Bremen

CG VR

Advanced Computer Graphics
Acceleration Data Structures (for Raytracing et al.)

G. Zachmann
University of Bremen, Germany
cgvr.informatik.uni-bremen.de

The Costs of Ray-Tracing

cost ≈ height * width *

num primitives *

intersection cost *

size of recursive ray tree *

num shadow rays *

num supersamples *

num glossy rays *

num temporal samples *

num focal samples *

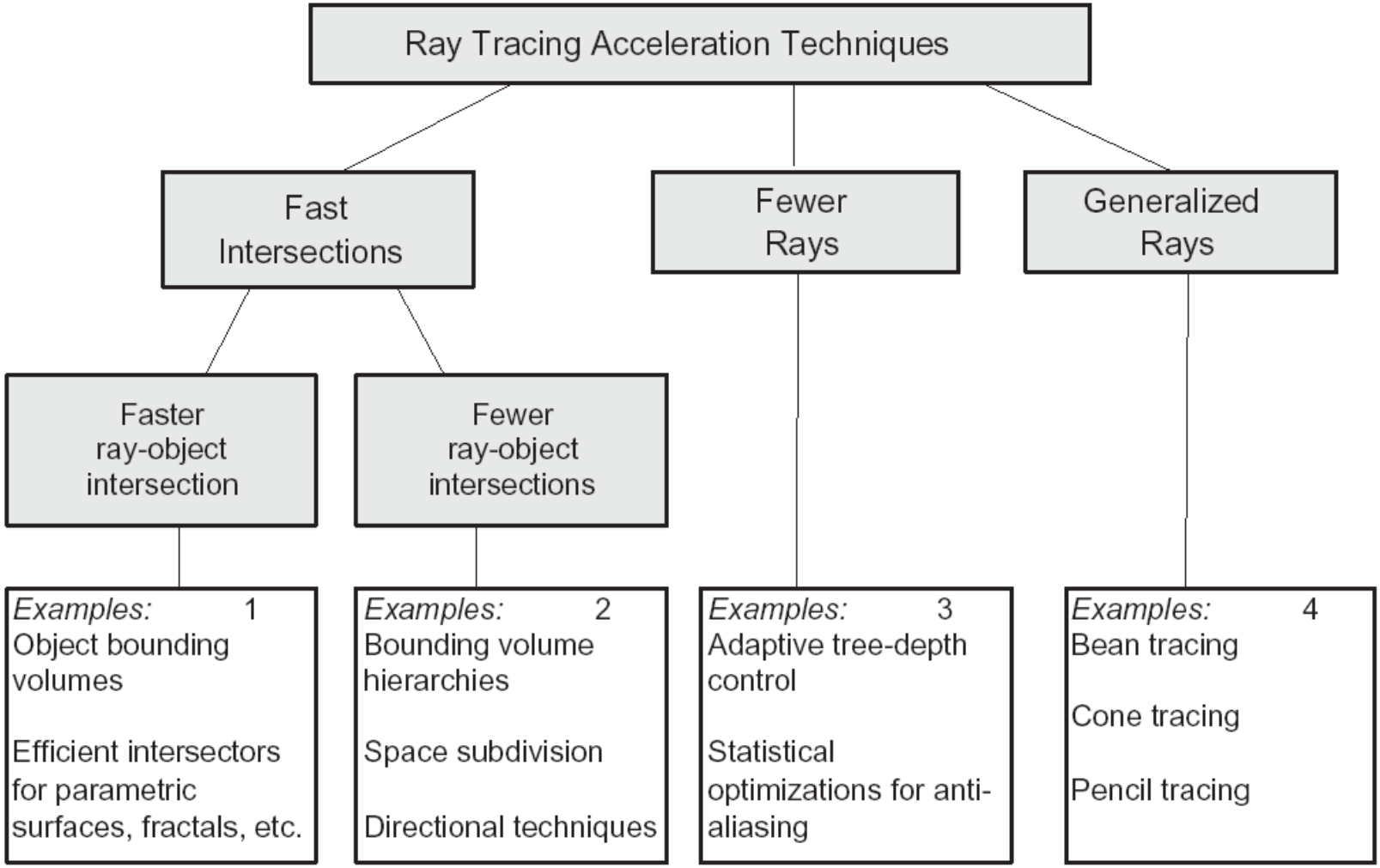
. . .

Can we decrease that?

"Rasterization is fast, but needs cleverness to support complex visual effects. Ray tracing supports complex visual effects, but needs cleverness to be fast."

