Data structures

CS594: Big Data Visualization & Analytics

Fabio Miranda

https://fmiranda.me

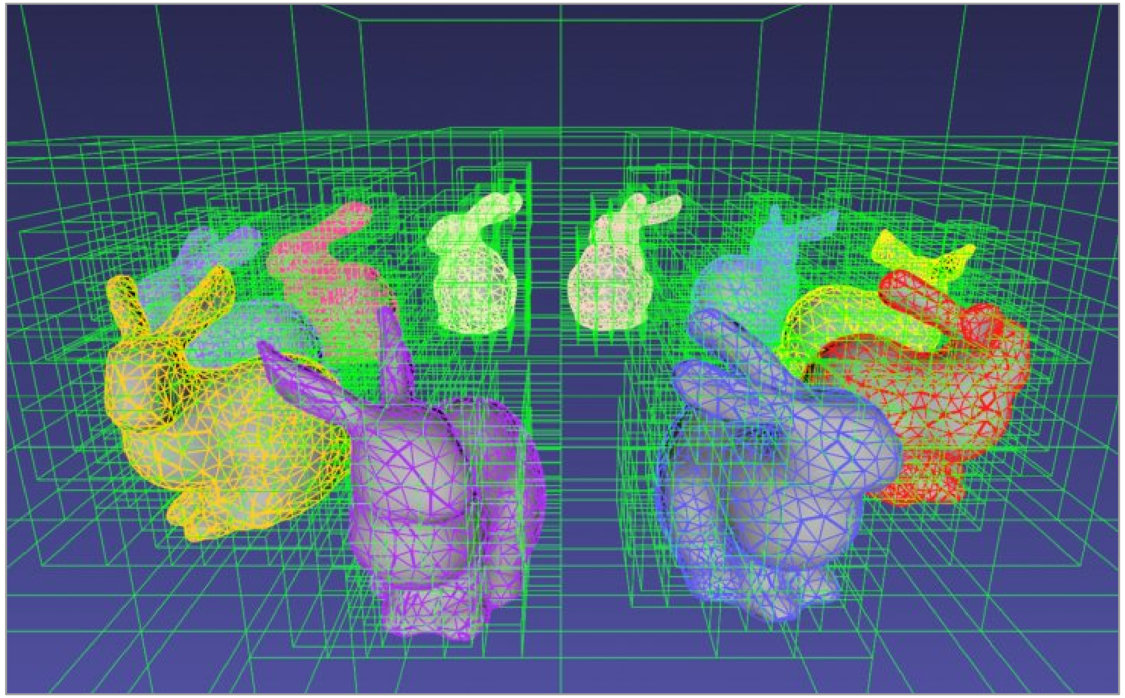
COMPUTER SCIENCE

Overview

- Spatial data structures:
  - Uniform grid
  - Nested grids
  - Quadtree / octree
  - K-d tree
  - BVH
  - …
- Visualization data structures:
  - Immens
  - Nanocubes
  - TopKube
  - …

How to efficiently organize objects?

- 2D/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.

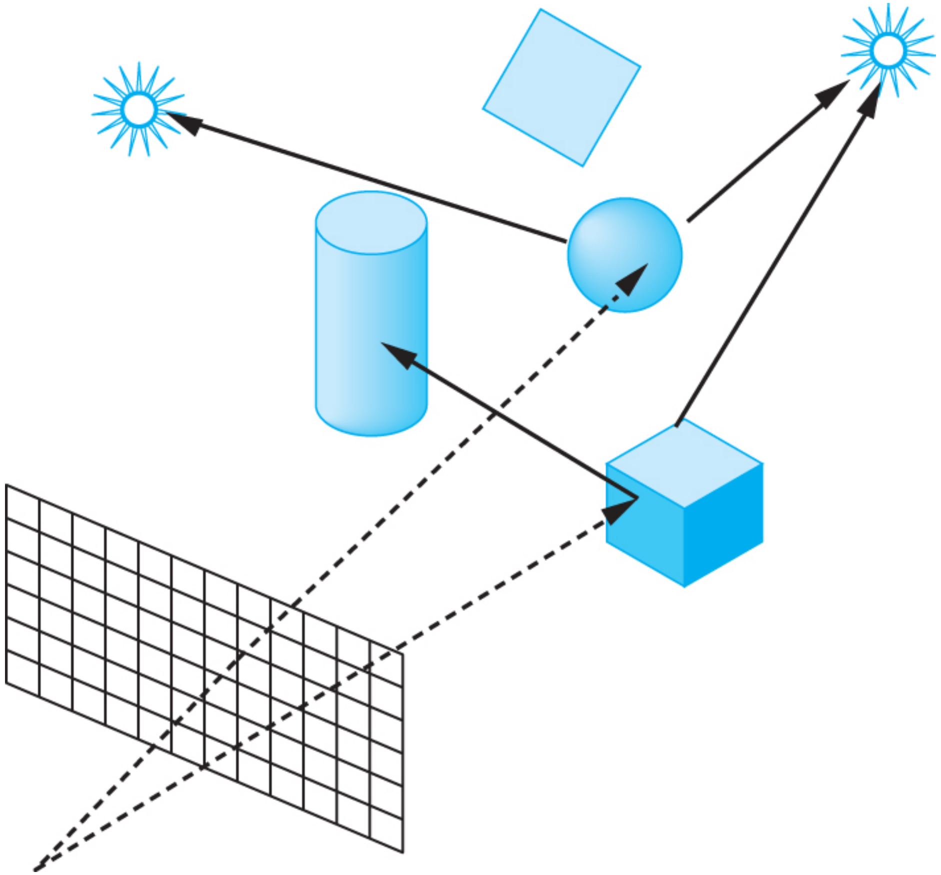


Ray-scene intersection

- 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?

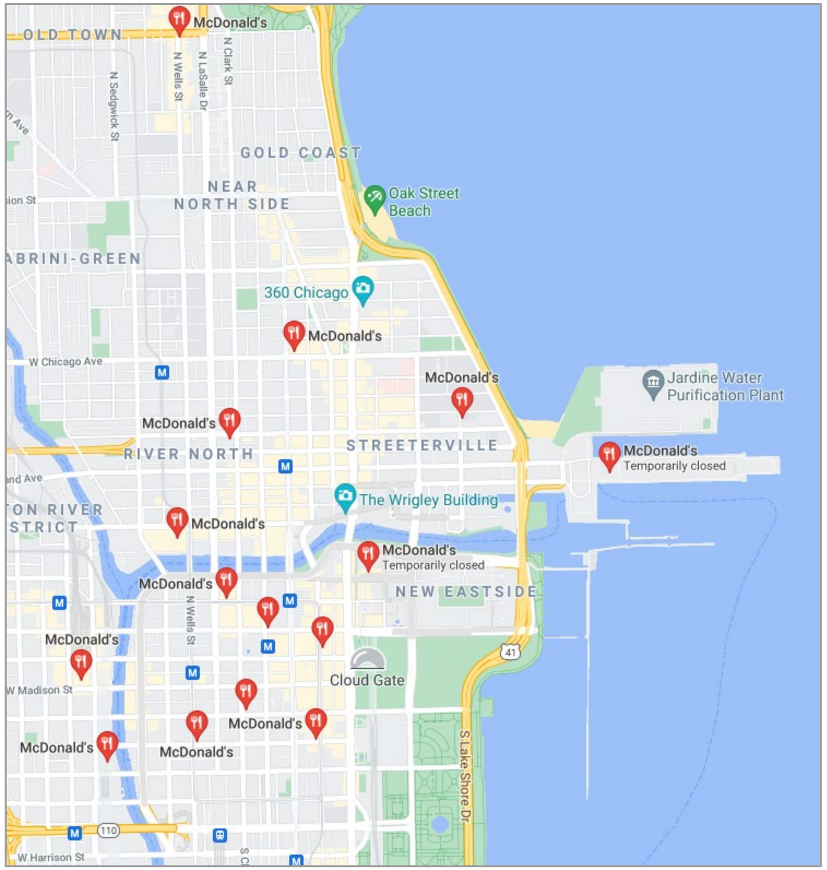


Proximity query

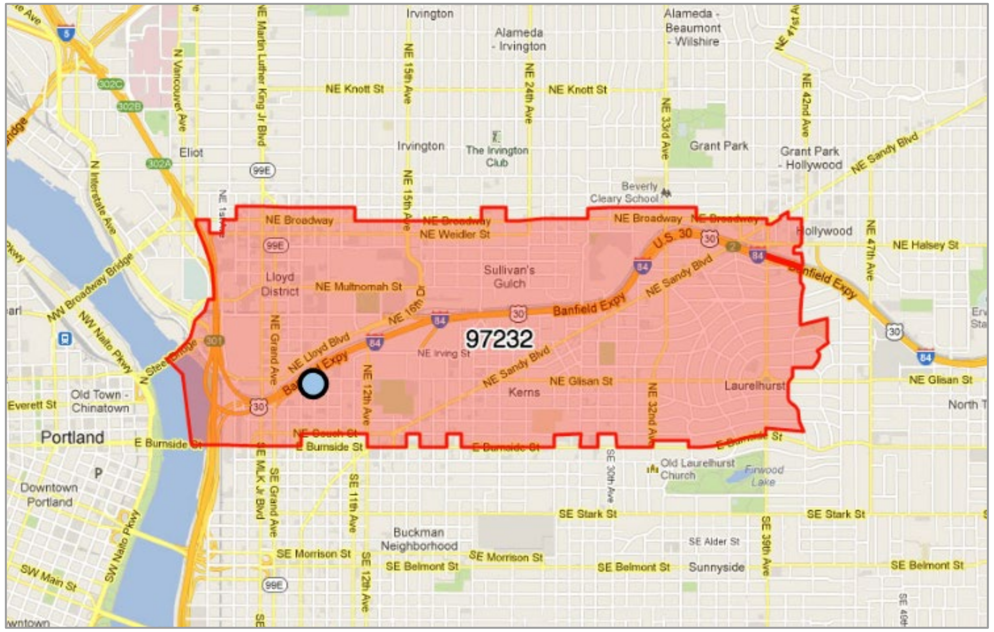
- Query based on proximity.
- “What is the closest McDonald’s?”

