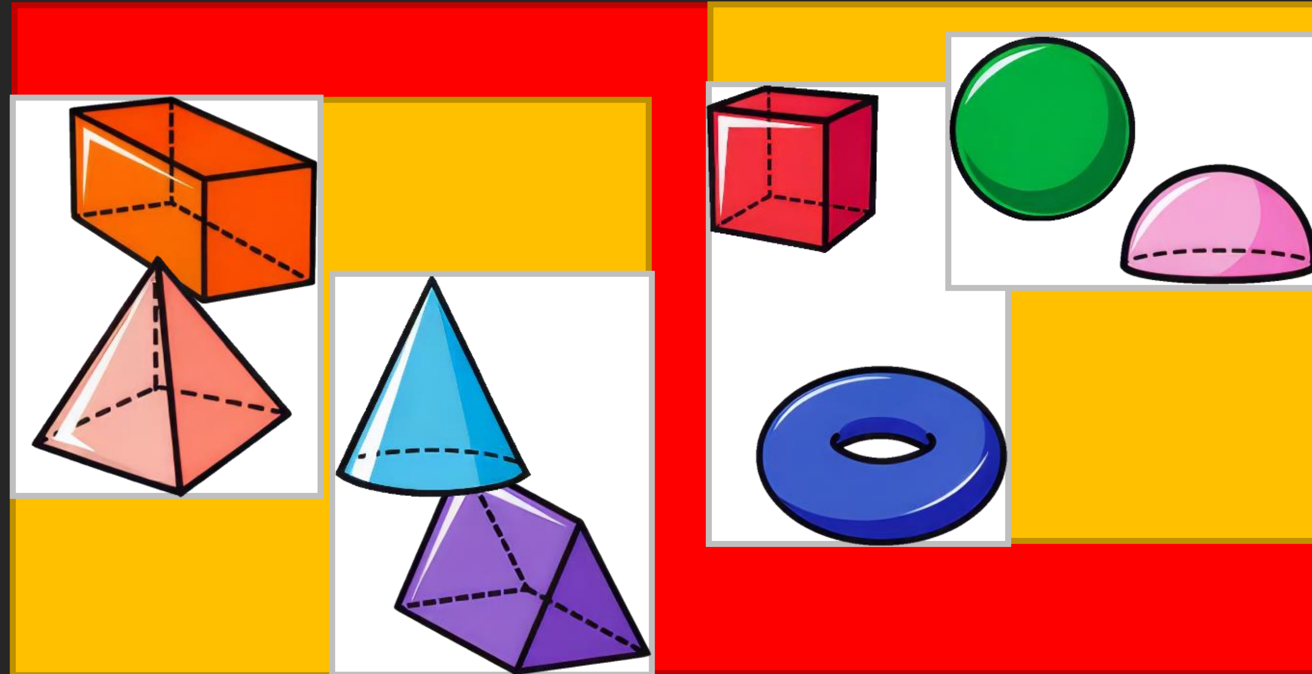
Bounding Volume Hierarchies

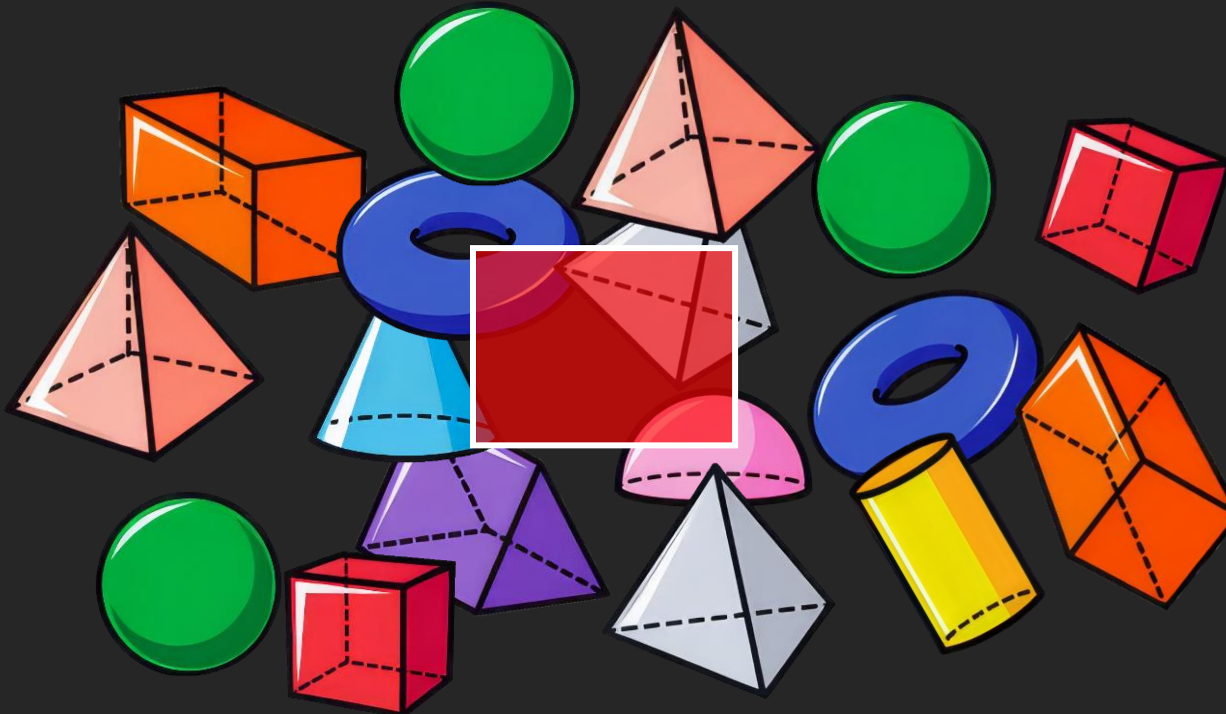


Matthias Müller, Ten Minute Physics

matthiasmueller.info/tenMinutePhysics