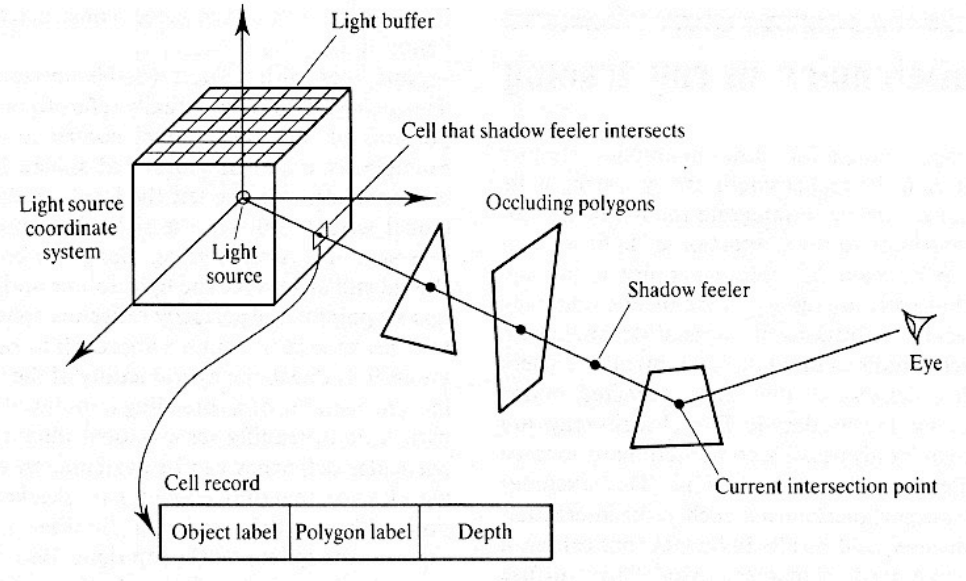
[David Luebke, Nvidia]

A Taxonomy of Acceleration Techniques



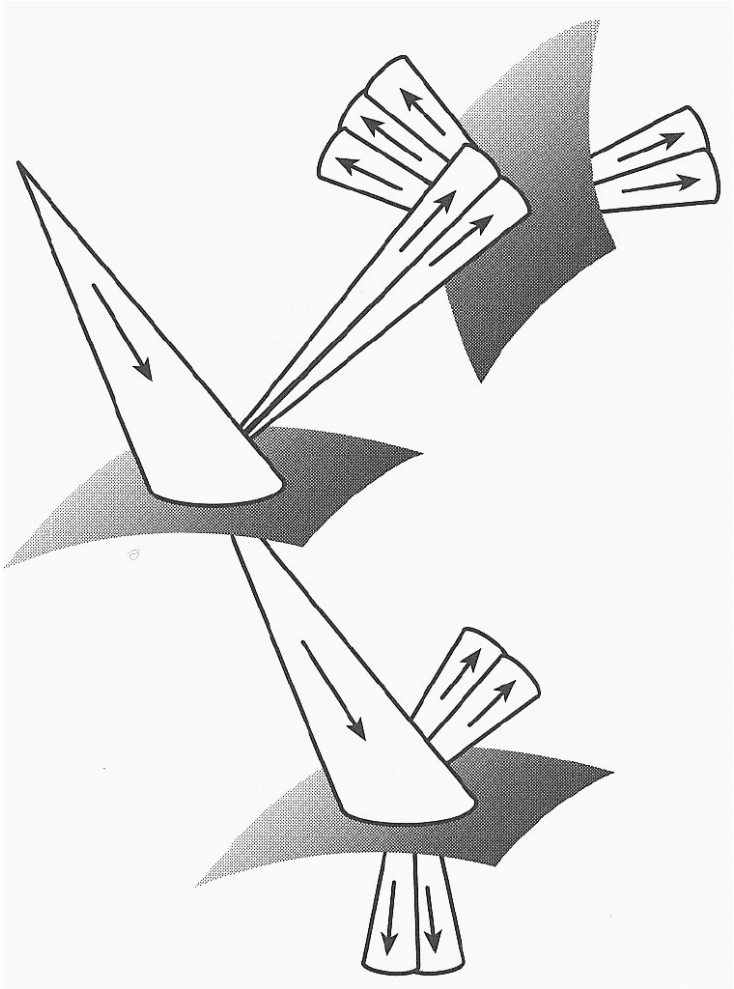
The Light Buffer

- Observation: when tracing shadow rays, it is sufficient to find any intersection with an opaque object
- Idea: for each light source, and for each direction, store a list of polygons lying in that direction when "looking" from the light source
  - The data structure of the light buffer: the "direction cube"
  - Construct either during preprocessing (by scan conversion onto the cube's sides), or construct "on demand" (i.e., insert occluder whenever found one)

