Spatial Subdivision

Graham Rhodes
Senior Software Developer,
Applied Research Associates, Inc.

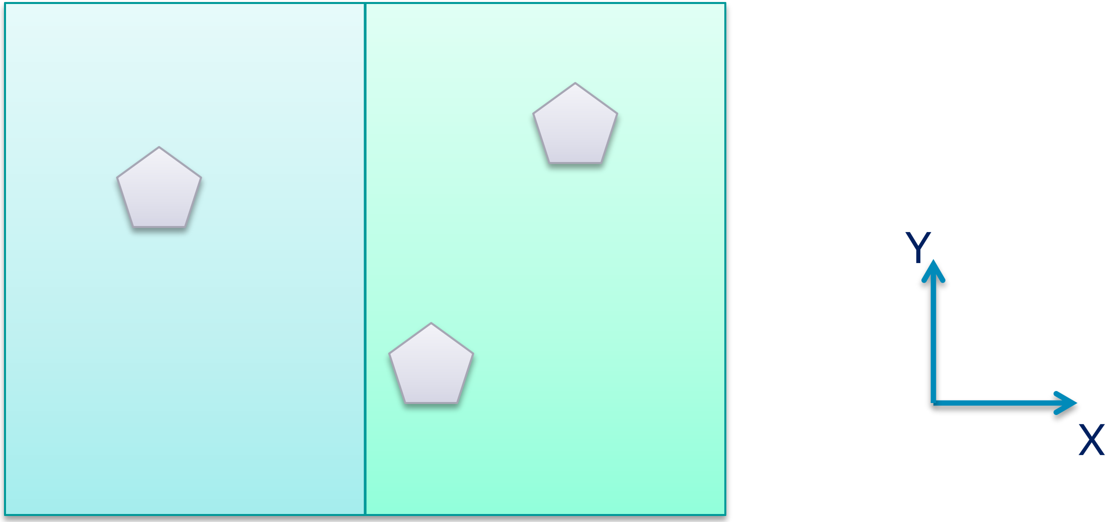
GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 17–21, 2014
EXPO DATES: MARCH 19–21
2014

How do you find a needle in a haystack?

Image is in the public domain. Painting date 1874. File source is http://commons.wikimedia.org/wiki/File:Haystacks_Autumn_1873_Jean-Francois_Millet.jpg

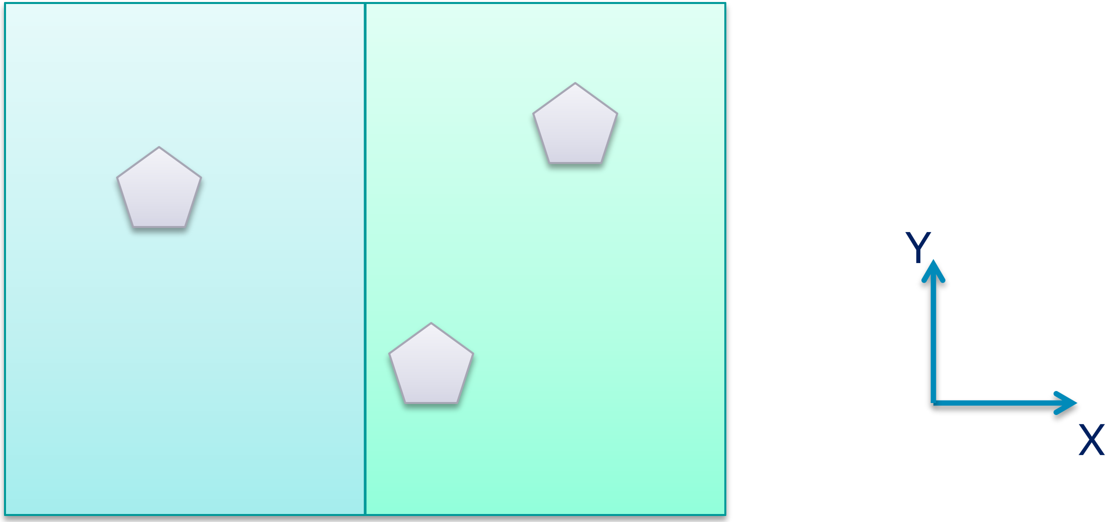
Spatial subdivision: what is it?

- A structured partitioning of geometry



Spatial subdivision: what is it for?

- Optimization!



Spatial subdivision: what is it for?