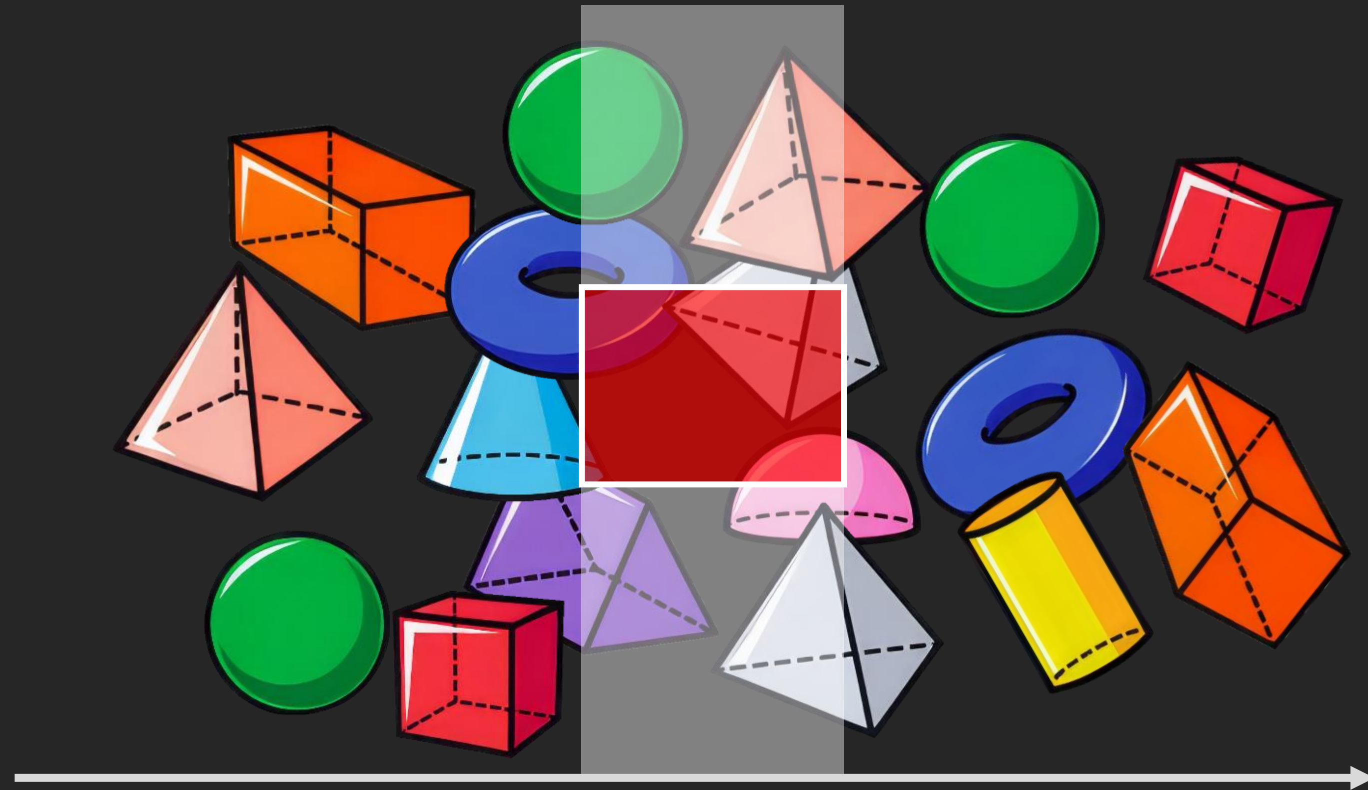
Problem

- Find all objects overlapping a region



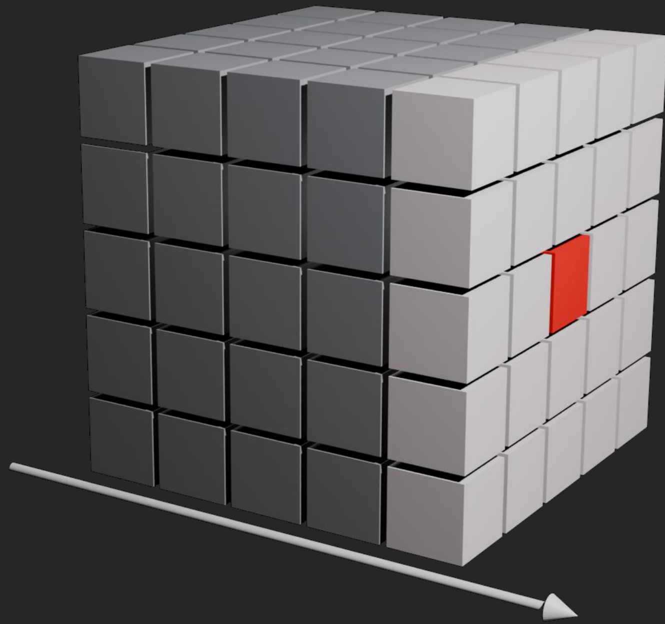
- Region = bounding box of an object → pair collision detection