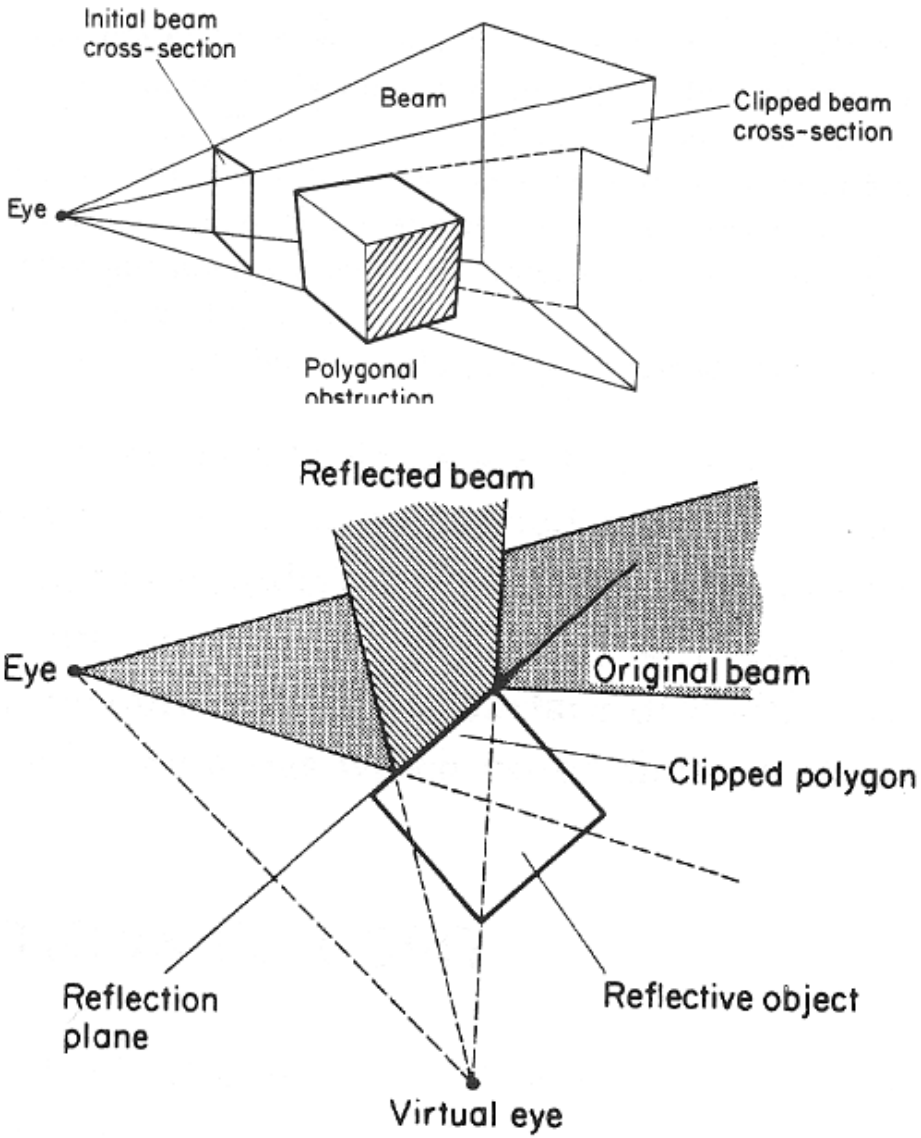


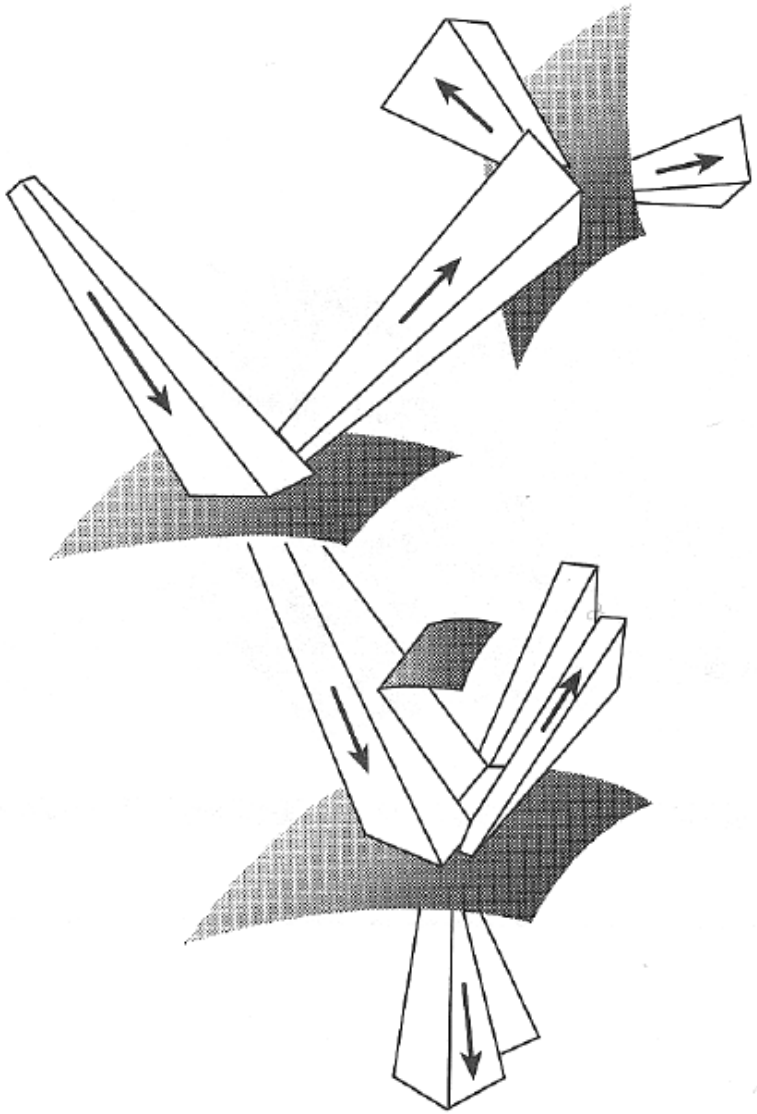
Beam and Cone Tracing

- The general idea: try to accelerate by shooting several or "thick" rays at once
- Beam Tracing:
  - Represent a "thick" ray by a pyramid
  - At the surfaces of polygons, create new beams
- Cone Tracing:
  - Approximate a thick ray by a cone
  - Whenever necessary, split into smaller cones
- Problems:
  - What is a good approximation?
  - How to compute the intersection of beams/cones with polygons?
- Conclusion (at the time): too expensive!



