CS354R

DR SARAH ABRAHAM

SPATIAL PARTITIONING

SPATIAL DATA STRUCTURES

- Data indexed by spatial location (e.g. location or polygons)
- Multitude of uses in video games
  - Visibility - What can I see?
  - Ray intersections - What did the player just shoot?
  - Collision detection - Did the player just hit a wall?
  - Proximity queries - Where is the nearest power-up?

USING DECOMPOSITIONS

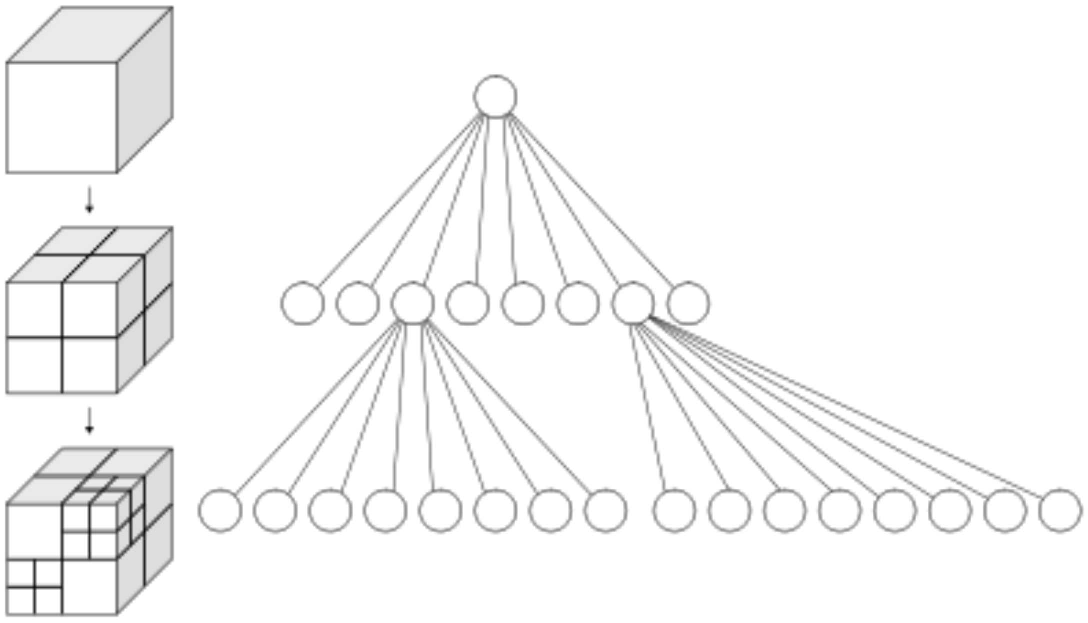
- Geometric queries are expensive
- Reduce the cost with fast, approximate queries that eliminate distant (or hidden) objects
- Trees with a containment property allow us to do this
  - The cell of a parent completely contains all the cells of its children
  - If a query fails for the cell, we know it will fail for all its children
  - If the query succeeds, we try for the children
  - If we get to a leaf, we do the expensive query

SPATIAL DECOMPOSITIONS

- Partition space into regions, or cells, of some type
- Octrees (Quadtrees): Axis aligned, regularly spaced planes cut space into cubes (squares)
- Kd-trees: Axis aligned planes cut space into rectilinear regions
- BSP trees: Arbitrarily aligned planes cut space into convex regions
- BVHs: Geometry hierarchically arranged within the tree

OCTREE

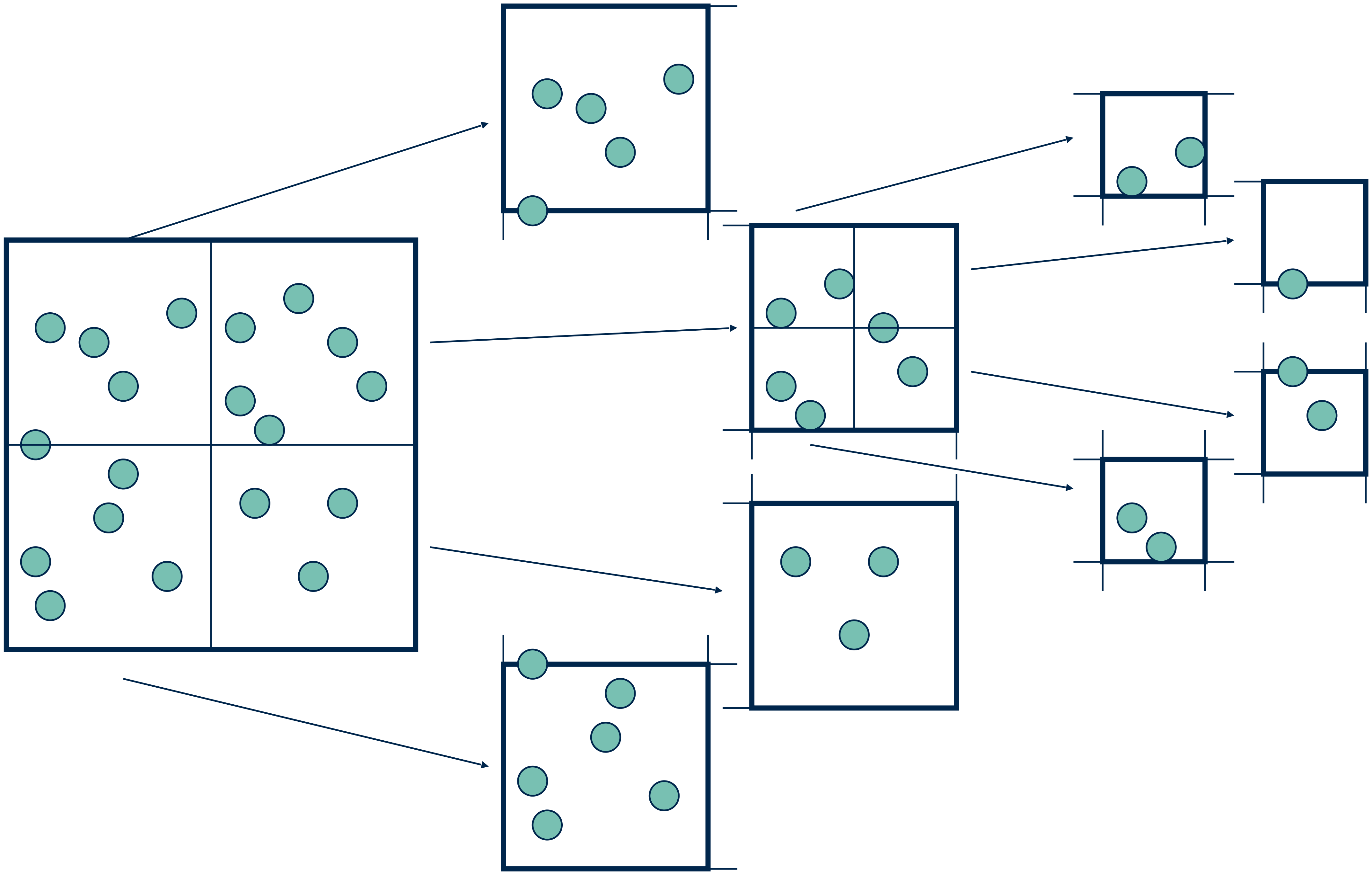
- Root node represents a cube containing entire world
- Each node has eight children nodes
  - Quadtree is for 2D decompositions - root is square and four children are sub-squares
- Objects assigned to nodes in one of two common ways:
  - All objects are in leaf nodes
  - Each object is in the leaf that partially contains it



OCTREE NODE DATA STRUCTURE

- What needs to be stored in a node?
  - Children pointers (at most eight)
  - Parent pointer
  - Extents of cube (inferable from tree structure, but easier to store)
  - Data associated with the contents of the cube
    - Contents might be whole objects or individual polygons, or even something else
  - Neighbors are useful in some algorithms (but not all)

QUADTREE EXAMPLE CONSTRUCTION

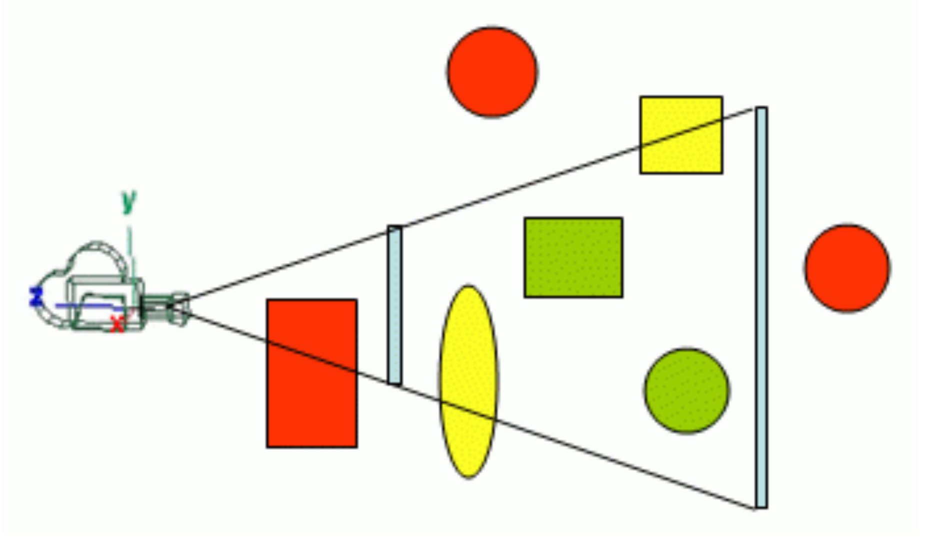


OBJECTS IN MULTIPLE CELLS

- Assume an object intersects more than one cell
- Typically store pointers to it in all the cells it intersects
  - Why can't we store it in just one cell?
- Object might be considered twice for some tests
  - Solution 1: Flag an object when it has been tested and not consider it again until the next round of testing
  - Solution 2: Tag it with the frame number it was last tested

FRUSTUM CULLING WITH OCTREES

- Eliminate objects that do not intersect the view frustum
- Have a test that succeeds if a cell may be visible
  - Test corners of cell against each clip plane
- Starting with the root node cell, perform the test
  - If it fails, nothing inside the cell is visible
  - If it succeeds, something inside the cell might be visible
  - Recurse for each child of a visible cell