Sweep and Prune 2d



- 100 x 100 objects in 2d:
  ~ 100 tests per object

Sweep and Prune 3d



- 100 x 100 x 100 objects in 3d:
  ~ 100 x 100 tests per object

- Problem: uses only one axis

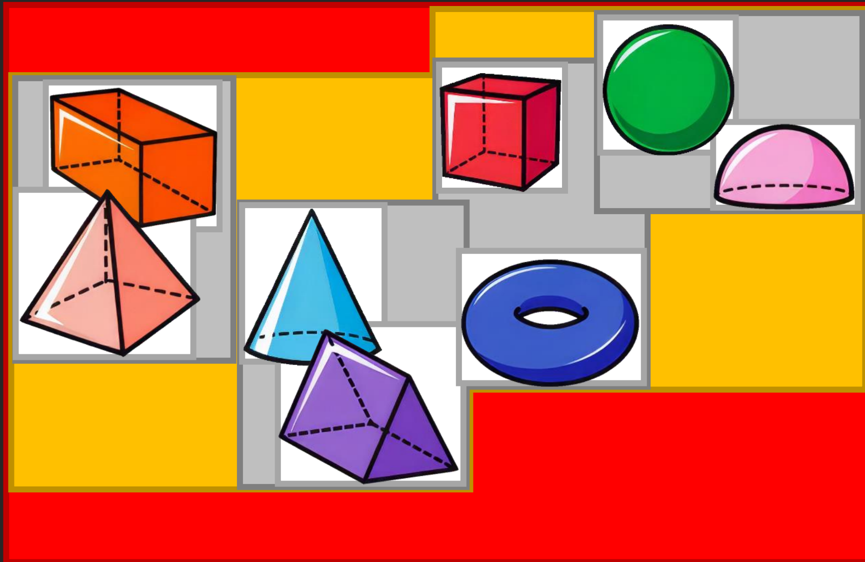
Bounding Volume Hierarchy

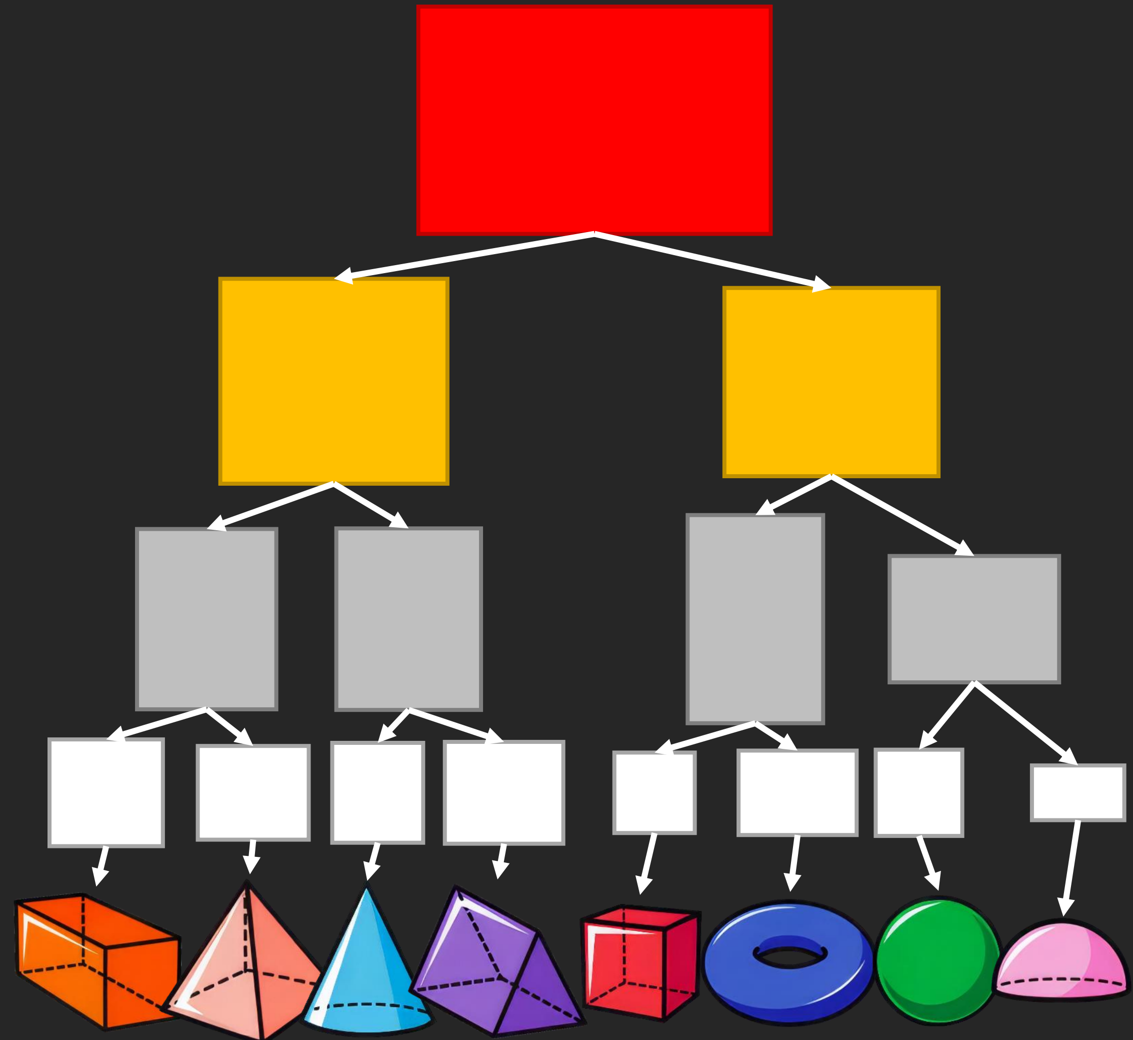
- n objects: $\sim\log_2(n)$ tests per object

- 2d: 100 x 100 objects
