# Spatial Data Structures

## CS425: Computer Graphics I

Fabio Miranda
https://fmiranda.me



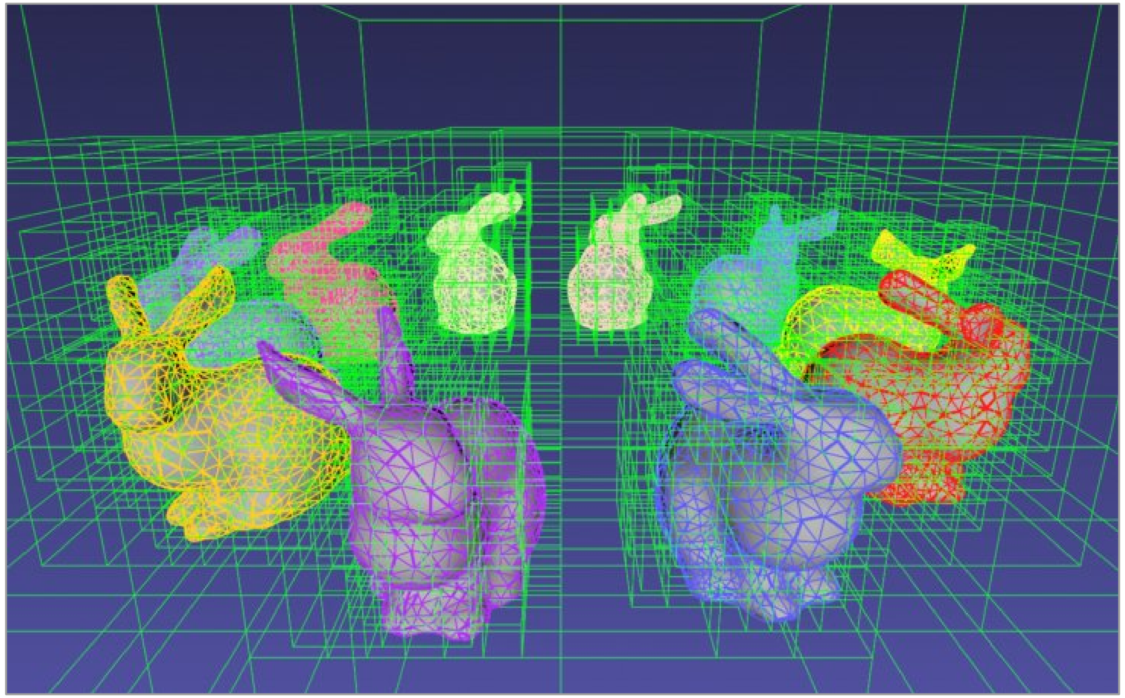
UIC COMPUTER SCIENCE

# Overview

- Spatial data structures
    - Uniform grid
    - Nested grids
    - Quadtree / octree
    - K-d tree
    - Bounding Volume Hierarchy
    - Efficient sparse voxel octrees
    - High resolution sparse voxel DAGs

# How to efficiently organize objects?

- 3D data contains spatial information.
- How to perform queries when there are thousands / millions of objects (points, polygons)?
  - Ray-scene intersection.
  - Proximity queries
  - Point in polygon.
  - Range query.

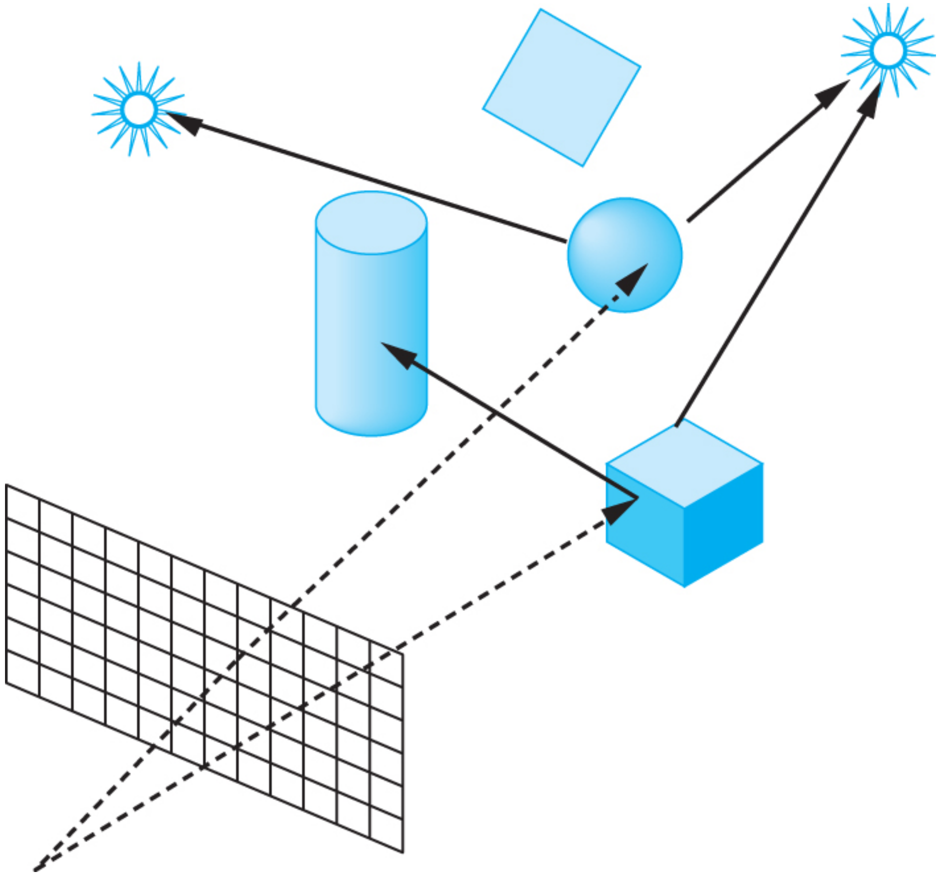


# Ray-scene intersection

- Given a scene with $n$ primitives and a ray $r$, find the closest point of intersection of $r$ with the scene.

```
function intersectObjects(ray, scene) {
    for(var i=0; i < scene.objects.length; i++) {
        var object = scene.objects[i];
        var dist = intersection(ray, object);
        // ...
    }
}
```

- Complexity: $O(n)$
- How to do better?





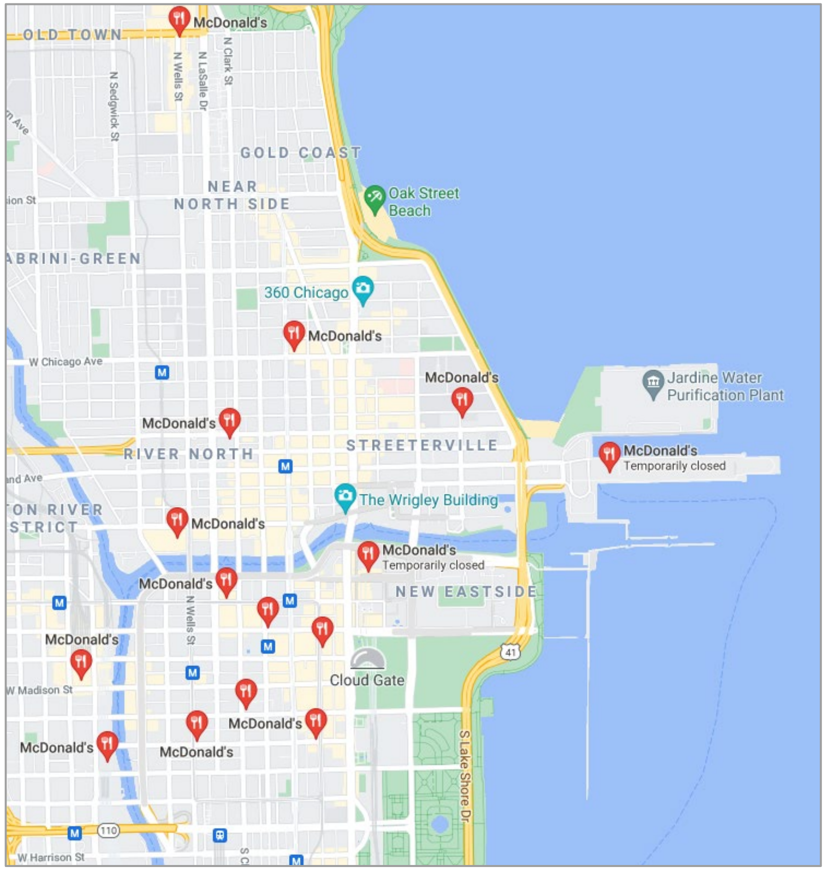
UIC COMPUTER SCIENCE

# Proximity query

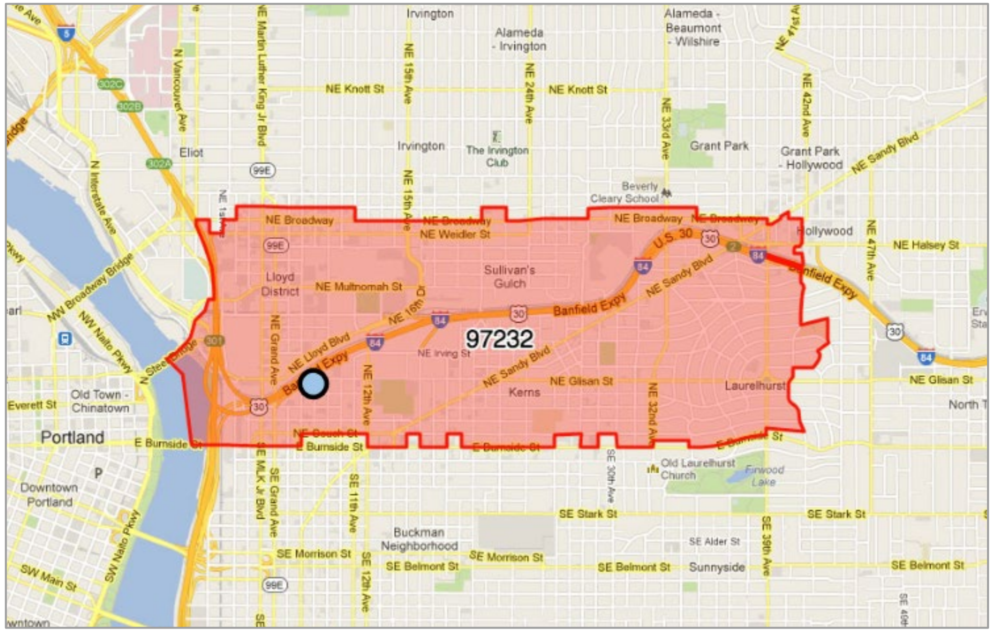
- Query based on proximity.
- “What is the closest McDonald’s?”

```
function findPlaces(query, scene) {
    for(var i=0; i < scene.places.length; i++) {
        var place = scene.places[i];
        var dist = satisfyQuery(query, place);
        // ...
    }
}
```

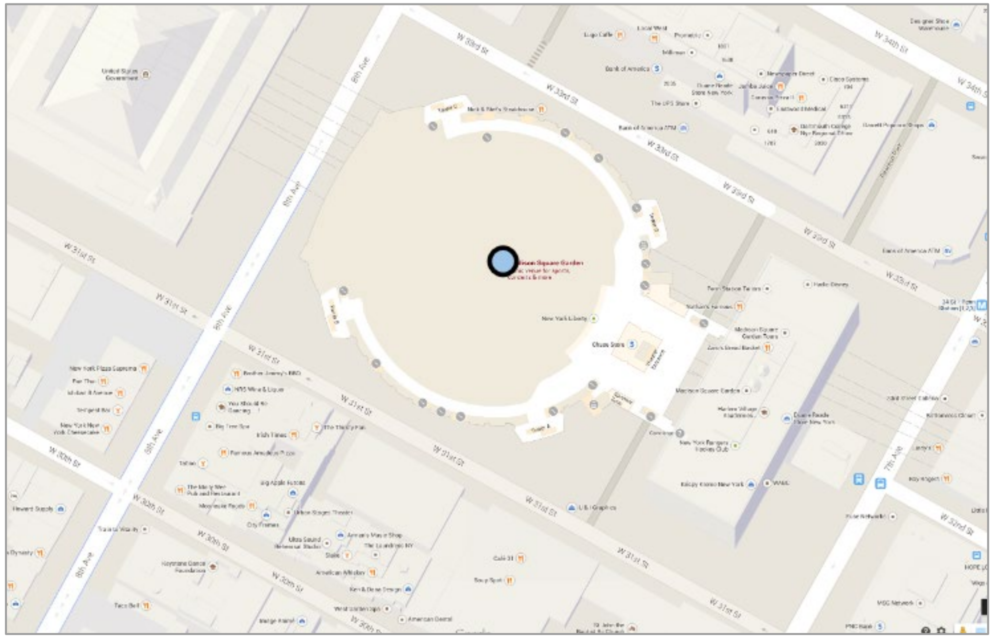
- Complexity: $O(n)$
- How to do better?



# Point in polygon




What is the zip code for this complaint?



Am I inside a specific building?



UIC COMPUTER SCIENCE

# Time complexity

- Ray-scene intersection: $O(n)$
- Proximity query: $O(n)$
- Point in polygon: $O(n)$

**How to reduce the time complexity?**

# Motivation

- Expensive operations (ray tracing, query).
  - Complex scenes (millions of objects).
  - Large number of operations (hundreds of millions per second).
- Reduce complexity through pre-processing data.
  - Spatial data structures: structures of objects in space.
  - Eliminate candidates as early as possible.
  - Reduce complexity to $O(log\ n)$ on average.
  - Worst case complexity still $O(n)$.
    - Can you come up with a worst case example?