- Optimization!
  - Manage rendering overhead
  - Support a geometry paging system
  - Minimize unnecessary geometry interrogation and pair-wise object tests for physics and AI
- **Not all games need this!**
  - (many do)

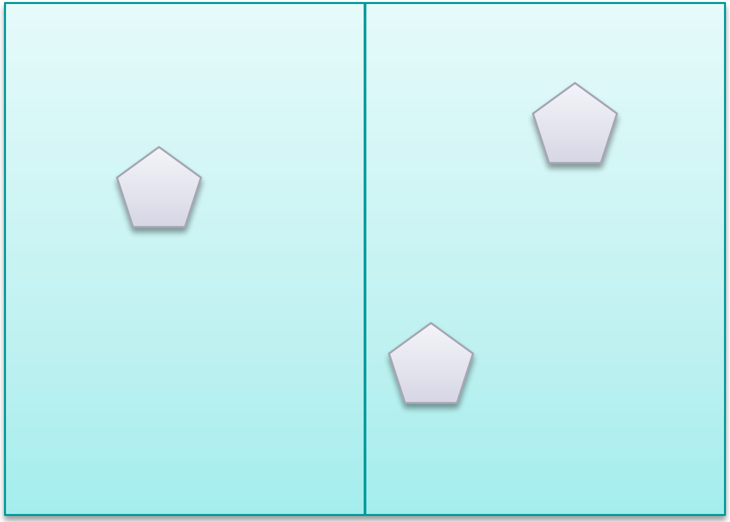
Classes of spatial subdivision

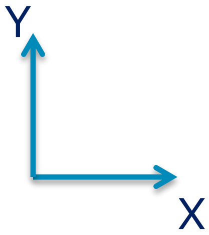
- Grid-based
- Tree-based
- Others
    - Bounding Volume Hierarchy
    - Scene Graph
    - Portals