  - Brute force: 10,000 test per object: 100,000,000 tests
  - SAP: ~100 test per object: 1,000,000 tests
  - BVH: ~13 test per object: 130,000 tests

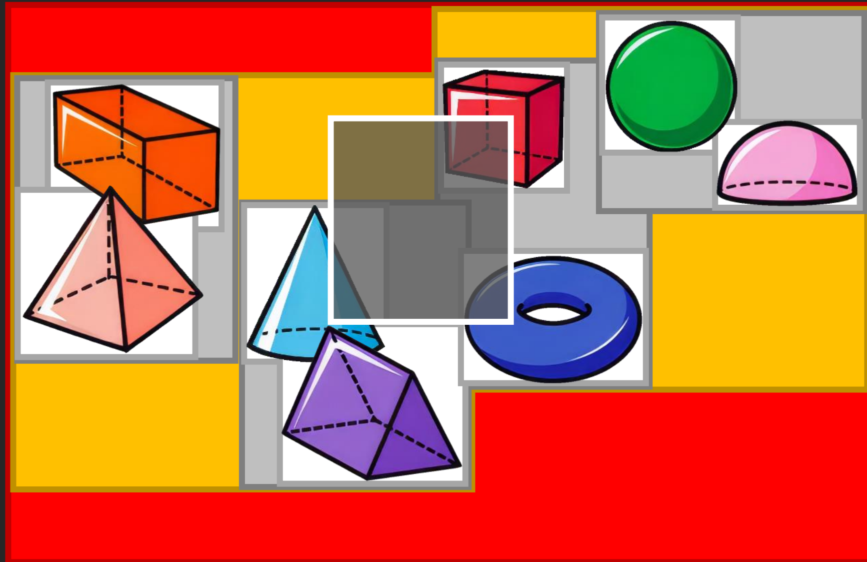
- 3d: 100 x 100 x 100 objects
  - Brute force: 1,000,000 1,000,000,000,000 tests
  - SAP: ~10’000 test per object: 10,000,000,000 total test
  - BVH: ~20 test per object: 20,000,000 total tests