# Spatial data structures

- Data structures to accelerate queries of the kind:

  **“I’m here, which object is around me?”**

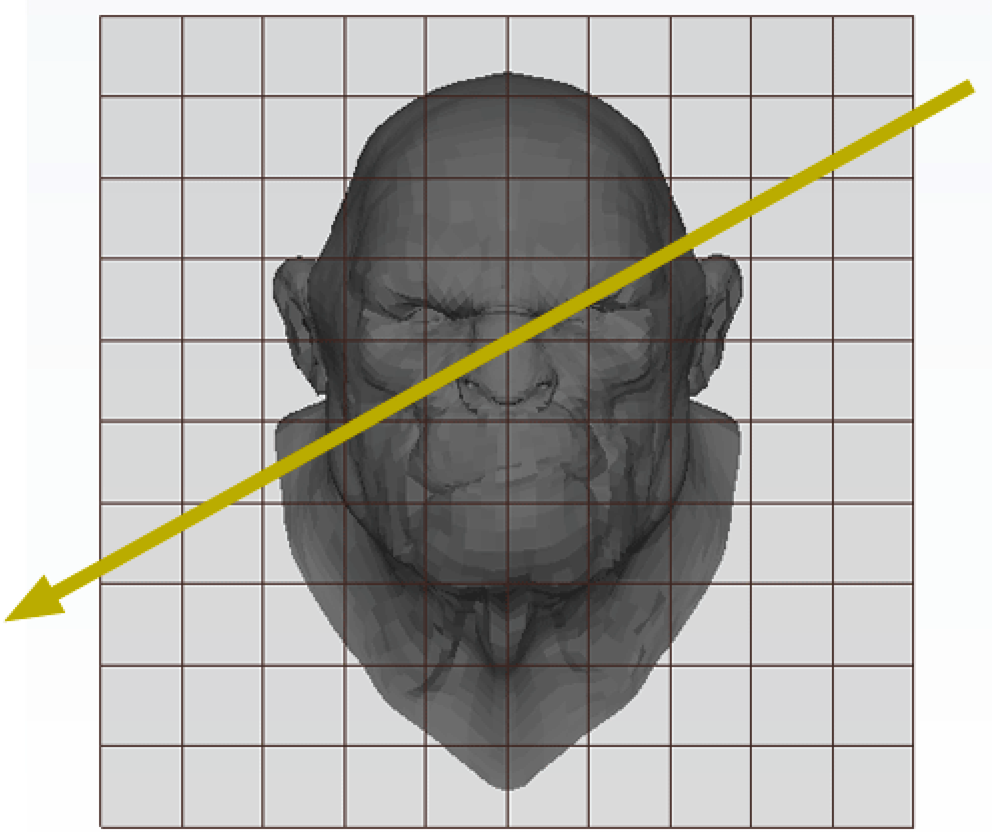
- Partition space or set of objects.
- Tasks:
  1. Construction / update:
     - Pre-processing for static parts of the scene.
     - Update for moving parts of the scene.
  2. Access:
     - Optimize so it is done as fast as possible.

# Spatial data structures

- Uniform grid: 2D/3D data, uniform distribution.
- Quadtree: 2D data, non-uniform distribution.
- Octree: 3D data, non-uniform distribution.
- KD-tree: 2D/3D data, avoid empty cells.

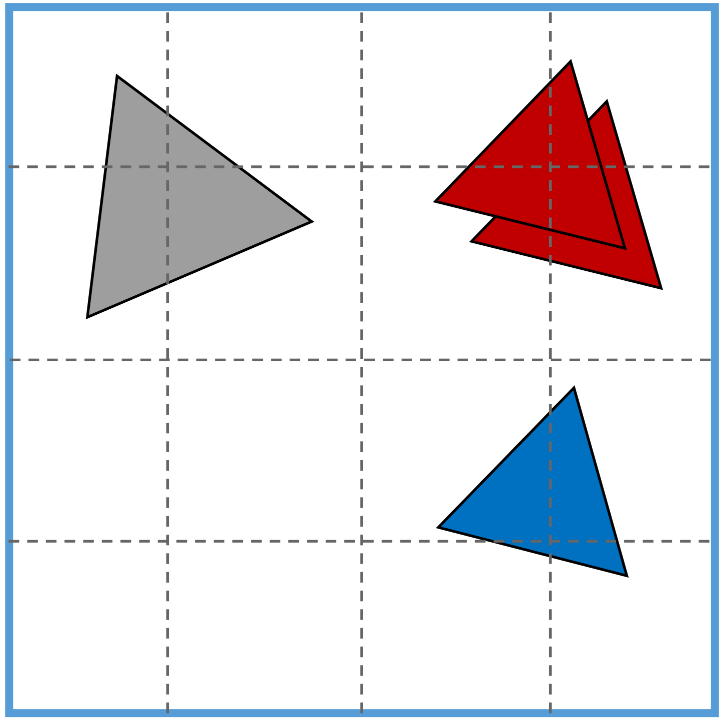
# Uniform grid

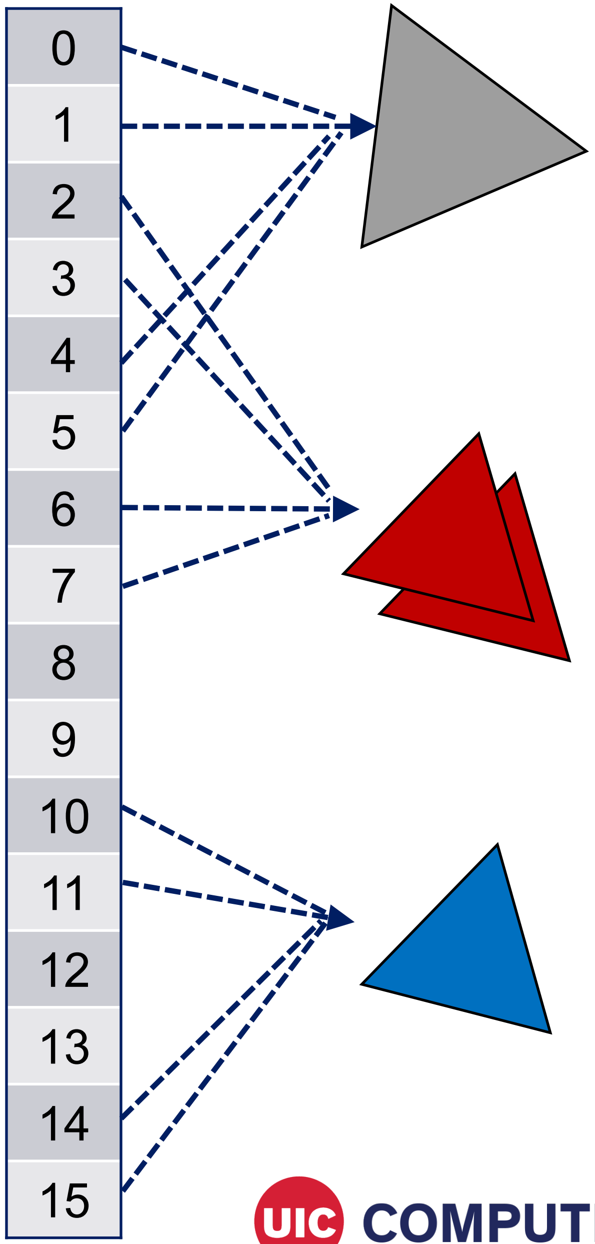
- Partition space into equal-sized volumes (i.e., voxels).
- Each cell will contain objects that overlap the voxel.
- Good for uniform data (points are evenly distributed in space).
- Fast construction and queries.



scratchpixel.com

# Uniform grid








UIC COMPUTER SCIENCE

# Uniform grid: construction and query

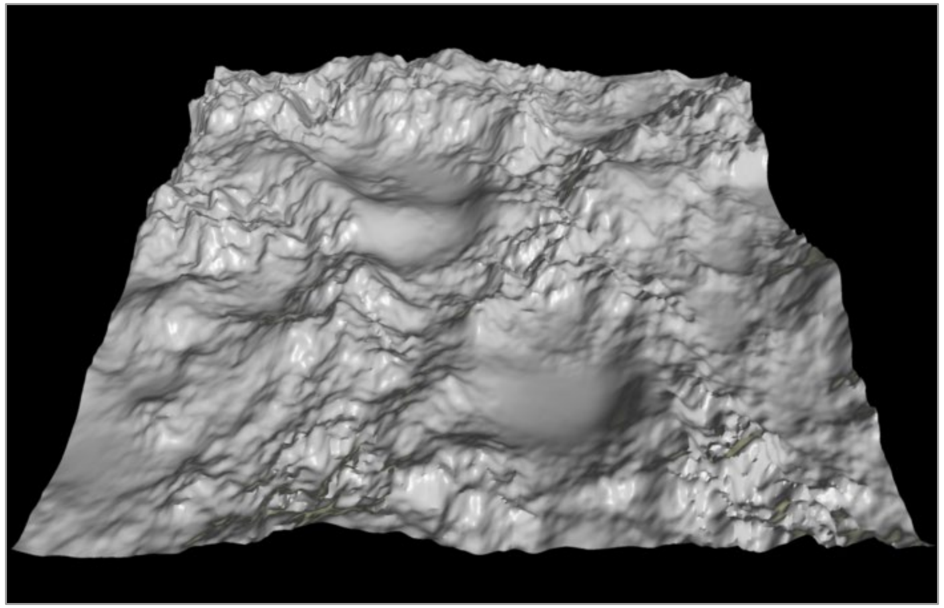
- Array of 3D voxels
  - Each voxel: list of pointers to colliding objects.
- Indexing function:
  - 3D point → cell index (constant time!)
- Construction:
  - Initialize cells for grid with size $w * h$
  - For each object $\mathrm{p}(x, y)$:
    - Compute grid cell using $(x, y)$.
    - Store $\mathrm{p}$ in cell.
- Query:
  - For query rectangle $(x_1, y_1) \times (x_2, y_2)$:
    - Compute subgrid for $(x_1, y_1)$ and $(x_2, y_2)$.
    - For all cells inside subgrid, report all objects.
    - For all cells on the border of the subgrid, test objects against rectangle.

# Uniform grid: complexity

- Build time: $O(n)$
- Space: $O(w * h) + O(n)$
- Query: $O(k)$

# Uniform grid: complexity

- When uniform grids work well? Uniform distribution of objects.

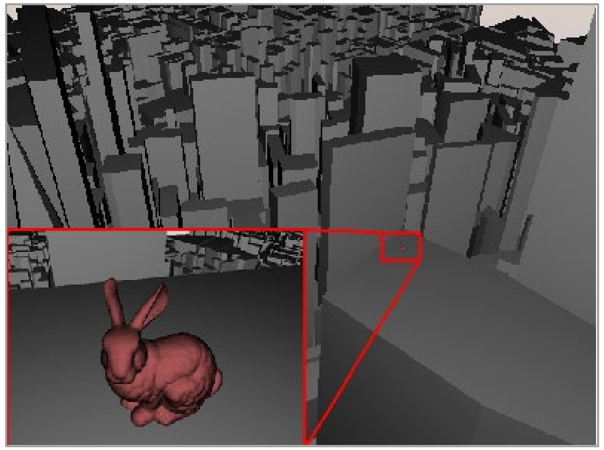


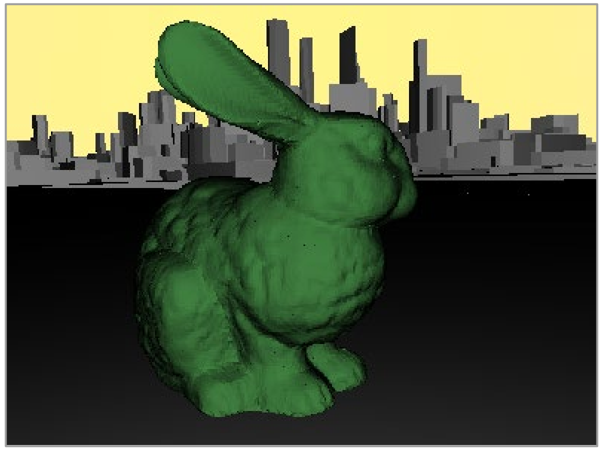
Mitsuba renderer

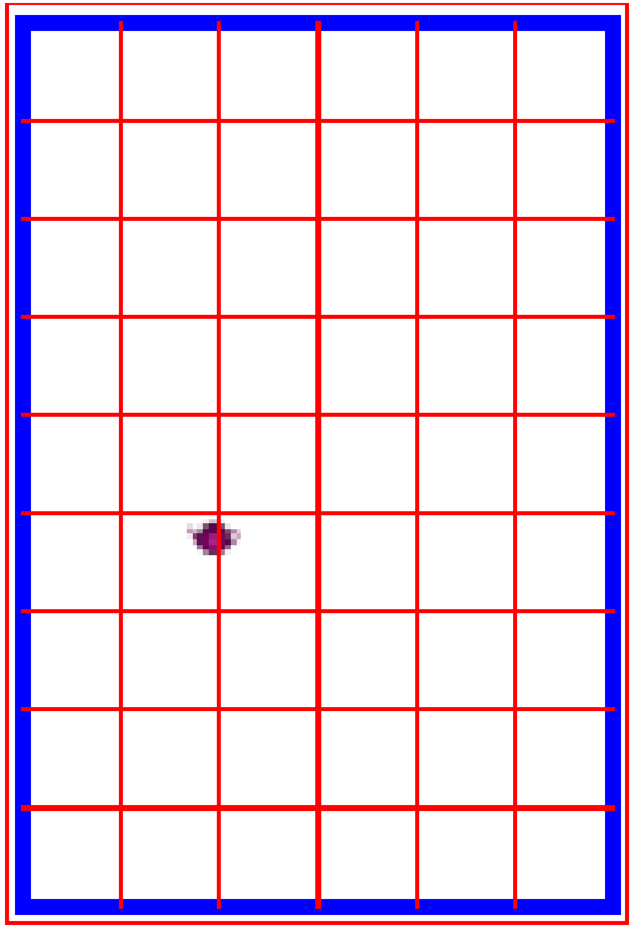
peterguthrie.net

# Uniform grid: drawbacks

- When uniform grids do not perform well? Non-uniform distribution of objects.
- “Teapot in a stadium” problem: uniform grids cannot adapt to local density of objects.