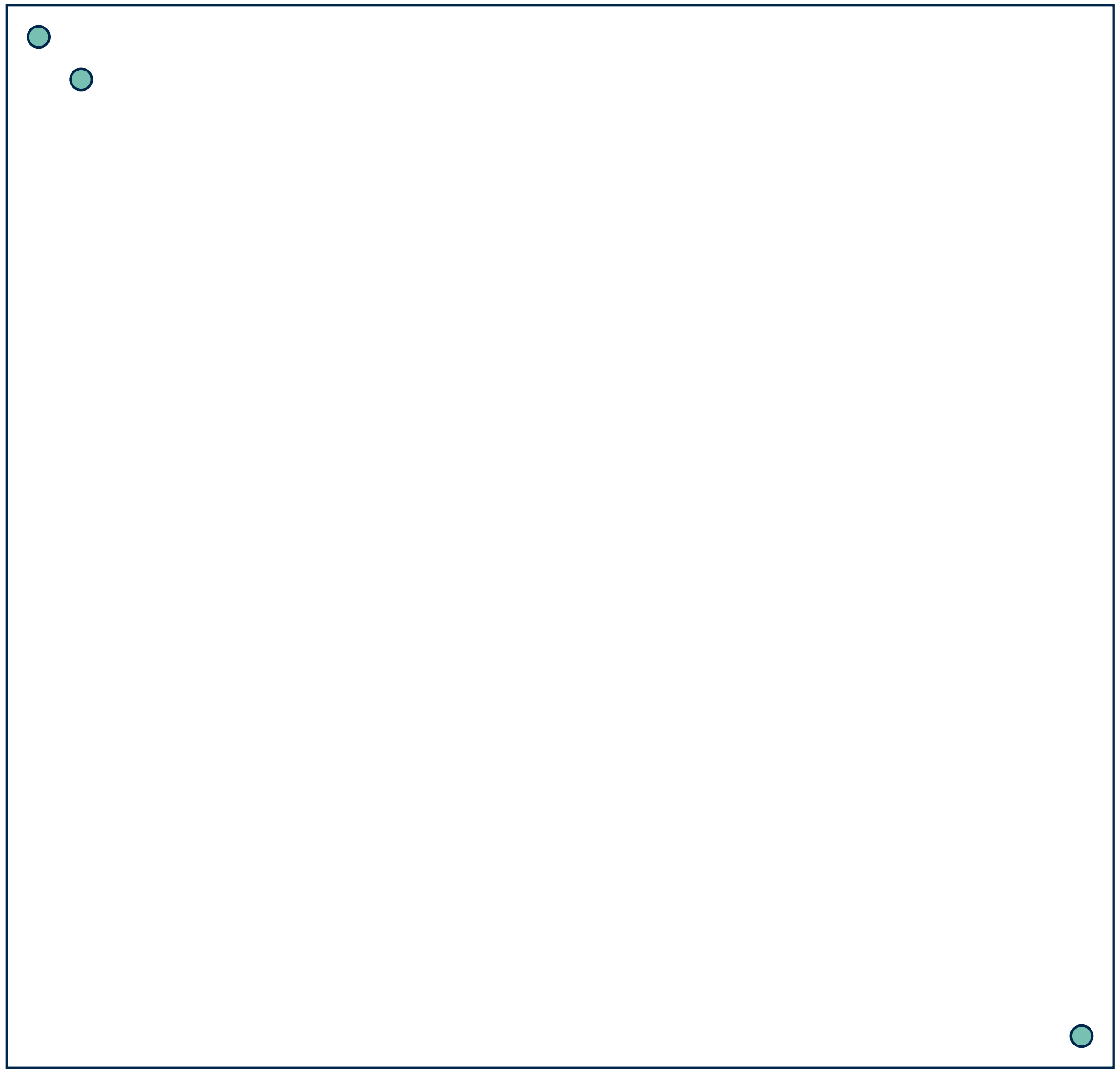
(Lighthouse3

OTHER COMMON TESTS

- Interference Testing (which cells an object collides with)
- Ray Intersection Testing (which cells a ray intersects)

OCTREE PROBLEMS

- Octrees become very unbalanced if the objects are far from a uniform distribution
- Problem is the requirement that cube always be equally split amongst children

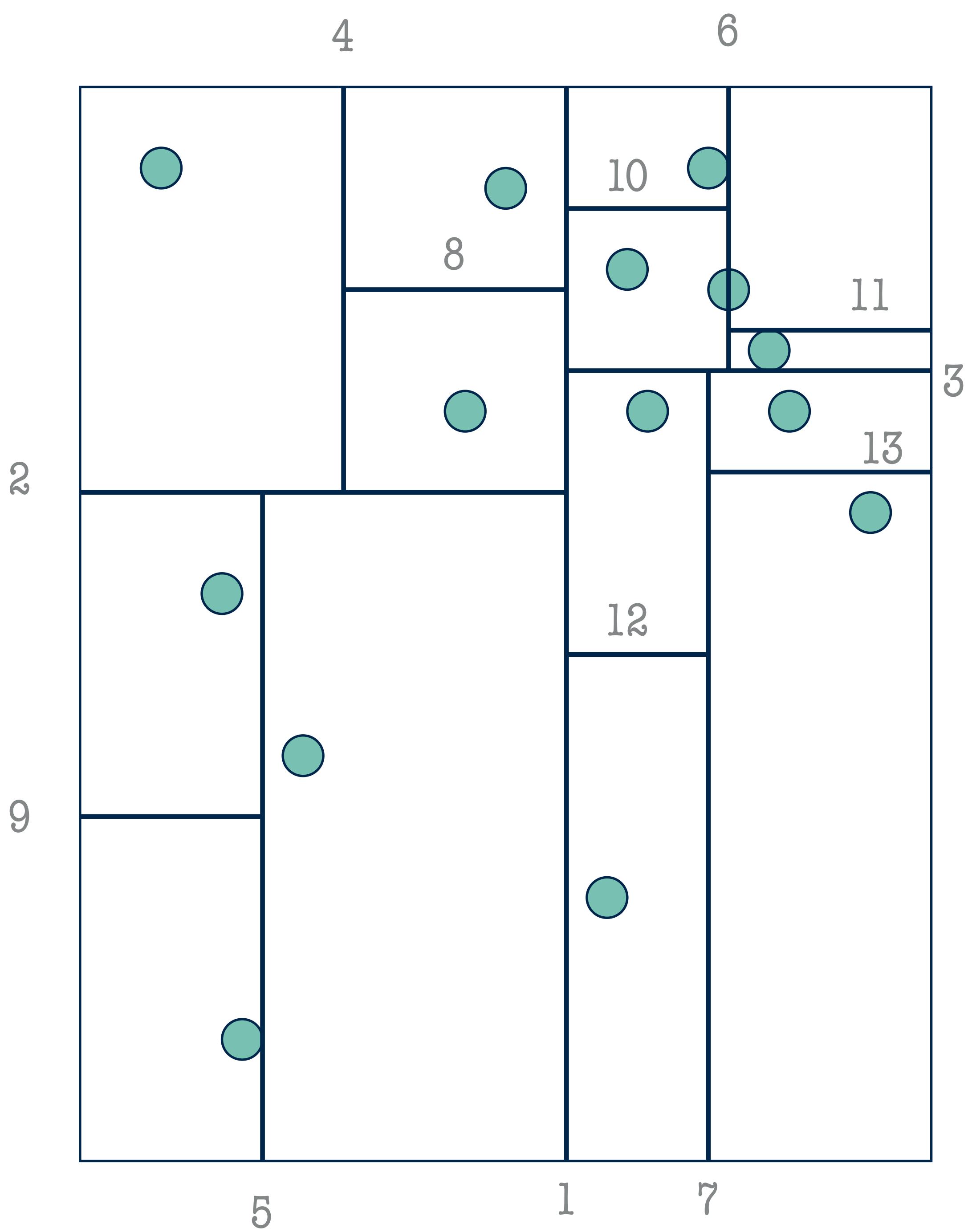


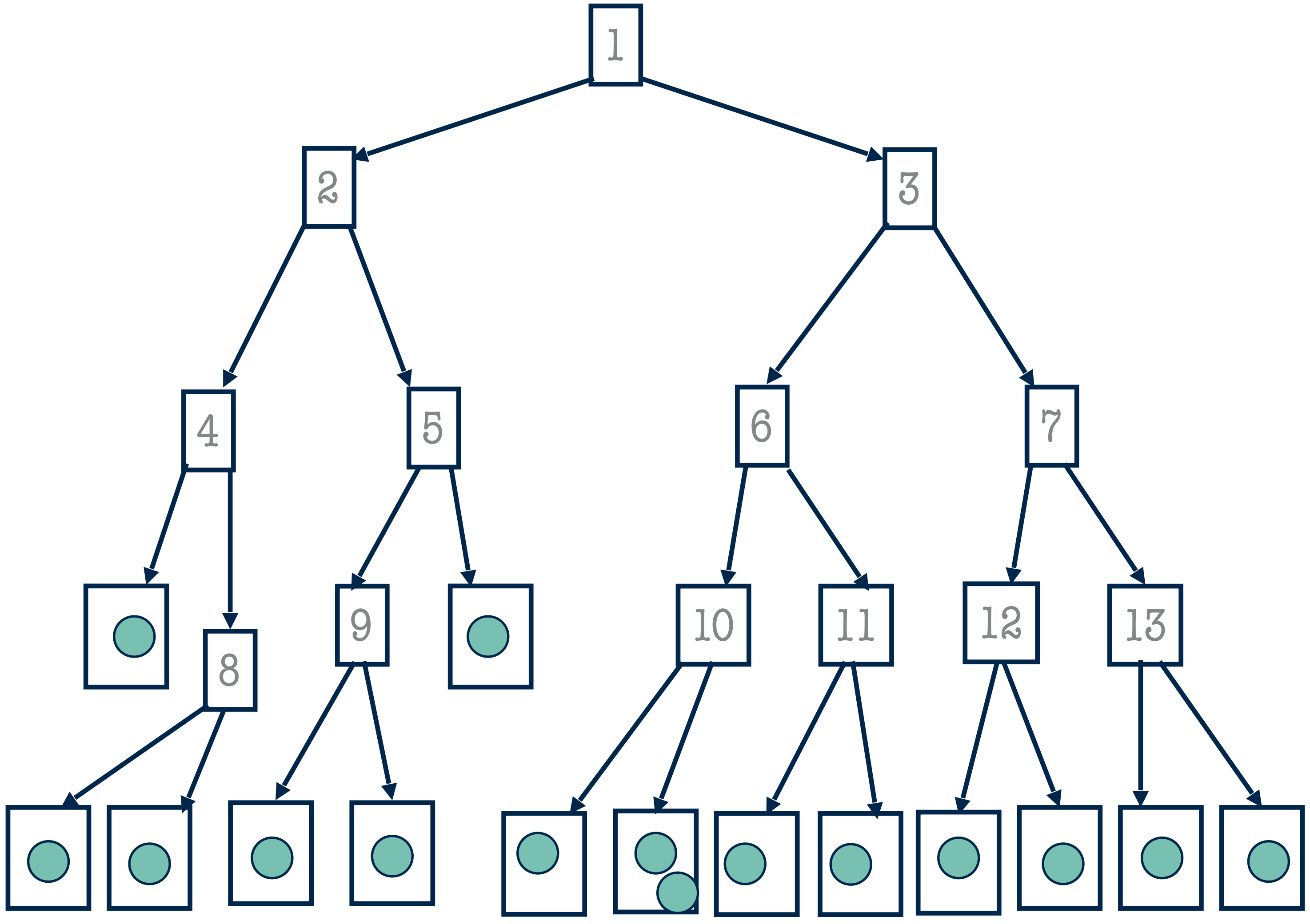
A bad octree case

KD-TREES

- A kd-tree has following properties:
  - Each node represents a rectilinear region (faces aligned with axes)
  - Each node is associated with an axis-aligned plane that cuts its region into two
  - Each node has a child for each sub-region
  - The directions of the cutting planes alternate with depth
- Kd-trees generalize octrees by allowing splitting planes at variable positions
  - Note that cut planes in different sub-trees at the same level need not be the same