Grid-based Spatial Subdivision

Overview of uniform grids

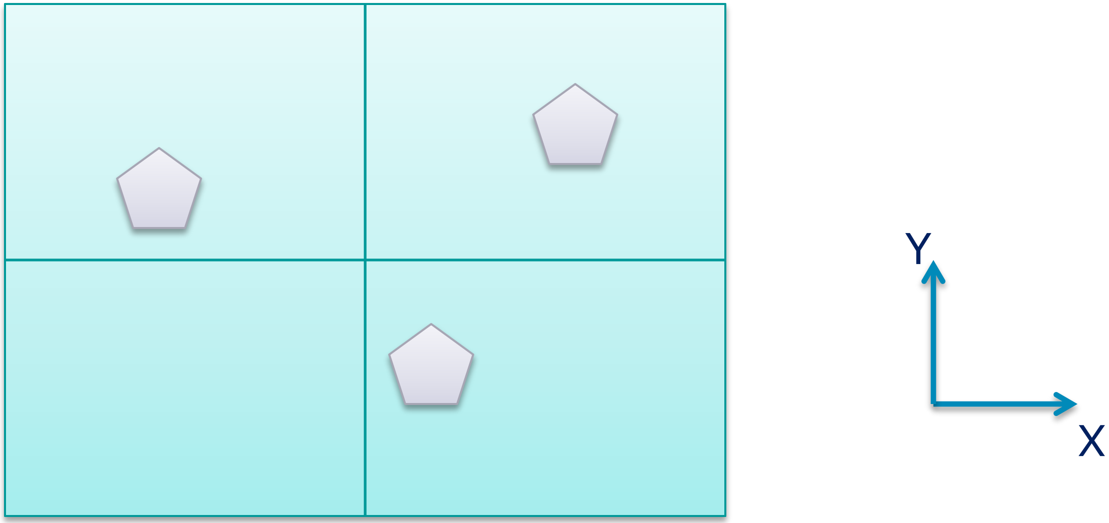
- A 2 x 1 grid





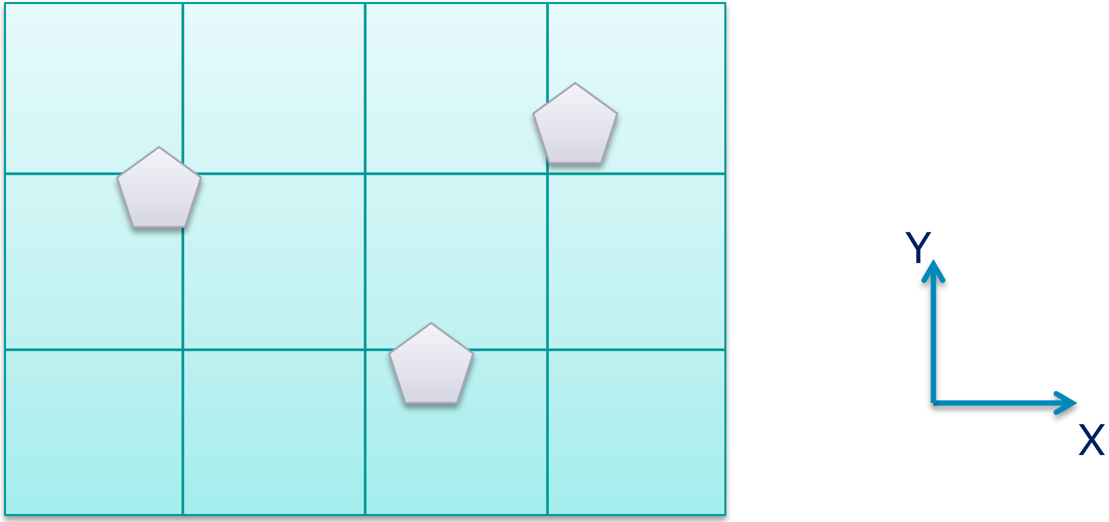
Overview of uniform grids

- A 2 x 2 grid



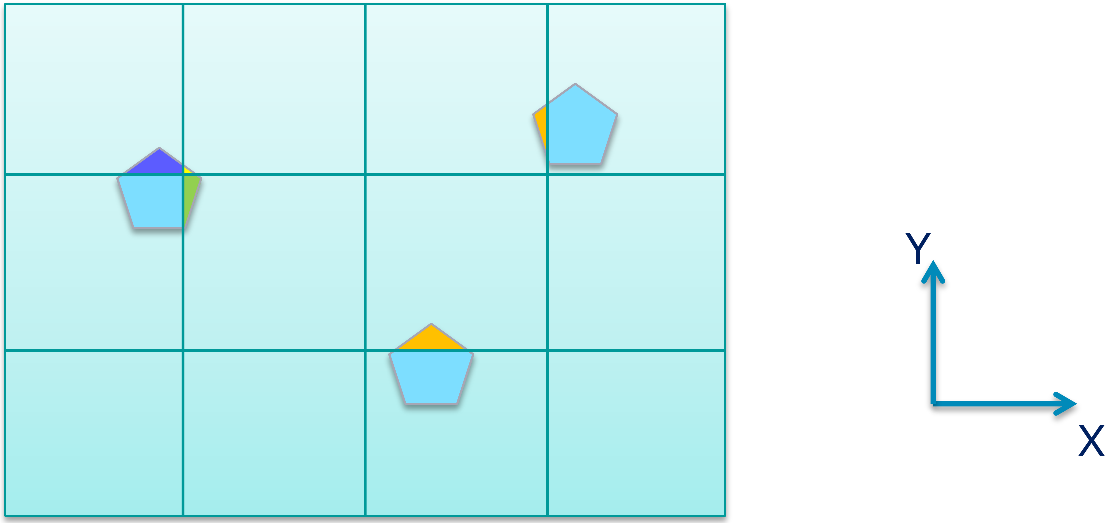
Overview of uniform grids

- A 4 x 3 grid



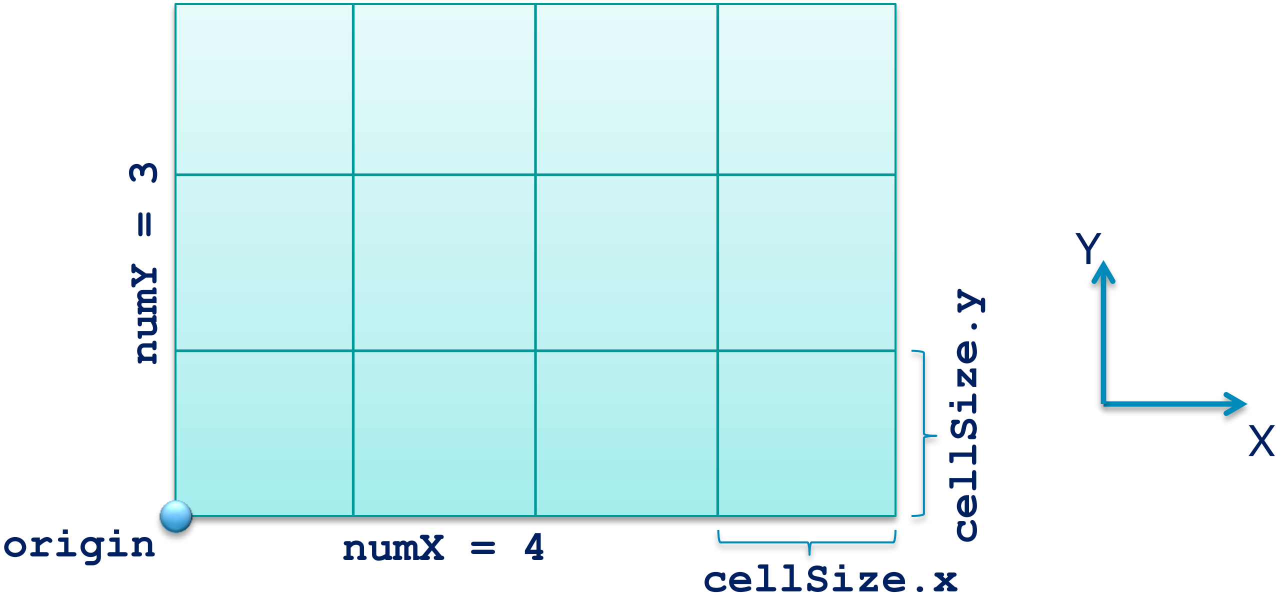