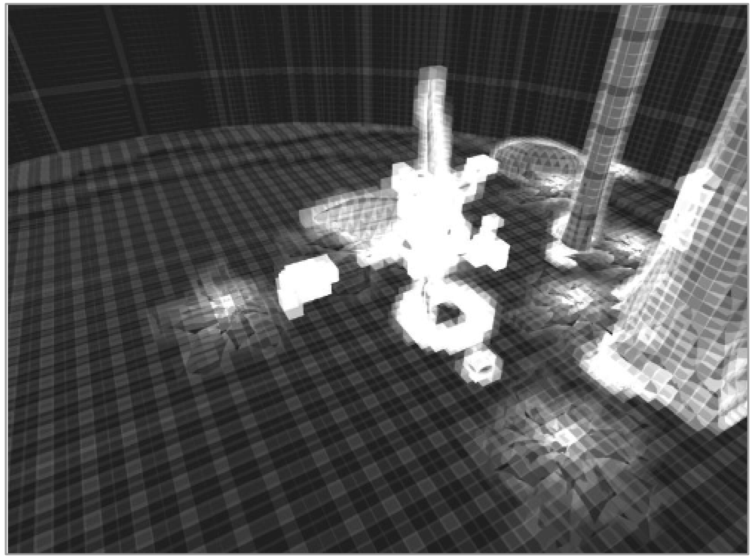


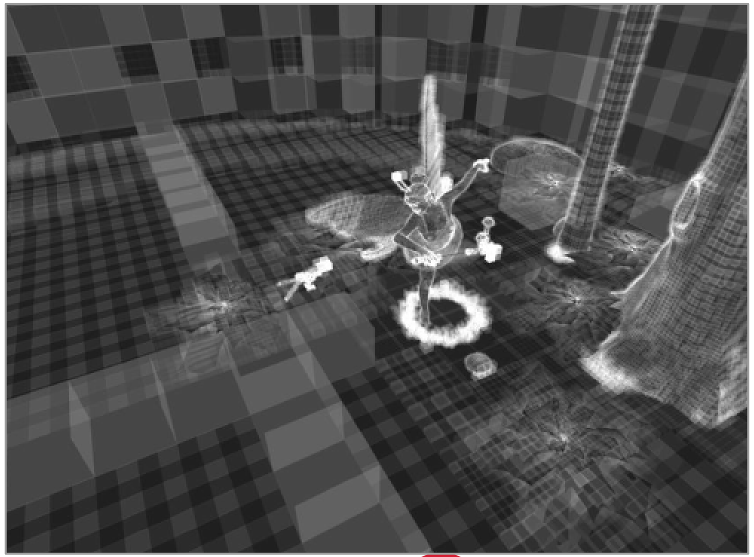
# Uniform grid: drawbacks

- Assumes objects uniformly distributed in space.
- What happens when assumption does not hold?
  - Many empty cells.
  - Few cells with too many points.
- Change cell size?
  - Too small: memory occupancy too large.
  - Too big: too many objects in one cell.

# Nested grids

- Possible solution to “teapot in a stadium” problem.
- Hierarchy of uniform grids: each cell is itself a grid.
- Fast building & traversal.

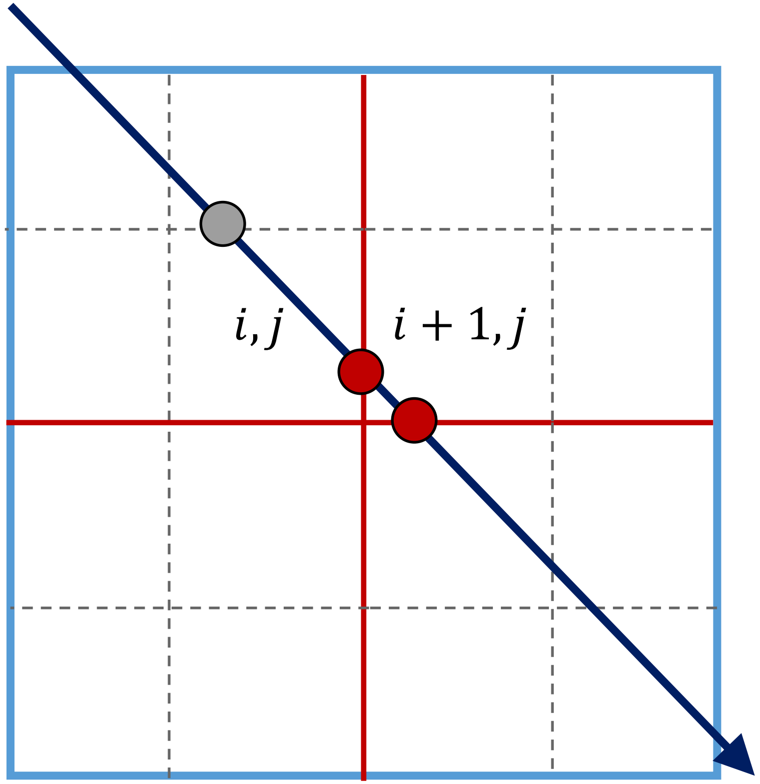




Philipp Slusallek

# Incremental traversal

- How to traverse the voxels?
- Incremental traversal similar to line rasterization:
  - From current intersection $P$ on cell $(i, j)$, perform ray-plane intersection tests with the next 2 grid planes along the ray direction.
  - Next intersection point is the nearest one among the 2 candidates.
  - Update cell index according to new intersection point.
  - Repeat process until the ray intersects with a primitive or reaches the boundary of the grid.

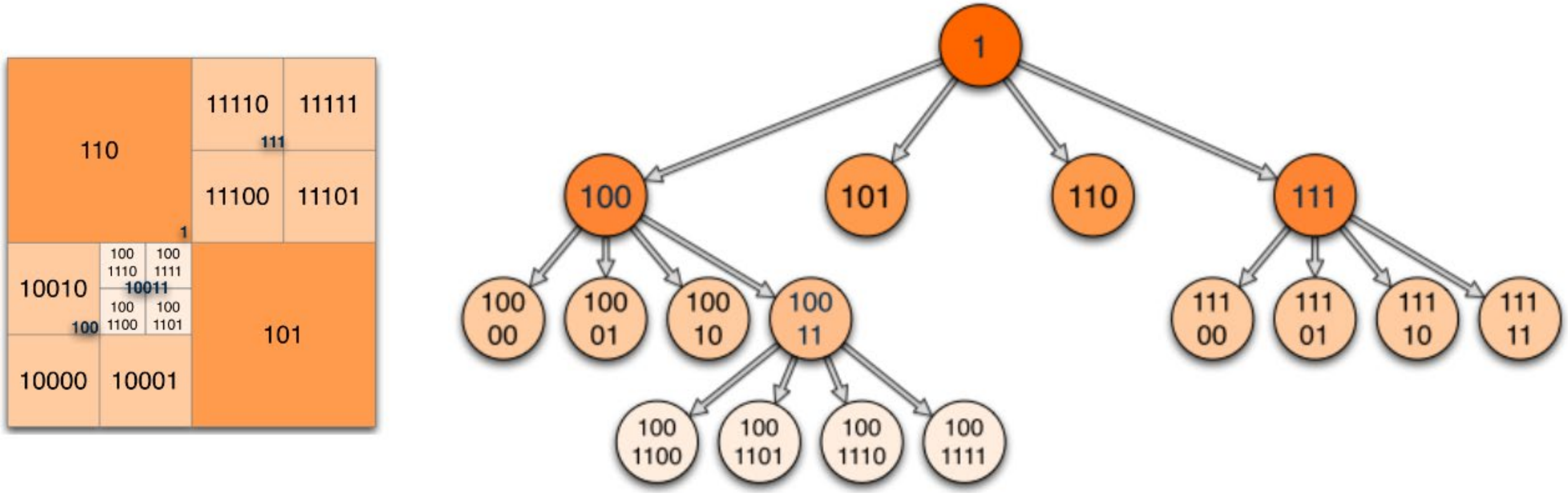





COMPUTER SCIENCE

# Quadtree

- Hierarchical structure that stores regular grids at each level.
- Adaptive subdivision: adjust depth to local scene complexity.

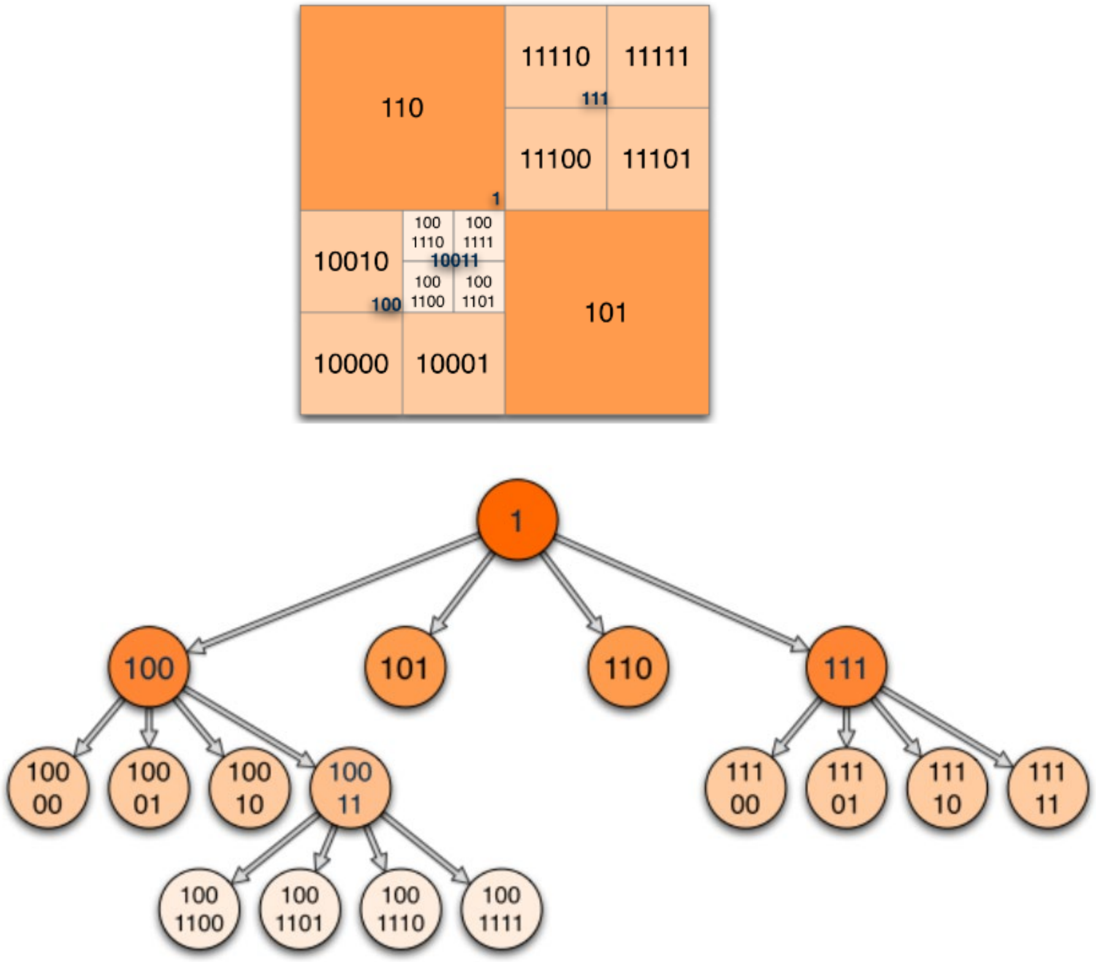





UIC COMPUTER SCIENCE

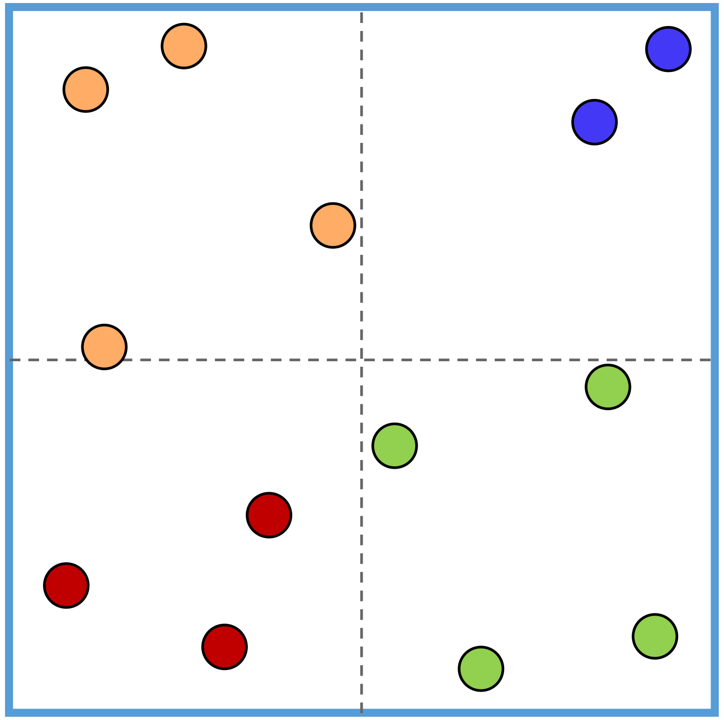
# Quadtree

- Rooted tree in which every internal node has four children.
- Every node corresponds to a square.
- Tree: branching factor 4 or 8.
- Each node: splits into all dimensions at once (in the middle).
- Construction: continue splitting until end nodes have few objects (or limit level reached).



