarchy.

A transformation hierarchy

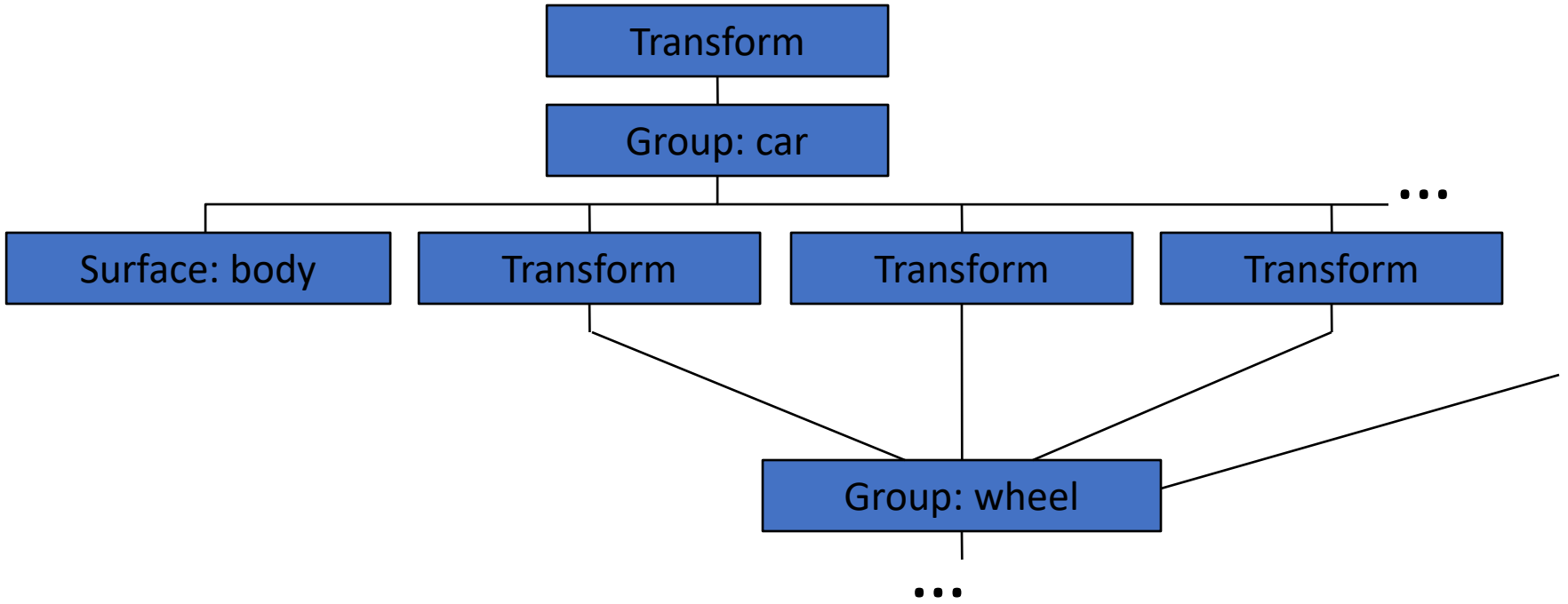


- Common optimization: merge transforms with groups
  - This is how we did it in the modeler assignment

Instancing

- Anything worth doing is worth doing *n* times
- If we can transform objects, why not transform them several ways?
  - Many models have repeated subassemblies
    - Mechanical parts (wheels of car)
    - Multiple objects (chairs in classroom, …)
  - Nothing stops you from creating two TransformedSurface objects that reference the same Surface
    - Allowing this makes the transformation tree into a DAG
      - (directed acyclic graph)
    - Mostly this is transparent to the renderer

Hierarchy with instancing



Hierarchies and performance

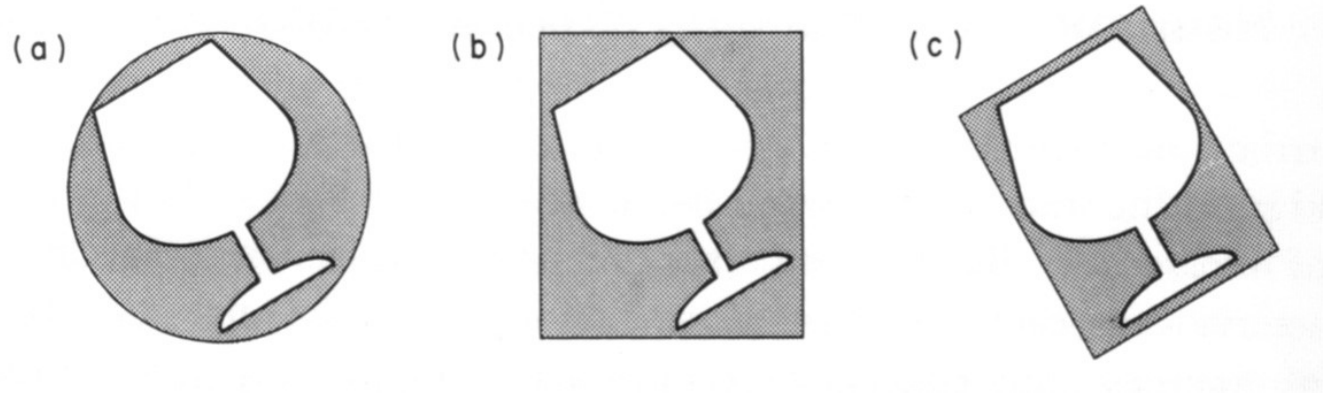
- Transforming rays is expensive
  - minimize tree depth: flatten on input
    - push all transformations toward leaves
    - triangle meshes may do best to stay as group
      - transform ray once, intersect with mesh
  - internal group nodes still required for instancing
    - can’t push two transforms down to same child!

Ray tracing acceleration

- Ray tracing is slow. This is bad!
  - Ray tracers spend most of their time in ray-surface intersection methods
- Ways to improve speed
  - Make intersection methods more efficient
    - Yes, good idea. But only gets you so far
  - Call intersection methods fewer times
    - Intersecting every ray with every object is wasteful
    - Basic strategy: efficiently find big chunks of geometry that definitely do not intersect a ray

Bounding volumes

- Quick way to avoid intersections: bound object with a simple volume
  - Object is fully contained in the volume
  - If it doesn't hit the volume, it doesn't hit the object
  - So test bvol first, then test object if it hits



[Glassner 89, Fig 4.5]

Bounding volumes

- Cost:
  - slightly more for hits and near misses,
  - ***much less*** for far misses
- Worth doing? It depends:
  - Cost of bvol intersection test should be small
    - Therefore use simple shapes (spheres, boxes, …)
  - Cost of object intersect test should be large
    - Bvols most useful for complex objects
  - Tightness of fit should be good
    - Loose fit leads to extra object intersections
    - Tradeoff between tightness and bvol intersection cost

Implementing bounding volume

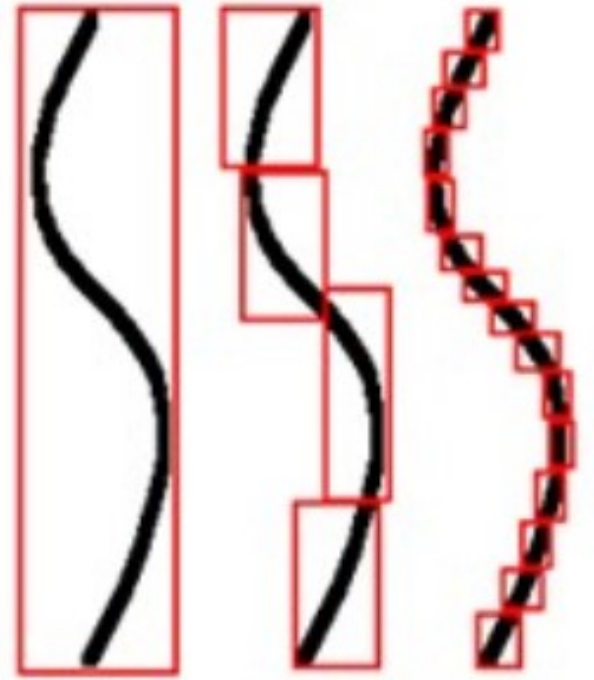
- Just add new Surface subclass, “BoundedSurface”
  - Contains a bounding volume and a reference to a surface
  - Intersection method:
    - Intersect with bvol, return false for miss
    - Return surface.intersect(ray)
  - Like transformations, common to merge with group
  - This change is transparent to the renderer (only it might run faster)
- Note that all Surfaces will need to be able to supply bounding volumes for themselves

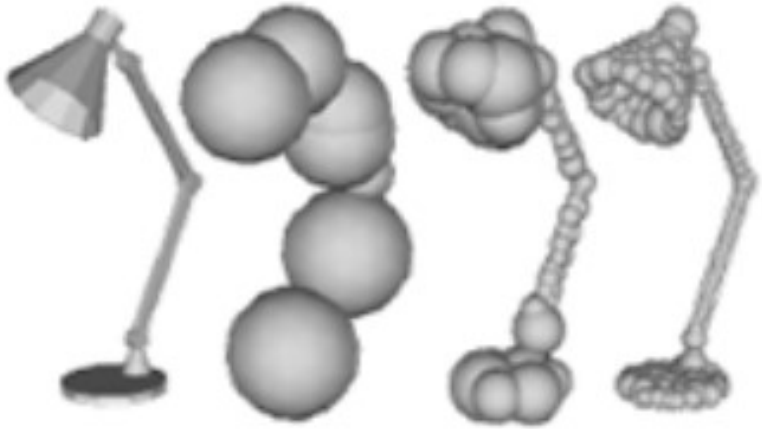
If it’s worth doing, it’s worth doing hierarchically!

- Bvols around objects may help
- Bvols around groups of objects will help
- Bvols around parts of complex objects will help
- Leads to the idea of using bounding volumes all the way from the whole scene down to groups of a few objects
- Meshes are a special case
  - We don’t add individual triangles into the DAG
  - We probably want to do something hierarchically at the level of an individual triangle soup!