KD-TREE EXAMPLE

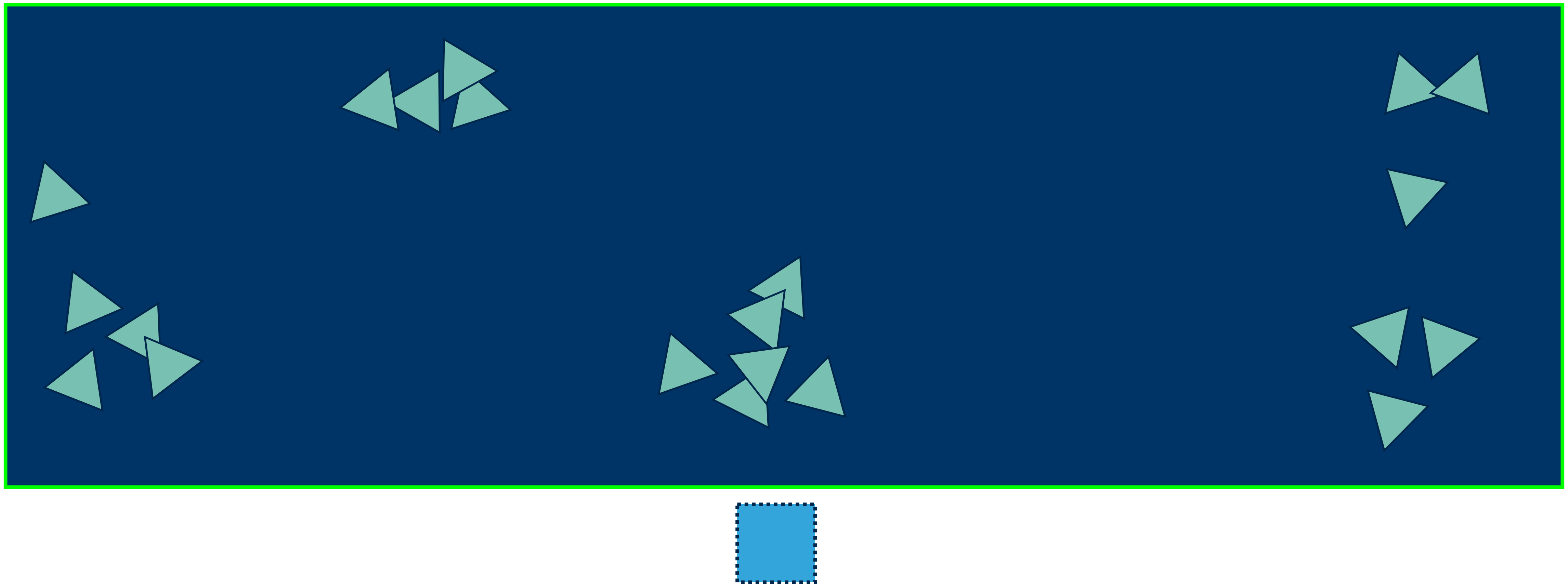




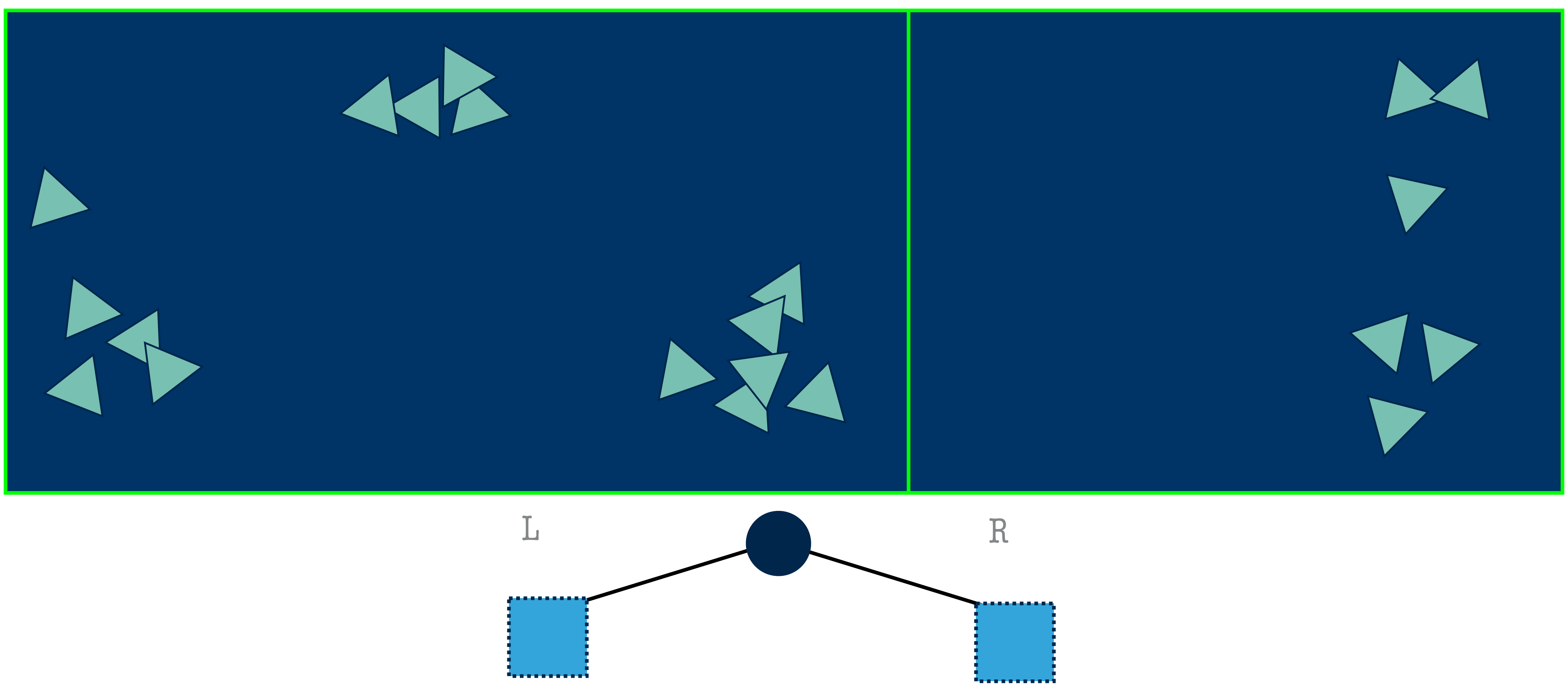
KD-TREE NODE DATA STRUCTURE

- What needs to be stored in a node?
  - Children pointers (always two)
  - Parent pointer - useful for moving about the tree
  - Extents of cell - x_{max}, x_{min}, y_{max}, y_{min}, z_{max}, z_{min}
  - List of pointers to the contents of the cell
  - Neighbors are complicated in kd-trees, so typically not stored