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

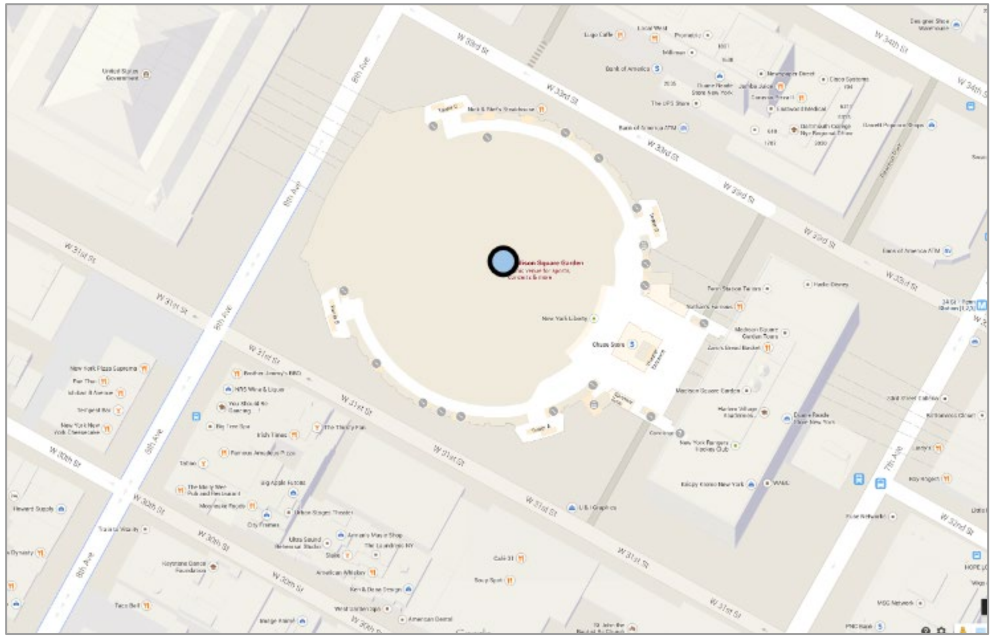
- Complexity: $O(n)$
- How to do better?



Point in polygon



What is the zip code for this complaint?



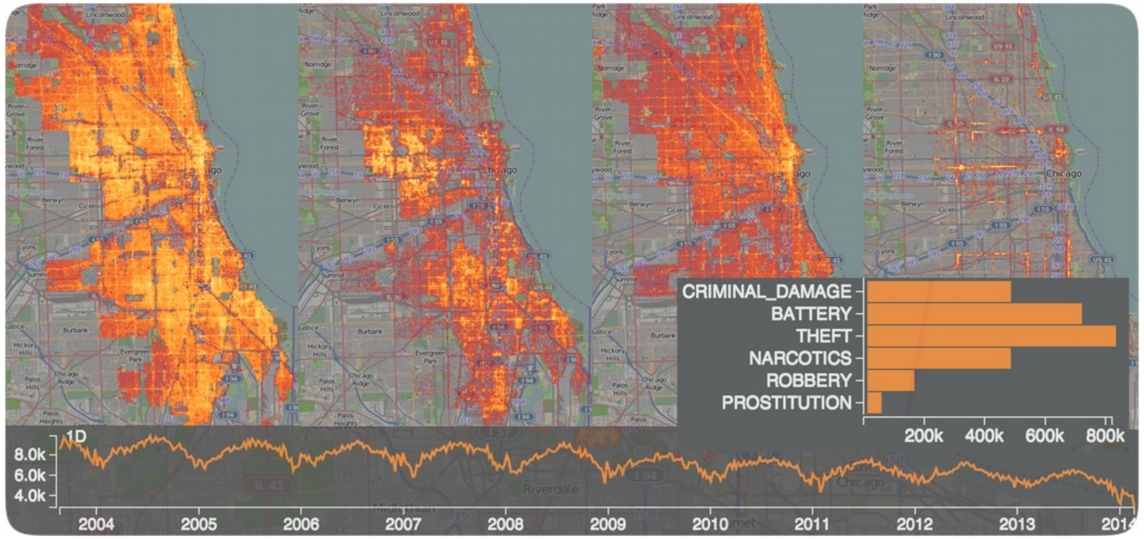
Am I inside a specific building?

Aggregation

- Aggregate spatiotemporal points.
- "Number of tweets in a region?"

```
function aggregate(points) {
    var map = AggregatedData(width, height, 0);
    for (var i=0; i < points.length; i++) {
        var coord = getCoord(points[i]);
        map.add(coord, 1);
    }
}
```

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



Time complexity

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

How to reduce the time complexity?

Motivation

- Expensive operations (ray tracing, query).
  - Complex datasets (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 $O(n)$.

Spatial data structures

- Data structures to accelerate queries of the kind:

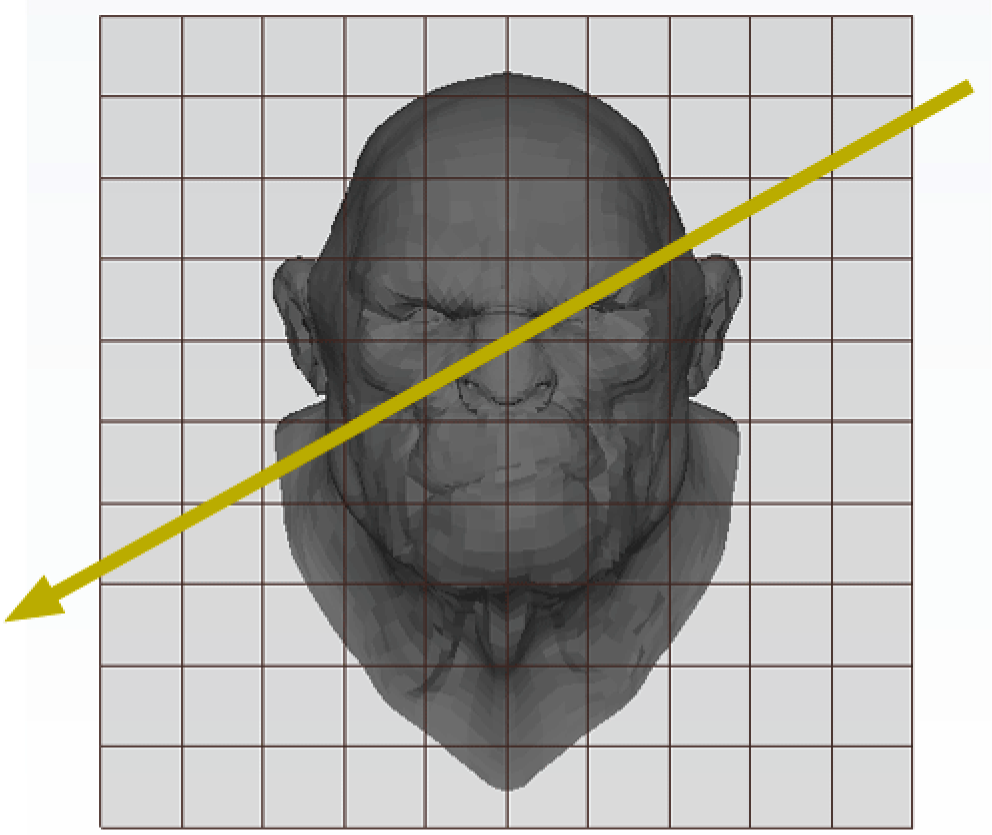
  "I'm here, which object is around me?"
- Partition space or set of objects.
- Tasks:
  1. Construction / update:
     - Pre-processing for static parts of the scene.
     - Update for moving parts of the scene.
  2. Access:
     - Optimize so it is done as fast as possible.

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.