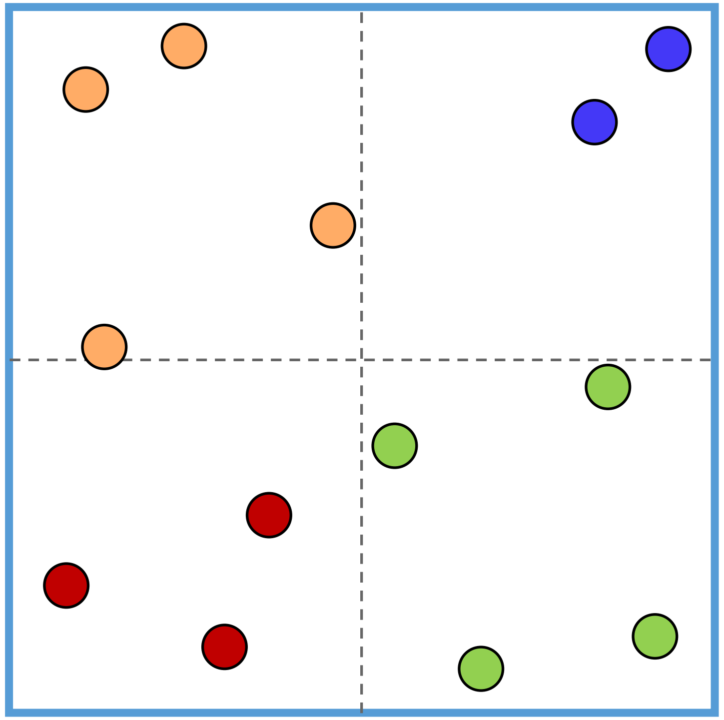
# Quadtree: construction

- Split the top level.



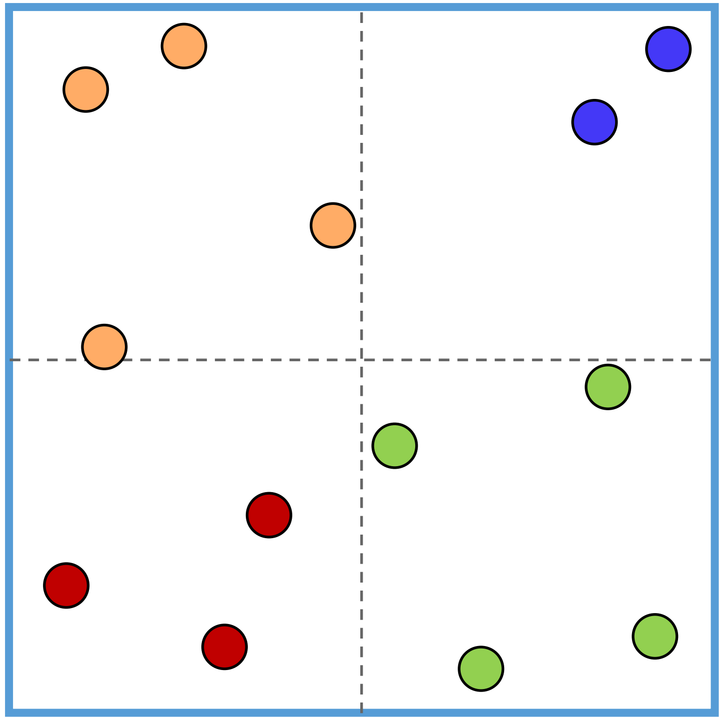
# Quadtree: construction

- Split the top level.
- Can we stop?



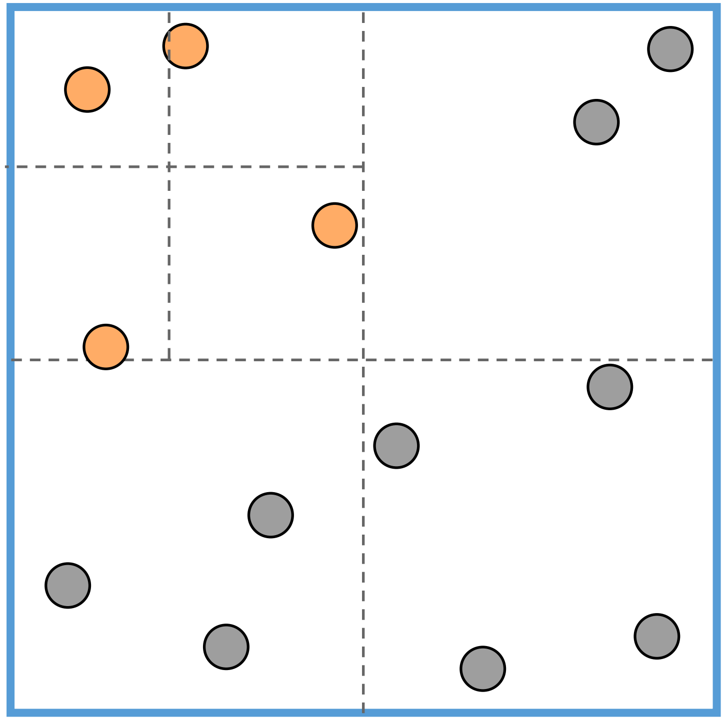
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.



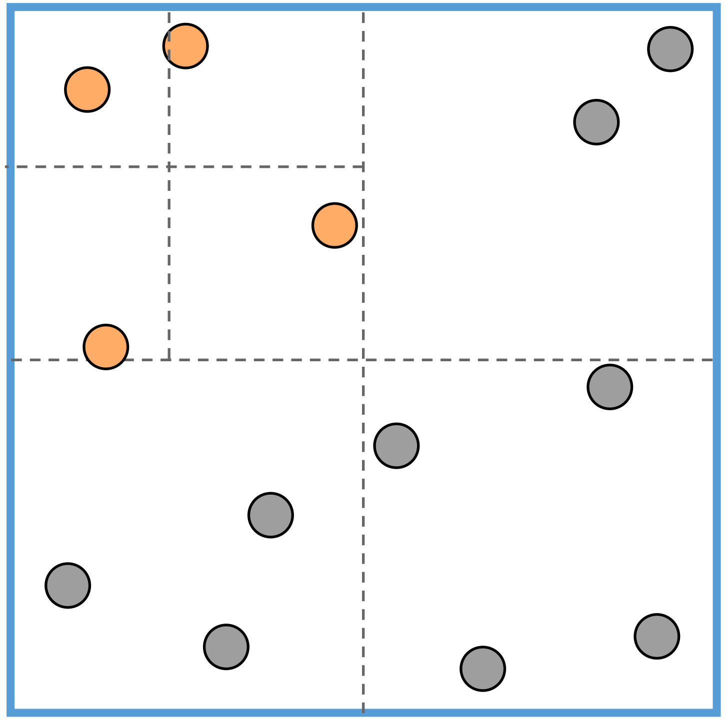
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.



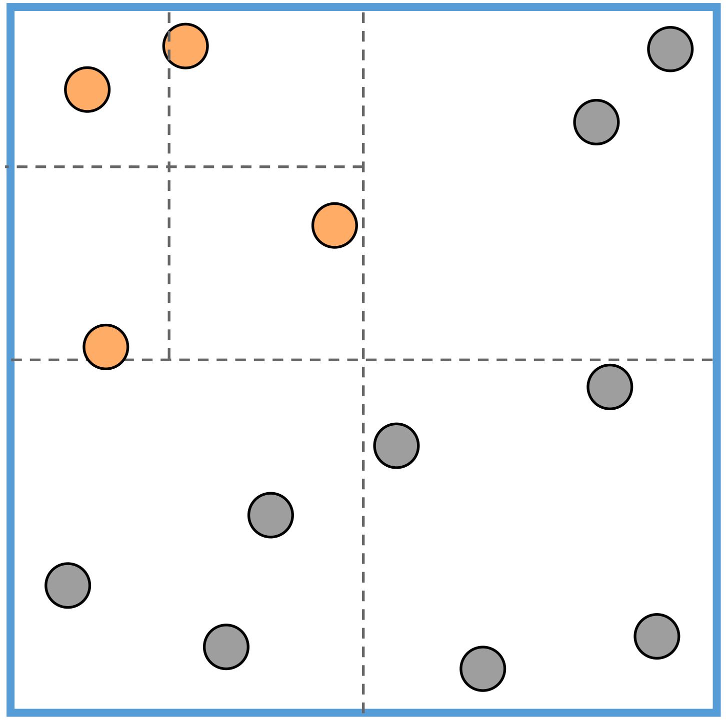
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left?



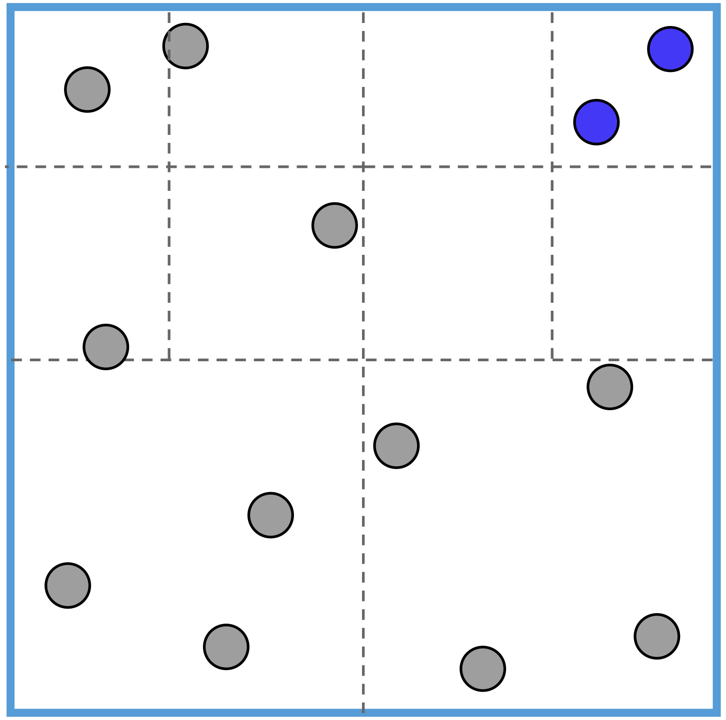
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.



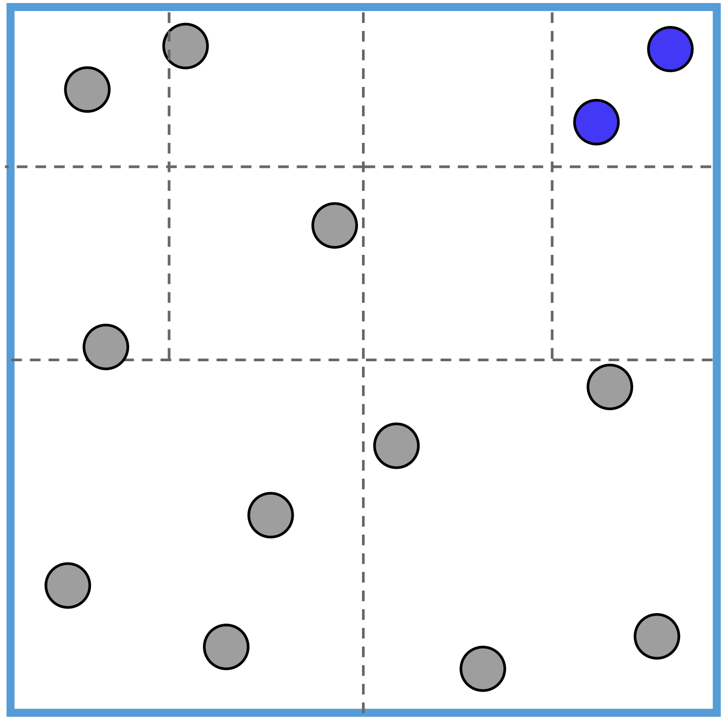
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.
- Split top-right.



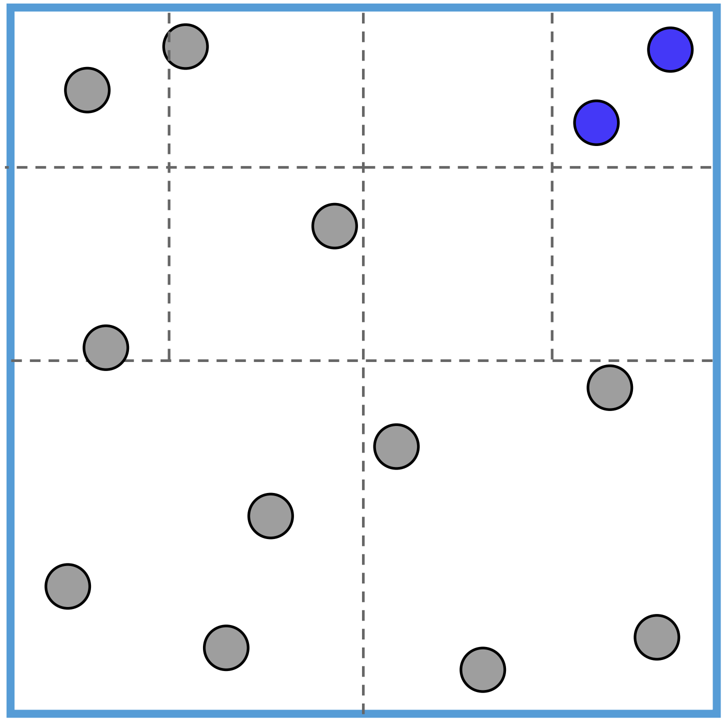
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.
- Split top-right.
- Can we stop top-right?



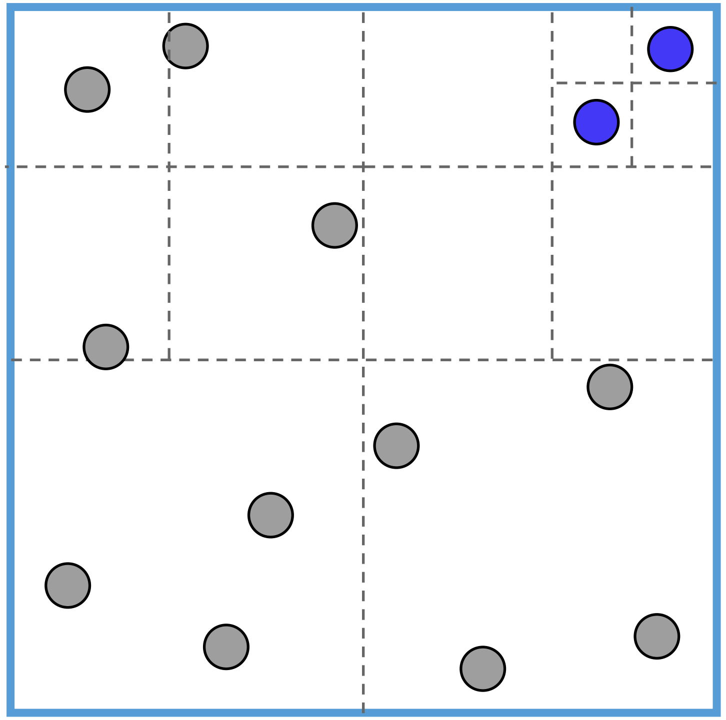
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.
- Split top-right.
- Can we stop top-right? No.