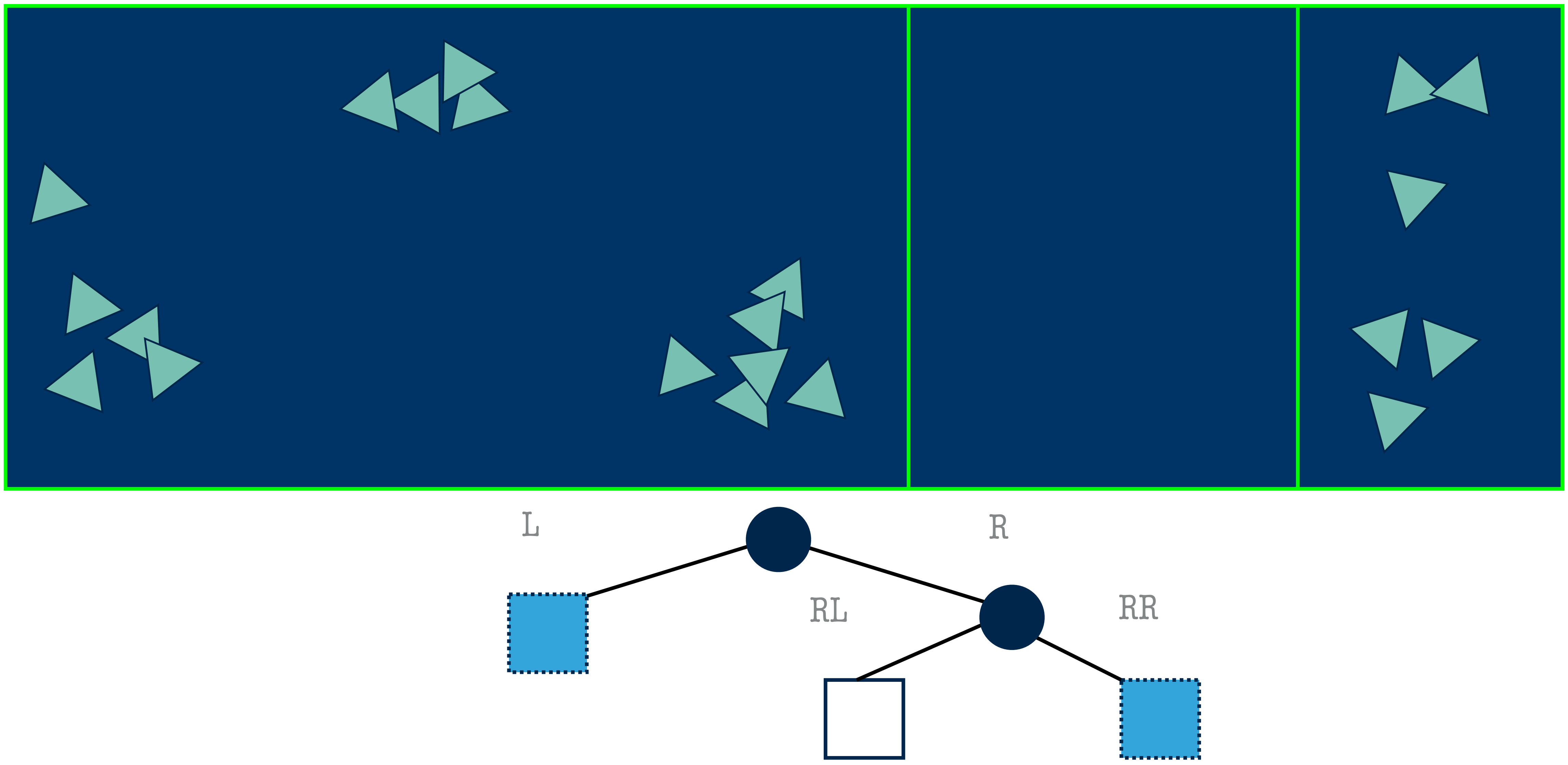
KD-TREE - BUILD



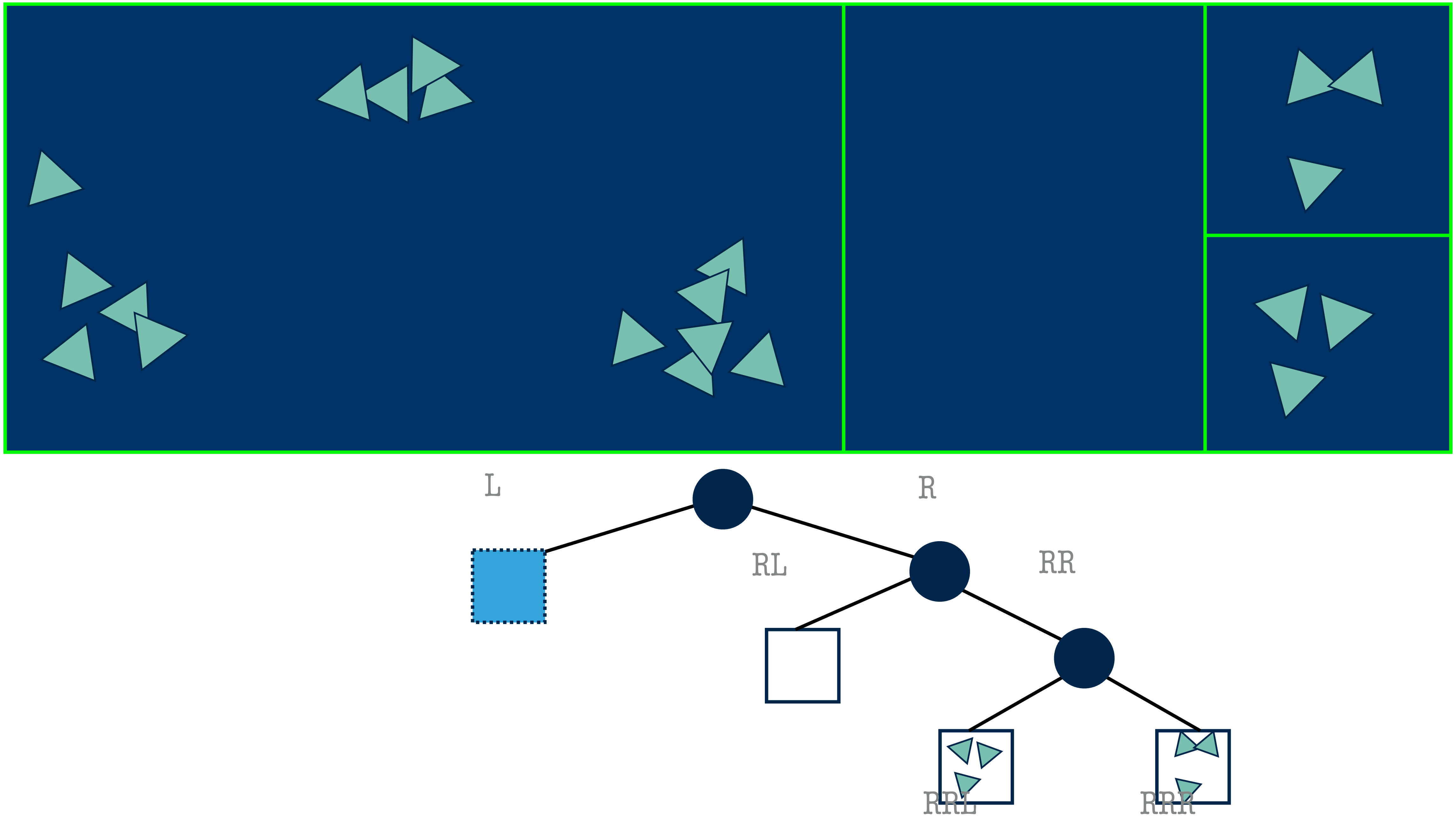
KD-TREE



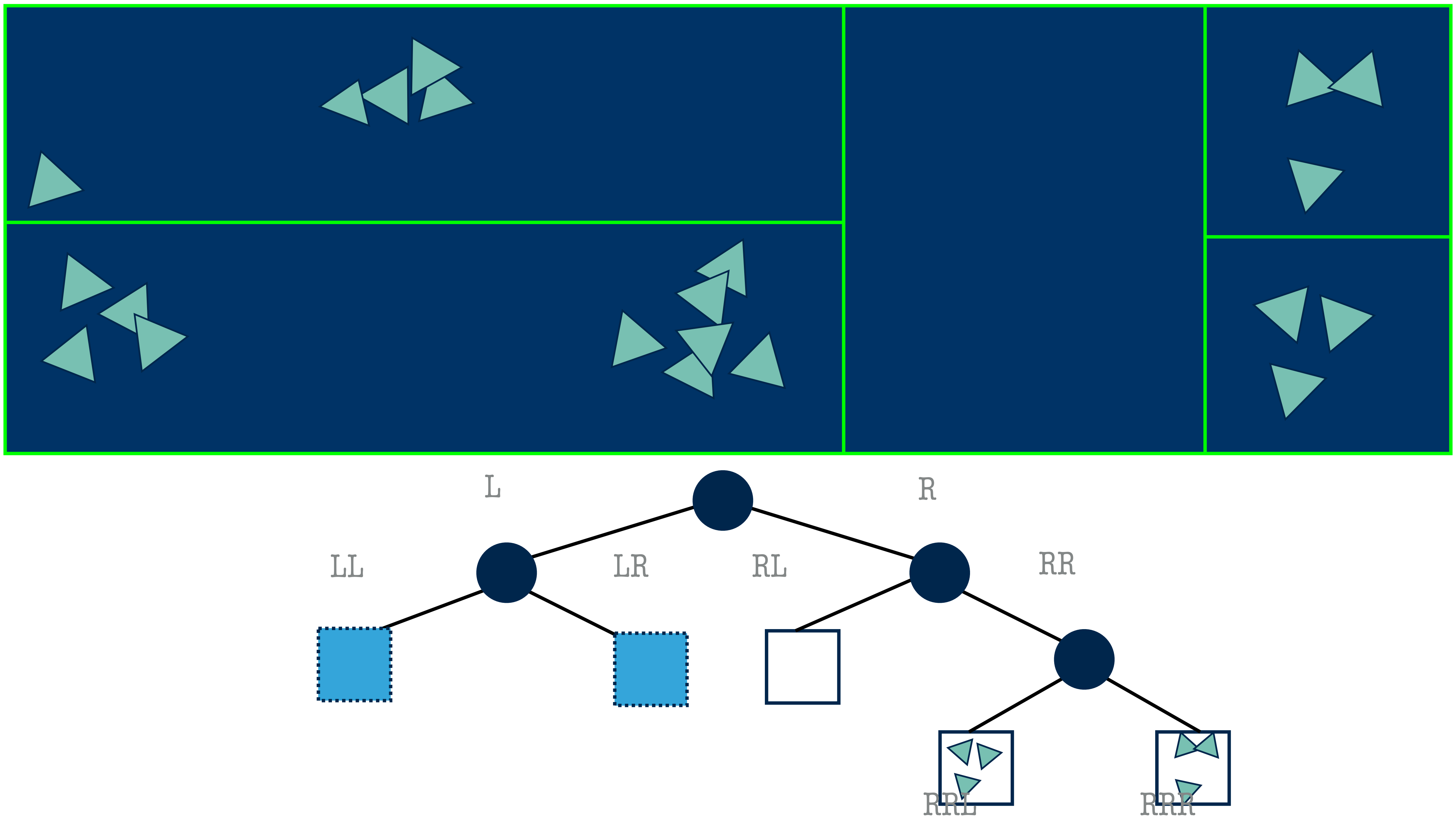
KD-TREE



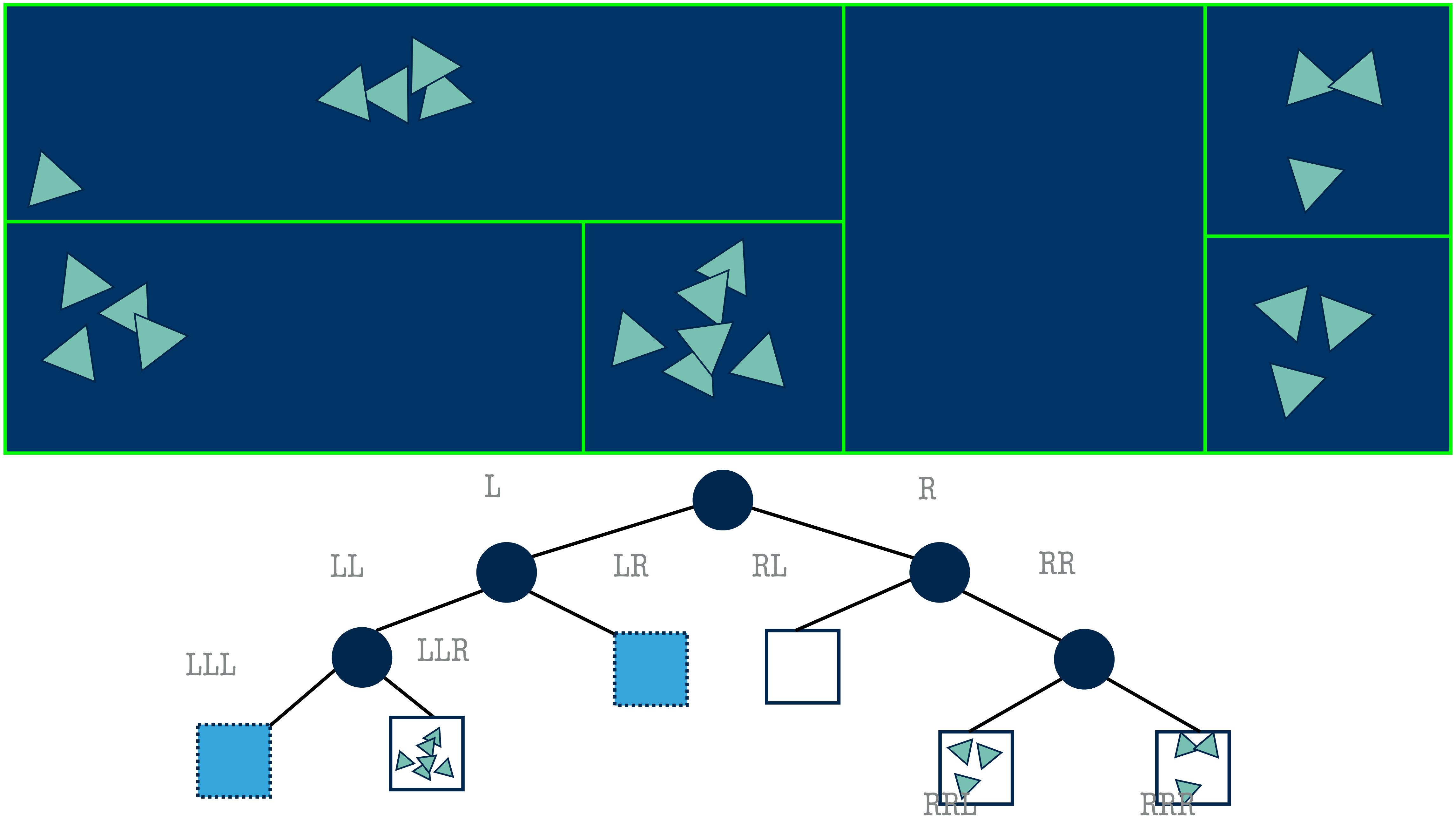
KD-TREE



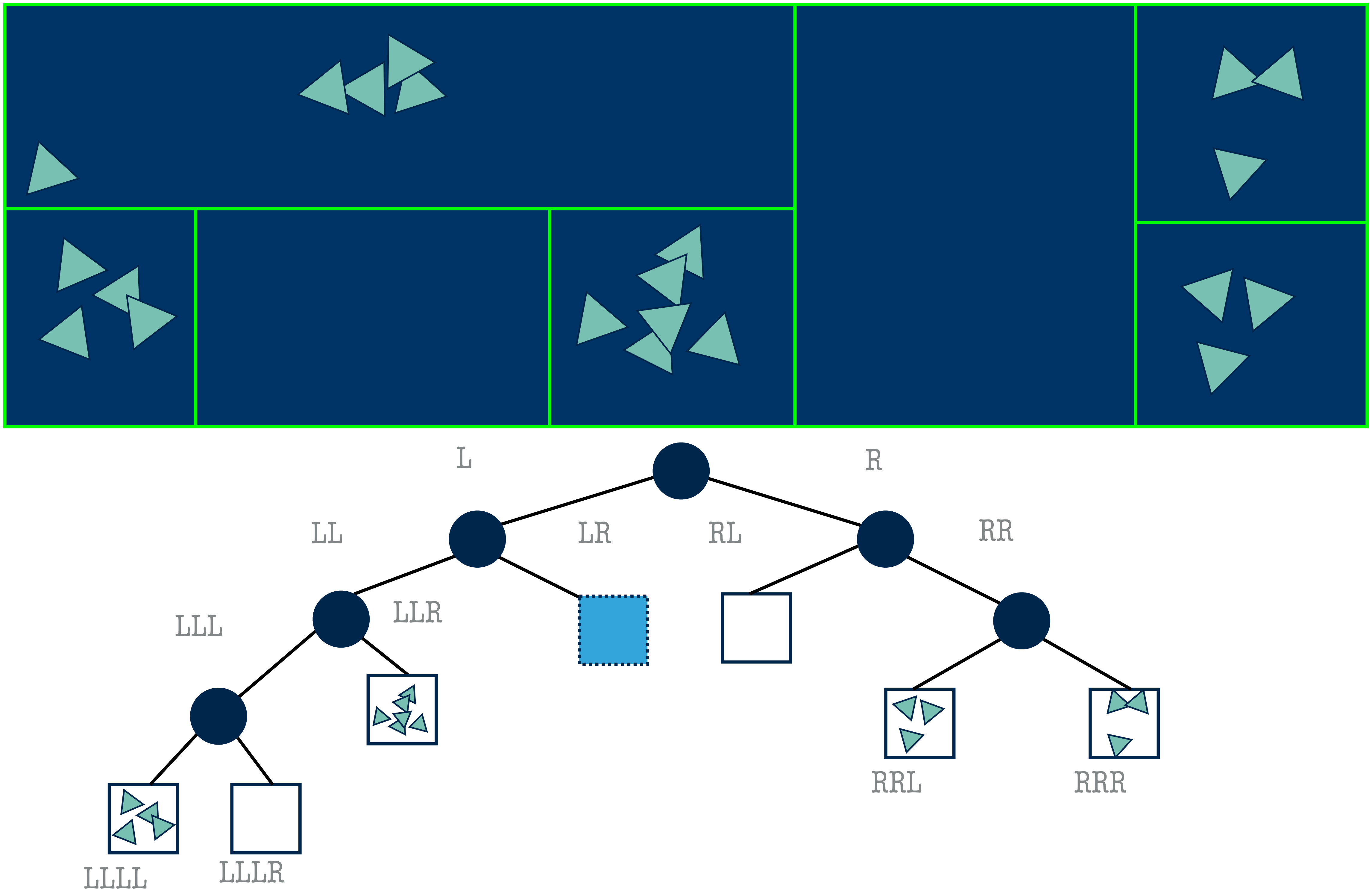
KD-TREE



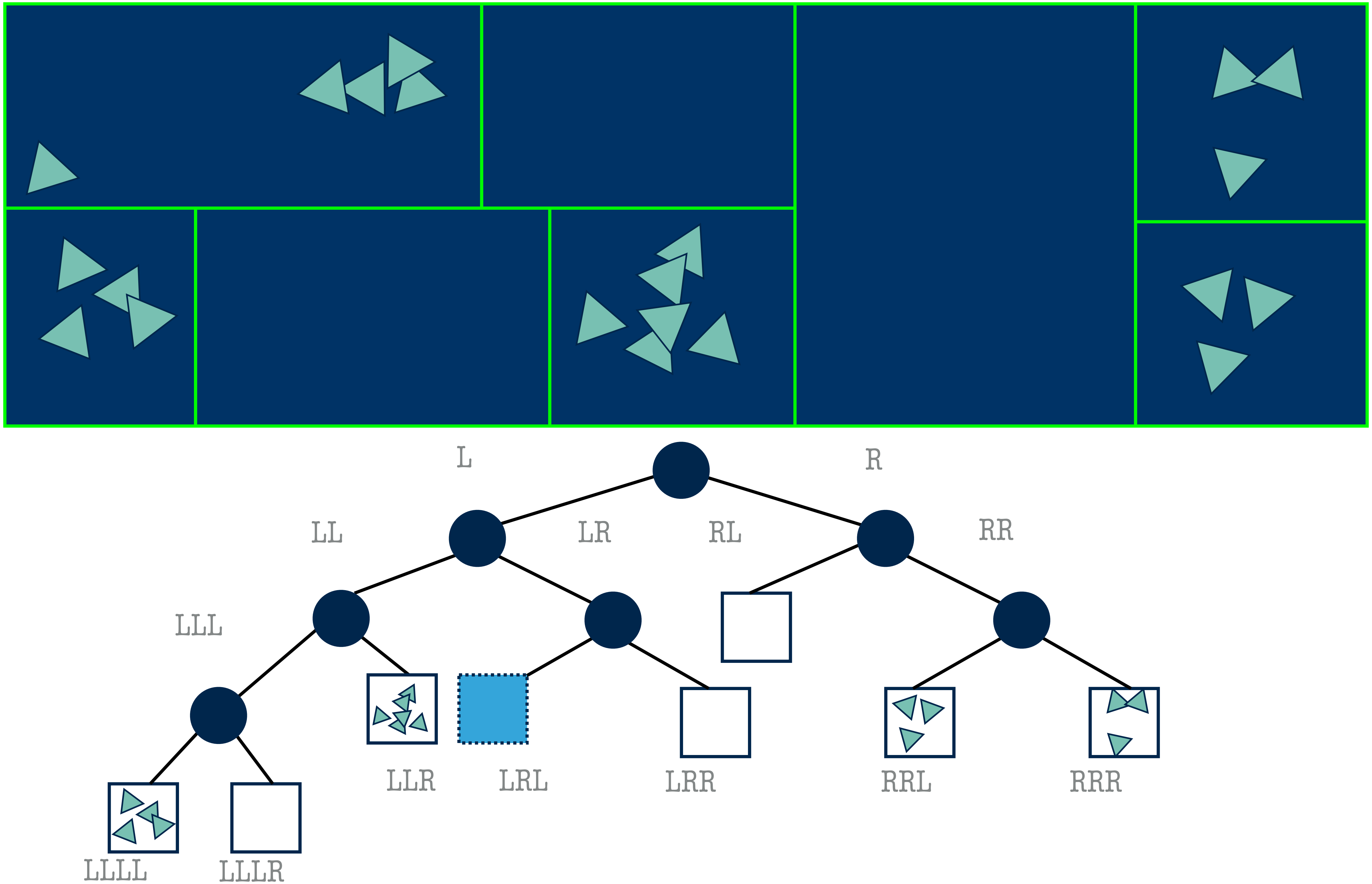