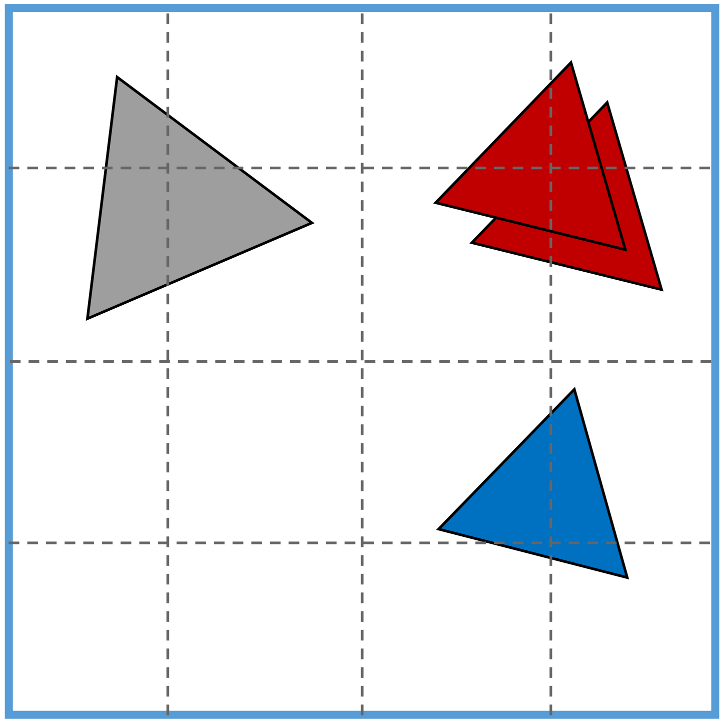
Uniform grid

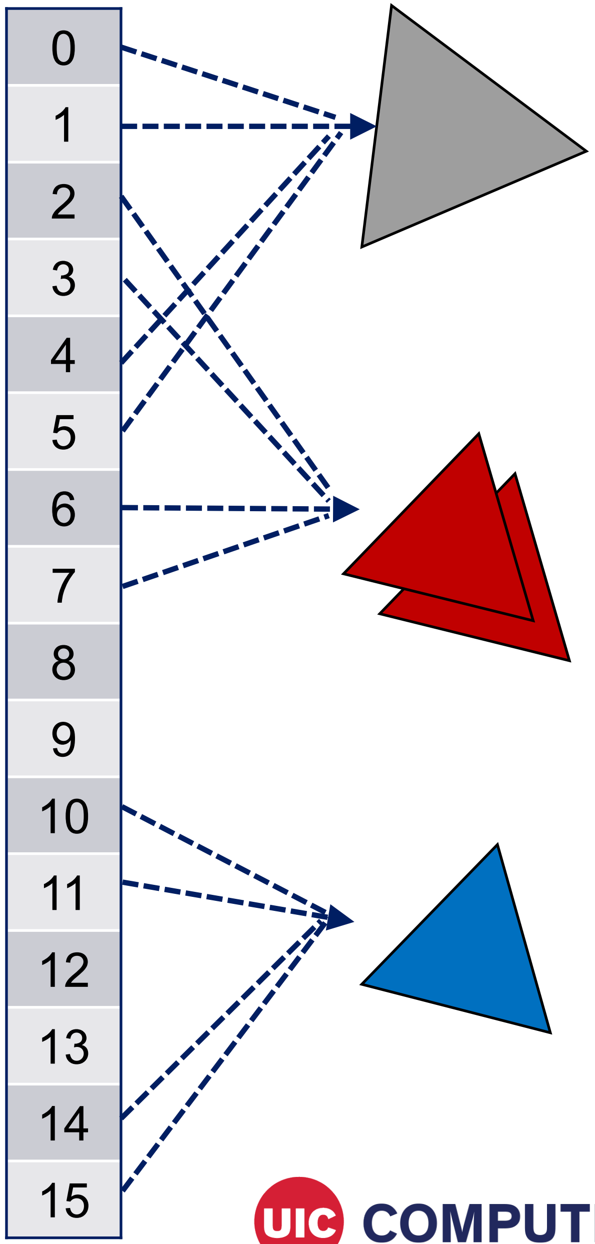
- Partition space into equal-sized volumes (i.e., voxels).
- Each cell will contain objects that overlap the voxel.
- Good for uniform data (points are evenly distributed in space).
- Fast construction and queries.



scratchpixel.com

Uniform grid





Uniform grid: construction and query

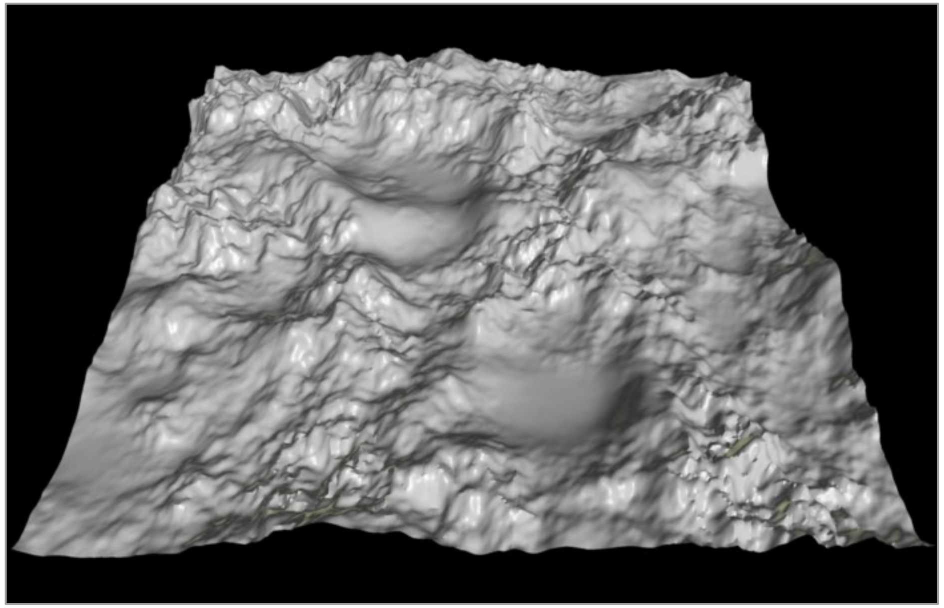
- Array of 3D voxels
  - Each voxel: list of pointers to colliding objects.
- Indexing function:
  - 3D point → cell index (constant time!)
- Construction:
  - Initialize cells for grid with size $w * h$
  - For each object $\mathrm{p}(x, y)$:
    - Compute grid cell using (x, y).
    - Store p in cell.
- 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.

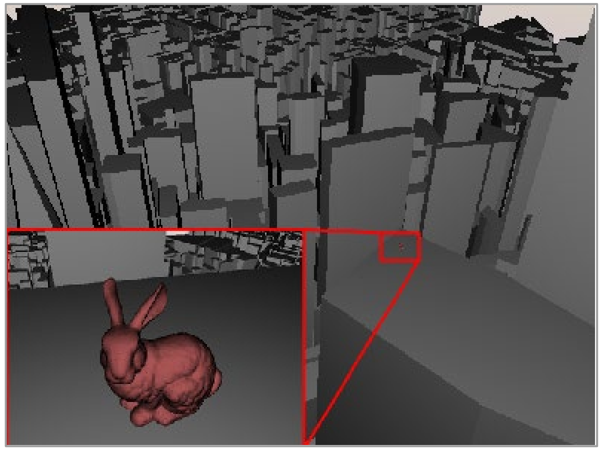


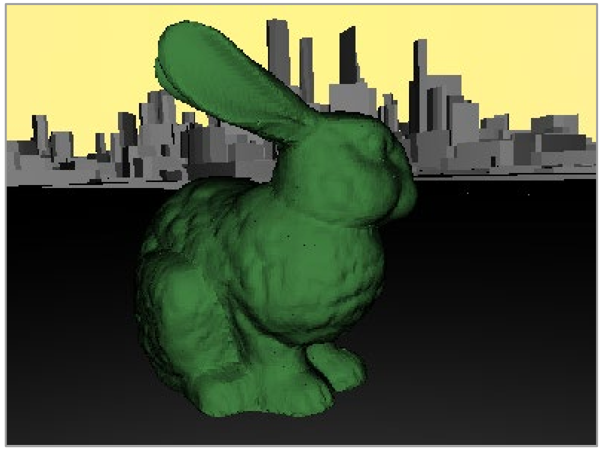
Mitsuba renderer

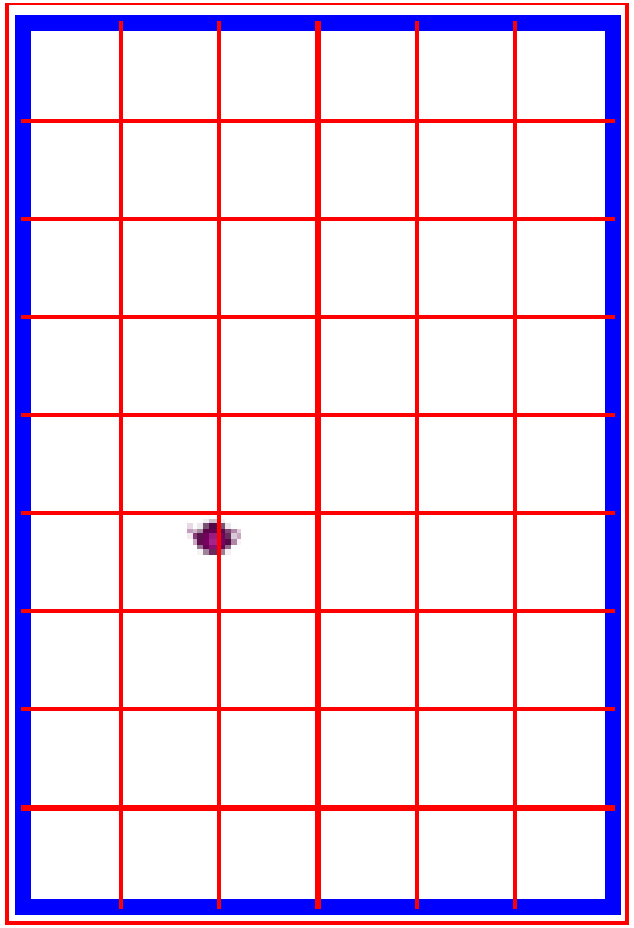
peterguthrie.net

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.





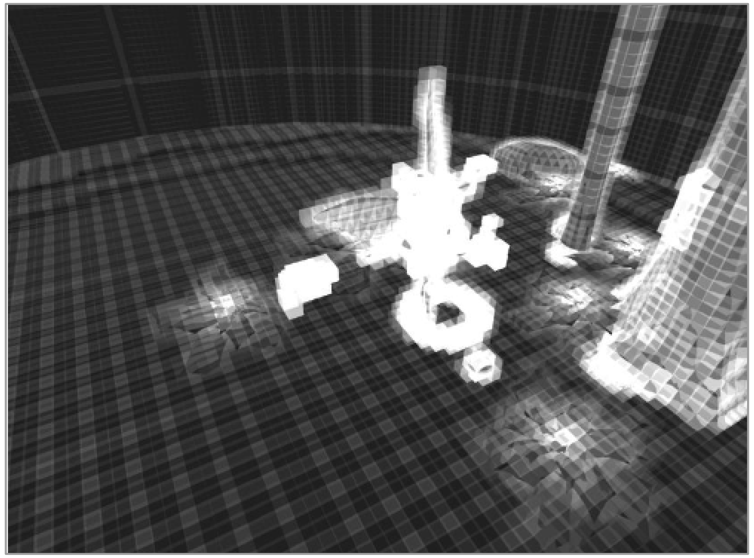


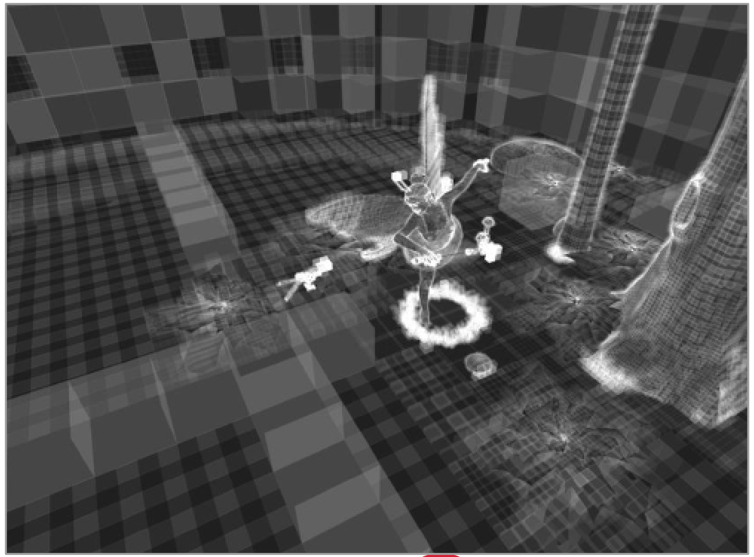
Uniform grid: drawbacks

- 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.