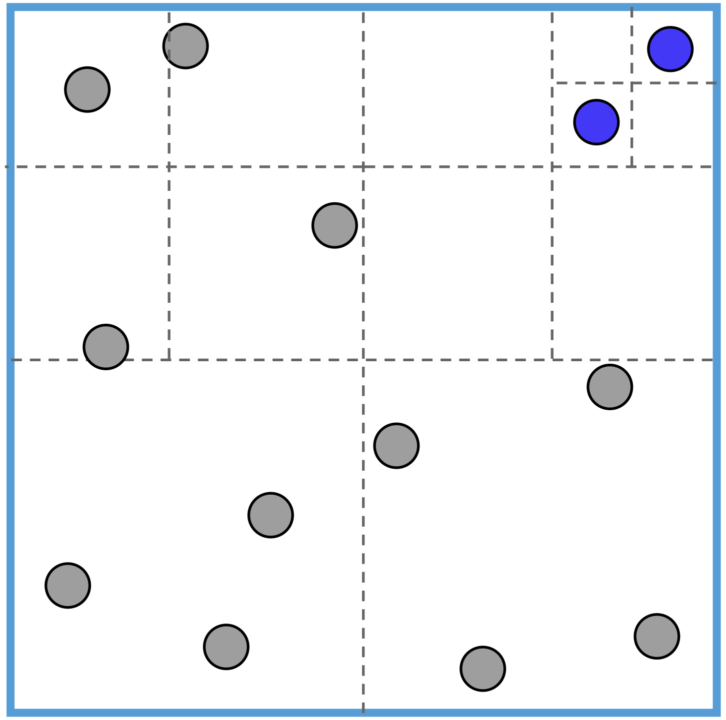
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.
- Split top-right.
- Can we stop top-right? No.
- Split top-right.



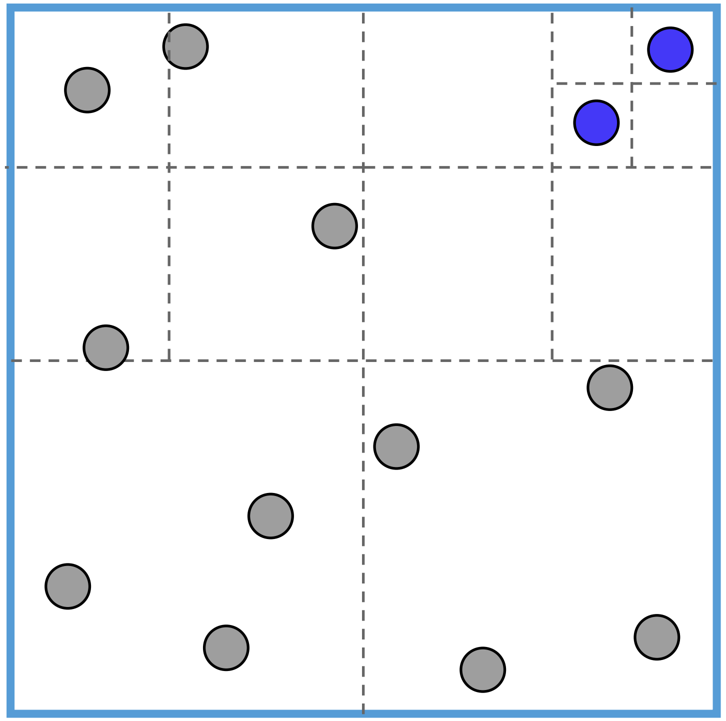
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.
- Split top-right.
- Can we stop top-right? No.
- Split top-right.
- Can we stop top-right?



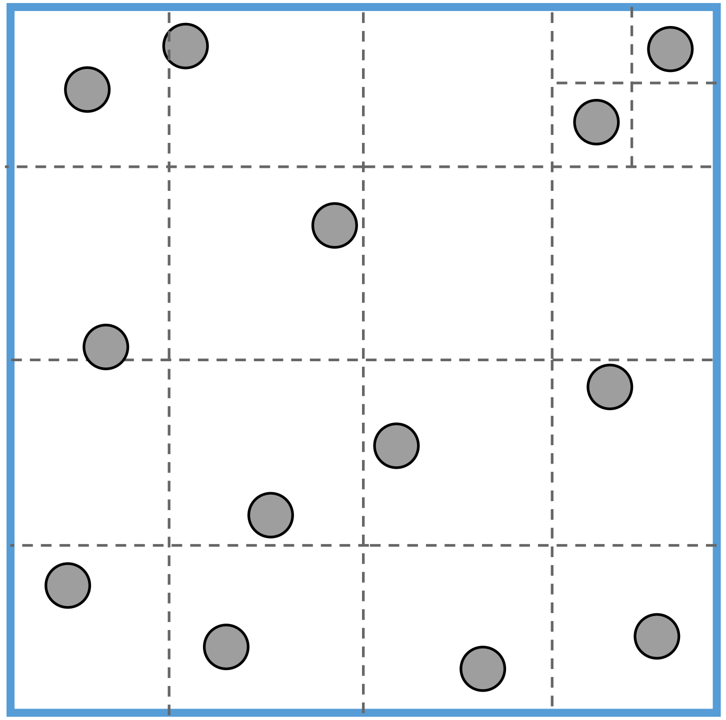
# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.
- Split top-right.
- Can we stop top-right? No.
- Split top-right.
- Can we stop top-right? Yes.

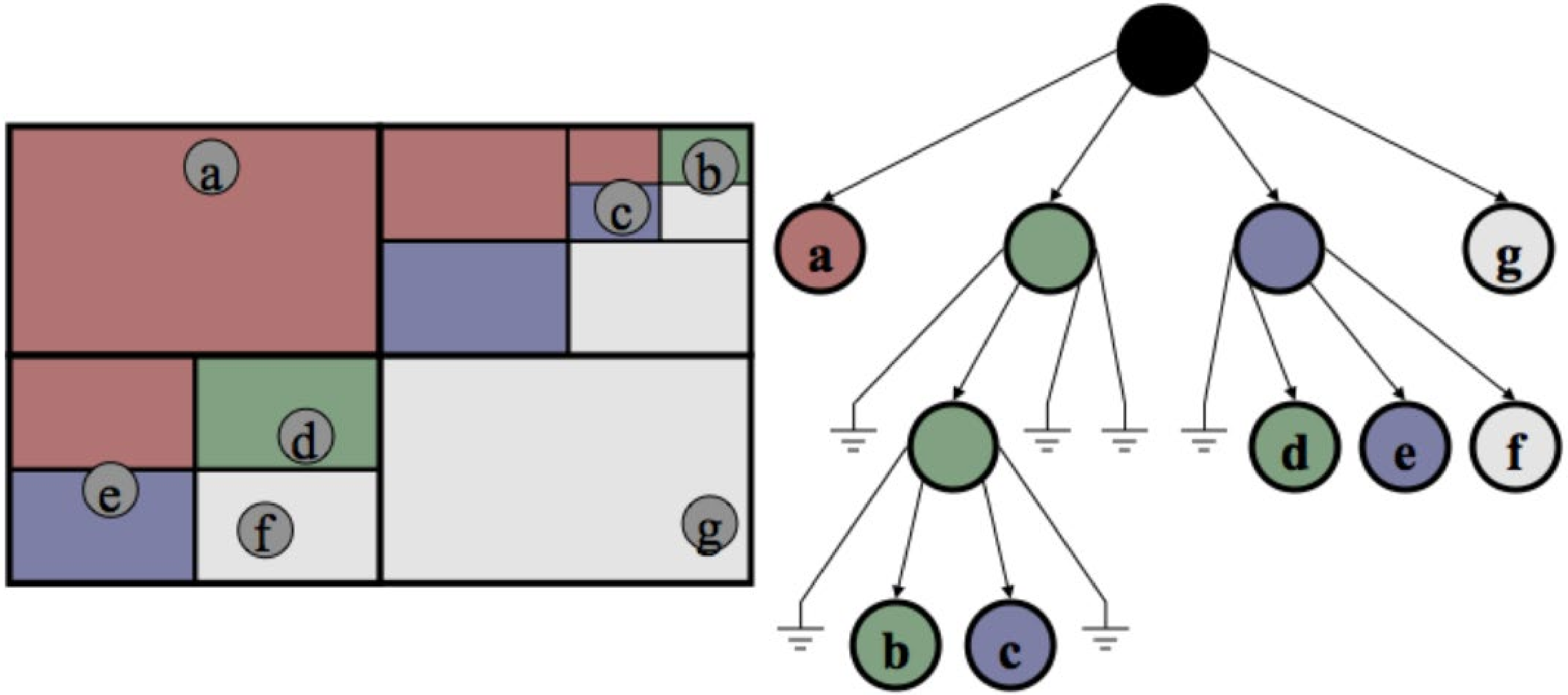


# Quadtree: construction

- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.
- Split top-right.
- Can we stop top-right? No.
- Split top-right.
- Can we stop top-right? Yes.



# Quadtree: construction



# Quadtree: construction

- Construction:
  - Input: set of objects $P$ inside a square $S$ $(x_1, y_1) \times (x_2, y_2)$, tree node $v$
  - If $|P| \leq 1$:
    - Quadtree consists of a single leaf with $P$.
  - Else:
    - $P_{00}$: set of points that fall in the bottom-left corner of $S$.
    - $P_{01}$: set of points that fall in the bottom-right corner of $S$.
    - …
    - $v_{00}$: node with points of $P_{00}$.
    - $v_{01}$: node with points of $P_{01}$.
    - …
    - Append $v_{00}, v_{01}, v_{10}, v_{11}$ to $v$.

# Quadtree: query

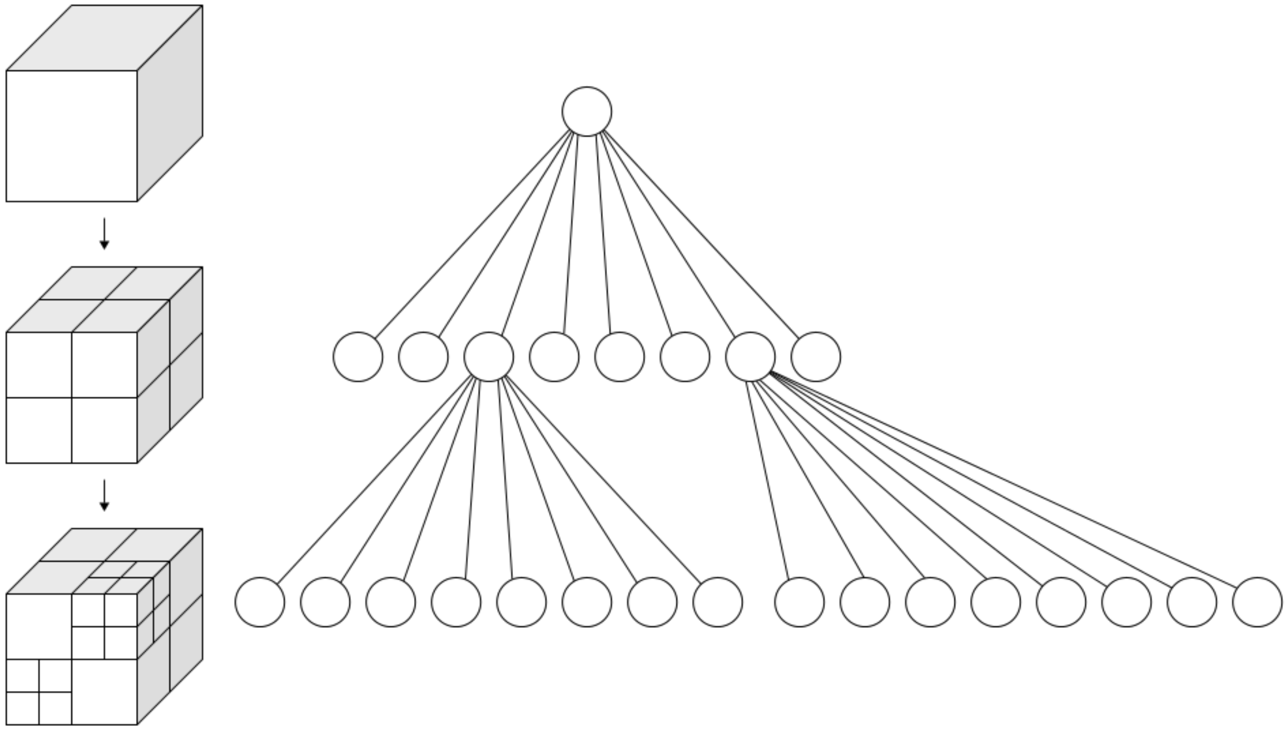
- Query:
  - Input: range query $r$ $(x_1, y_1) \times (x_2, y_2)$, tree node $v$.
  - If $v$ is a leaf:
    - Search points of $v$ inside range $r$.
  - If $v_{00}$ inside range $r$ :
    - Query($v_{00}$, $r$)
  - If $v_{01}$ inside range $r$ :
    - Query($v_{01}$, $r$)
  - If $v_{10}$ inside range $r$ :
    - Query($v_{10}$, $r$)
  - If $v_{11}$ inside range $r$ :
    - Query($v_{11}$, $r$)

# Quadtree: complexity

- Build time: $O(n)$
- Space: $O(n)$
- Range query: $O(\sqrt{n} + k)$
- Leaf traversal: $O(logn)$

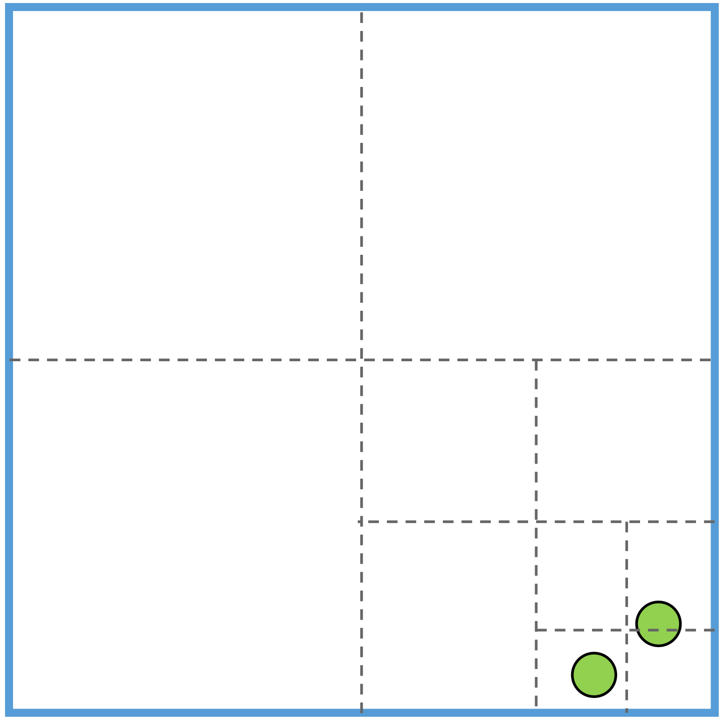
# Octree

- Each inner node contains 8 equally sized voxels.
- A 3D quadtree.