KD-TREE



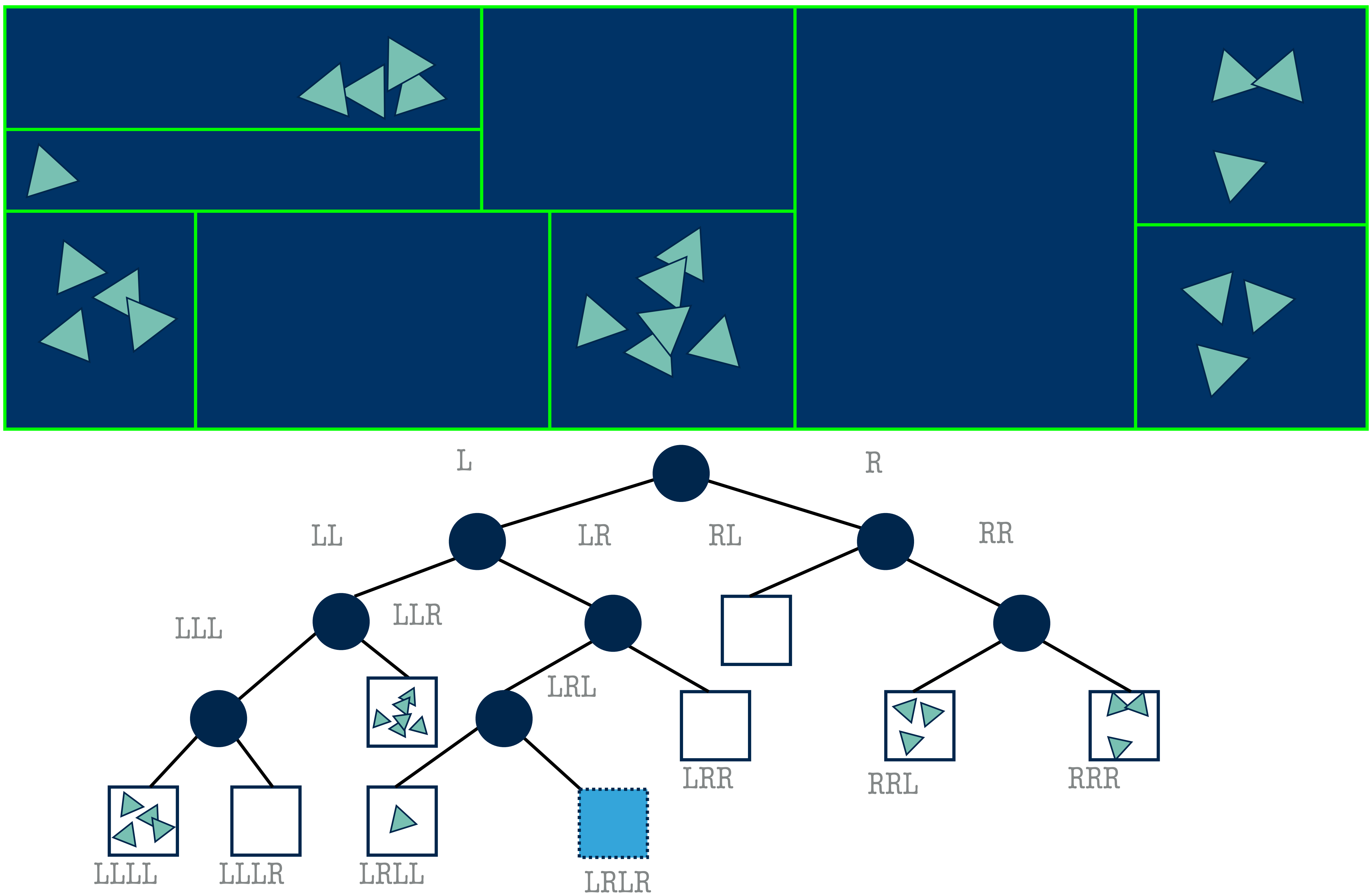
KD-TREE



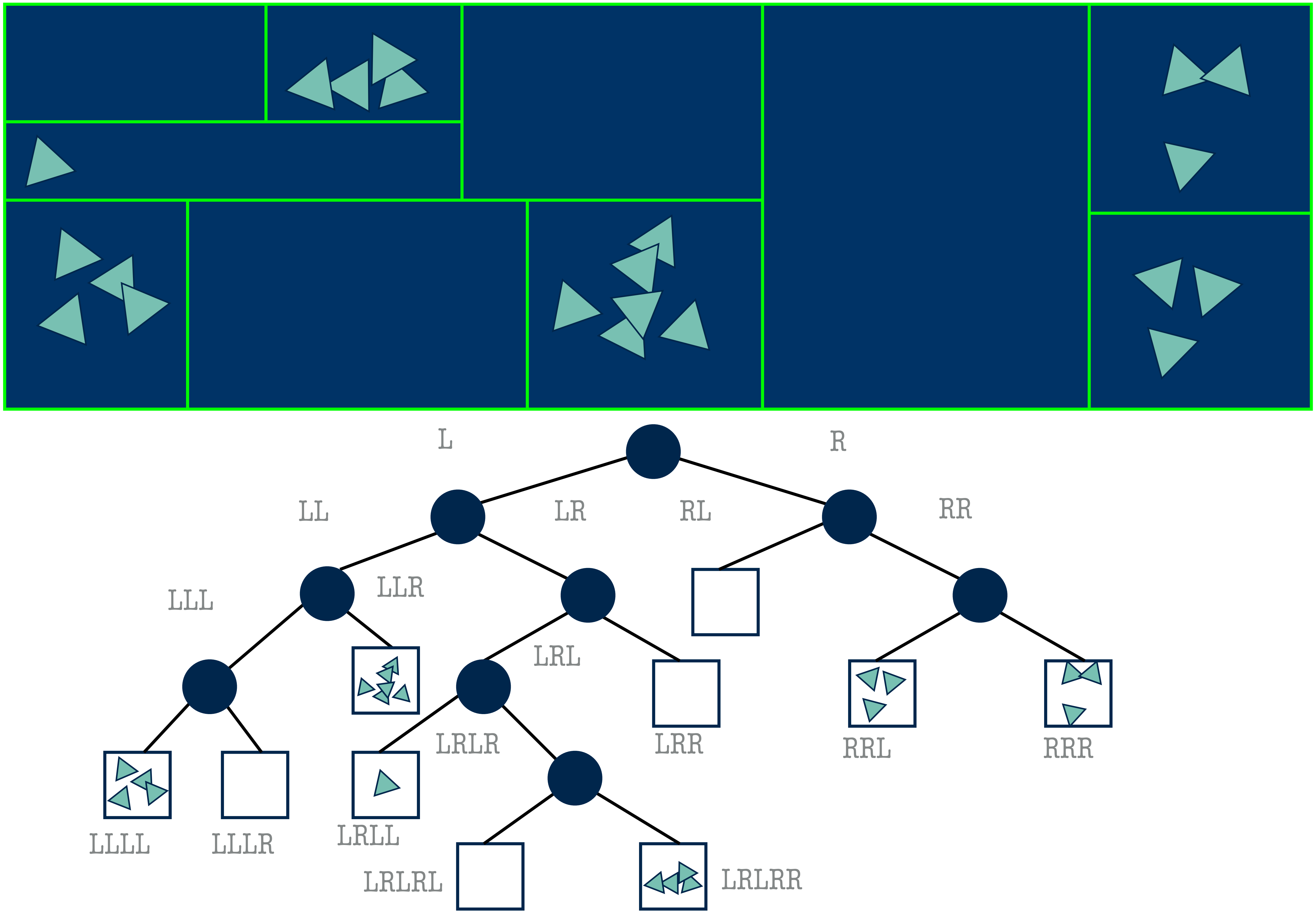
KD-TREE



KD-TREE



KD-TREE

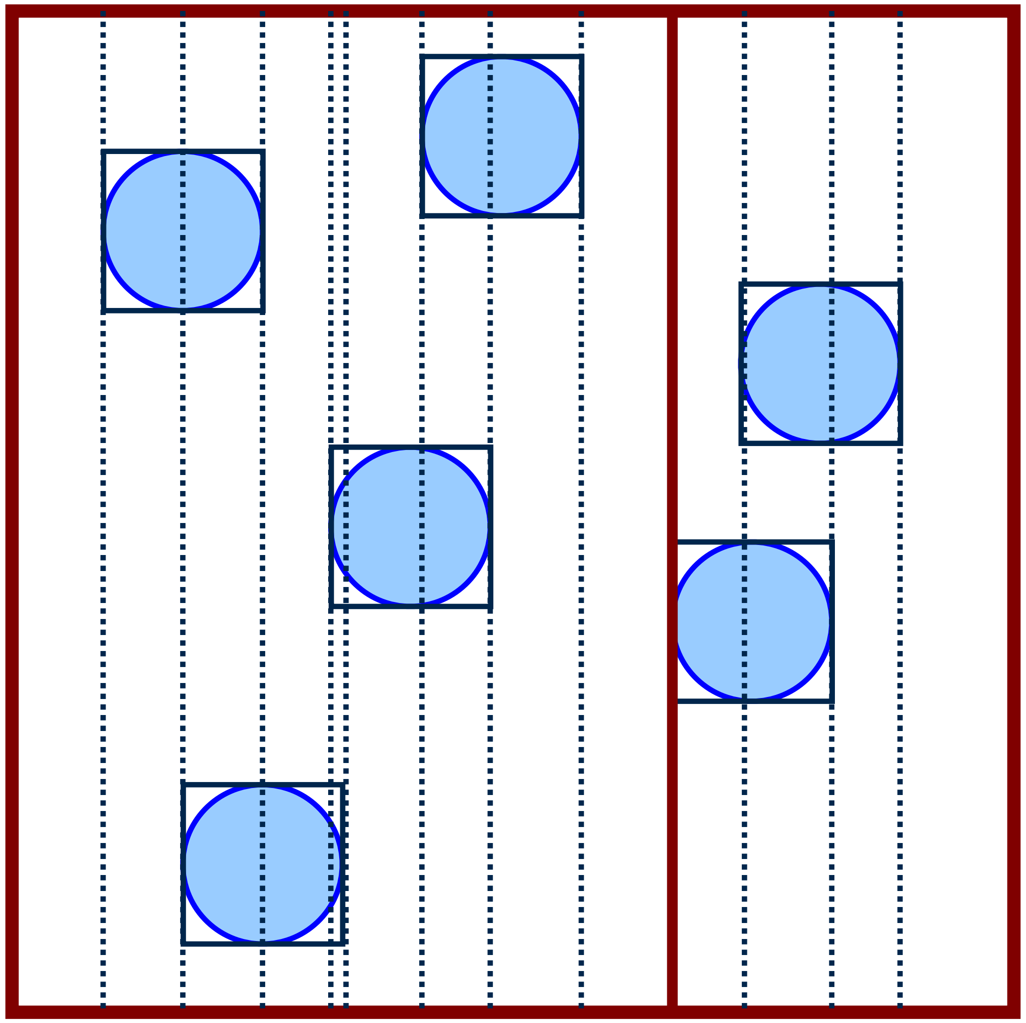

L
R
LL
LR
RL
RR
LLL
LLR
LRL
LLLL
LLLR
LRLR
LRLL
LRR
RRL
RRR
LRLRL
LRLRR

CHOOSING A SPLIT PLANE

- Goals in selecting a splitting plane for each cell:
  - Minimize the number of objects cut by the plane
  - Balance the tree: Use the plane that equally divides the objects into two sets (the median cut plane)
- Generally NP-complete, so we approximate
  - Axis-Aligned Bounding Boxes (AABBs)
  - Suface Area Heuristic