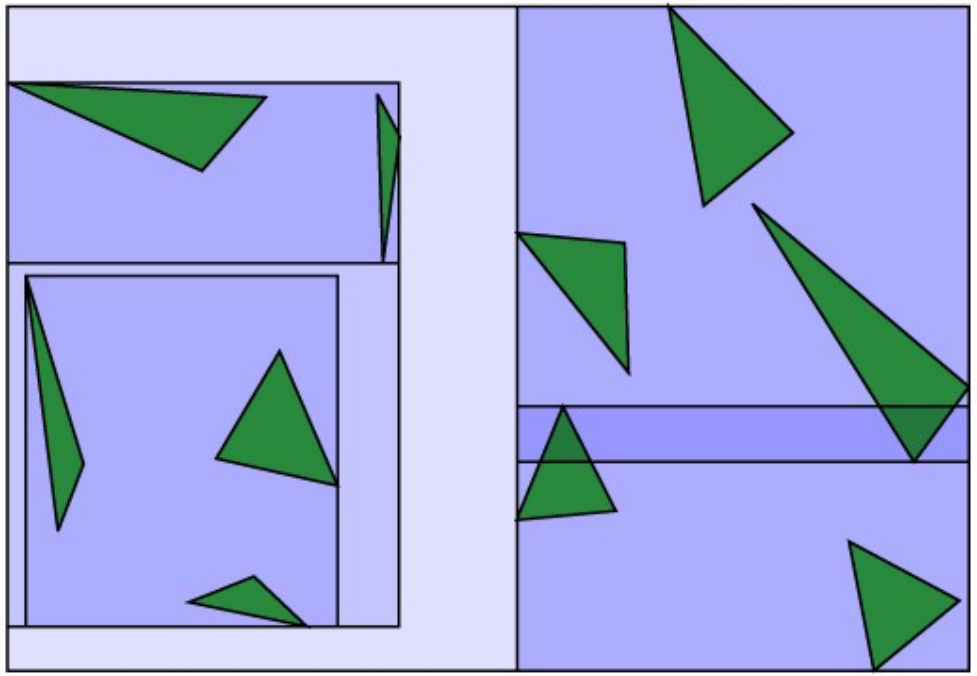
Implementing a bvol hierarchy

- A BoundedSurface can contain a list of Surfaces
- Some of those Surfaces might be more BoundedSurfaces
  - Axis Aligned Bounding Boxes (AABB), Easy!
  - Spheres, can computed in expected O(n) time
    - WELZL, E. 1991. Smallest enclosing disks (balls and ellipsoids)
    - Other fast approximate methods
  - Oriented Bounding Boxes (OBBs), and other types also possible

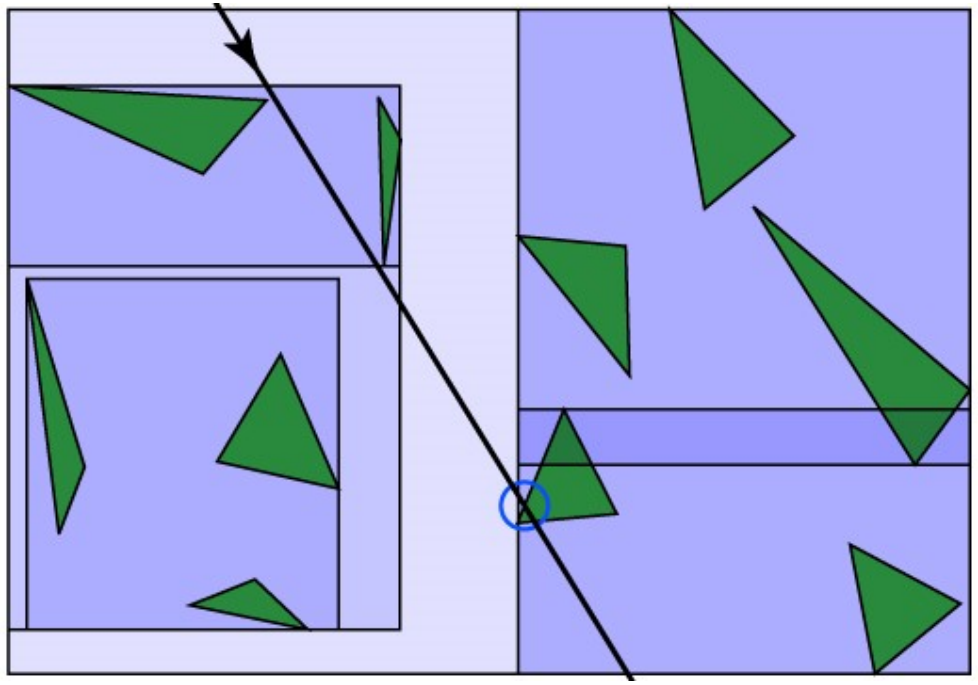




BVH construction example



BVH ray-tracing example



Choice of bounding volumes

- Spheres
  - easy to intersect, not always so tight,
  - Tricky to compute tight bounds (google Welzl's smallest enclosing disk algorithm)
- Axis-aligned bounding boxes (AABBs) –
  - easy to intersect, often tighter (esp. for axis-aligned models)
- Oriented bounding boxes (OBBs) –
  - easy to intersect (but cost of transformation), tighter for arbitrary objects
- Computing the bvols
  - For primitives -- generally pretty easy
  - For groups -- not so easy for OBBs (to do well)
  - For transformed surfaces -- not so easy for spheres

Axis aligned bounding boxes

- Probably easiest to implement
- Computing for primitives
  - Cube: very straightforward!
  - Sphere, cylinder, etc.: easy to approximate
  - Groups or meshes: min/max of component parts
- AABBs for transformed surface
  - Easy to do conservatively: bbox of the 8 corners of the bbox of the untransformed surface
- Can computer intersection as we saw previously as the intersection of 3 slabs

Building a hierarchy

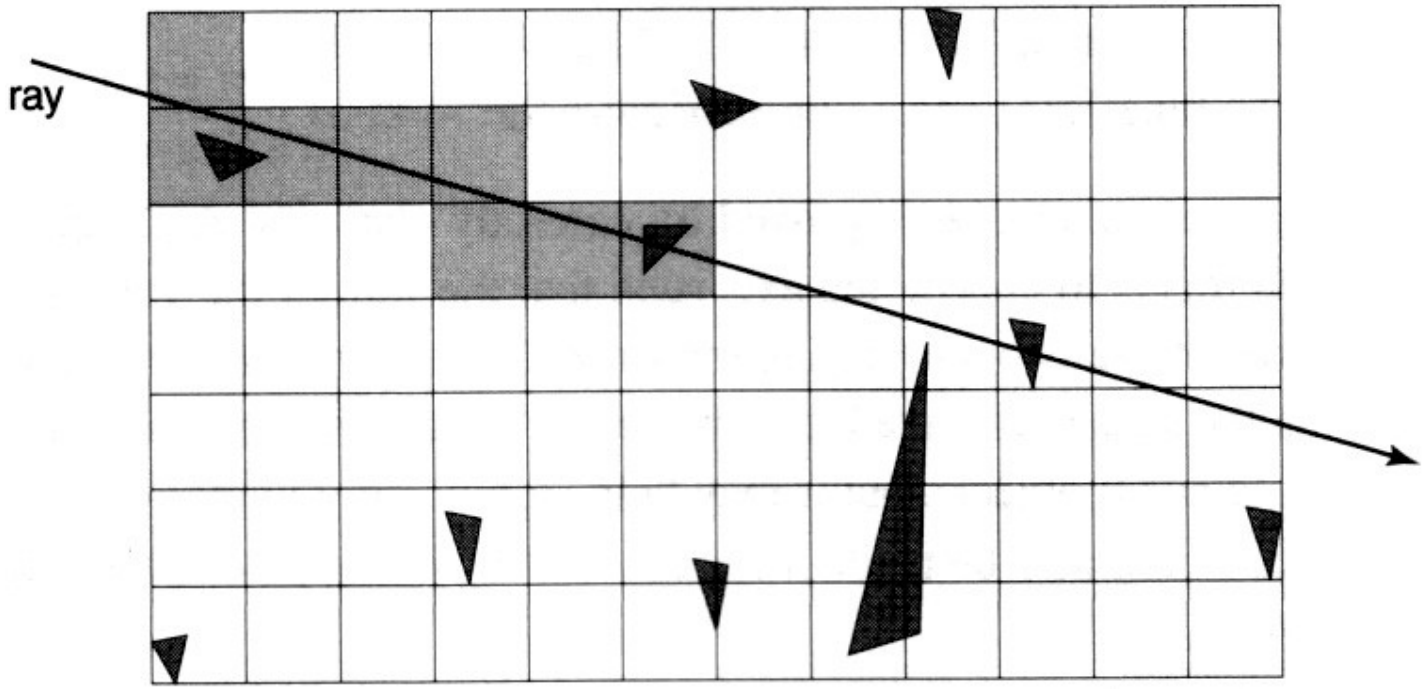
- Usually do it top-down
- Make bbox for whole scene, then split into (maybe 2) parts
  - Recurse on parts
  - Stop when there are just a few objects in your box

Building a hierarchy

- How to partition?
  - Ideal: clusters
  - Practical: partition along axis
    - Median partition
      - More expensive
      - More balanced tree
    - Center partition
      - Less expensive, simpler
      - Unbalanced tree, but that may actually be better