# Quadtree and octree: drawbacks

- Grater ability to adapt to location of scene geometry than uniform grid.
- But very long tree to store points that are concentrated in a small region.
- Many nodes will contain zero objects.

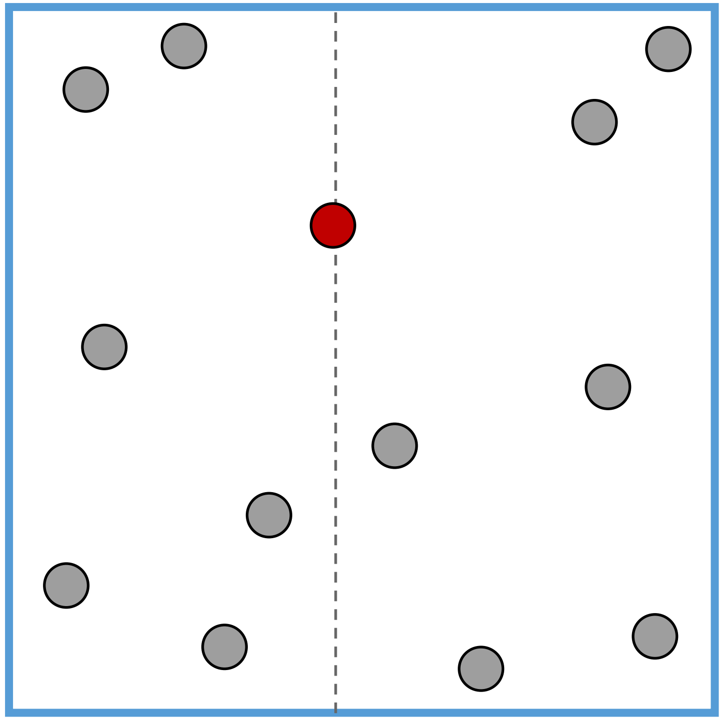


# K-d tree

- Differently from quadtrees and octrees, k-d trees only split **one** dimension at each level.
- Where to split? Middle? Median? Proportional to surface area?
- At each level:
  - Quadtree creates 4 equal sized cells.
  - Octree creates 8 equal sized cells.
  - K-d tree creates 2 non-equal sized cells (2D case).

# K-d tree: construction

- First split: x dimension (median point).

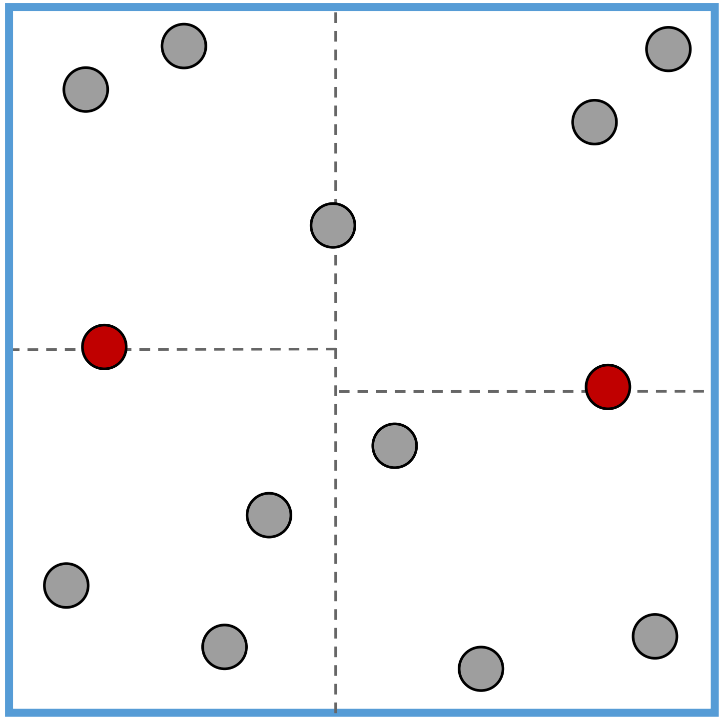




UIC COMPUTER SCIENCE

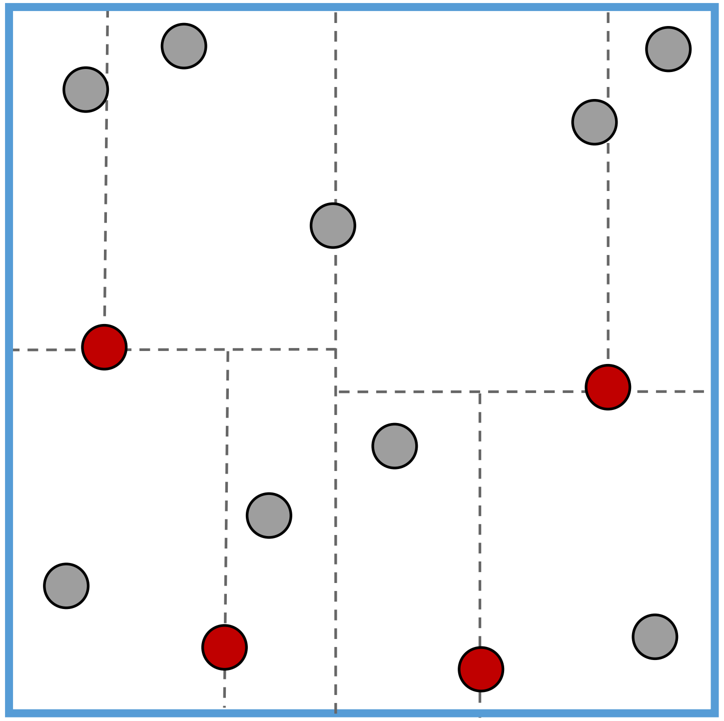
# K-d tree: construction

- First split: x dimension (median point).
- Second split: y dimension.



# K-d tree: construction

- First split: x dimension (median point).
- Second split: y dimension.
- Repeat, alternating split dimensions



# K-d tree: construction

- Construction:
  - Input: set of objects $P$ inside a square $S$ $(x_1, y_1) \times (x_2, y_2)$, tree node $v$
  - If $|P| \leq 1$:
    - K-d tree consists of a single leaf with $P$.
  - Else:
    - If depth is even:
      - Split $P$ into $P_0$ and $P_1$, along a vertical line through the $y$ axis.
    - Else:
      - Split $P$ into $P_0$ and $P_1$, along a vertical line through the $\mathrm{x}$ axis.
    - $v_0$: $build(v, P_0, depth + 1)$.
    - $v_1$: $build(v, P_1, depth + 1)$.
    - …
    - Append $v_0$, $v_1$ to $v$.

# K-d tree: query

- Query:
  - Input: range query $r$ $(x_1, y_1) \times (x_2, y_2)$, tree node $v$.
  - If $v$ is a leaf:
    - Search points of $v$ inside range $r$.
  - If $v_0$ inside range $r$ :
    - Query($v_0$, $r$)
  - If $v_1$ inside range $r$ :
    - Query($v_1$, $r$)

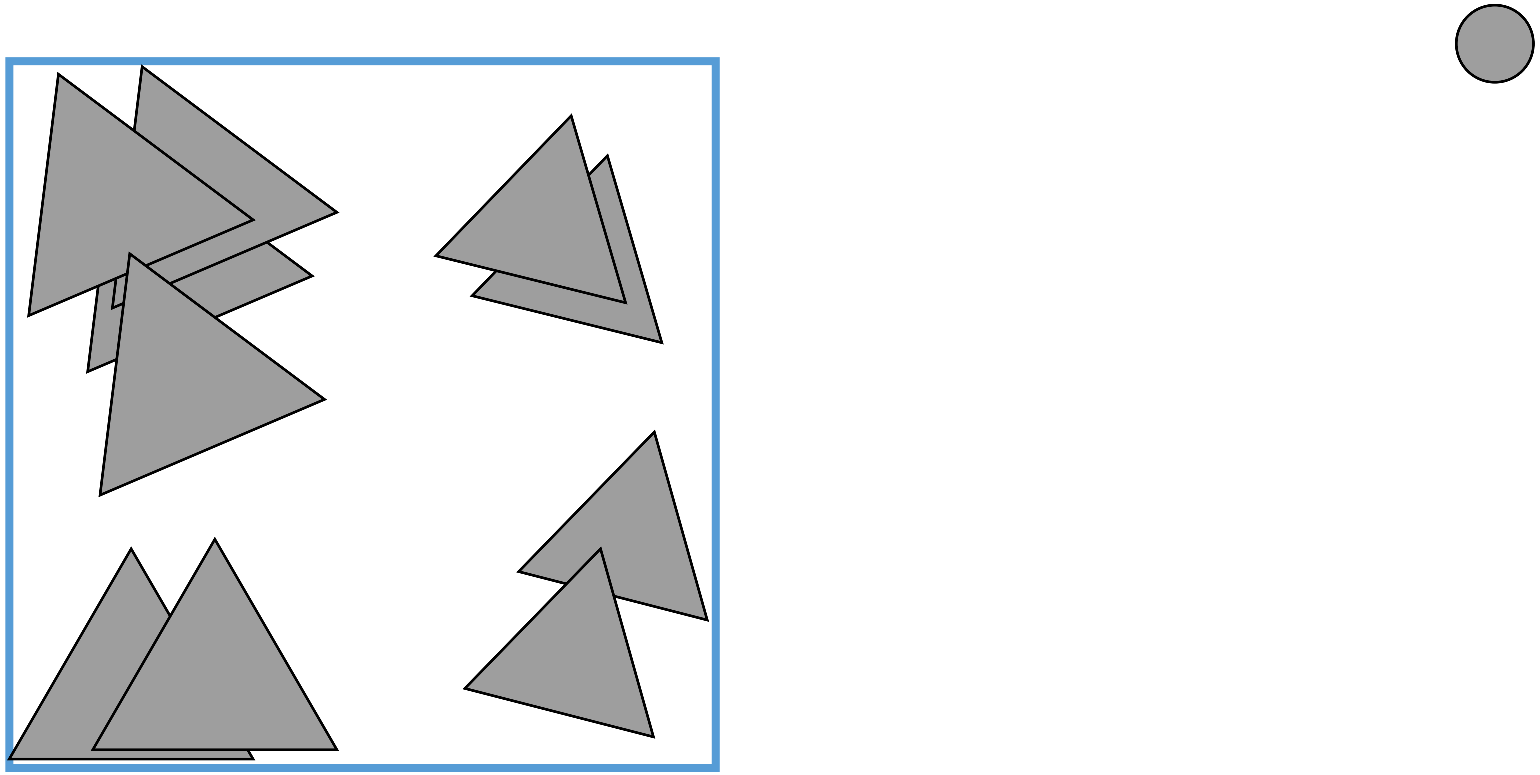
# K-d tree: complexity

- Build time: $O(nlogn)$
- Space: $O(n)$
- Range query: $O(\sqrt{n}+k)$
- Leaf traversal: $O(logn)$

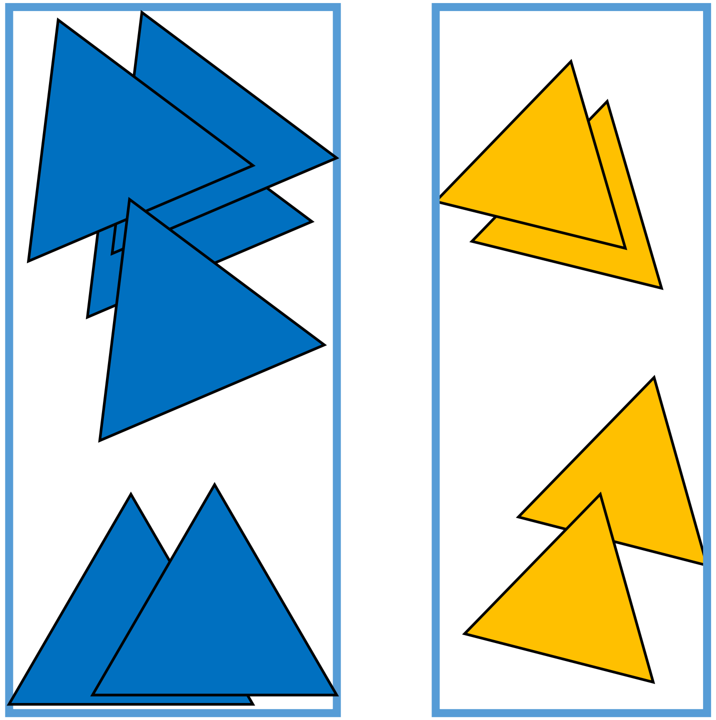
# Bounding volume hierarchies (BVH)

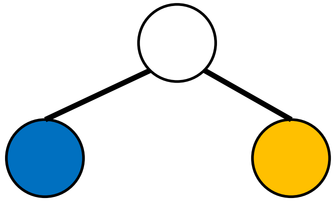
- Goal: use the scene hierarchy given by the scene graph (instead of a spatial derived one).
- Associate a bounding volume to each node.
  - A bounding volume of a node bounds **<u>all</u>** objects in the subtree.
  - Nodes are aggregated objects.
- Construction and update is fast.
  - Bottom-up: recursive.
- Querying it:
  - Top-down: visit.