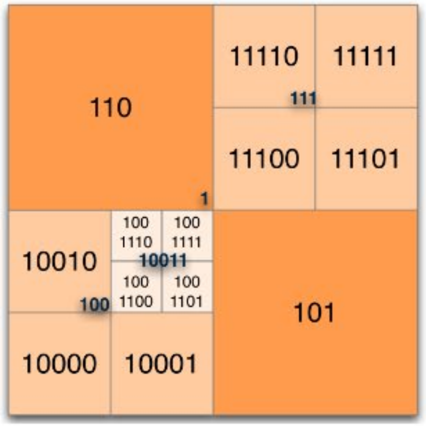


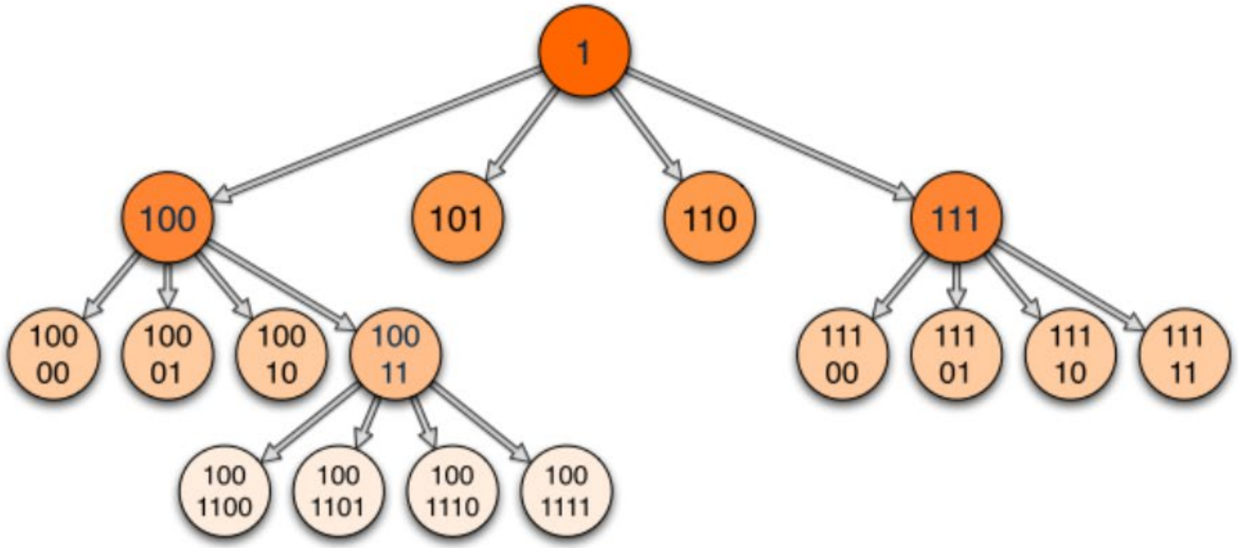


Philipp Slusallek

Quadtree

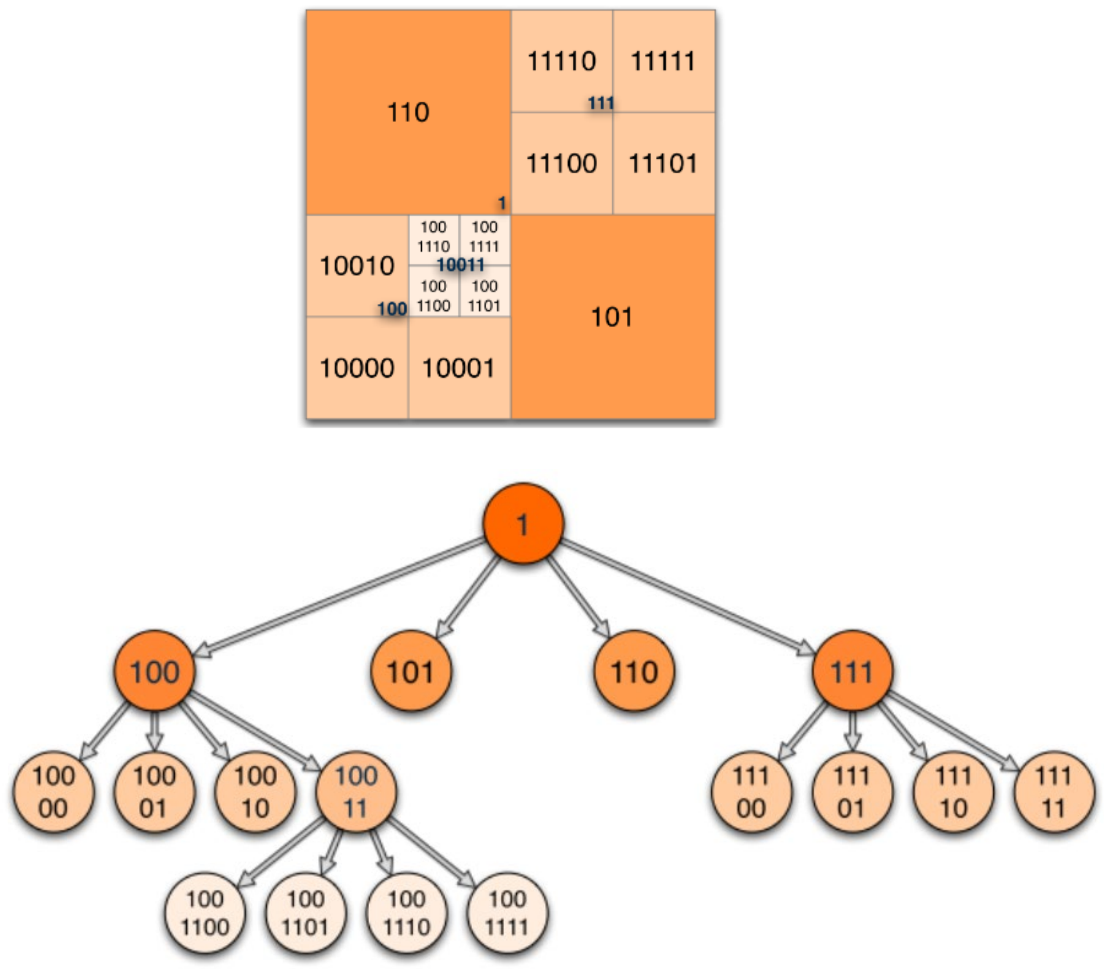
- Hierarchical structure that stores regular grids at each level.
- Adaptive subdivision: adjust depth to local scene complexity.





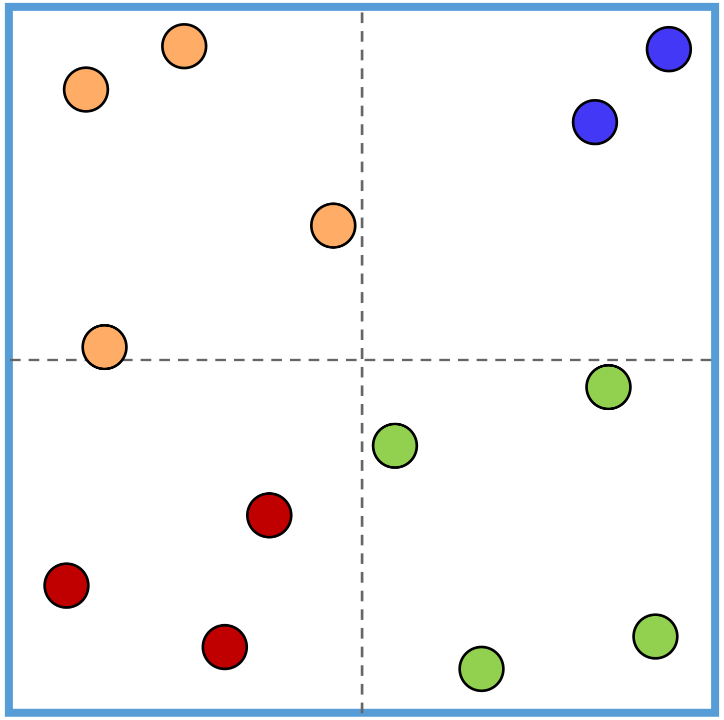
Quadtree

- 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).