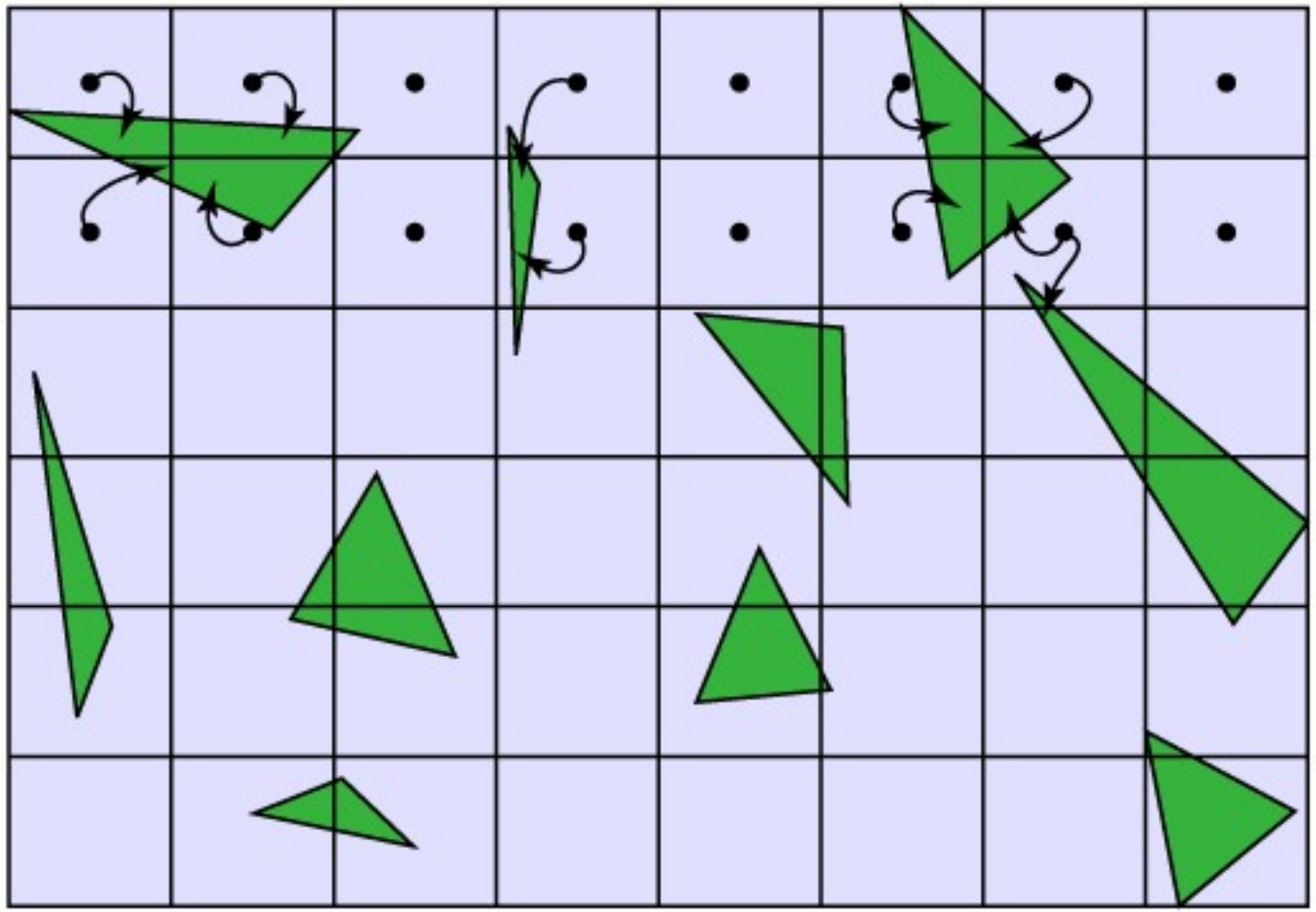
Regular space subdivision

- An entirely different approach: uniform grid of cells

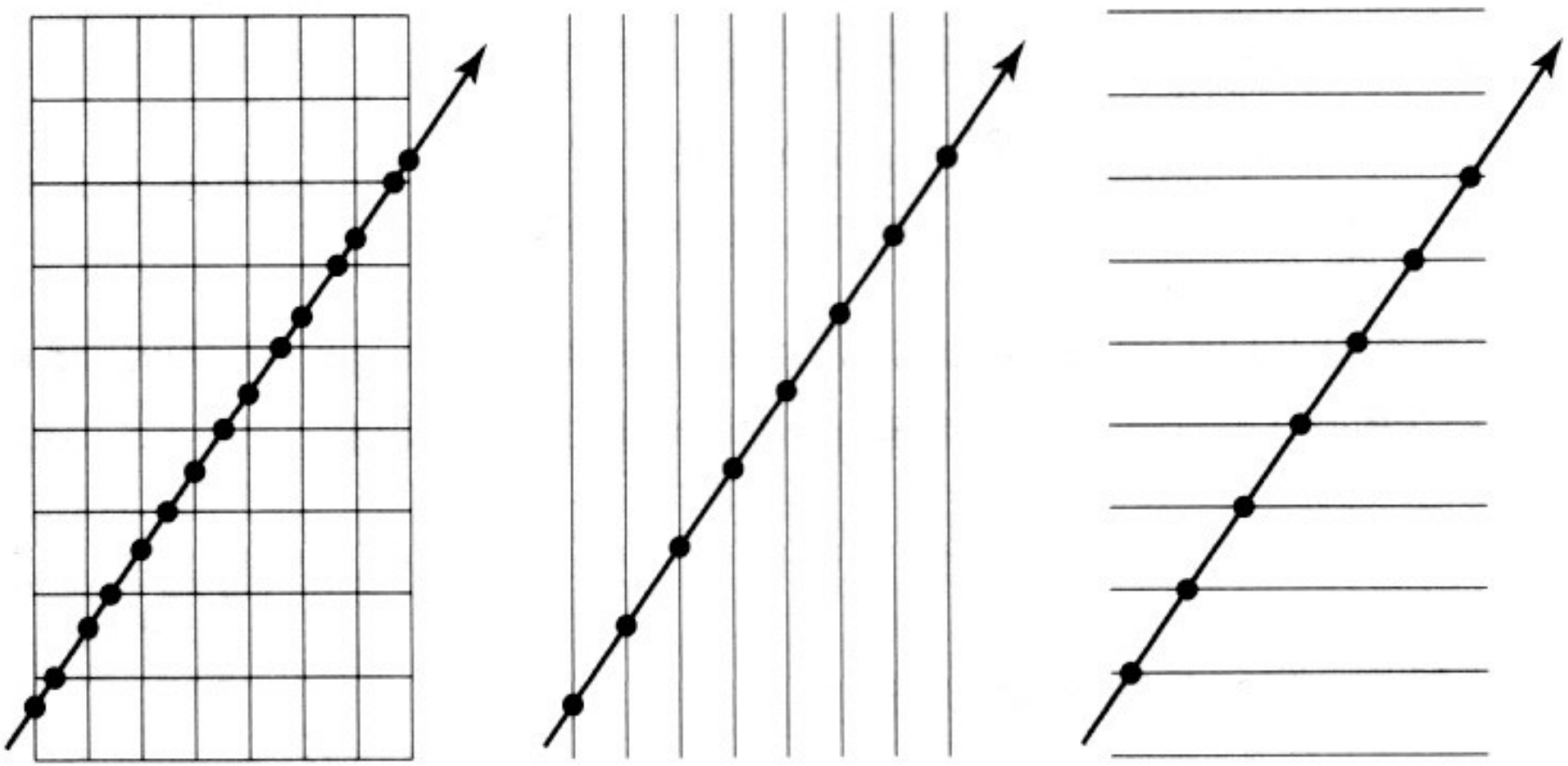


Regular grid example

- Grid divides space, not objects

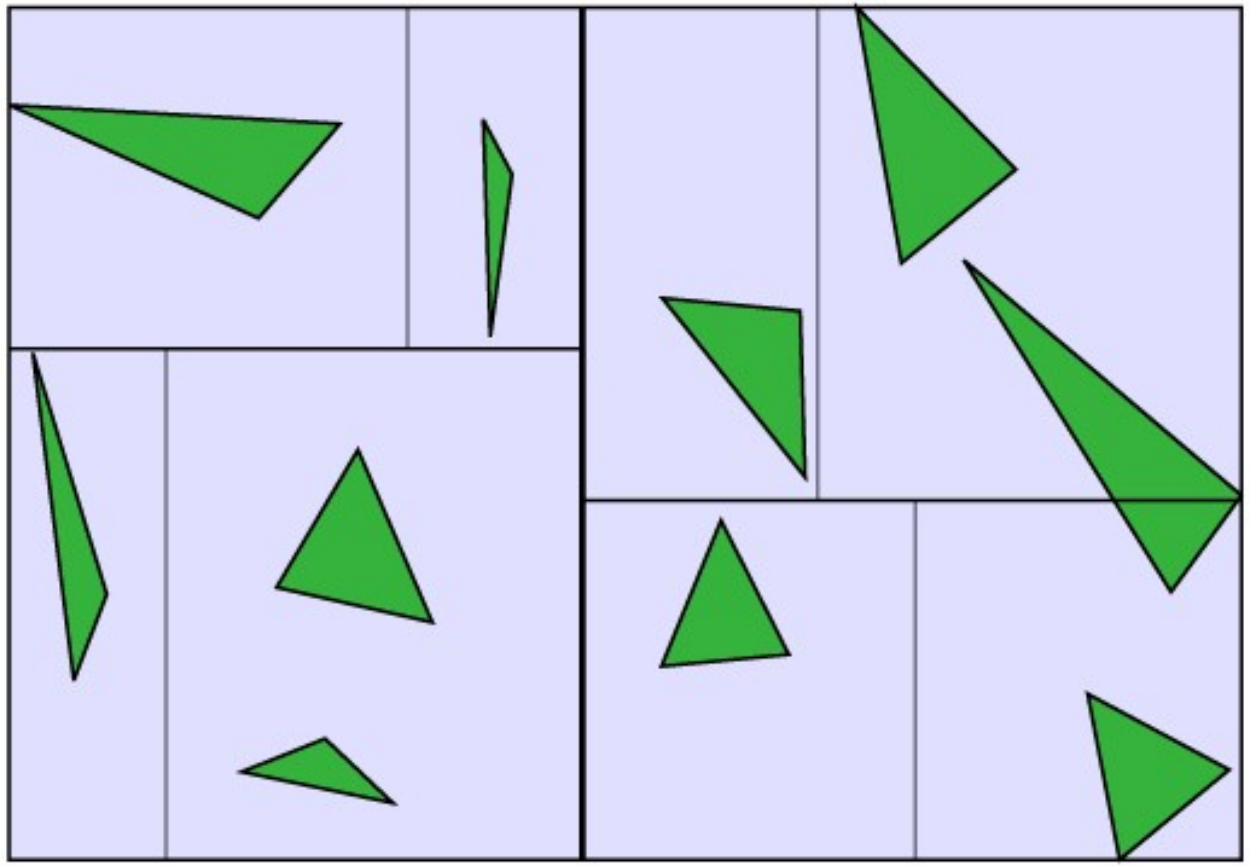


Traversing a regular grid



Non-regular space subdivision

- *k*-d Tree
  - subdivides space, like grid
  - adaptive, like BVH



Implementing acceleration structures

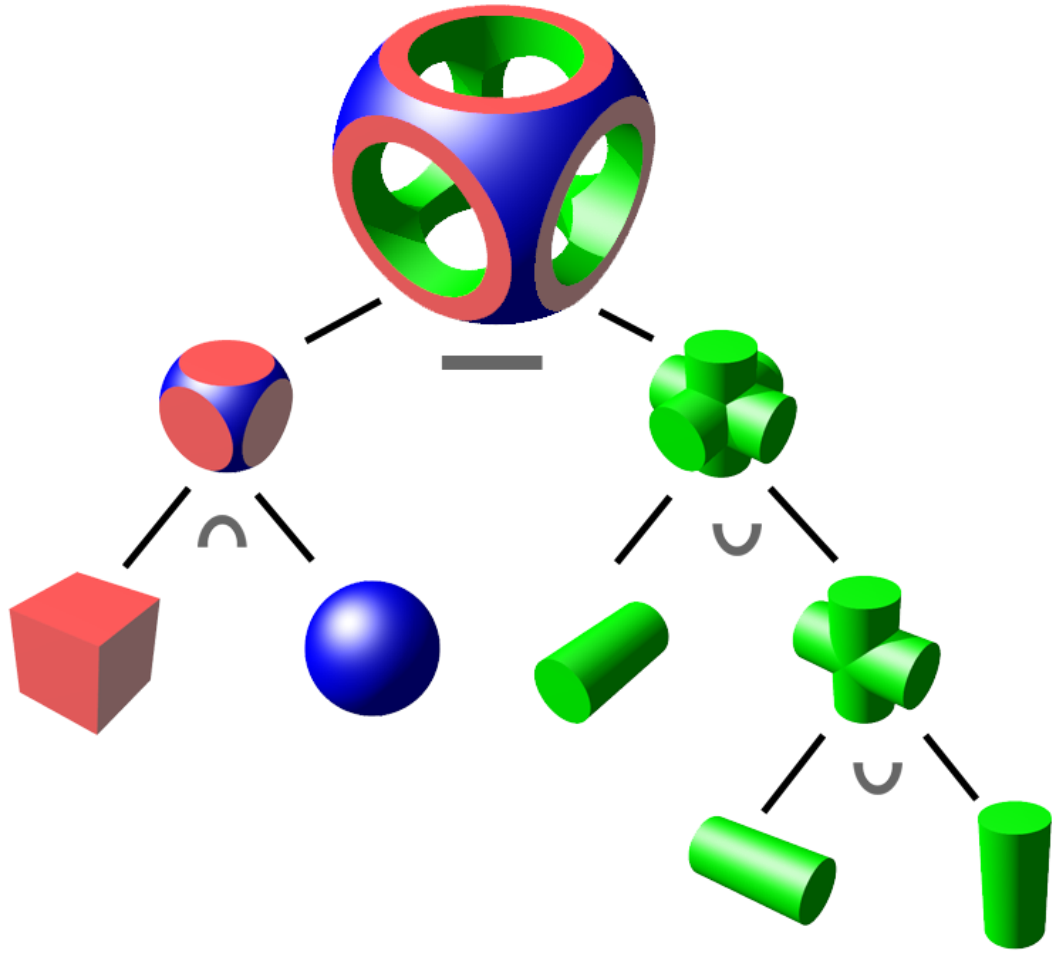
- Conceptually simple to build acceleration structure into scene structure
- Better engineering decision to separate them

Other ray tracing features

- Fresnel reflection and refraction
- Area light sources / soft shadows
- Super sampling for anti aliasing & jittering
- Depth of field blur / real lens modelling
- Motion blur
- Transparency and compositing
- Cartoon shaders
- Texture maps (colour, displacement, normal)
- Other implicit surfaces (metaballs)
- Subdivision surfaces / parametric surfaces
- Constructive solid geometry
- Perlin Noise

Constructive Solid Geometry

- Exactly like intersecting a cube using 3 slabs
- Assemble binary set operations in a tree (works nicely with a dag scene graph)