# Bounding volume hierarchies (BVH)

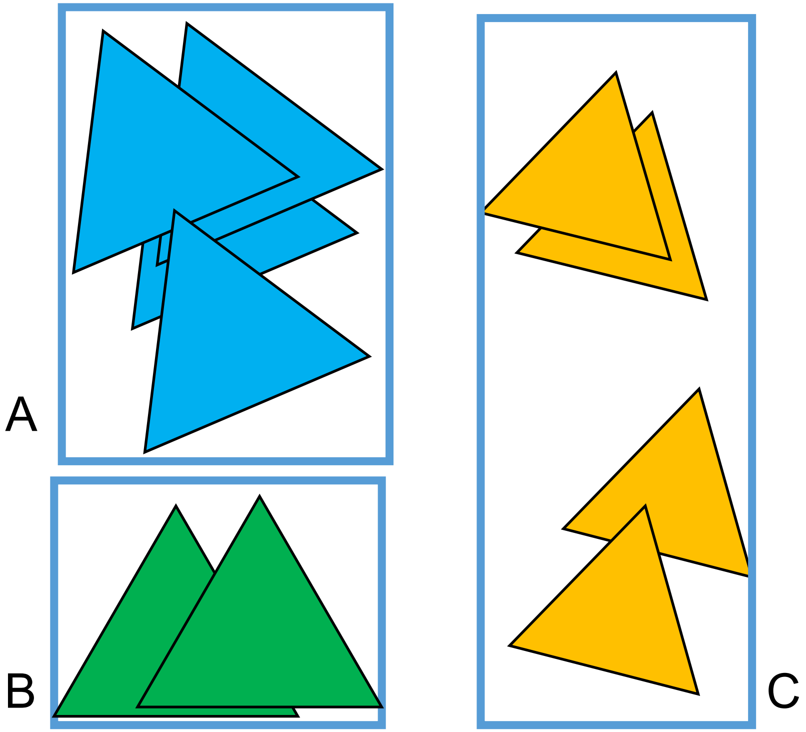


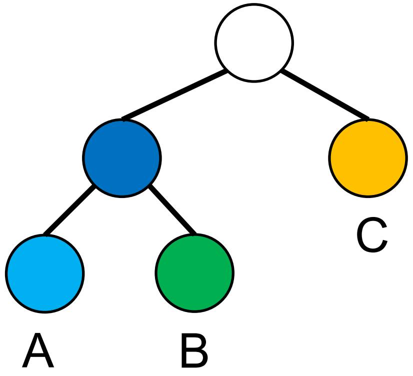
# Bounding volume hierarchies (BVH)





# Bounding volume hierarchies (BVH)




- Leaf nodes: contain *small* list of primitives.
- Interior nodes:
  - Proxy for a large subset of primitives.
  - Stores bounding box for all primitives in subtree.

# Bounding volume hierarchies (BVH)

- How to split the scene?
- Primitive list of current node $P$ is sorted based on the centroids of primitive bounding boxes. Ordered list is split into $P_0$ and $P_1$.
- Different heuristics:
  - Edge volume heuristic (EVH).
  - Split bounding volume hierarchy (SBVH).
  - Surface area heuristic (SAH):

$$C = C_t + \frac{SA(B_1)}{SA(B)}|P_1|C_i + \frac{SA(B_2)}{SA(B)}|P_2|C_i$$

# Bounding volume hierarchies (BVH)

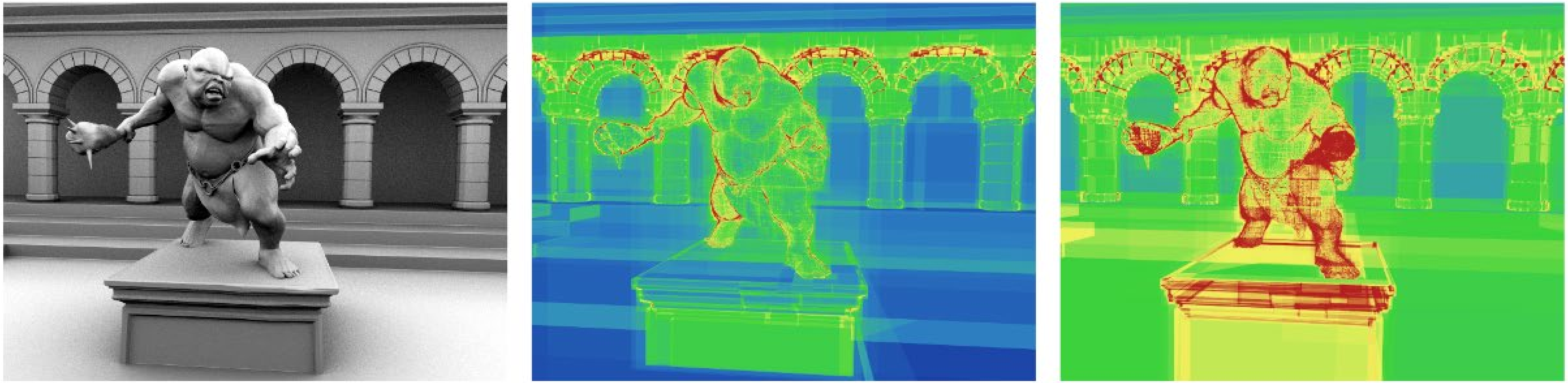


**Figure 1:** *Sample scene consisting of roughly 1.9 million triangles (left). Our method (middle) results in a significant reduction of ray shooting costs compared to a regular bounding volume hierarchy (right). The heat views visualize the summed number of traversal steps and primitive intersections for primary rays.*

[Stich et al., 2009]

# Object vs. space partitioning

- Object partitioning:
  - Hierarchically partition **objects** into groups.
  - Spatial index is created by spatially bounding each subgroup.
- Space partitioning:
  - Hierarchically partition **space** into subspaces.
  - Subspaces are non-overlapping and completely fill parent space.
  - Tree or table structure.

# Summary

- Choose the right structure considering the operations and data.
- Uniform grid:
  - The most parallelizable (to update, construct, use).
  - Constant time access (best!).
  - Quadratic / cubic space (2D, 3D).
  - Good performance under uniform distribution of objects.
- Quadtree, octree, k-d tree:
  - Compact.
  - Simple.
  - Non-constant accessing time.
  - Good performance under non-uniform distribution of objects.
- BVH:
  - Simple construction.
  - Ideal for dynamic parts of the scene.
  - Not necessarily very efficient to access.
    - May need to traverse multiple children.
    - If you do not have a scene-graph you need to create one.

# Voxel-based rendering

- Outcast, 1999.
- *“At this point Yves started wondering if dedicated hardware would ever catch-up the exponential acceleration of CPU power and flexibility”.*
- Flexibility to program effects not available on GPUs at the time:
  - Ripple effects
  - Real-time shadows
  - Curved surfaces

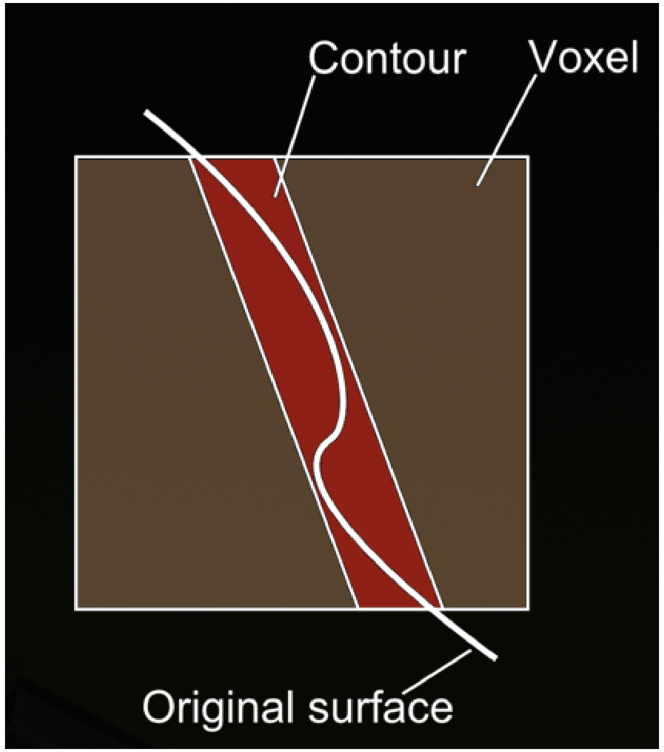
http://francksauer.com/index.php/games/test/15-games/published-games/47-outcast-pc

# Sparse voxel octrees

- Using voxel representations as a generic way for expressing complex and feature-rich geometry.
- Voxels as a representation of opaque surfaces.
  - Fine-grained resolution control
  - Compact representation
  - Easy to downsample
- Efficient sparse voxel octrees avoid building the octree with maximum depth by providing each voxel with a contour.
  - If contours are good approximation to the original geometry, subdivision is stopped.

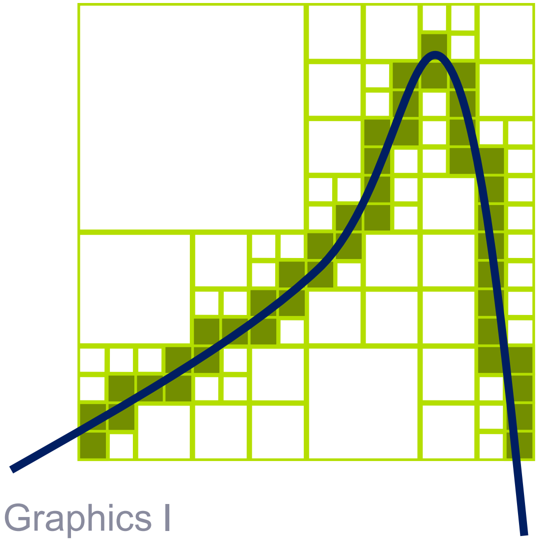
# Sparse voxel octrees: contours

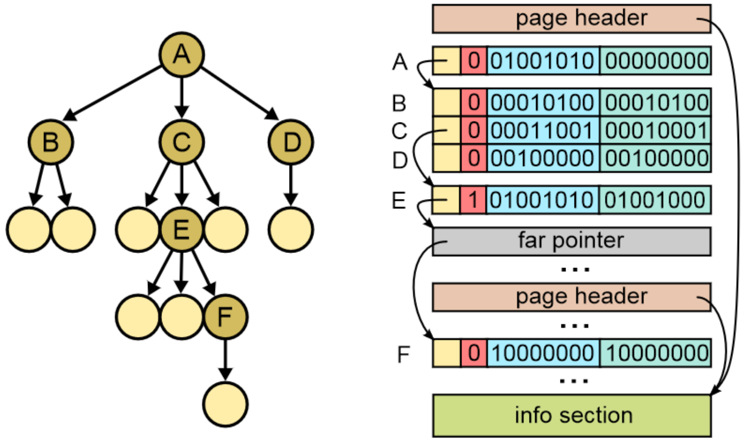
- Contours are slabs that modify voxel's shape.
- Improve accuracy for representing surfaces.
- Simple for rendering.



# Efficient sparse voxel octrees

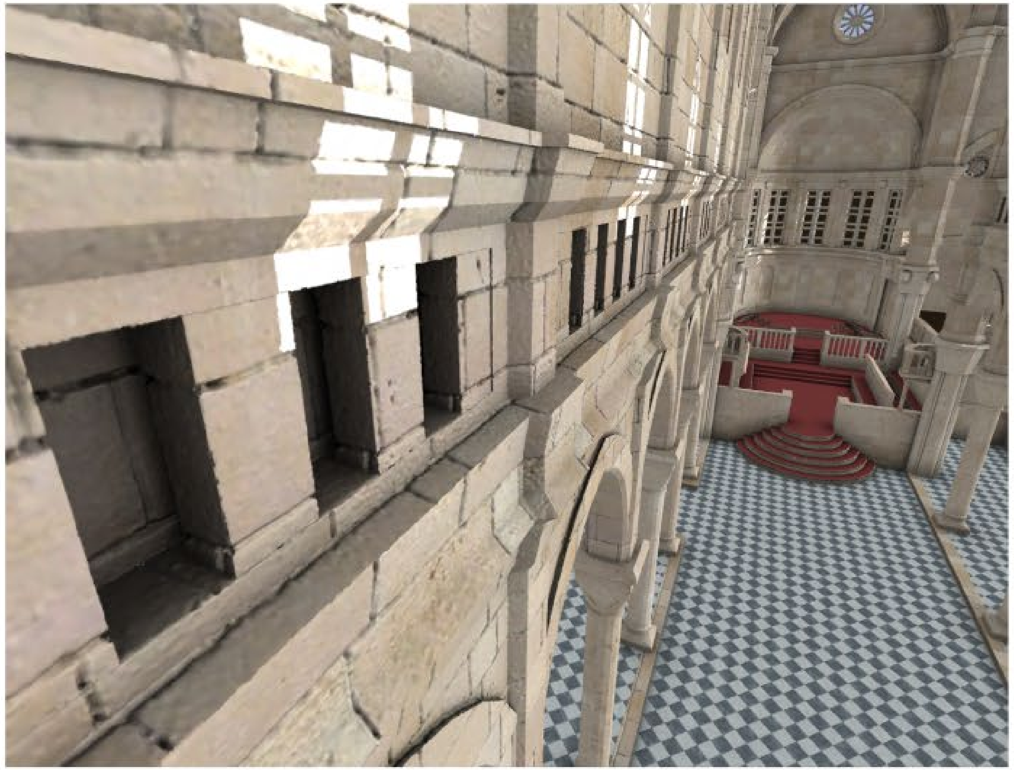
- How to efficiently store octrees?
  - One child pointer per voxel.
  - Efficient encoding of empty regions.
  - Culling away nodes that correspond to empty space.