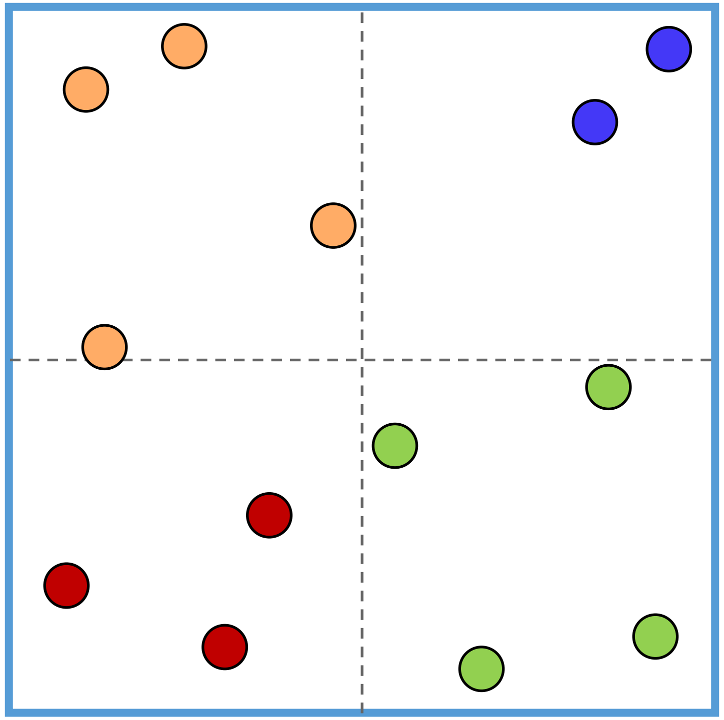
Quadtree: construction

- Split the top level.



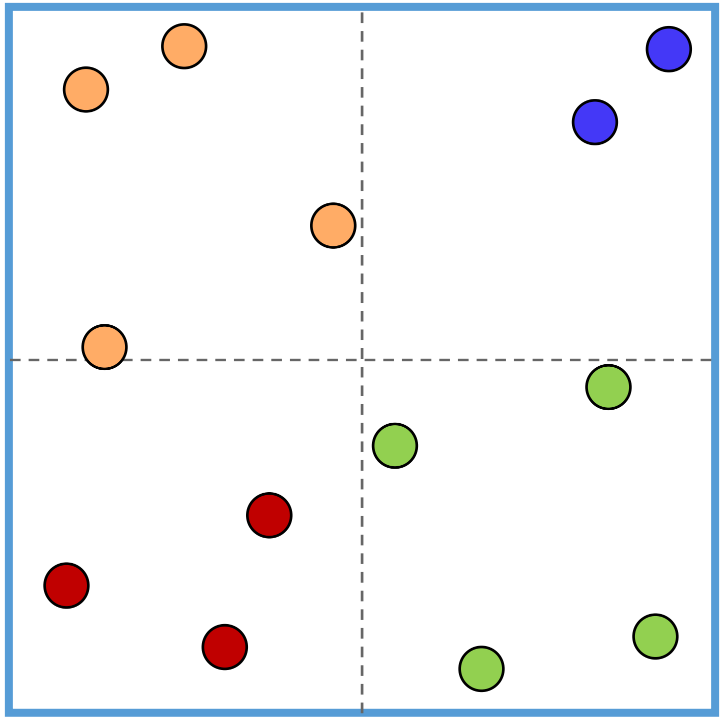
Quadtree: construction

- Split the top level.
- Can we stop?



Quadtree: construction

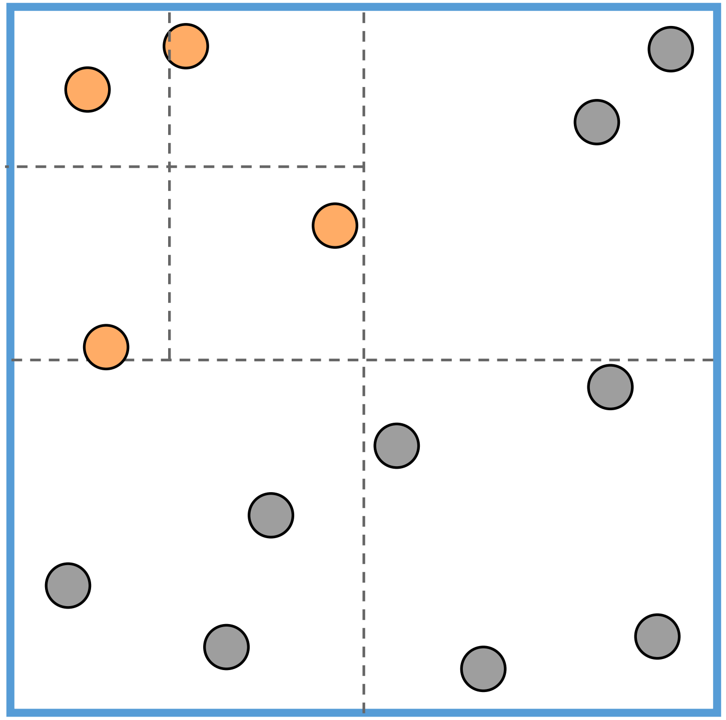
- Split the top level.
- Can we stop? No, split the next level.



UIC COMPUTER SCIENCE

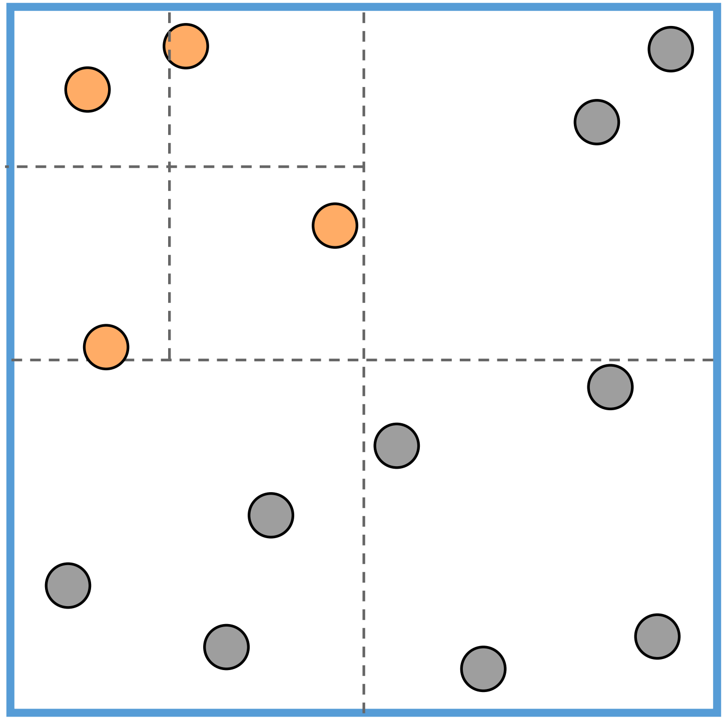
Quadtree: construction

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



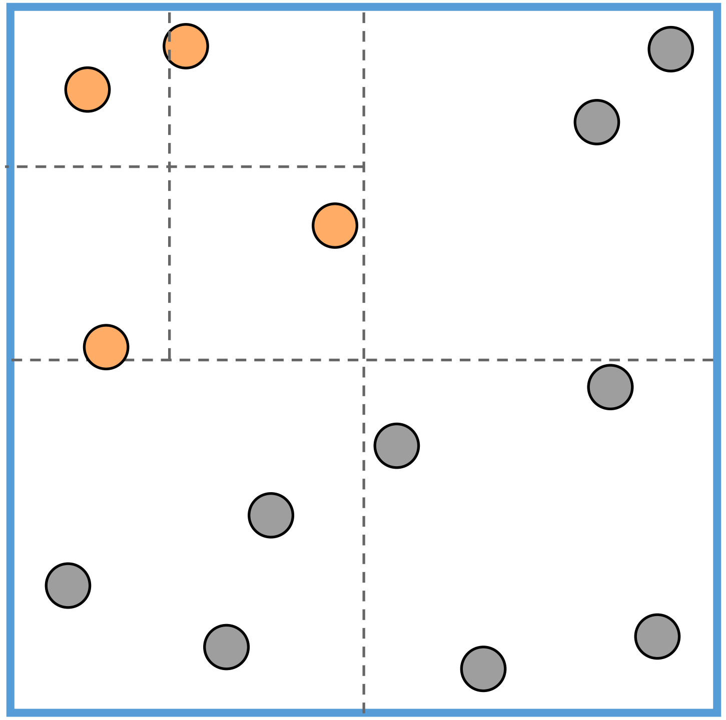
Quadtree: construction

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



Quadtree: construction

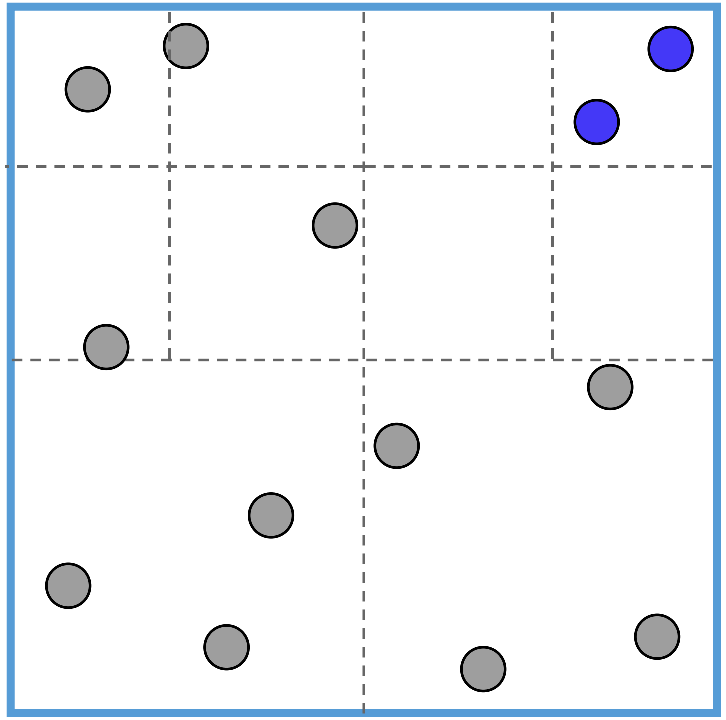
- Split the top level.
- Can we stop? No, split the next level.
- Split top-left.
- Can we stop top-left? Yes.