- Ray intersection class must maintain a list of all intersections instead of closest intersection.



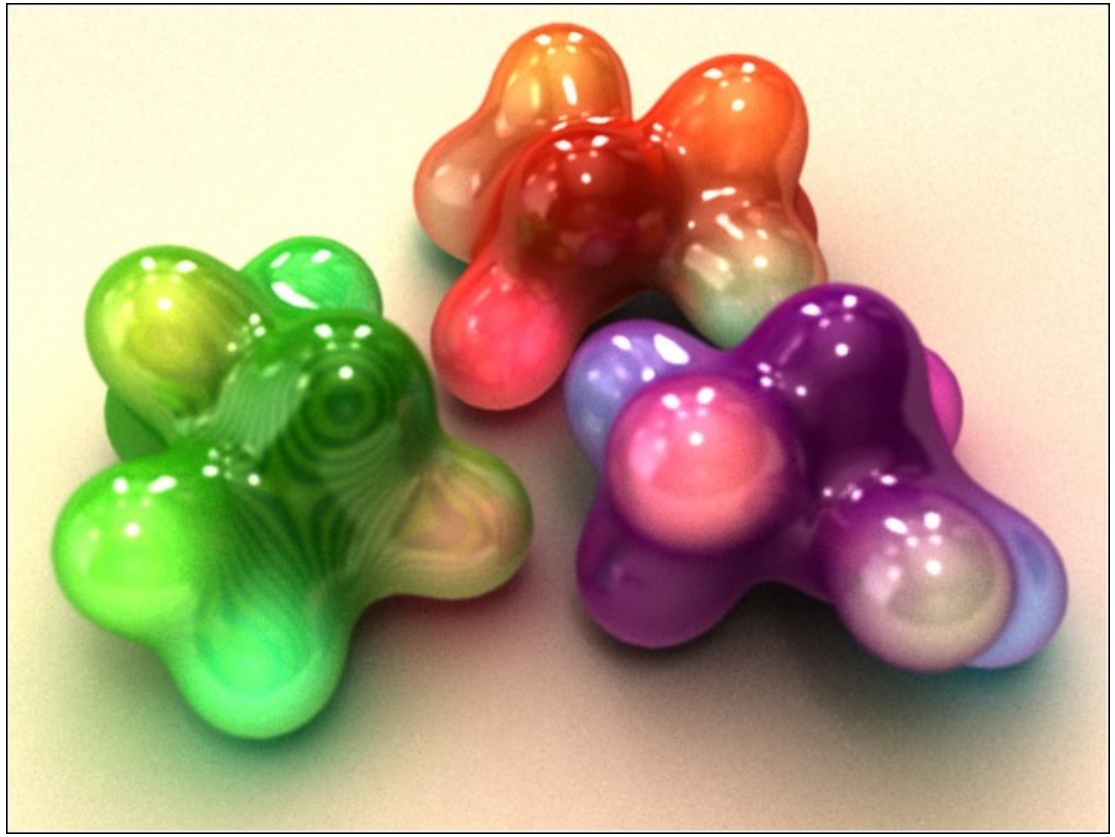
Meta Balls

- Implicit function (other (better) choices exist!)

$$\sum_{i=0}^{n} f_i(\boldsymbol{x}) \leq \text{threshold}$$

$$f_i(\boldsymbol{x}) = \frac{1}{\|\boldsymbol{x} - \boldsymbol{x}_i\|^2}$$

- ***Consider the picture at right***
  - ***What will be the normal?***
  - ***How do you find the intersection?***
  - ***Where do the colours come from?***

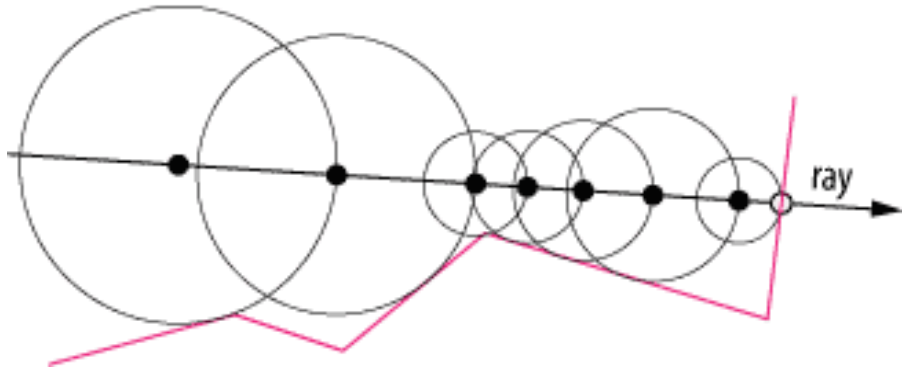


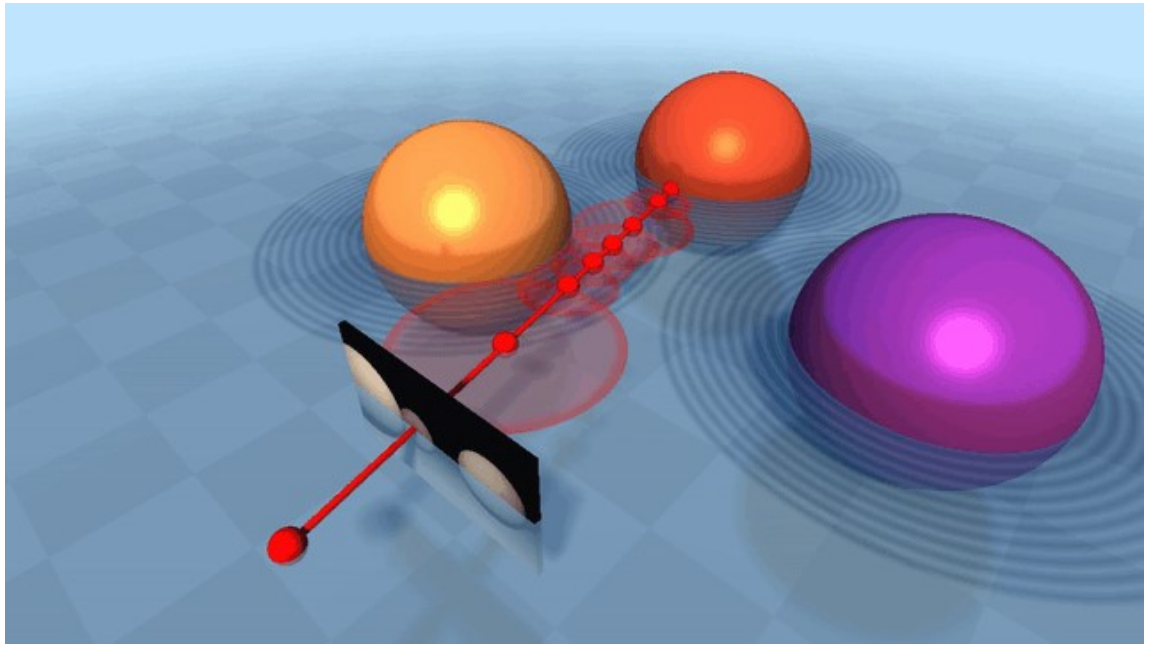
Meta Balls - Normals

- Not knowing the equation for $f_i(\boldsymbol{x})$ can we approximate $\nabla_{\boldsymbol{x}} f_i$?
  - Finite differences?
- If $f_i(\boldsymbol{x}) = \frac{1}{\|\boldsymbol{x}-\boldsymbol{x}_i\|^2}$, then what is $\nabla_{\boldsymbol{x}} f_i$?
- For $r > 0$, let $p(r) = \begin{cases} 6r^5 - 15r^4 + 10r^3 & r < 1 \\ 0 & \text{otherwise} \end{cases}$
  - Why is this nice? Finite support!
  - If $f_i(\boldsymbol{x}) = p(\|\boldsymbol{x} - \boldsymbol{x}_i\|)$, then what is $\nabla_{\boldsymbol{x}} f_i$?