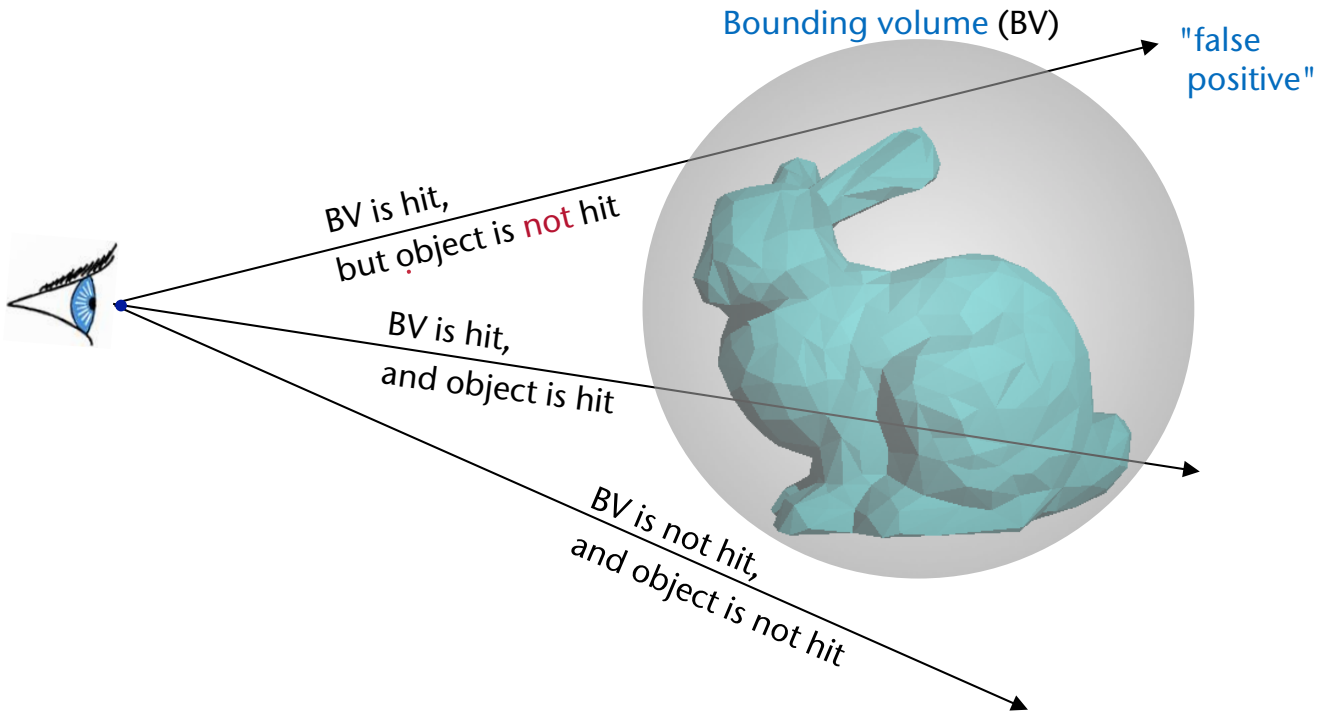
Beam Tracing





Bounding Volumes (BVs)

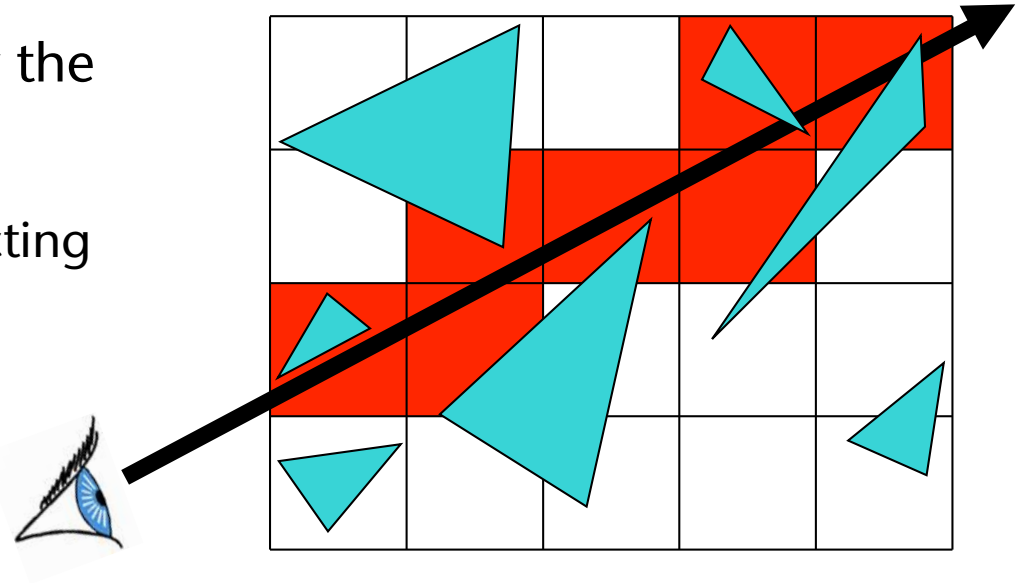
- Basic idea: save costs by precomputations on the scene and filtering of the rays during run-time