UIC COMPUTER SCIENCE

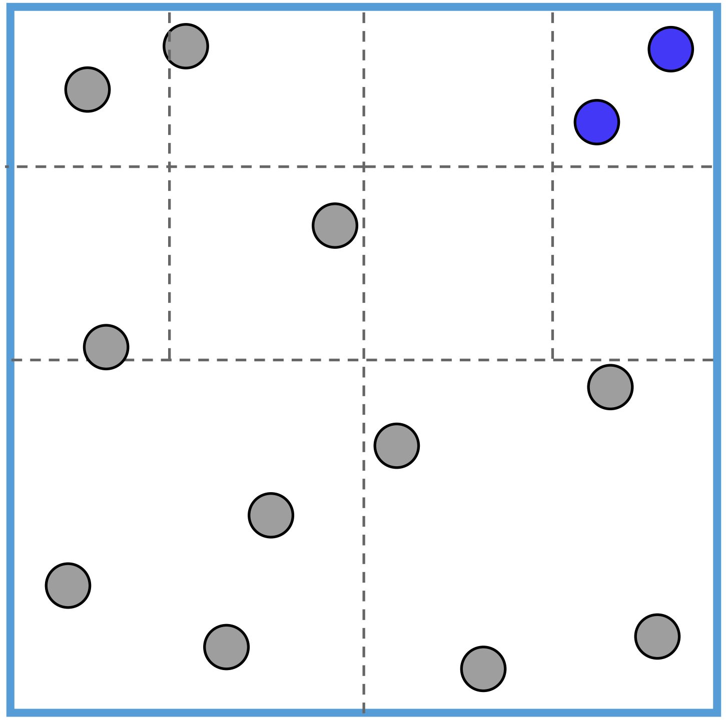
Quadtree: construction

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



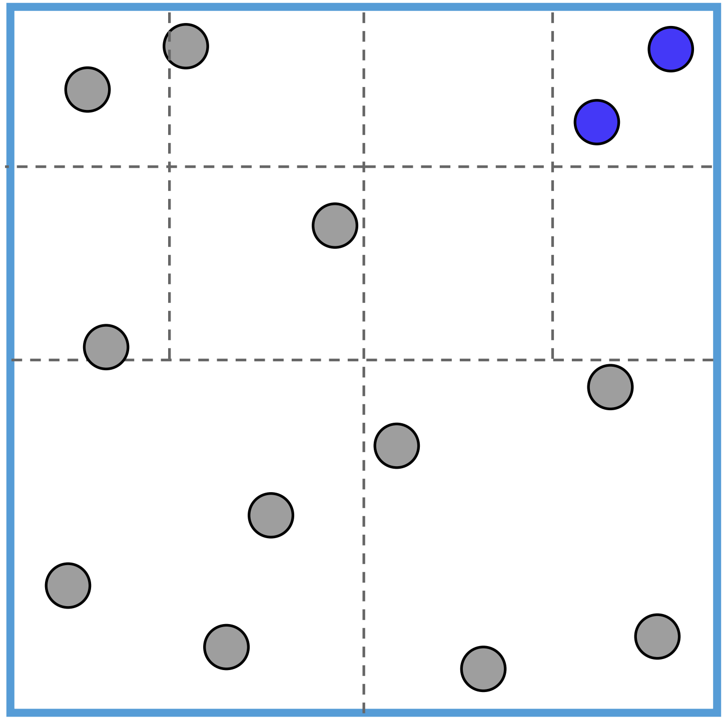
Quadtree: construction

- 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?



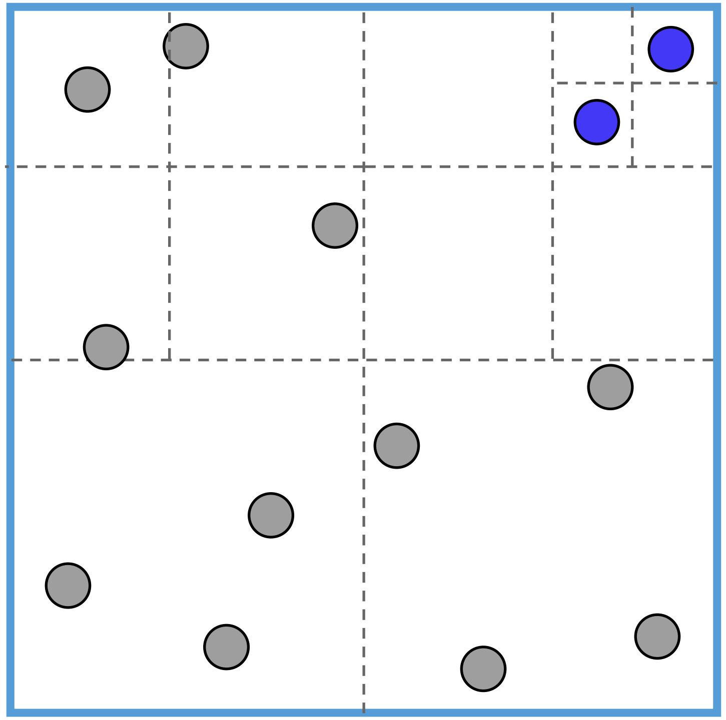
Quadtree: construction

- 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.



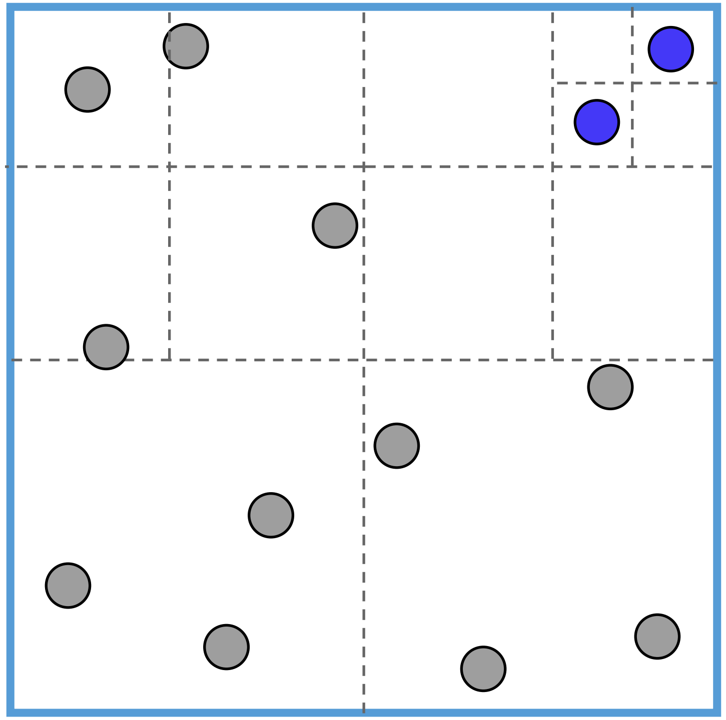
Quadtree: construction

- 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.



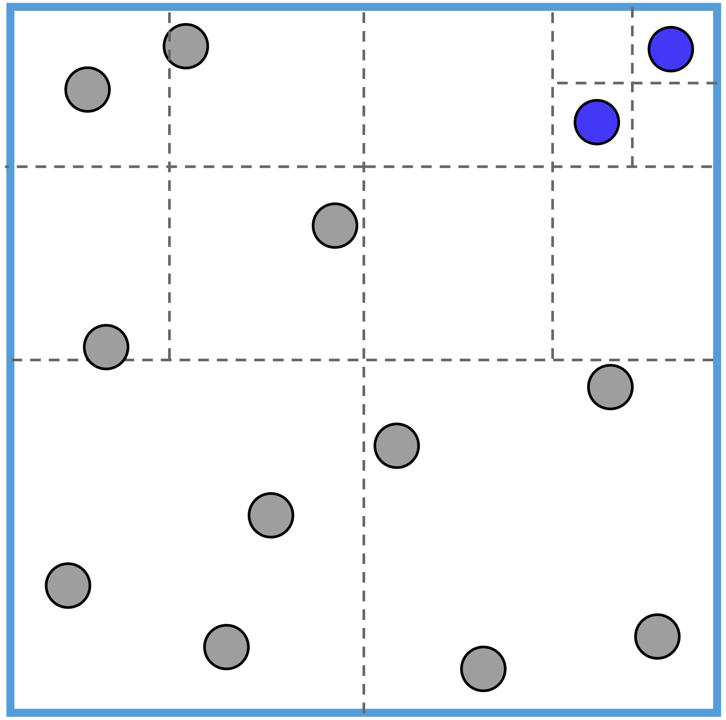
Quadtree: construction

- 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?



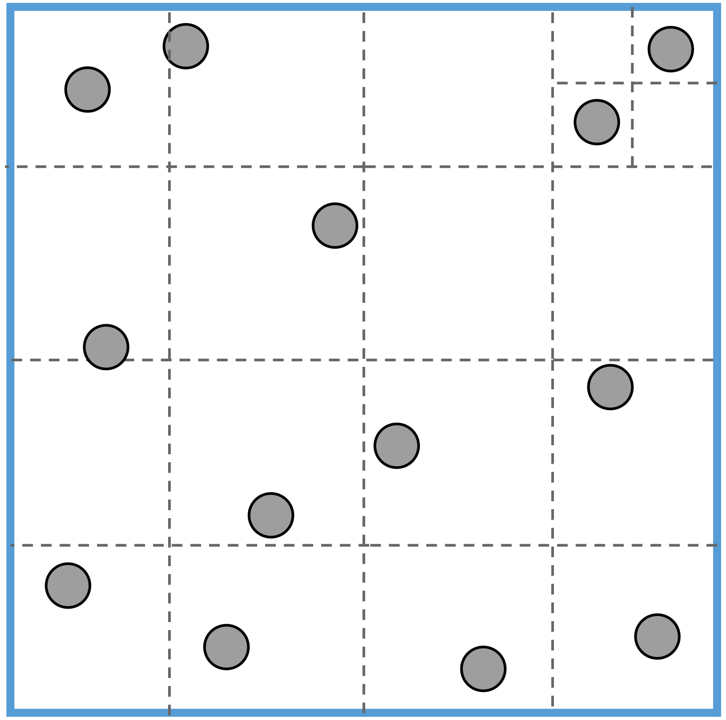
Quadtree: construction

- 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.