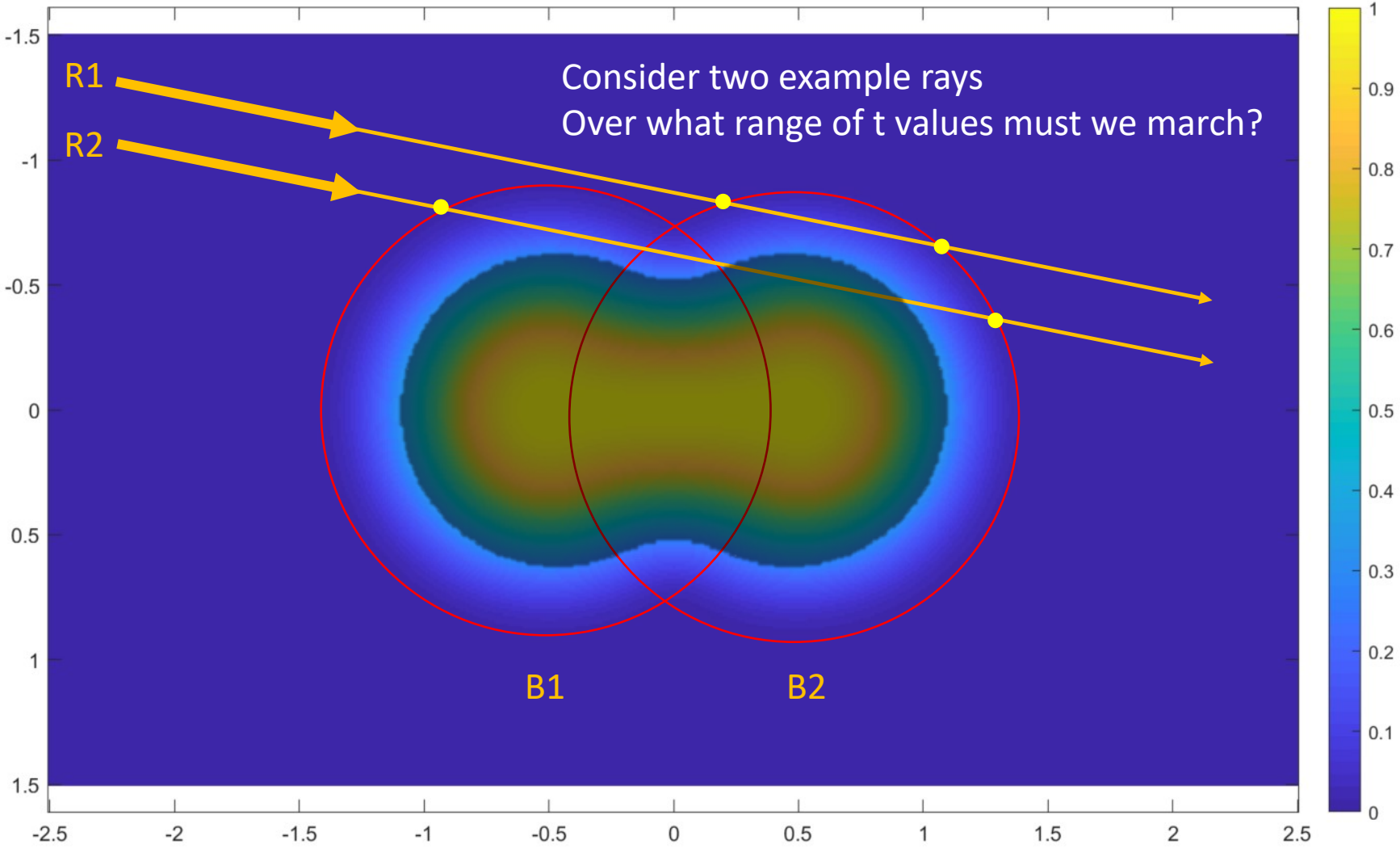
Meta Balls – Ray marching

- Easiest approach to meta ball intersection is ray marching
  - Given a signed distance function, then at any point we can compute the **distance** and know that we can step ***at least this far*** along the ray





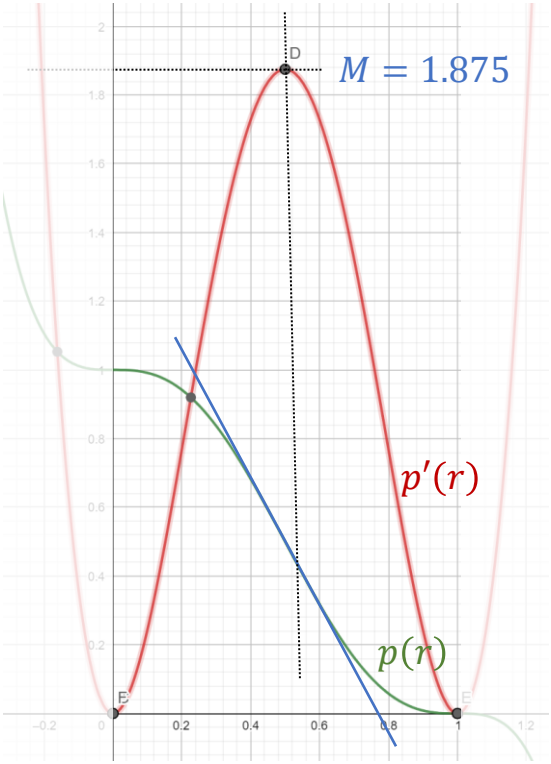
Meta Balls – Ray marching



- Computing the exact distance? Tricky!
  - But can underestimate!
  - Use Lipschitz constant

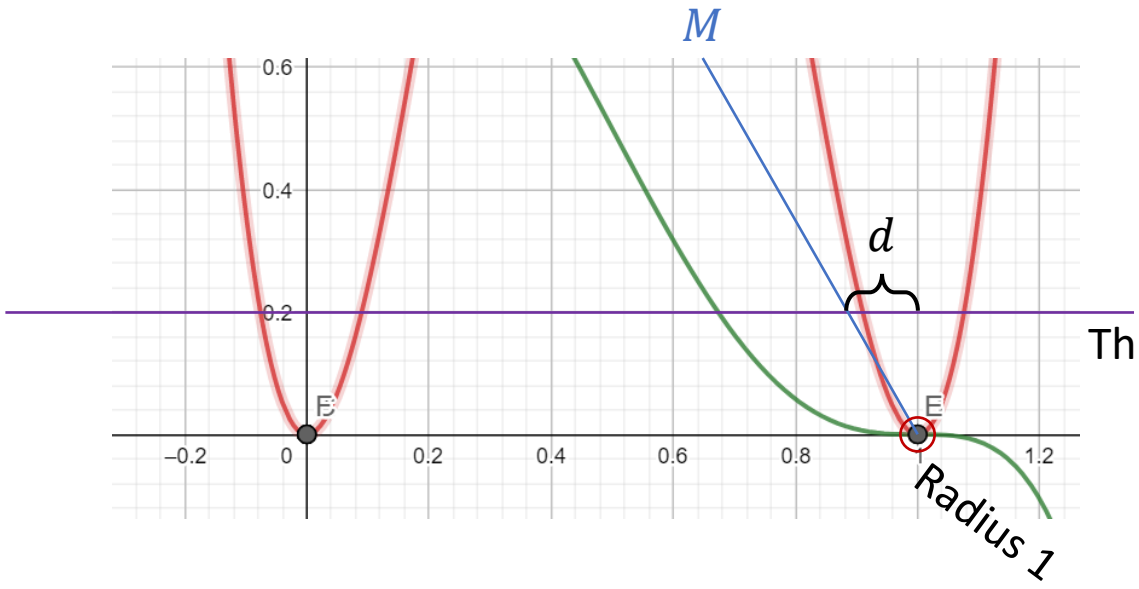
$$|p(r_1) - p(r_2)| \leq M|r_1 - r_2|$$

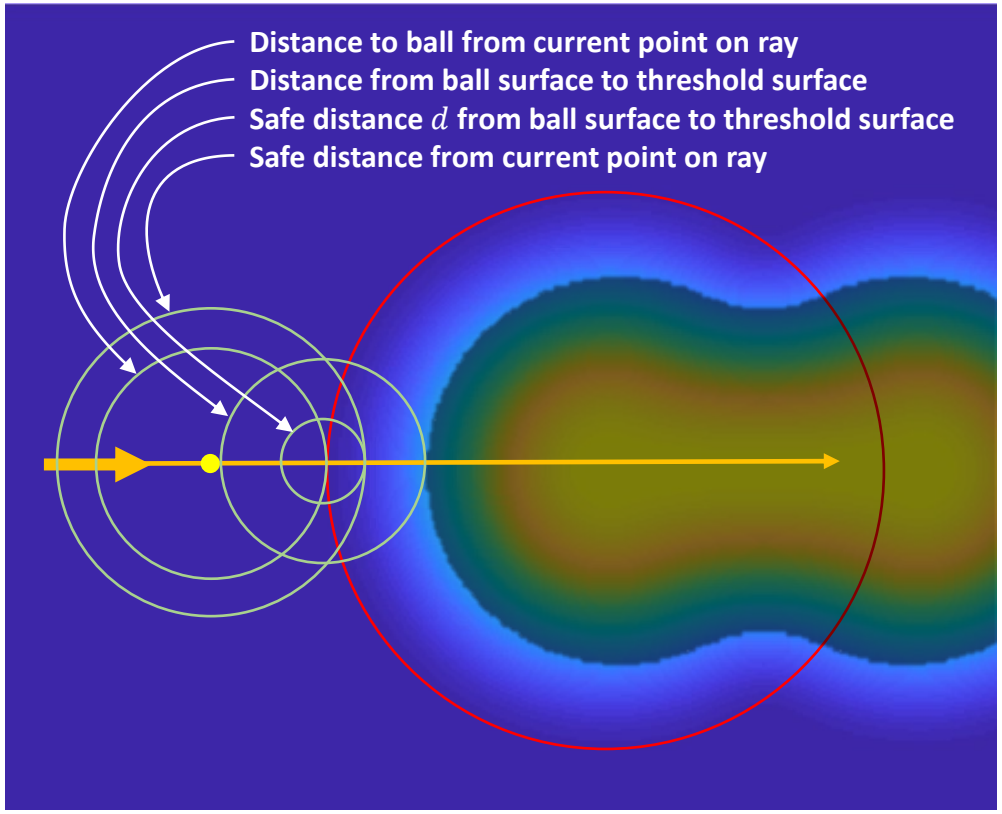
  - Lipschitz constant M is the maximum of 1st derivative on [0,1]