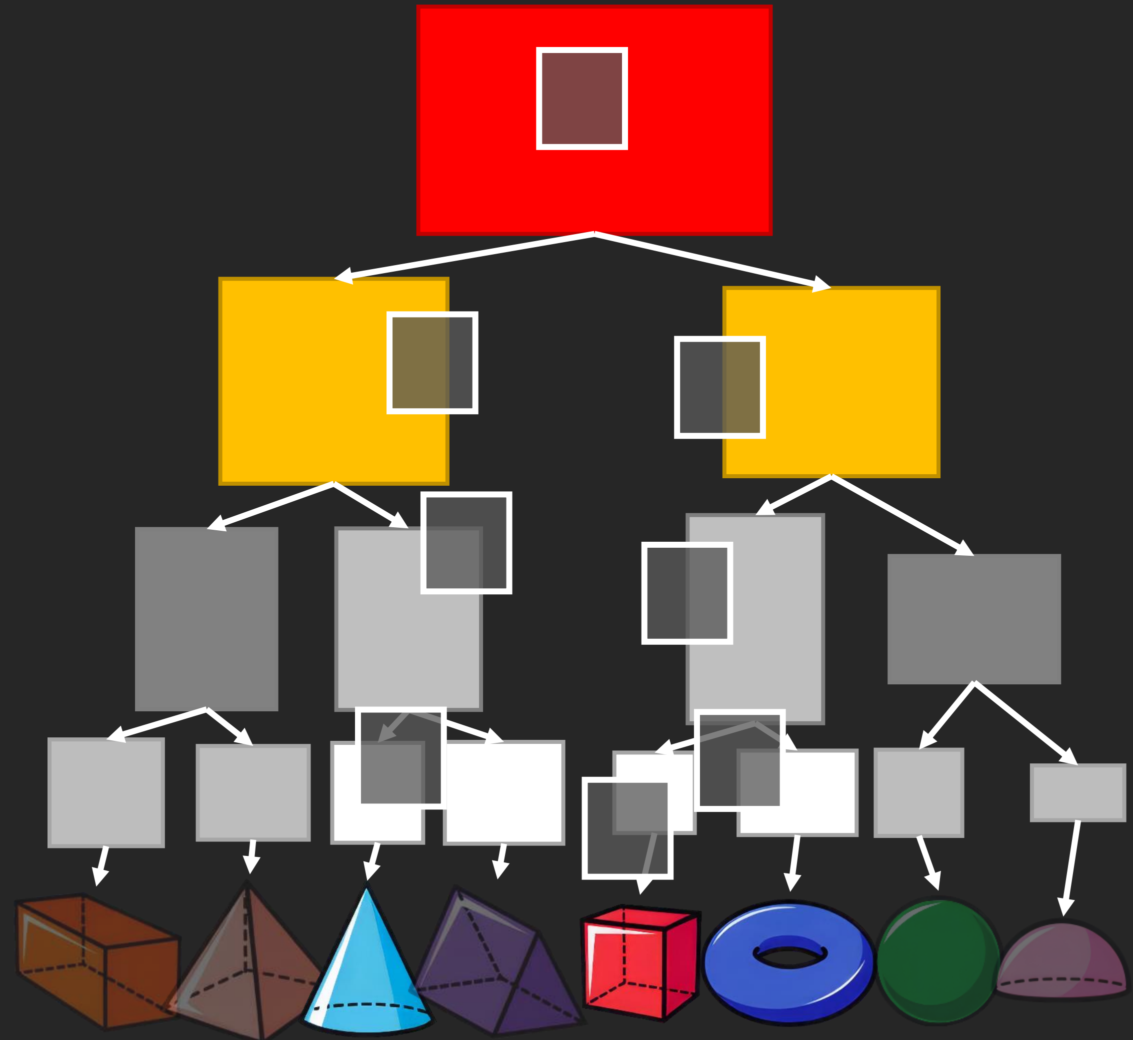
Data Structure



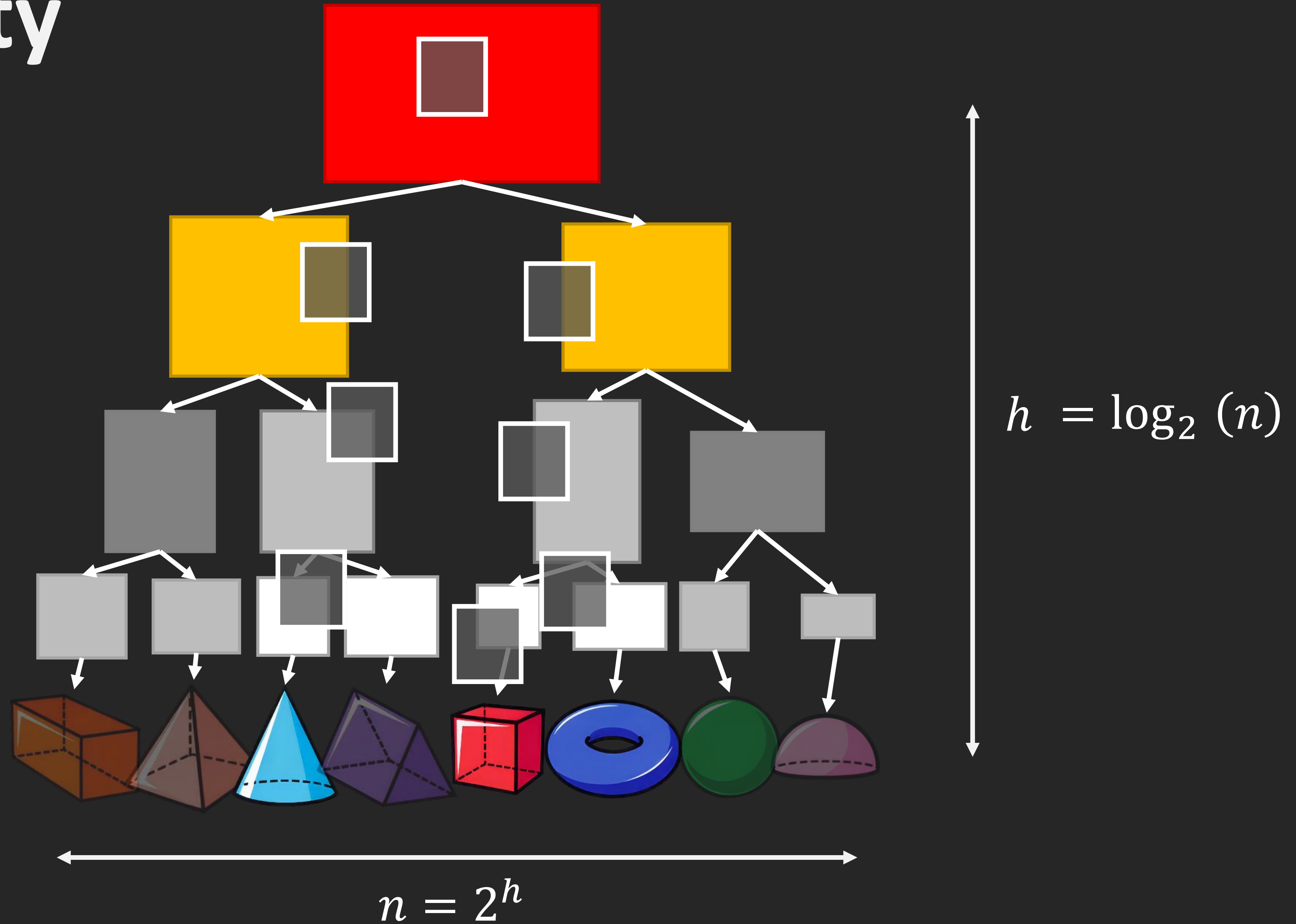


Query

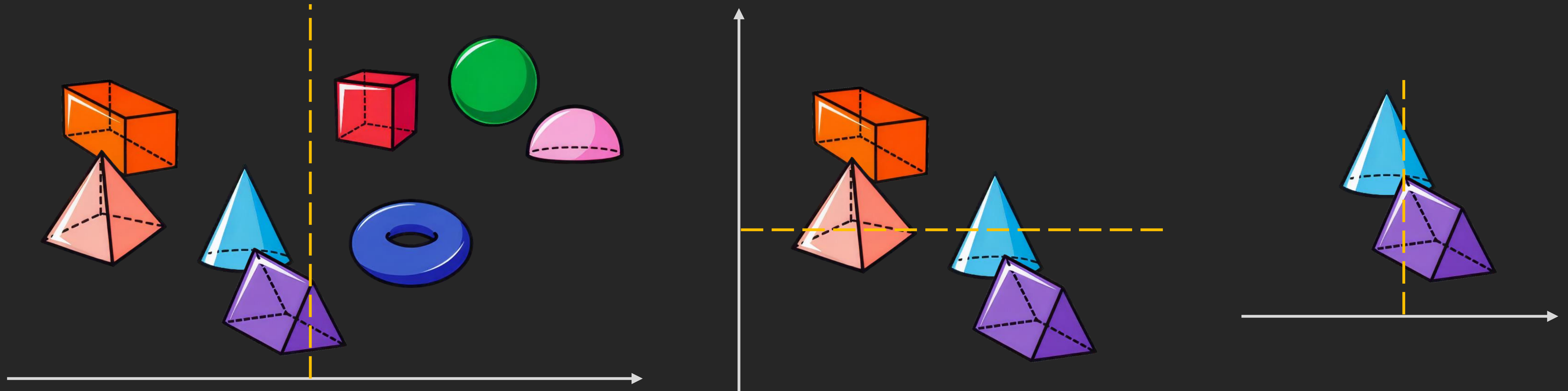




Complexity



Creation

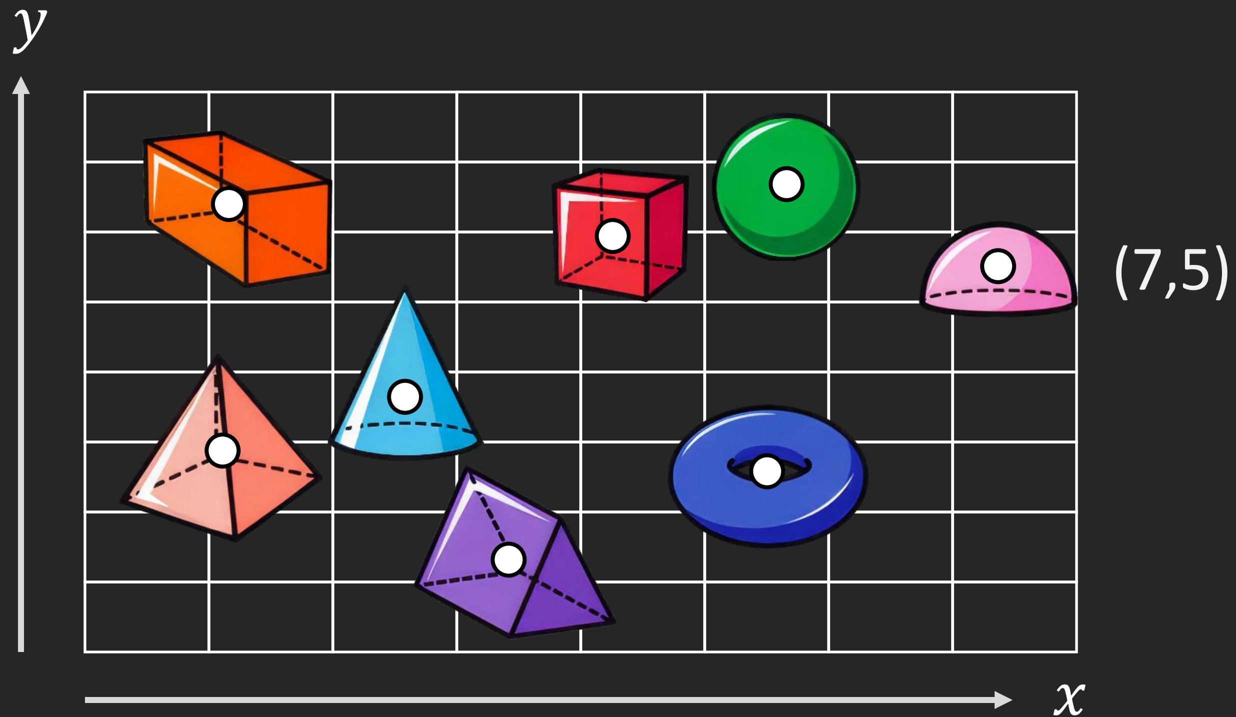


- Top down:
  - Alternate axes
  - Sort objects, split into equal sets
  - Recurse on both sets

Dynamic Scenes

- Reorder parts of the tree dynamically
  - Complicated, heterogenious algorithm
  - Not well suited for parallel implementation
- Reconstruct from scratch
  - Every frame
  - Every n^{th} frame, expand bounds
  - Need of fast construction
  - Can we construct the entire tree with one sort?

Use Integer Coordinates



- Split global bounding box into $2^b \times 2^b$ virtual cells
- Replace object with center
- Compute integer coordinates

```
let xi = Math.floor((centerX - minX) / (maxX - minX) * (2 ** bits));