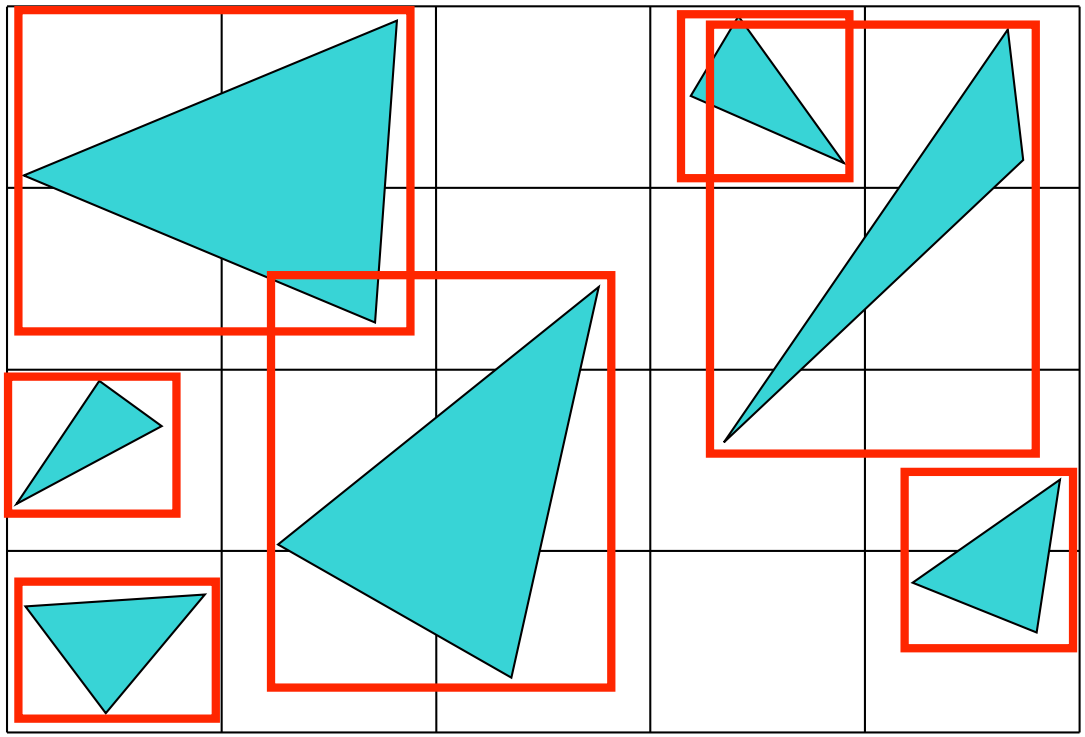
- If the ray misses the BV, then it must also miss the enclosed object

Regular 3D Grids

- Construction of the grid:
  - Calculate BBox of the scene
  - Choose a (suitable) grid resolution (n_x, n_y, n_z)
- For each cell intersected by the ray:
  - Is any of the objects intersecting the cell hit by the ray?
  - Yes: return closest hit
  - No: proceed to next cell



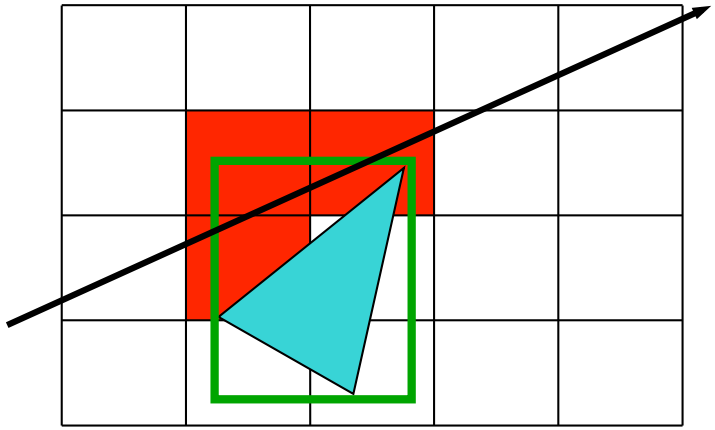
- Precomputation: for each cell store all objects intersecting that cell in a list with that cell⟶ "insert objects in cells"
  - Each cell has a list that contains pointers to objects
- How to insert objects: use bbox of objects
  - Exact intersection tests are not worth the effort
- Note: most objects are inserted in many cells

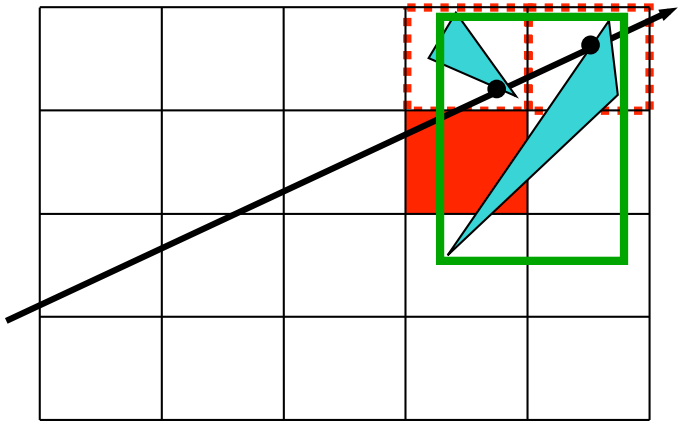


Problems

- Objects could be referenced from many cells

1. Consequence: a ray-object intersection need not be the closest one (see bottom right)
   - Solution: disregard a hit, if the intersection point is outside the current cell
2. Consequence: we need a method to prevent the ray from being intersected with the same object several times (see bottom left)