Ray marching with Lipschitz bound

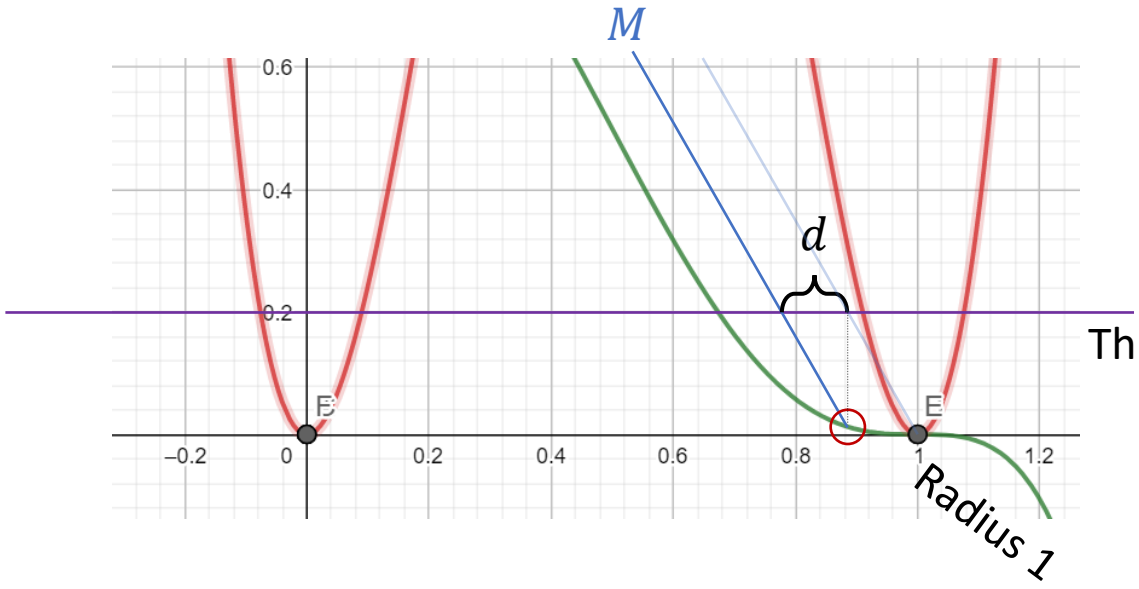
- The Lipschitz bound M gives us worst case growth, for which we can identify a safe underestimate of the distance d to the surface

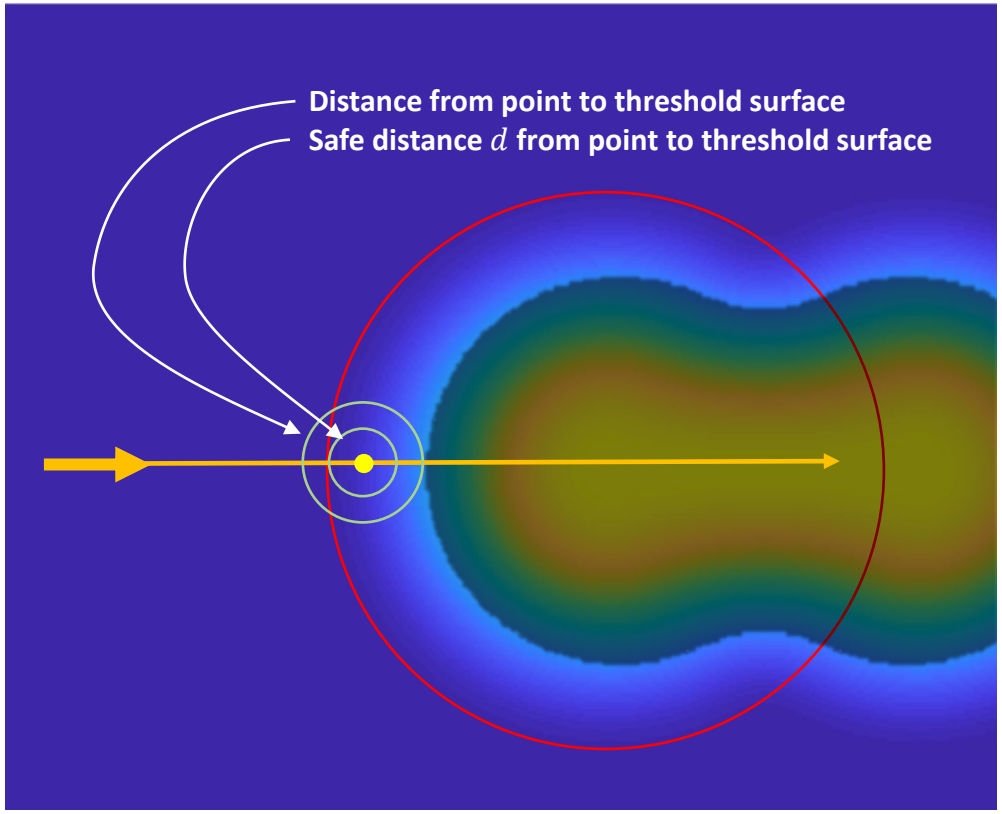




Ray marching with Lipschitz bound

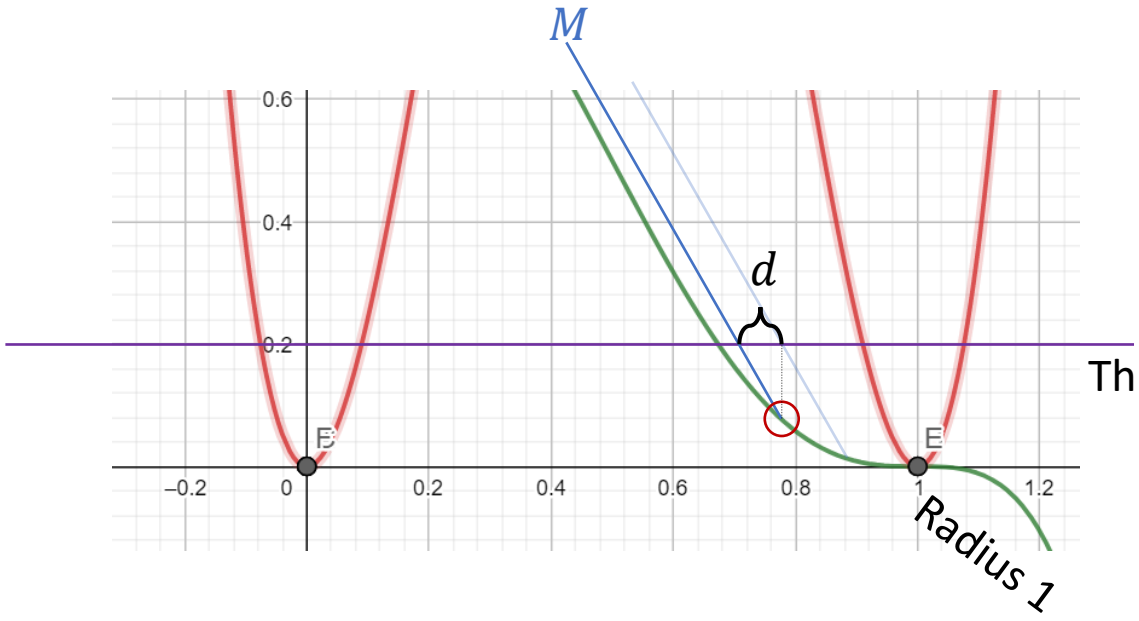
- The Lipschitz bound M gives us worst case growth, for which we can identify a safe underestimate of the distance d to the surface