COMMON APPROXIMATIONS

- Axis-Aligned Bounding Boxes (AABBs)
  - Simplify objects to “fat points”
  - Reduces candidate split planes
- Surface Area Heuristic (SAH)
  - Greedy strategy to estimate traversal cost



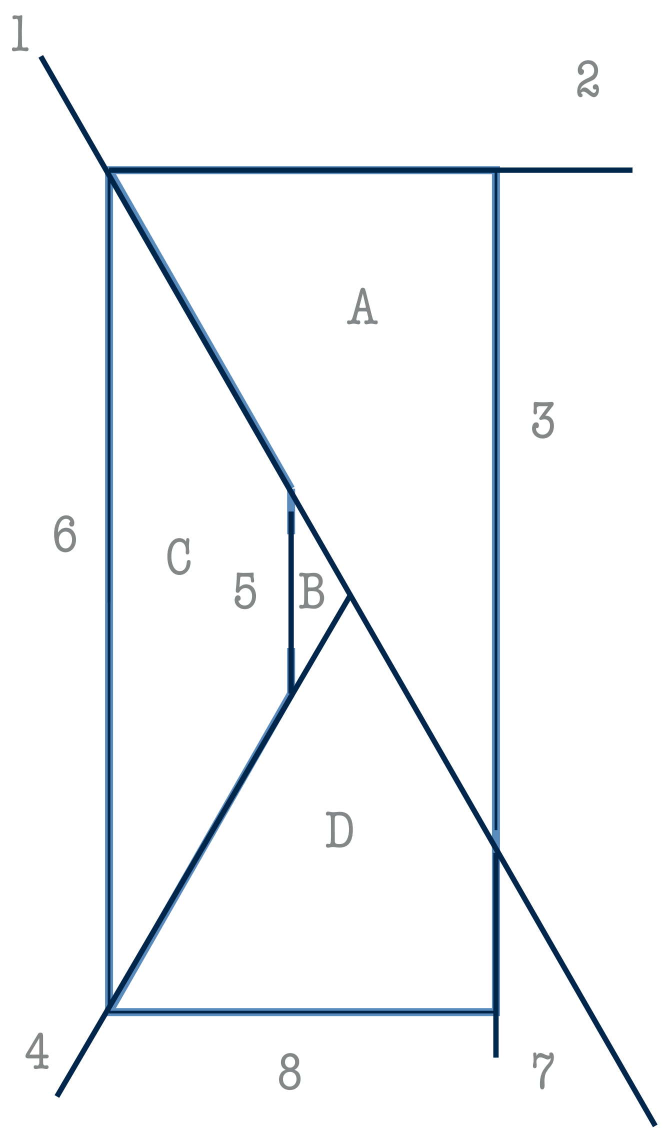
KD-TREE APPLICATIONS

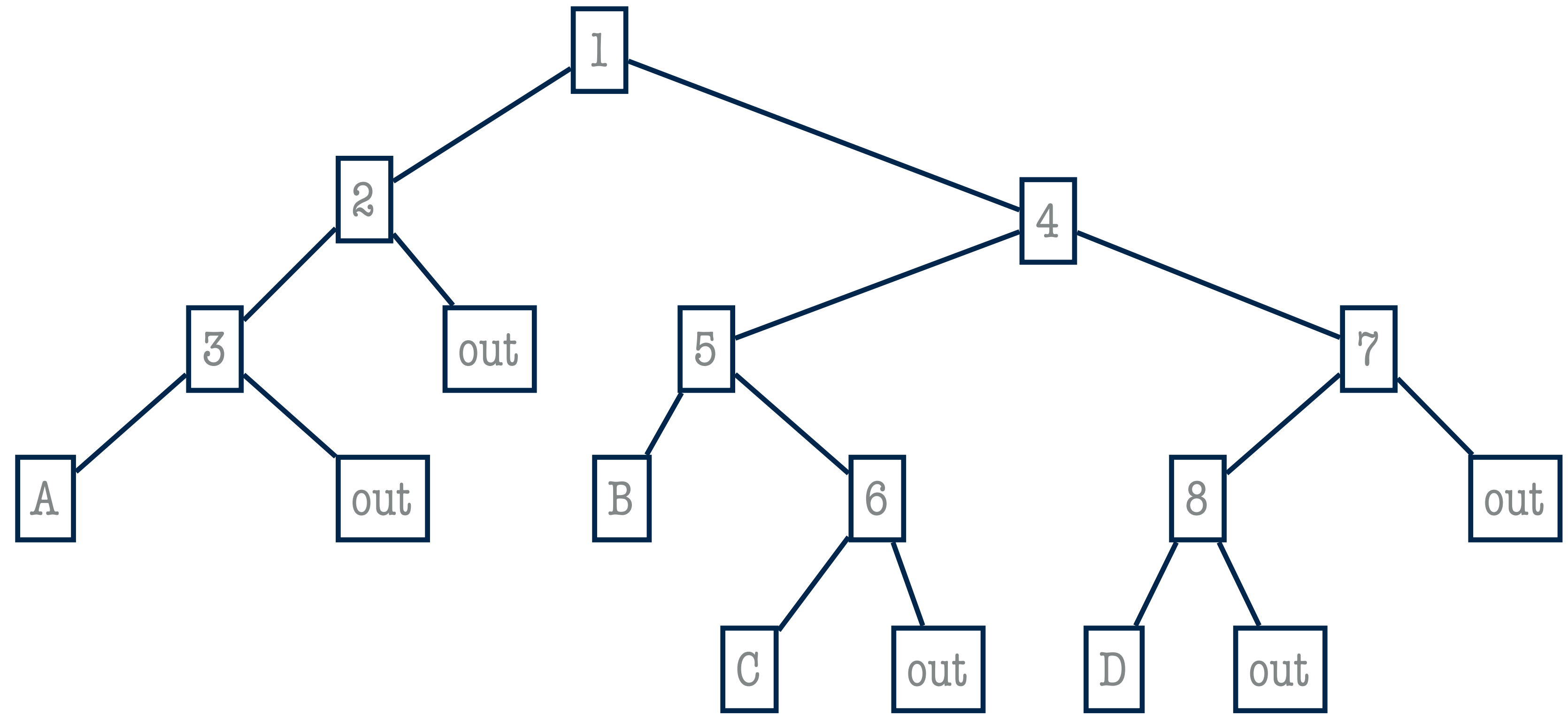
- Kd-trees work well when axis aligned planes cut things into meaningful cells
- View frustum culling extends trivially to kd-trees
- Kd-trees are frequently used as data structures for other algorithms – particularly in visibility

BSP TREES

- Binary Space Partition trees
  - Sequence of cuts that divide a region of space into two
- Cutting planes can be of any orientation
  - Generalization of kd-trees (kd-tree is an axis-aligned BSP tree)
- Divides space into convex cells
- Industry standard for spatial subdivision in many game environments
  - General enough to handle most common environments
  - Easy enough to manage and understand
  - Big performance gains

BSP EXAMPLE





- Notes:
  - Splitting planes end when they intersect their parent node's planes
  - Internal node labeled with planes, leaf nodes with regions

BSP TREE NODE DATA STRUCTURE

- What needs to be stored in a node?
  - Children pointers (always two)
  - Parent pointer
  - If a leaf node: Extents of cell
  - If an internal node: The split plane
  - List of pointers to the contents of the cell
  - Neighbors are useful in many algorithms
    - Store neighbors at leaf nodes
    - Cells can have many neighboring cells
  - Portals are also useful (holes that see into neighbors)

CHOOSING SPLITTING PLANES

- Goals:
  - Trees with few cells
  - Planes that are mostly opaque (best for visibility calculations)
  - Objects not split across cells
- Some heuristics:
  - Choose planes that are also polygon planes
  - Choose large polygons first
  - Choose planes that don't split many polygons
  - Choose planes that evenly divide the data
  - User selects or otherwise guides the splitting process
  - Random choice of splitting planes doesn't do too badly