The Mailbox Technique

- Solution: assign a mailbox with each object (e.g., just an integer instance variable), and generate a unique ray ID for each new ray
  - For the ray ID: just increment a counter in the constructor of the ray class
- After each intersection test with an object, store the *ray ID* in the object's *mailbox*
- Before an intersection test, compare the ray ID with the ID stored in the object's mailbox:
  - Both IDs are equal ⟶ the intersection point can be read out from the mailbox;
  - IDs are not equal ⟶ perform new ray-object intersection test, and save the result in the mailbox (together with the ray ID)

Optimization of the Mailbox Technique

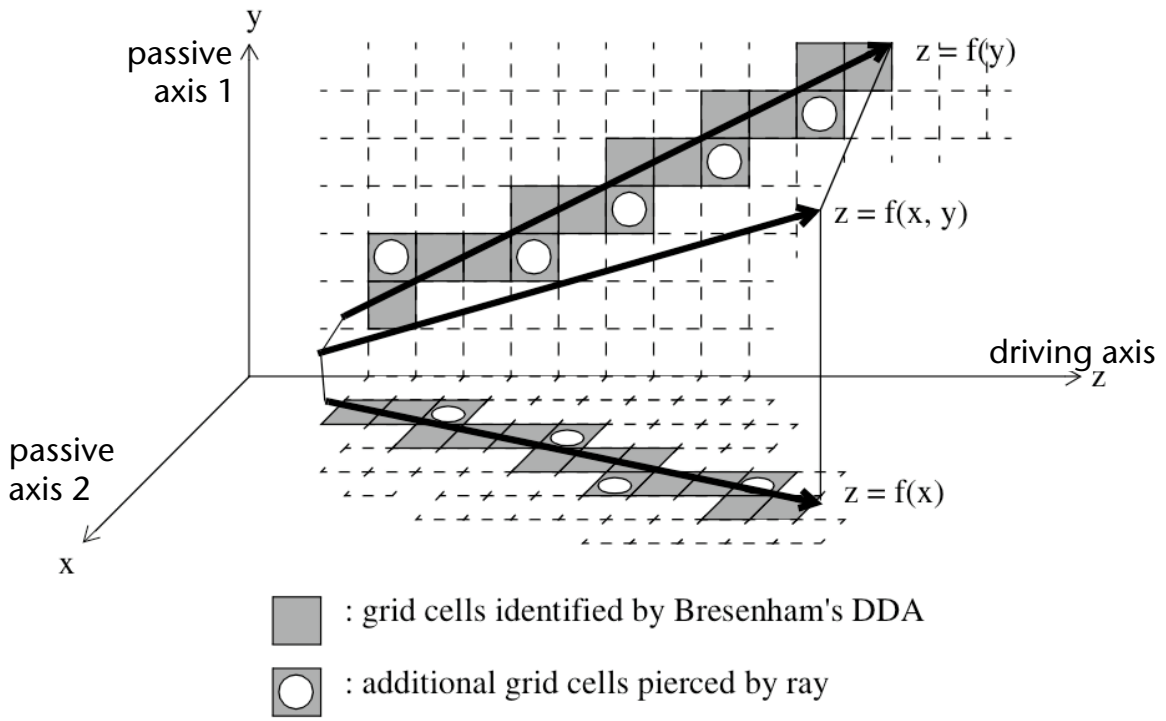
- Problems with the naive method:
  - Writing the mailbox invalidates the cache
  - You cannot test several rays in parallel
- Solution: store mailboxes seperately from geometry
  - Maintain a small hash-table with each ray that stores object IDs
    - Works, because only few objects are hit by a ray
    - So, the hashtable can reside mostly in level 1 cache
  - A simple hash function is sufficient
  - Now, checking several rays in parallel is trivial
- Remark: this is another example of the old question, whether one should implement it using an

  "Array of Structs" (AoS) or a "*Struct of Arrays*" (SoA)

  ?