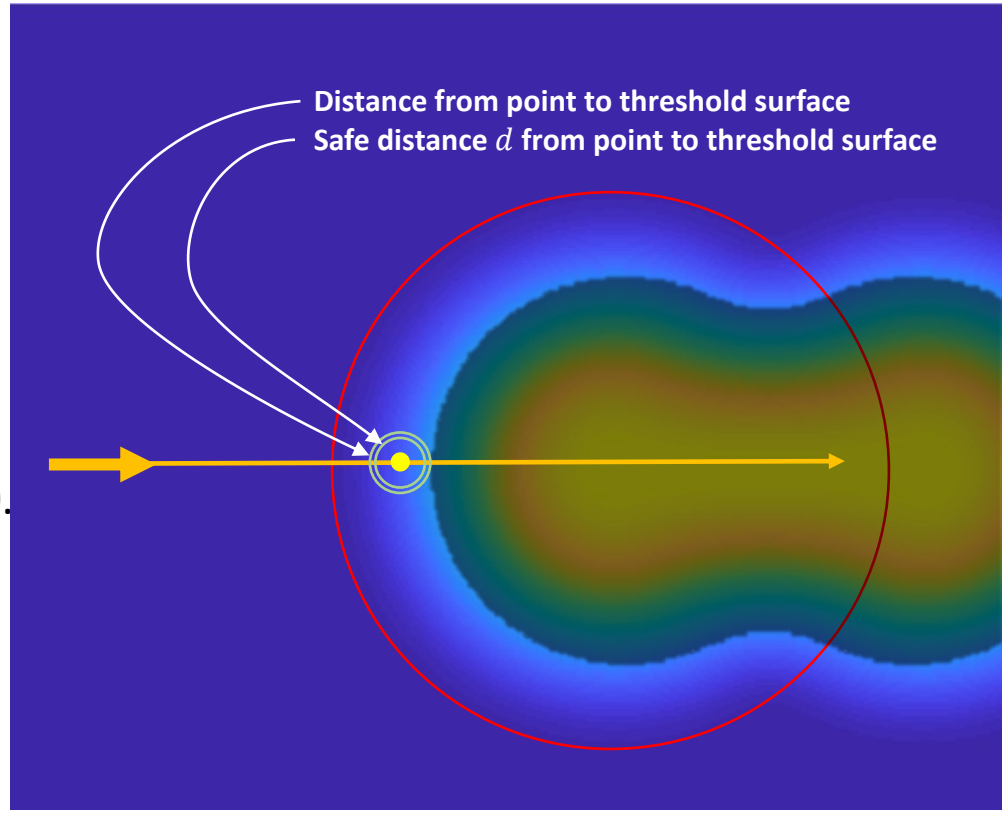




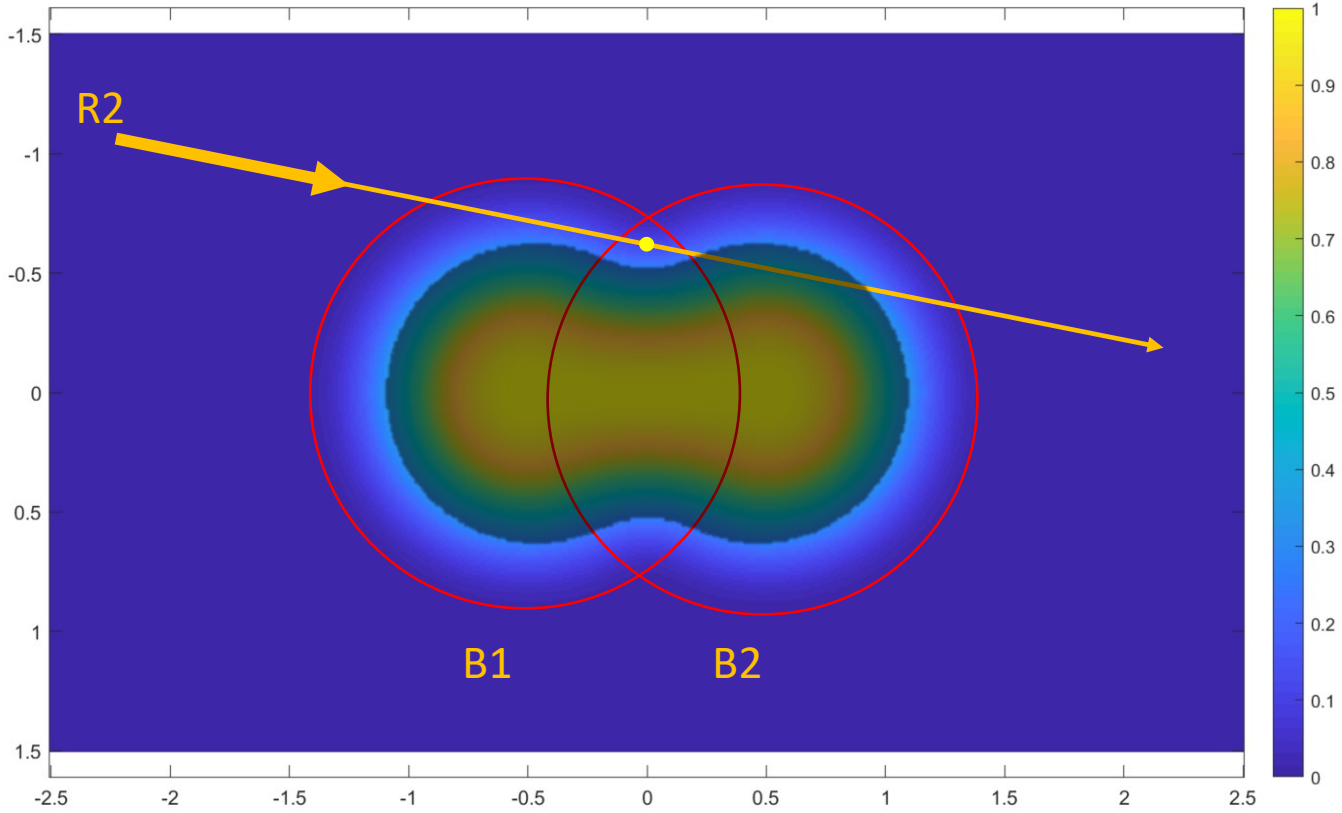
Ray marching with Lipschitz bound

- The Lipschitz bound M gives us worst case growth, for which we can identify a safe underestimate of the distance d to the surface





Ray marching – be careful!

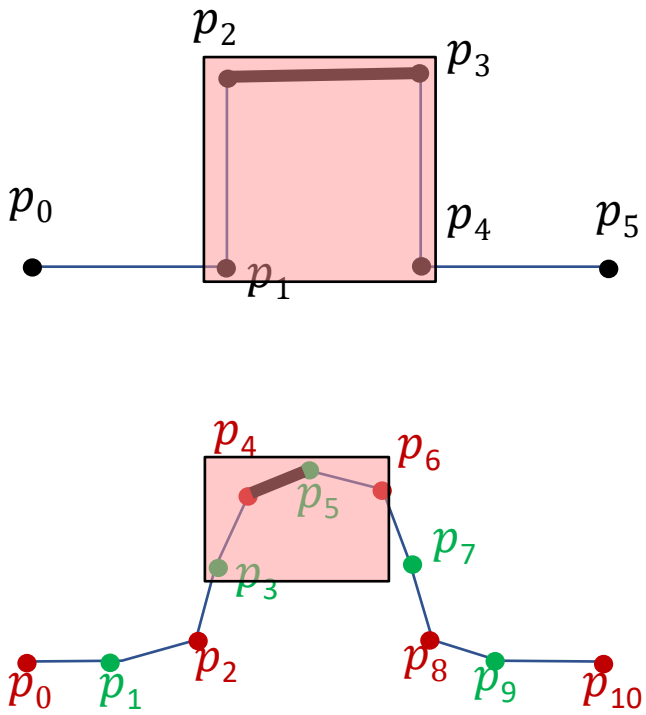


- When the ray point falls into multiple balls, the implicit field function is the sum of both
- The Lipschitz bound is likewise the sum of the bounds.

Review and more information on sphere-tracing of implicit surfaces:
https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields

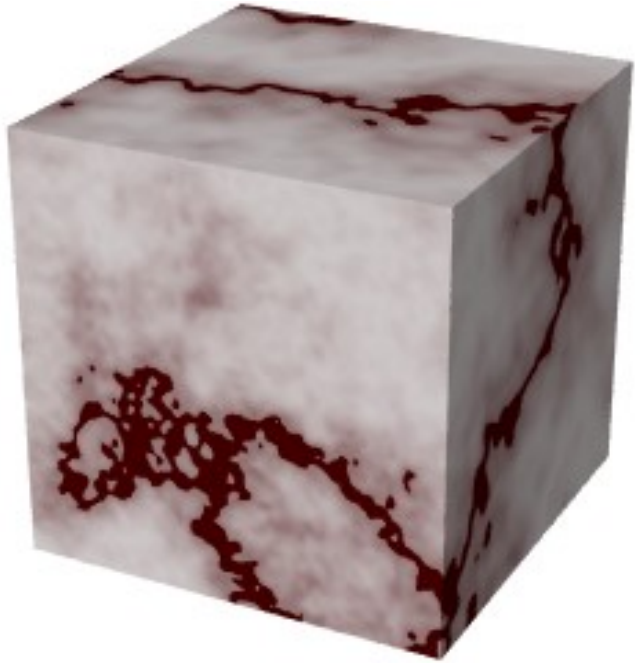
Subdivision Surfaces

- Many subdivision schemes simply involve a ***convex combination of points***
  - Chaikin, Lane-Reisenfeld, Loop, Catmull-Clark
- Convex hull of points can be used for ray-intersection test
  - Recursively subdivide checking with bounding box of all child faces
  - Once sufficiently flat (or small), use ray polygon intersection



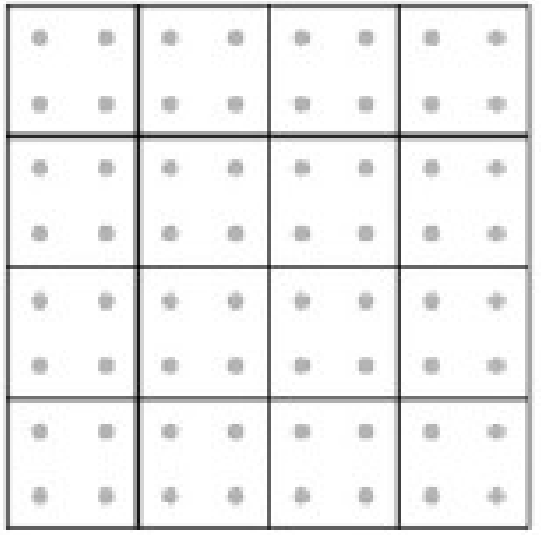
Perlin Noise

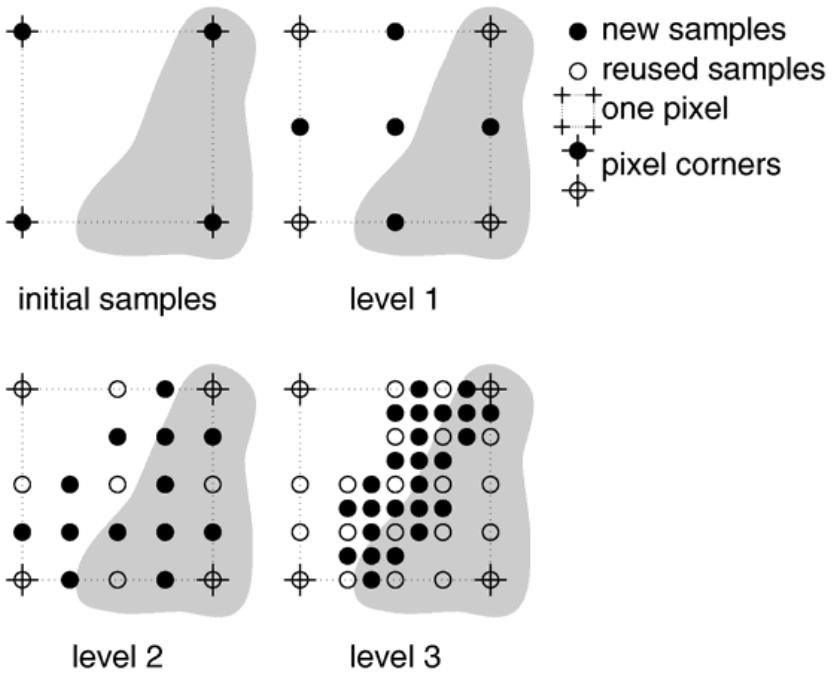
- Useful for clouds, marble, and other random varying effects
- Code is very small… noise is easy to compute!