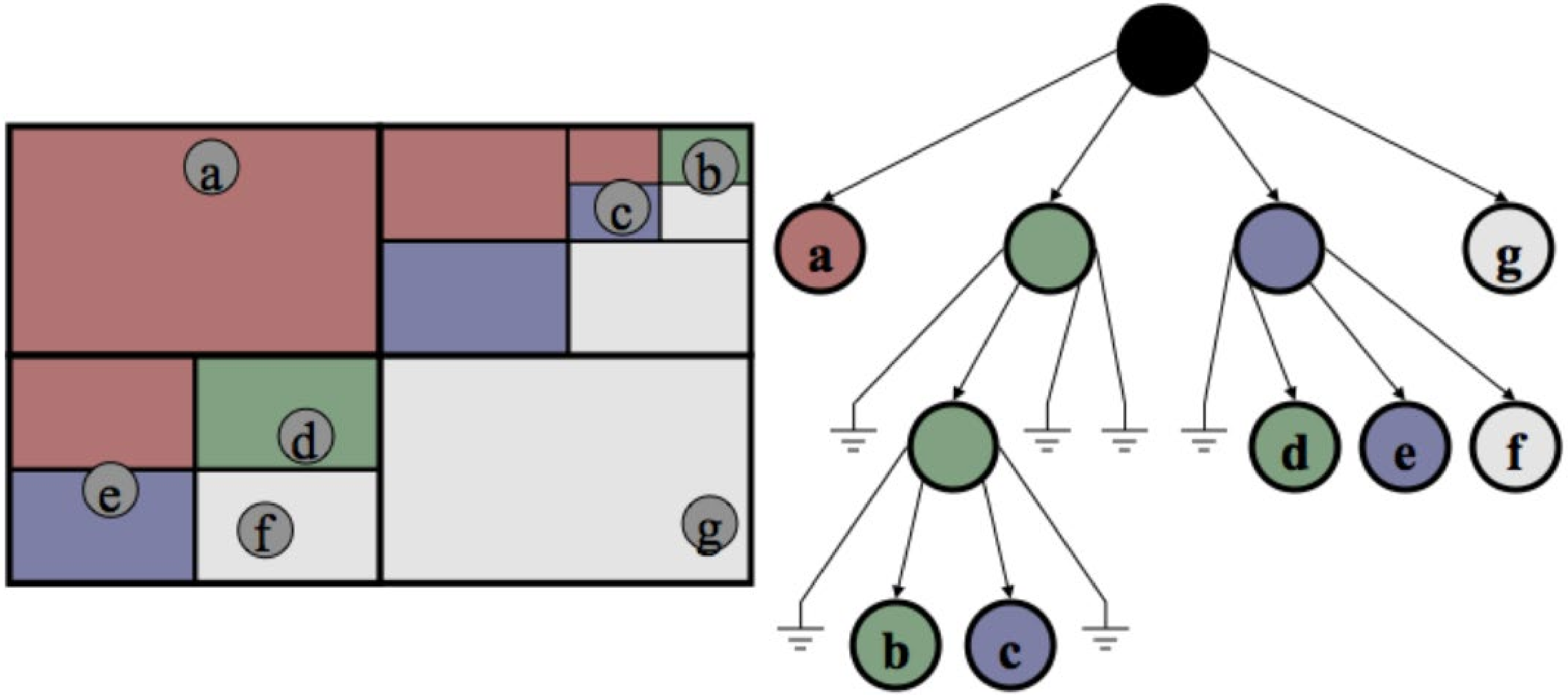


Quadtree: construction

- 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

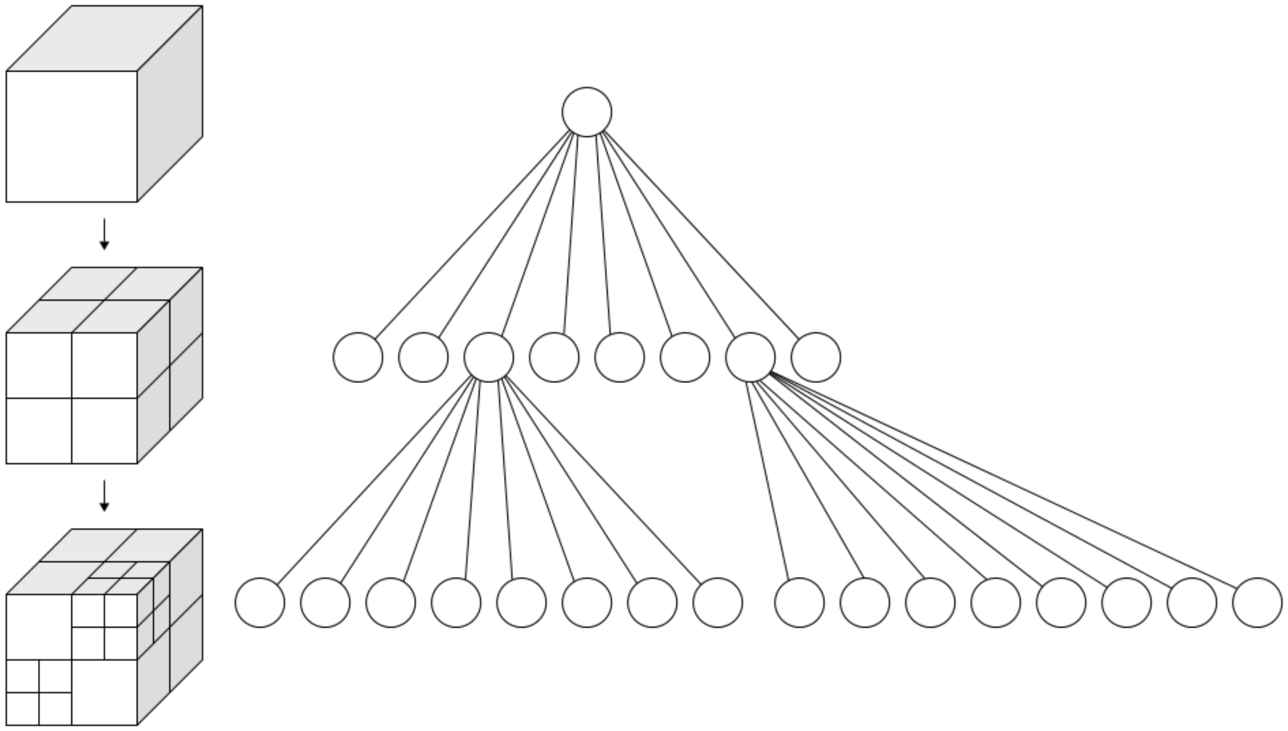
- Query:
  - Input: range query r $(x_1, y_1) \times (x_2, y_2)$, tree node v.
  - If v is a leaf:
    - Search points of v inside range r.
  - If v_{00} inside range r :
    - Query(v_{00}, r)
  - If v_{01} inside range r :
    - Query(v_{01}, r)
  - If v_{10} inside range r :
    - Query(v_{10}, r)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

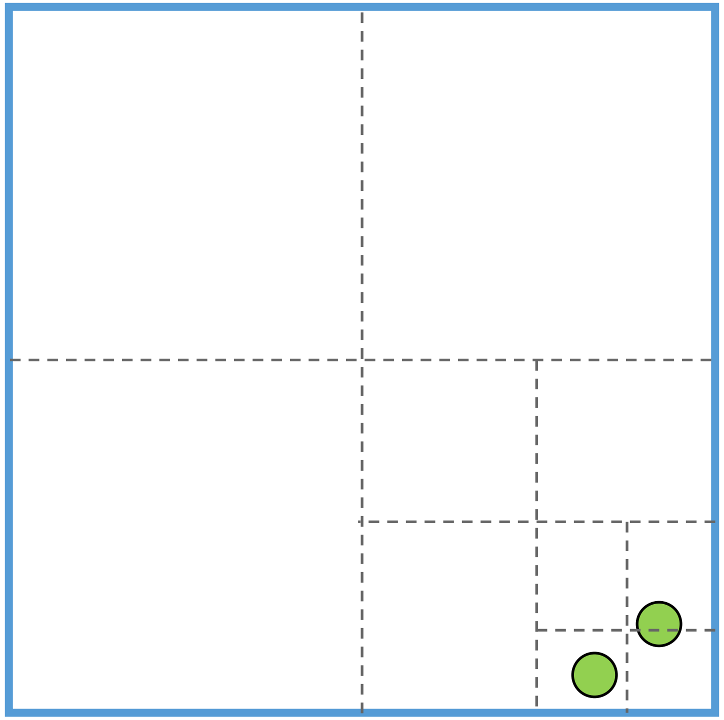
Octree

- 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.