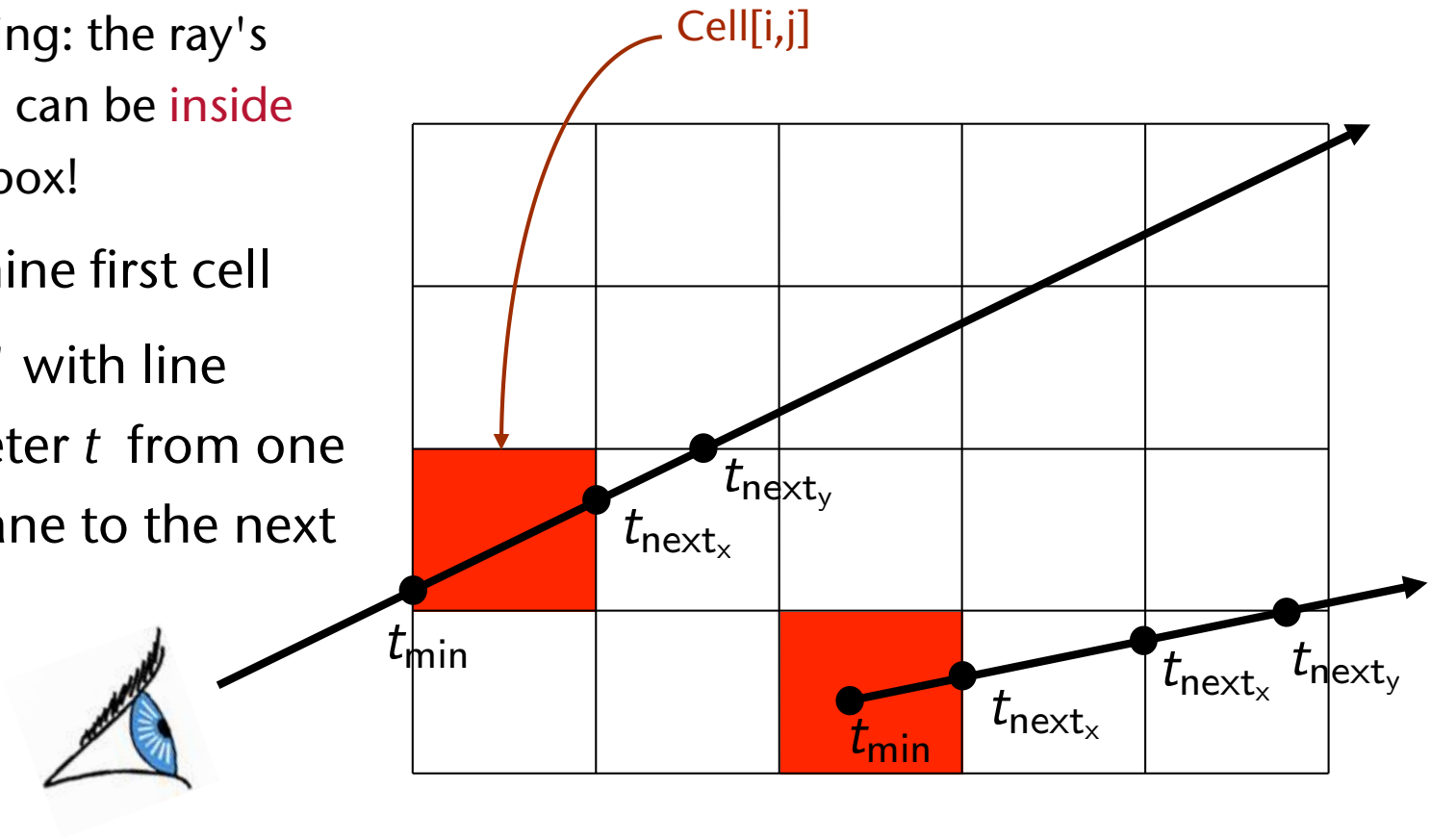
Traversal of a 3D Grid

- Simple idea: utilize 2 synchonized DDA's → 3D-DDA
  - Just like in 2D, there is a "driving axis"
  - In 3D, there are now two "passive axes"