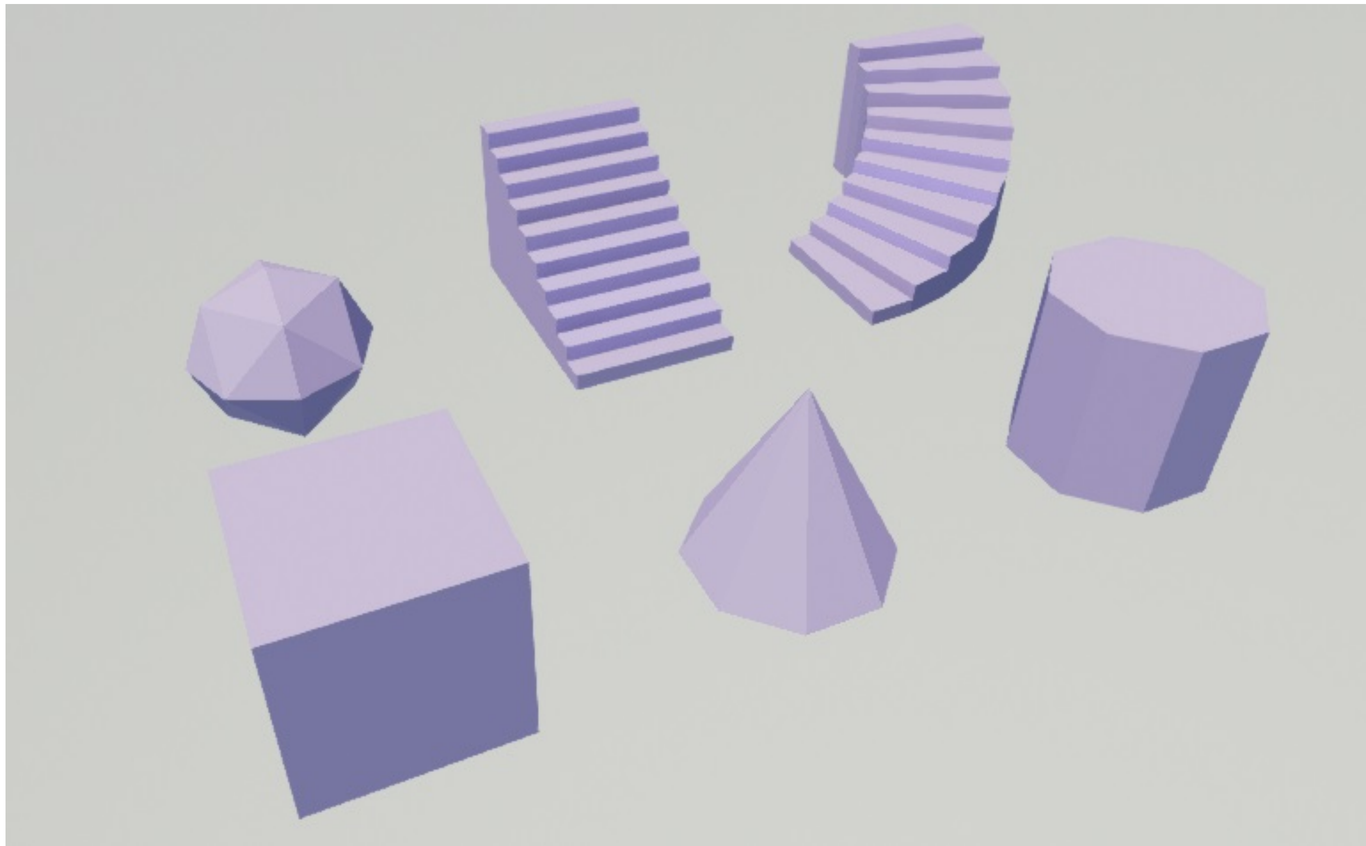
DRAWING ORDER FROM BSP TREES

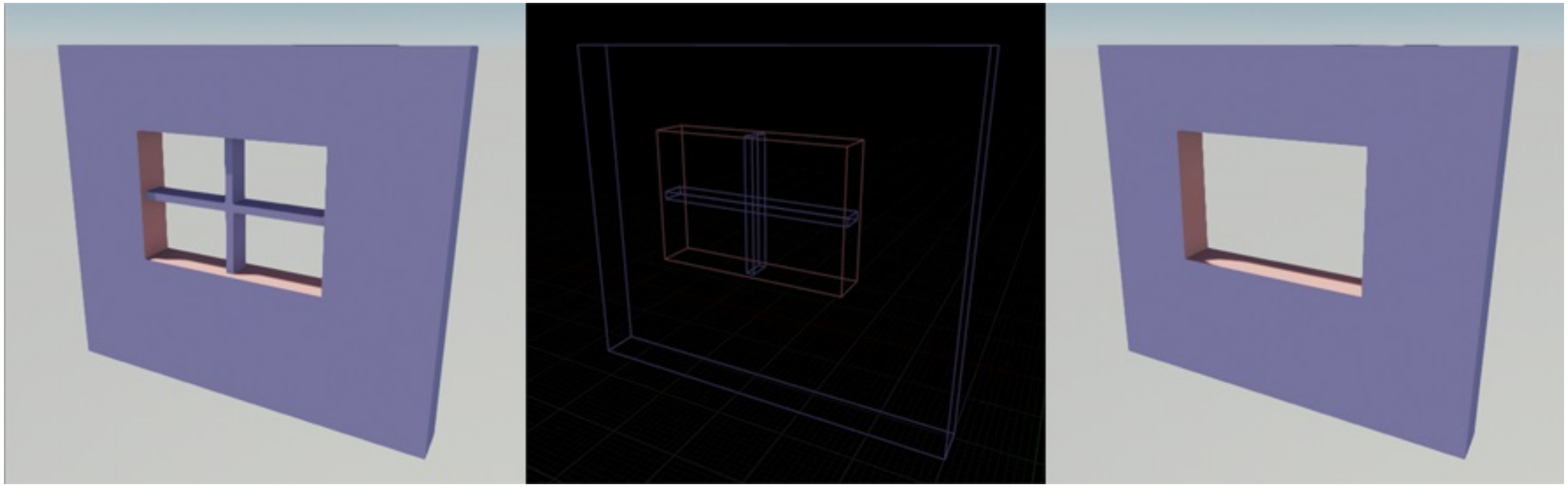
- BSP trees can order polygons from back to front, or visa-versa
  - Descend tree with viewpoint
  - Things on the same side of a splitting plane as the viewpoint are always in front of things on the far side
- Can draw from back to front
  - Removes need for z-buffer (but few people care any more)
  - Gives the correct order for rendering transparent objects with a z-buffer, and by far the best way to do it
- Can draw front to back
  - Use info from front polygons to avoid drawing back ones
  - Useful in software renderers

BSPS IN GAMES

- BSP trees can partition space as you would with an octree or kd-tree
  - Leaf nodes are cells with lists of objects
  - Cells typically correspond to “rooms” but don’t have to
  - Fast visibility and ray-trace queries
- Polygons used in the partitioning are defined by the level designer
  - A **brush** is a region of space that contributes planes to the BSP
  - Artists lay out brushes, then populate them with objects
  - Additional planes may be specified
    - Sky planes for outdoor scenes to block off visibility
    - Planes defined to block sight-lines, but not visible themselves

BSP BRUSHES IN UE4

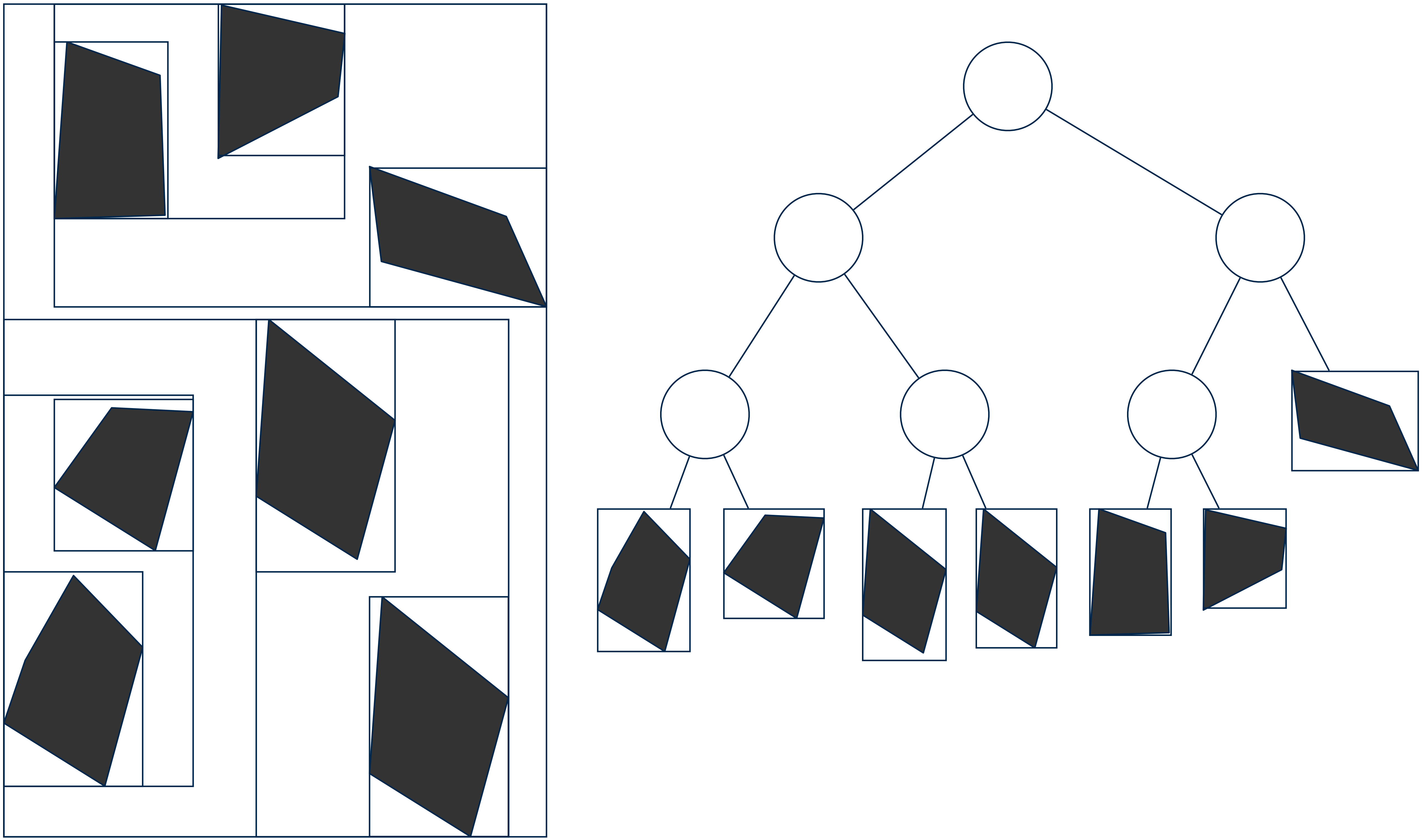




BOUNDING VOLUME HIERARCHIES

- BVHs have a bounding volume for each object
  - Spheres, AABBs etc
- Parent bounds bound their children's bounds
  - Children bounds the same type as their parent's
  - Fixed or variable number of children per node
- No notion of cells

BVH EXAMPLE



BVH OPERATIONS

- Some of the operations work with BVHs
  - Frustum culling
  - Collision detection
- BVHs are good for moving objects
  - Updating the tree is easier than for other methods
  - Incremental construction to avoid complete rebuilds
- BVHs lack some convenient properties
  - Not all space is filled so algorithms that "walk" through cells won't work