let yi = Math.floor((centerY - minY) / (maxY - minY) * (2 ** bits));
```

Morton Codes

- For a single sort we need one key from two coordinates
- Alternate axes → interleave bits:

Cell Morton Codes

010101	010111	011101	011111	110101	110111	111101	111111
010100	010100	011100	011110	110100	110110	111100	111110
010001	010011	011001	011011	110001	110011	111001	111011
010000	010010	011000	011010	110000	110010	111000	111010
000101	000111	001101	001111	100101	100111	101101	101111
000100	000110	001100	001110	100100	100110	101100	101110
000001	000011	001001	001011	100001	100011	101001	101011
000000	000010	001000	001010	100000	100010	101000	101010

Level 1 Split

010101	010111	011101	011111	110101	110111	111101	111111
010100	010100	011100	011110	110100	110110	111100	111110
010001	010011	011001	011011	110001	110011	111001	111011
010000	010010	011000	011010	110000	110010	111000	111010
000101	000111	001101	001111	100101	100111	101101	101111
000100	000110	001100	001110	100100	100110	101100	101110
000001	000011	001001	001011	100001	100011	101001	101011
000000	000010	001000	001010	100000	100010	101000	101010

Level 2 Split

010101	010111	011101	011111
010100	010100	011100	011110
010001	010011	011001	011011
010000	010010	011000	011010
000101	000111	001101	001111
000100	000110	001100	001110
000001	000011	001001	001011
000000	000010	001000	001010

Level 1 Split

110101	110111	111101	111111
110100	110110	111100	111110
110001	110011	111001	111011
110000	110010	111000	111010
100101	100111	101101	101111
100100	100110	101100	101110
100001	100011	101001	101011
100000	100010	101000	101010

Level 3 Split

010101	010111	011101	011111
010100	010100	011100	011110
010001	010011	011001	011011
010000	010010	011000	011010