Super Sampling and Jittering Samples

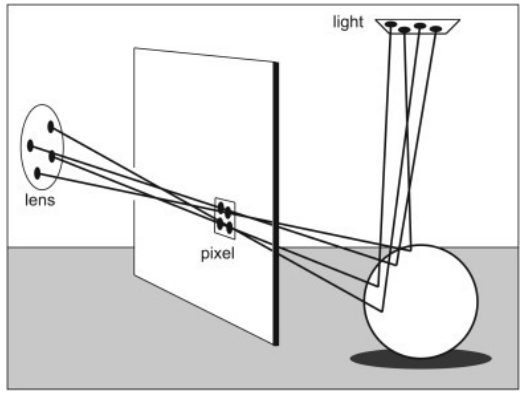
- Cast more rays through sub pixel positions.
- Watch out for aliasing!
  - Jittering samples will replace aliasing with noise, which is preferable
  - Noise added to sub-pixel position should not be too large!
- Super sampling can be done adaptively based on scene complexity at the pixel

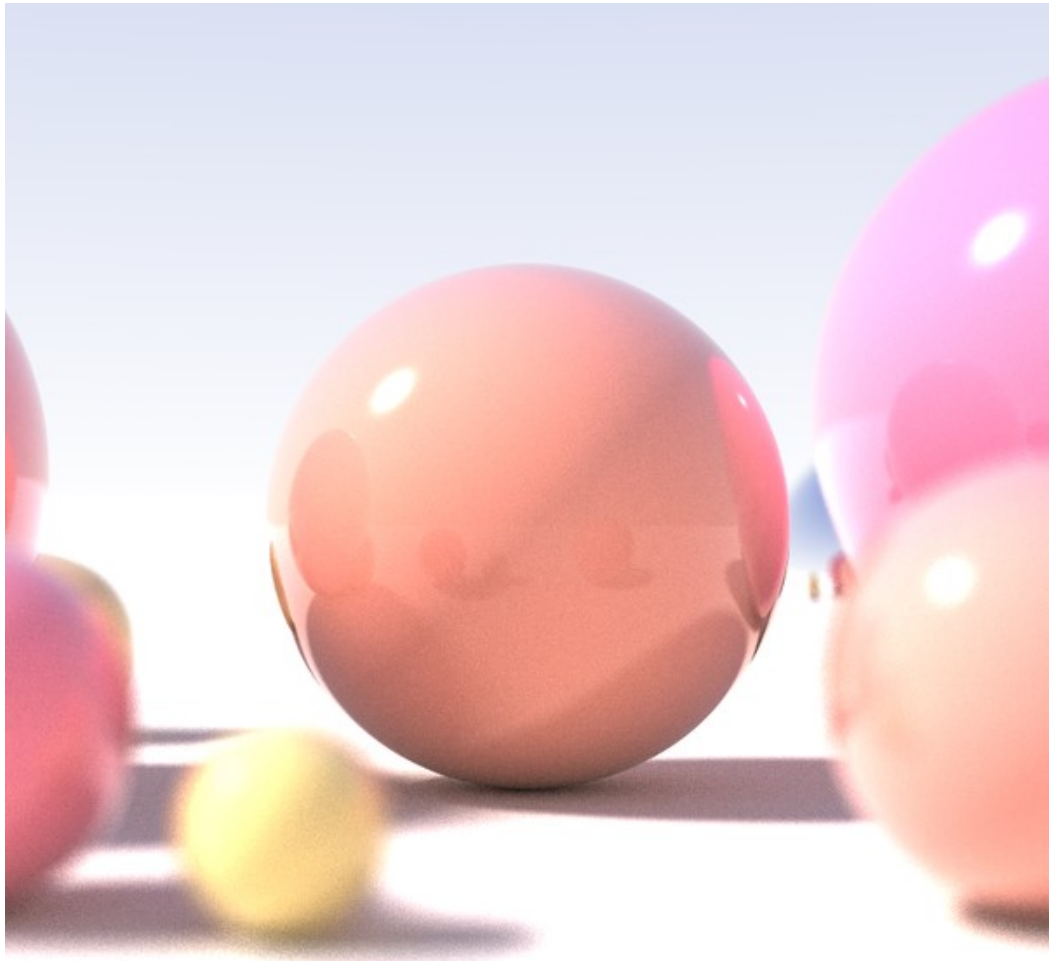




Depth of Field, Area Lights

- Cast more rays!
  - Sample different ***random positions*** on lens aperture by changing the eye position
  - For an area light source, do shadow ray test and lighting computation for a different ***random positions*** on the area light source.





Motion Blur

- Knowing object trajectories over a time interval, cast rays at different times and average.
- Can also just average a sequence of images, but this would produce temporal aliasing!
- Can also be smart about not casting few rays where nothing is in motion.

[Thomas Porter 1984]


4

Reflection and Refraction

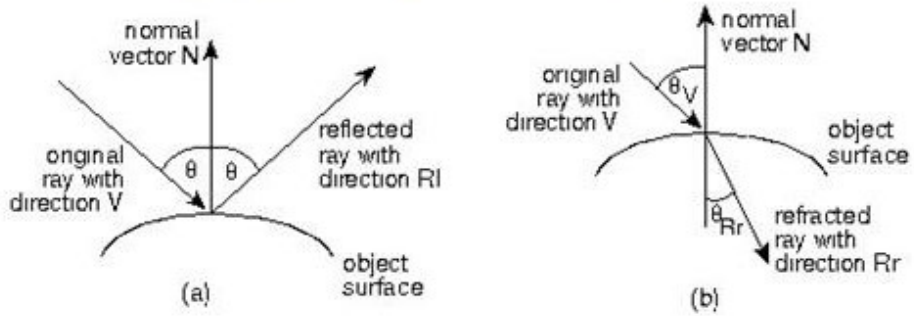
n1 = index of refraction of original medium

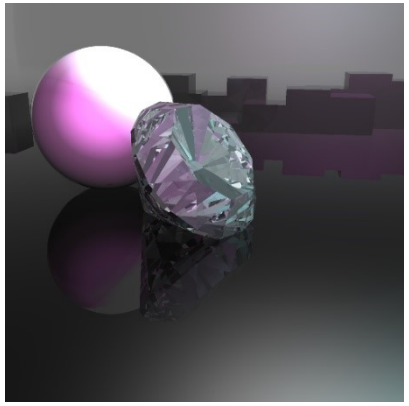
n2 = index of refraction of new medium

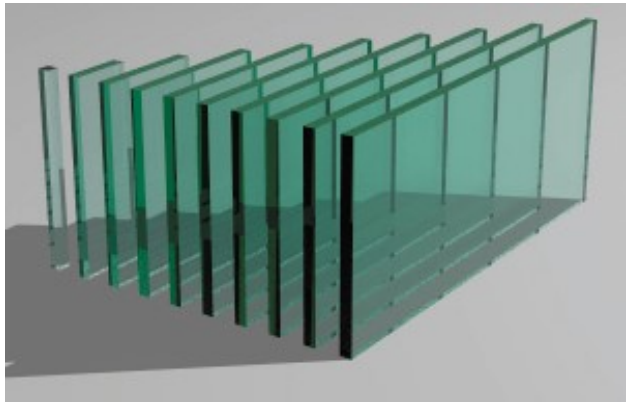
$$n = \frac{n1}{n2}$$

$$c2 = \sqrt[2]{1 - n^2 \times (1 - c1^2)}$$

$$Rr = (n \times V) + (n \times c1 - c2) \times SN$$

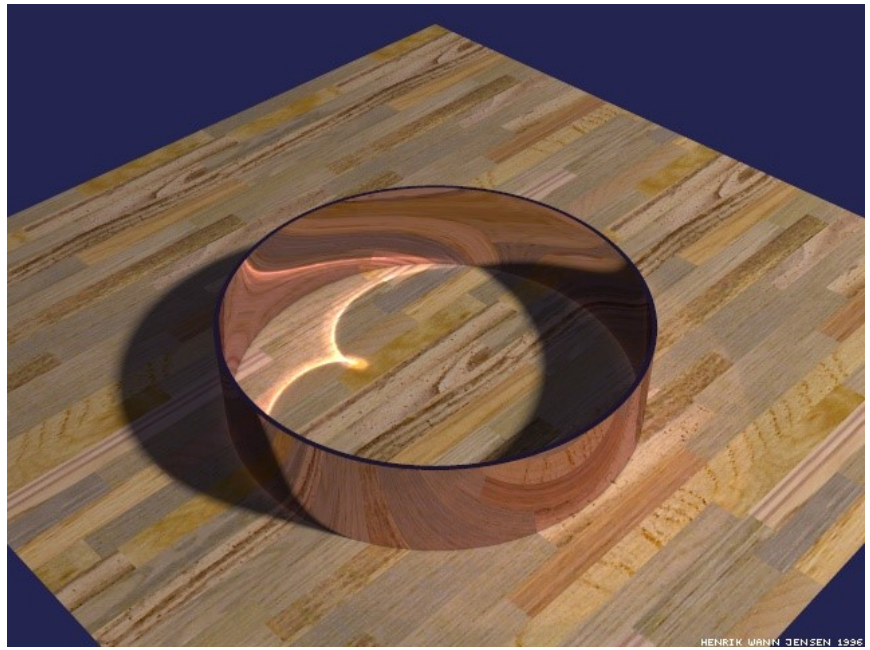






Caustics

- Reflection and refraction scatters light too
  - Very challenging without photon mapping, or path tracing, etc.



Anisotropic BRDFs

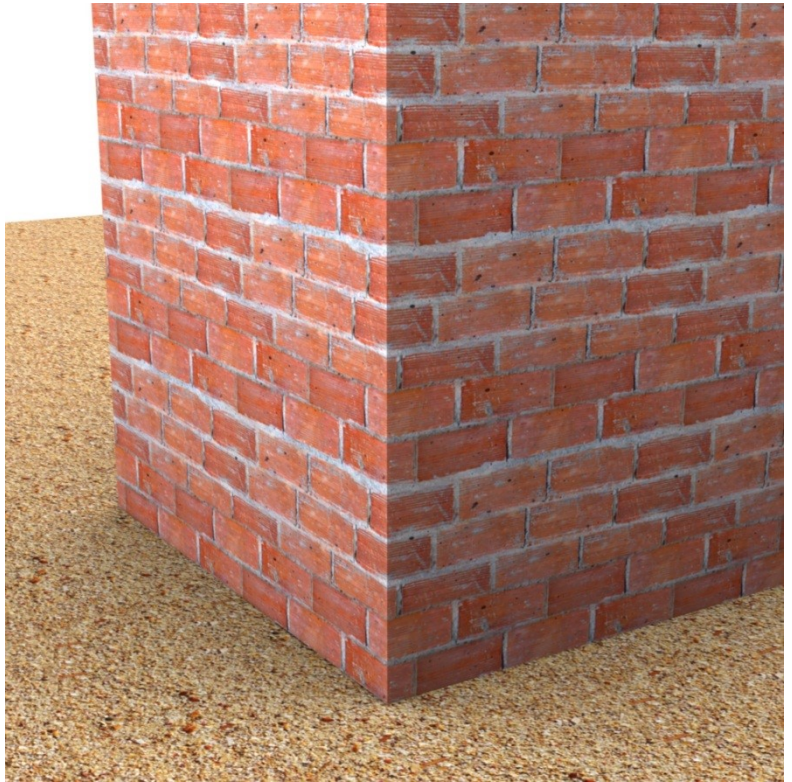
- Can use a varying reflectance function for computing the amount of reflected light.

Cartoon Shaders

- Change lighting computations so that they produce a smaller number of colours.
- Add black edges at boundaries between objects.

Texture Maps

- Can download texture images from a variety of repositories.
- Easy to texture certain primitives (cubes, spheres, planes)
- Meshes often have texture coordinates attached
- ***Sampling is an issue!!***
  - Resolution may be too high or too low giving a poor quality result



Review and More Information

- FCG Chapters 12 and 13
- https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields
- CGPP Chapter 27 Material scattering models
- CGPP Chapter 37 Spatial data structures