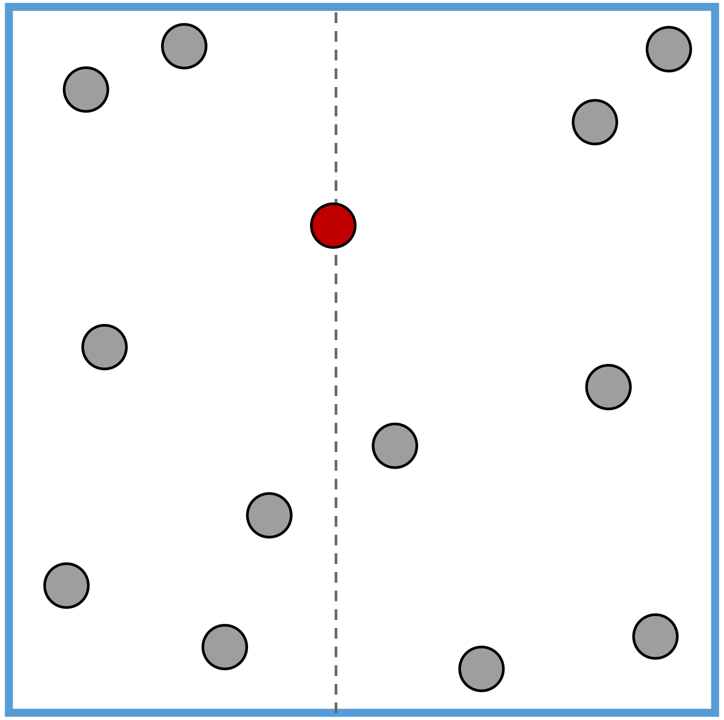
UIC COMPUTER SCIENCE

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).

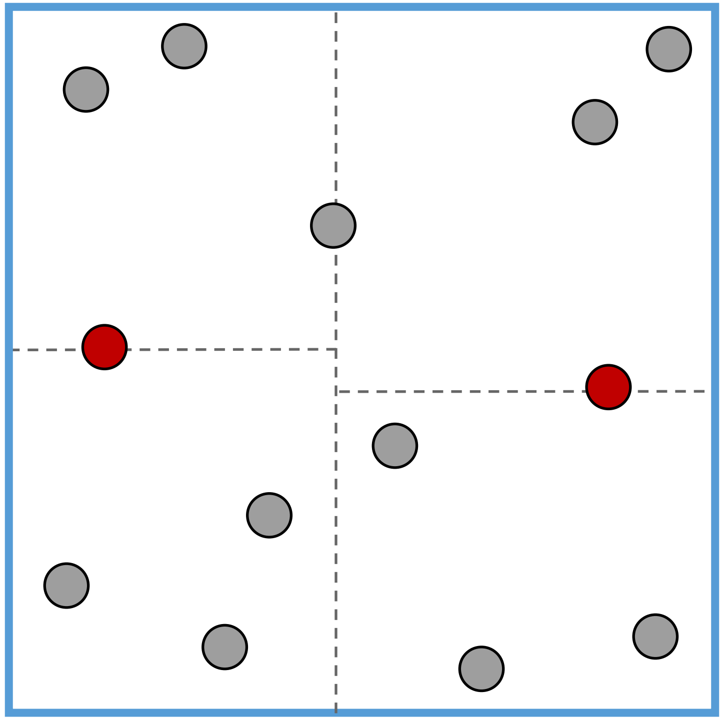
K-d tree: construction

- First split: x dimension (median point).



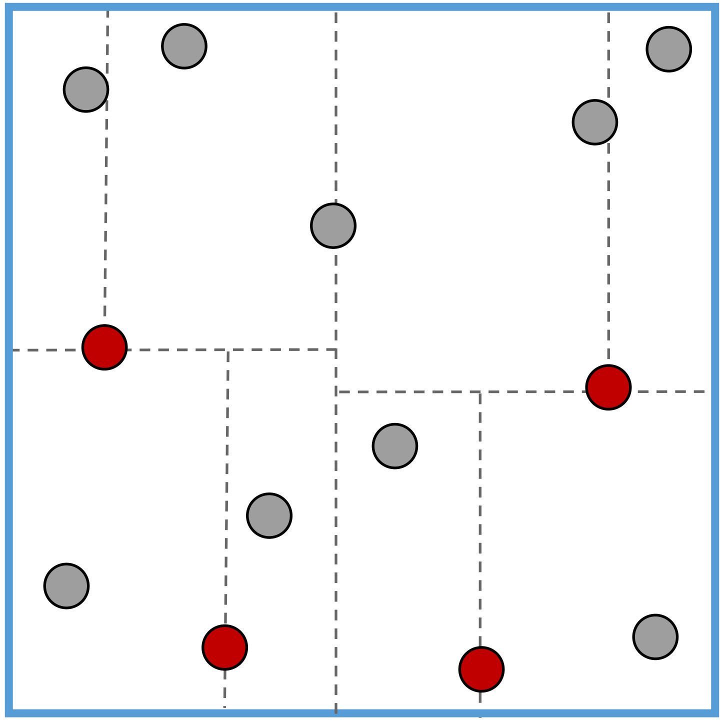
K-d tree: construction

- 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 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: $O(logn)$

K-d tree and Scikit-learn

- K-nearest neighbors and neighbors within a radius:

```
from sklearn.neighbors import KDTree
import numpy as np

rng = np.random.RandomState(0)
X = rng.random_sample((1000, 2))
tree = KDTree(X, leaf_size=2)
dist, ind = tree.query(X[:1], k=3)
```

```
from sklearn.neighbors import KDTree
import numpy as np

rng = np.random.RandomState(0)
X = rng.random_sample((1000, 2))
tree = KDTree(X, leaf_size=2)
points = tree.query_radius(X[:1], r=0.3)
```

- Kernel density estimation:

```
from sklearn.neighbors import KDTree
import numpy as np

rng = np.random.RandomState(0)
X = rng.random_sample((1000, 2))
tree = KDTree(X, leaf_size=2)
estimate = tree.kernel_density(X[:3], h=0.1, kernel='gaussian')
```

K-d tree and Scikit-learn

- tSNE

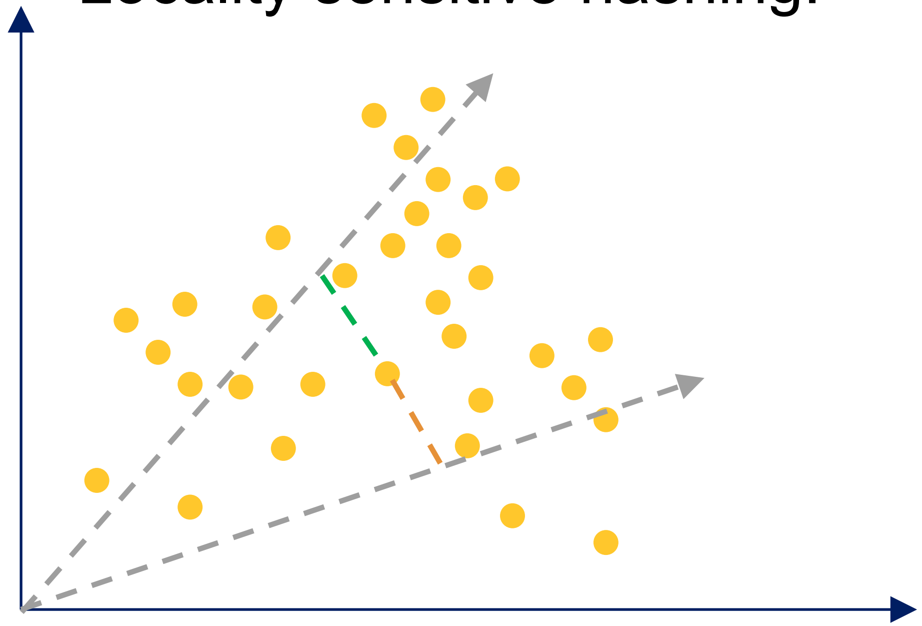
```
import numpy as np
from sklearn.neighbors import KNeighborsTransformer
from sklearn.pipeline import make_pipeline

rng = np.random.RandomState(0)
X = rng.random_sample((1000, 2))
transformer = make_pipeline(
    KNeighborsTransformer(n_neighbors=n_neighbors, mode='distance',metric=metric),
    TSNE(metric='precomputed', **tsne_params)
)

transformer.fit_transform(X)
```

Annoy and Scikit-learn

- Annoy: approximate nearest neighbords C++ library with Python bindings.
- Locality sensitive hashing:



0.37	0.12	…	0.93	0.44

4 kbytes per embedding

0	0	…	1	0

64 bytes per embedding

$$\alpha_{1,2} = \cos^{-1}\left(\frac{\overrightarrow{v_1} \cdot \overrightarrow{v_2}}{|\overrightarrow{v_1}||\overrightarrow{v_2}|}\right)$$

Annoy and Scikit-learn

- tSNE

```
class AnnoyTransformer(TransformerMixin, BaseEstimator):
    def fit(self, X):
        self.n_samples_fit_ = X.shape[0]
        self.annoy_ = annoy.AnnoyIndex(X.shape[1], metric=self.metric)
        for i, x in enumerate(X):
            self.annoy_.add_item(i, x.tolist())
        self.annoy_.build(self.n_trees)
        return self
    (...)
rng = np.random.RandomState(0)
X = rng.random_sample((1000, 2))
transformer = make_pipeline(
    AnnoyTransformer(n_neighbors=n_neighbors, metric=metric),
    TSNE(metric='precomputed', **tsne_params)
)

embedded = transformer.fit_transform(X)
```

Annoy and Scikit-learn

- tSNE

```
class AnnoyTransformer(TransformerMixin, BaseEstimator):
    def fit(self, X):
        self.n_samples_fit_ = X.shape[0]
        self.annoy_ = annoy.AnnoyIndex(X.shape[1], metric=self.metric)
        for i, x in enumerate(X):
            self.annoy_.add_item(i, x.tolist())
        self.annoy_.build(self.n_trees)
        return self
    (...)
rng = np.random.RandomState(0)
X = rng.random_sample((1000, 2))
transformer = make_pipeline(
    AnnoyTransformer(n_neighbors=n_neighbors, metric=metric),
    TSNE(metric='precomputed', **tsne_params)
)

embedded = transformer.fit_transform(X)
```

TSNE with AnnoyTransformer: 30.225 sec
TSNE with KNeighborsTransformer: 64.845 sec

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.

Data structures for visualization

- Immens
- Nanocube
- TopKube
- Learned cubes