Level 4 Split

011101	011111
011100	011110
011001	011011
011000	011010

Level 5 Split

011001	011011
011000	011010

Level 6 Split

011001

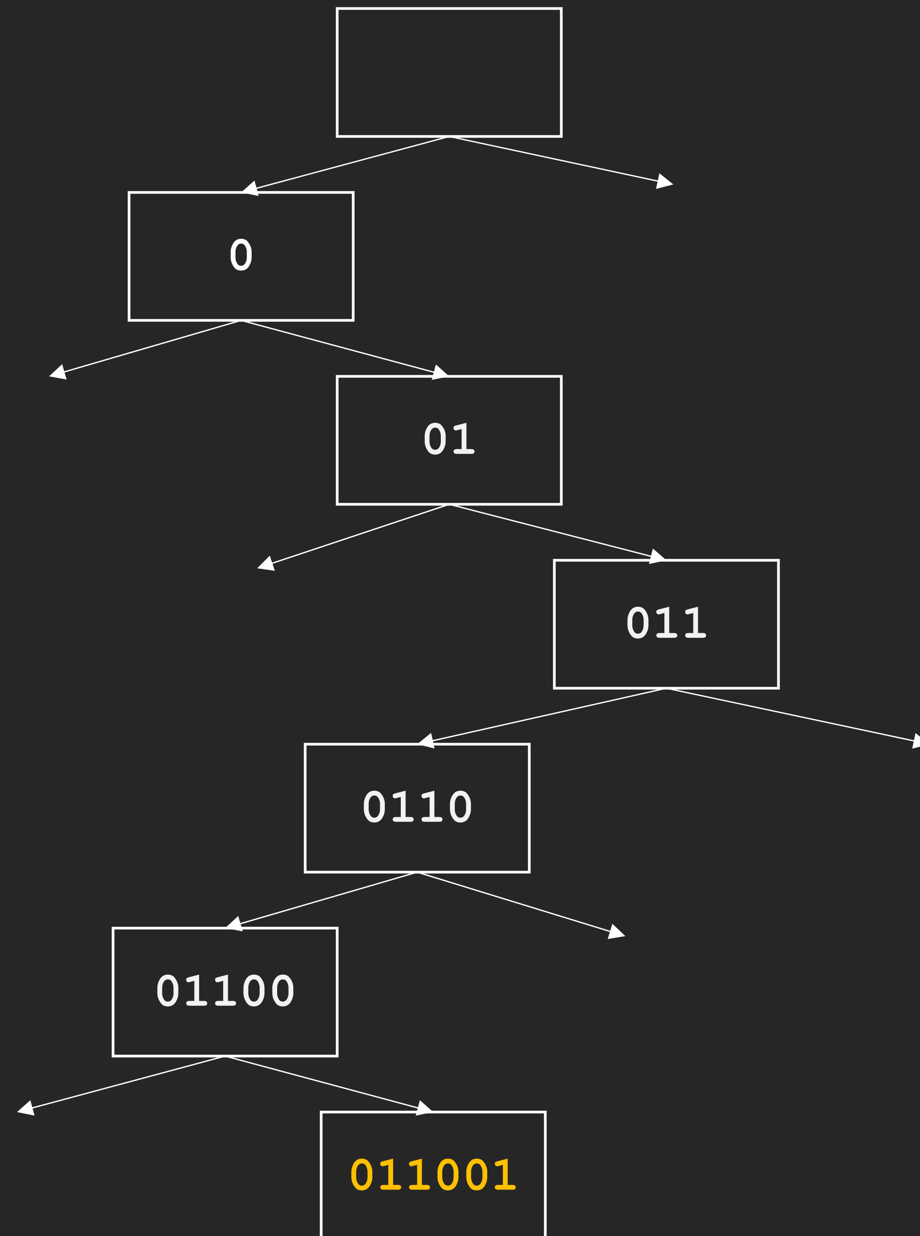
011000

Leaf Node

011001

Resulting Tree

- The Morton code tells us where in the tree we are!



Splitting

- Sorted Morton codes:

- The highest matching bits define the sub-tree, here **10**
- The highest switching bit determines the cut
- If all keys are equal, split in the middle

Algorithm

```
class Node { bounds, id, left, right)

func createTree()
    list = [{id1, mortonCode1}, {id2, mortonCode2}, …]
    sortByMortonCode(list)
    return createSubTree(list, 1, n)

func createSubTree(list, begin, end)
    if begin == end
        id = list[begin].id
        return  { objectBounds[id], id, null, null }
    else
        m = getSplitPos(list, begin, end)
        left = createSubTree(list, begin, m-1)
        right = createSubTree(list, m, end)
        return { union(left.bounds, right.bounds), -1, left, right  }
```

Demo

3D Morton BVH Collision Detection
Boxes: 10000
Collisions: 6154
BVH Build Time: 8.00 ms
Collision Check Time: 23.30 ms
Naive Checks: 0
BVH Checks: 948962
Using full 3D Morton code for spatial sorting
FPS: 5

Thanks for watching!

See you in the next tutorial…