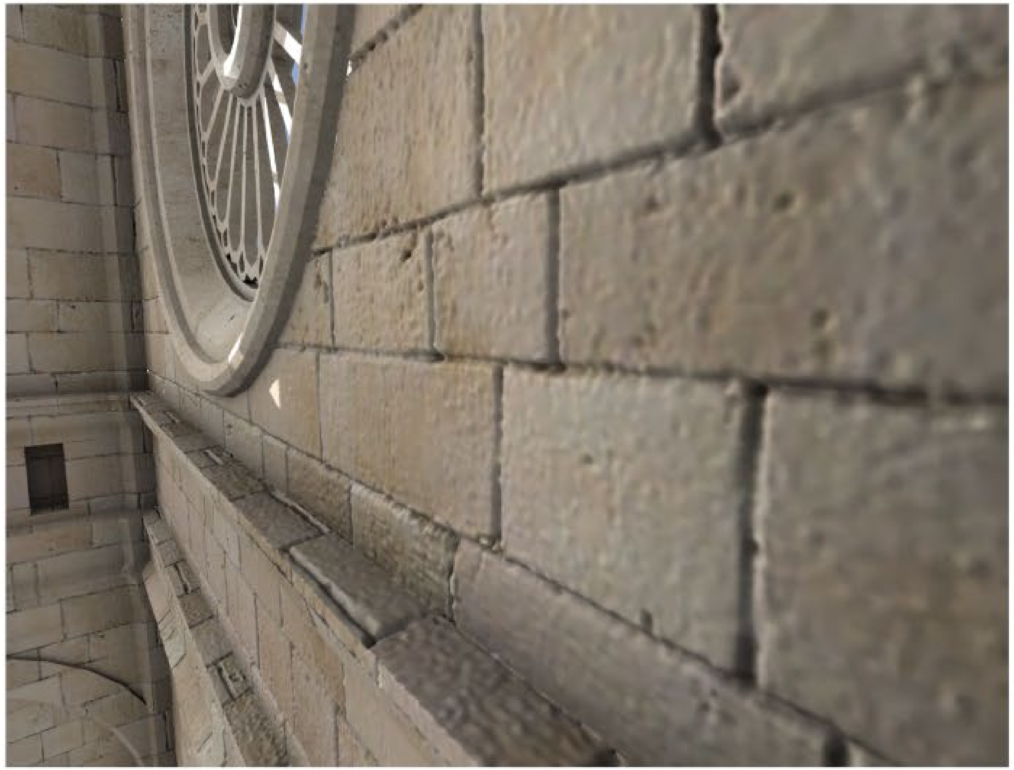





Laine and Karras, 2010

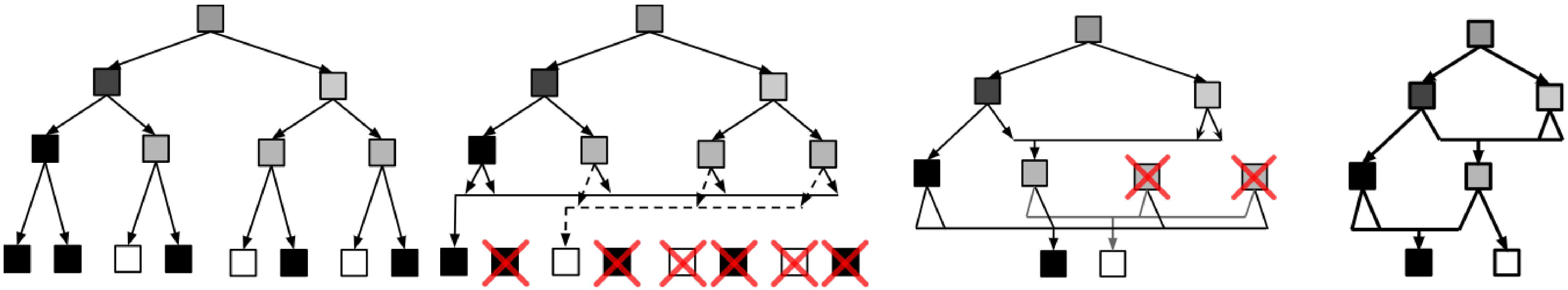
# Efficient sparse voxel octrees





# High resolution sparse voxel DAGs

- Binary voxel grid – generalization of octree to a directed acyclic graph.
- Assumption that SVO contains redundant subtress.
  - Searches the tree for common subtrees and only reference unique instances.
- DAG representation decreases memory consumption by up to 38x.





UIC COMPUTER SCIENCE

# High resolution sparse voxel DAGs

- SVO to DAG:
  - Top down approach:
    - Start at the root node and test whether its children correspond to identical subtrees, and proceed down recursively.
    - Expensive.
  - Bottom up approach:
    - Leaf nodes are uniquely defined by their childmasks (8-bit mask where bit $i$ tells us if child $i$ contains geometry). Then at most $2^8 = 256$ unique leaf nodes in an SVO.
    - First step: merge identical leaves.
    - Proceed to level above, identifying nodes that have identical childmasks and identical pointers (roots of identical subtrees that can be merged).
    - Iteratively perform previous step.

# High resolution sparse voxel DAGs

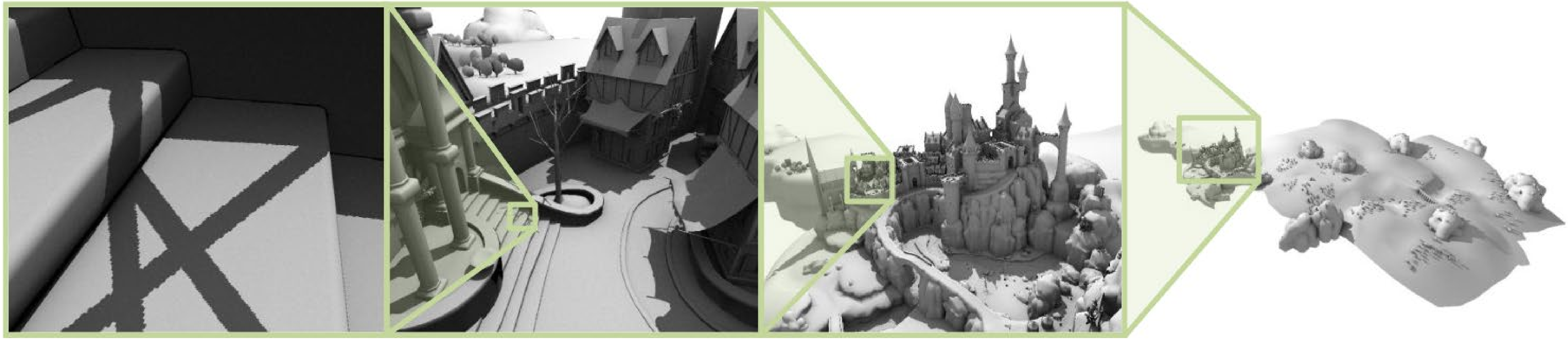
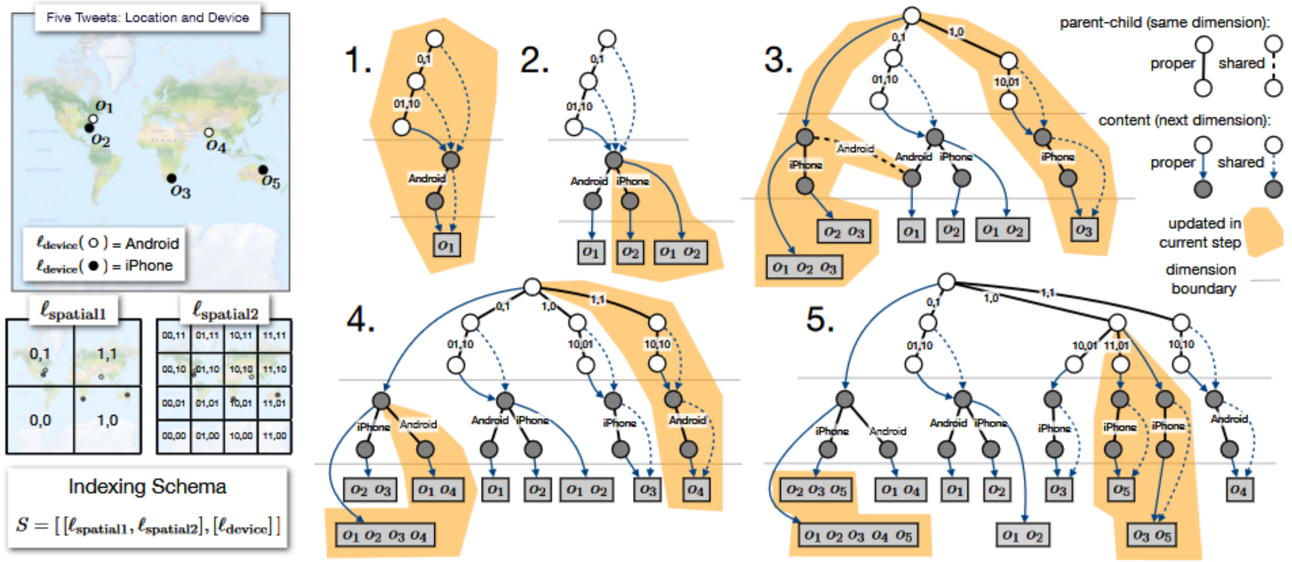


**Figure 1:** *The* EpicCitadel *scene voxelized to a* 128K$^3$ *(131 072*$^3$*) resolution and stored as a Sparse Voxel DAG. Total voxel count is 19 billion, which requires 945MB of GPU memory. A sparse voxel octree would require 5.1GB without counting pointers. Primary shading is from triangle rasterization, while ambient occlusion and shadows are raytraced in the sparse voxel DAG at 170 MRays/sec and 240 MRays/sec respectively, on an NVIDIA GTX680.*

Kampe et al., 2013

# Beyond graphics: nanocubes

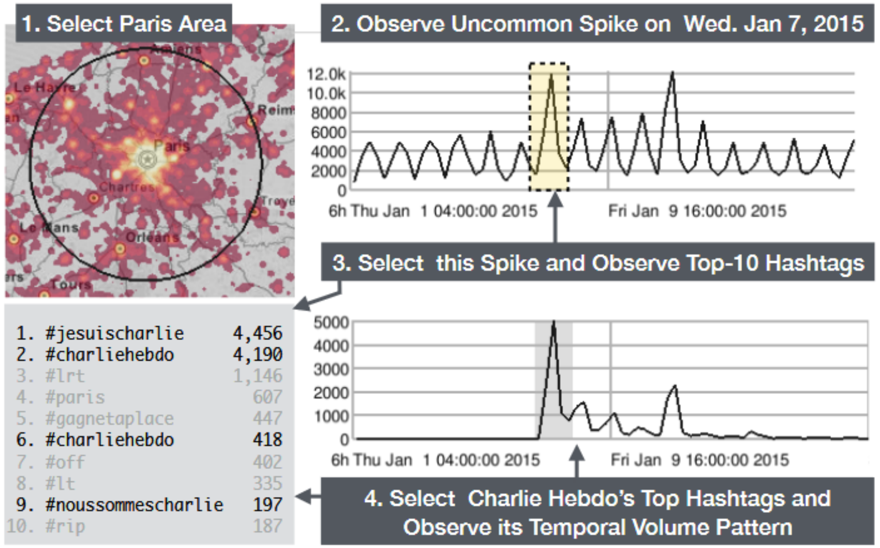
- Data structure for real-time exploration of spatiotemporal datasets. Key insight: pre-compute possible aggregations and store in a sparse data structure (store shared links across dimensions).




Lins et al., 2013

# Beyond graphics: topkube

- Data structure for real-time exploration of top-k queries of spatiotemporal datasets, also using sparse structure.



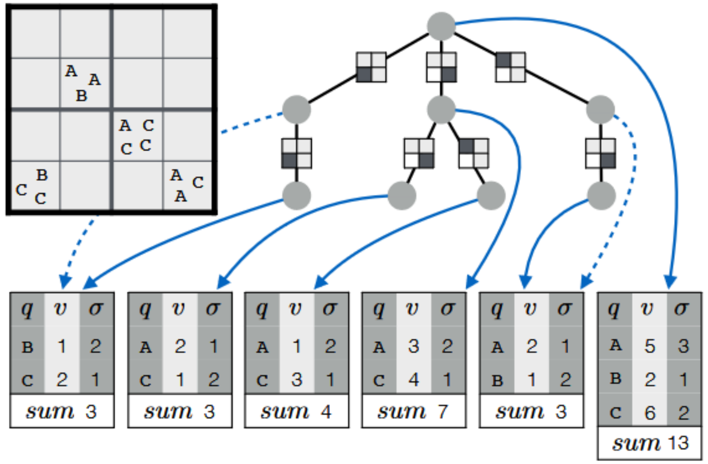



Fig. 3. Concrete example of a TopKube with one spatial dimension and the special key-dimension for counting and ranking the event types: A, B, or C. The additional ranking information $(q, v, sigma)$ from Equation 2 is shown in the tables associated with each product-bin.

Miranda et al., 2018



UIC COMPUTER SCIENCE