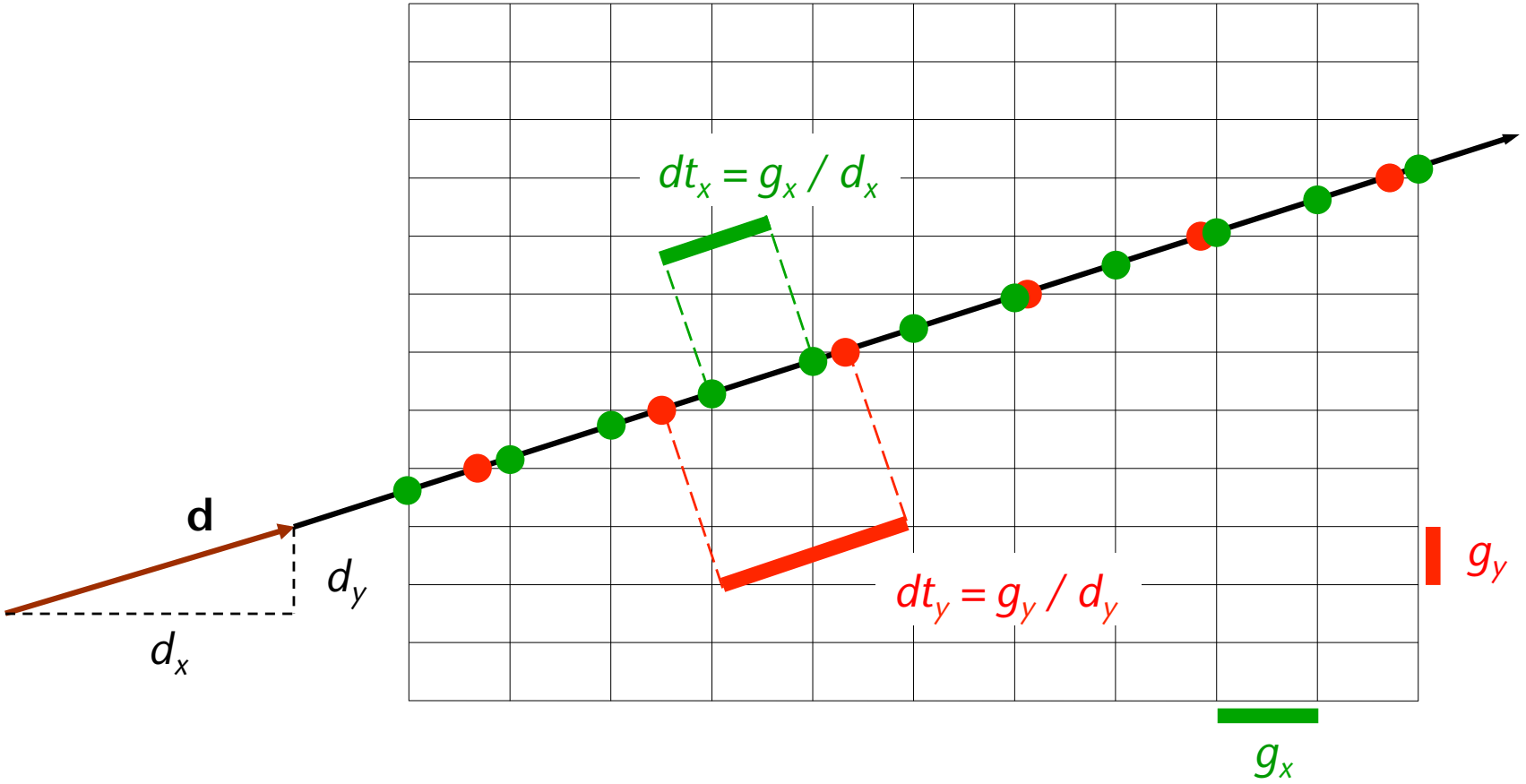


Better Grid Traversal Algorithm

- Intersect ray with Bbox of the whole scene
  - Warning: the ray's origin can be inside the Bbox!
- Determine first cell
- "Jump" with line parameter t from one grid plane to the next

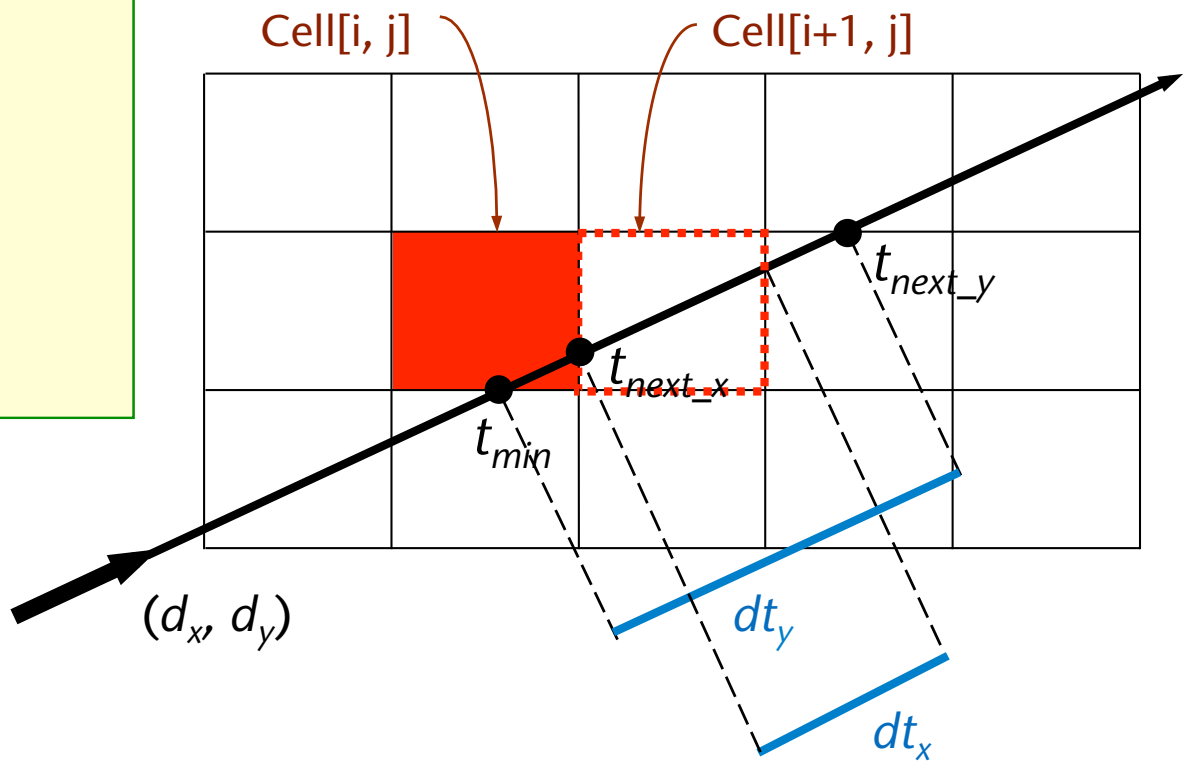


- Is there a pattern in the cell transitions?
- Yes, all horizontal and all vertical transitions have the same distance (among themselves)



The Algorithm

```
if t_next_x < t_next_y :
  i += s_x
  t_min = t_next_x
  t_next_x += dt_x
else:
  j += s_y
  t_min = t_next_y
  t_next_y += dt_y
```



Storage

- Lots of empty cells → represent grid by hash table