Overview of uniform grids

- A 4 x 3 grid



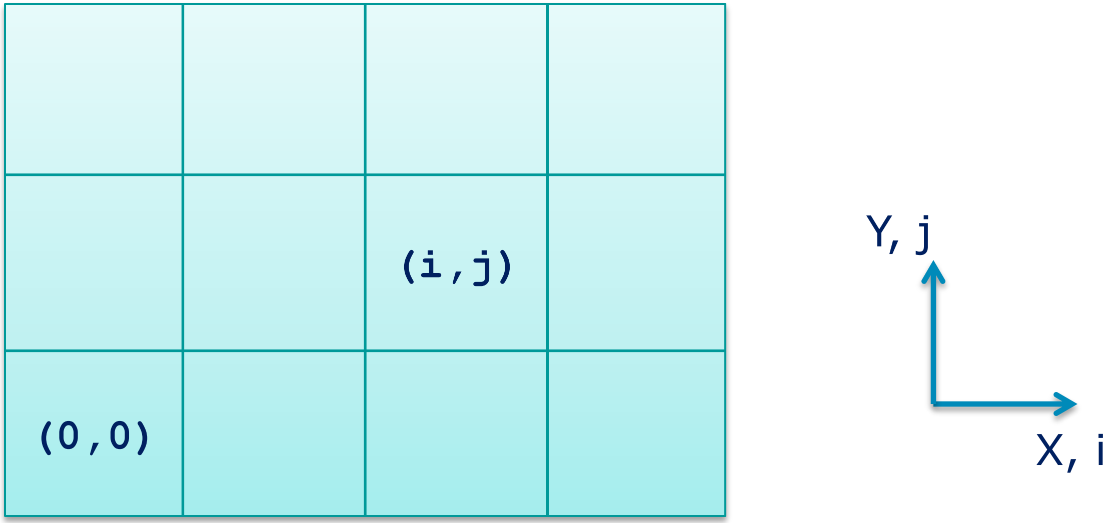
Overview of uniform grids

- The spatial and dimensional properties



Overview of uniform grids

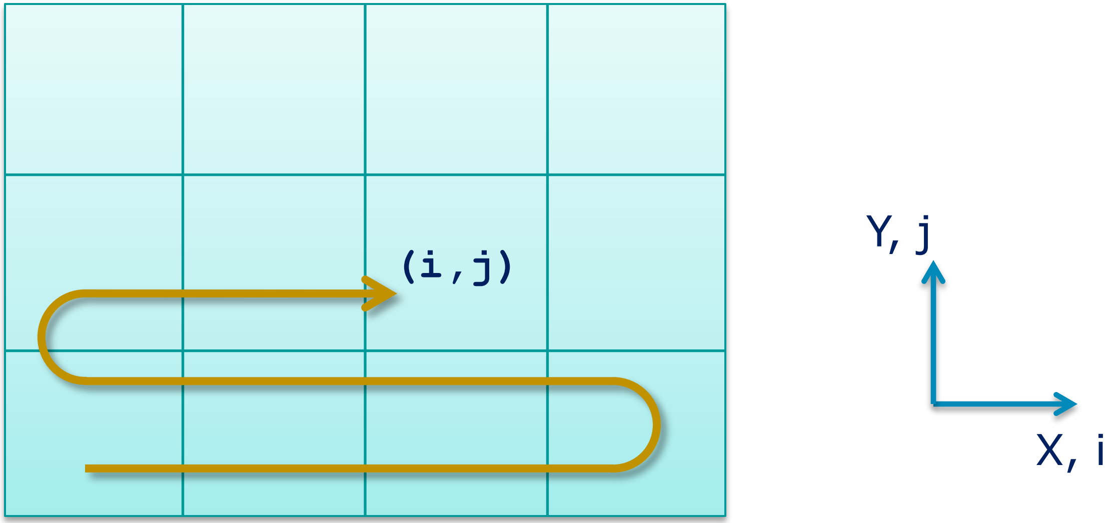
- Spatial index of a cell: (i, j) or (i, j, k)



Overview of uniform grids

- Logical address of a cell: memory location

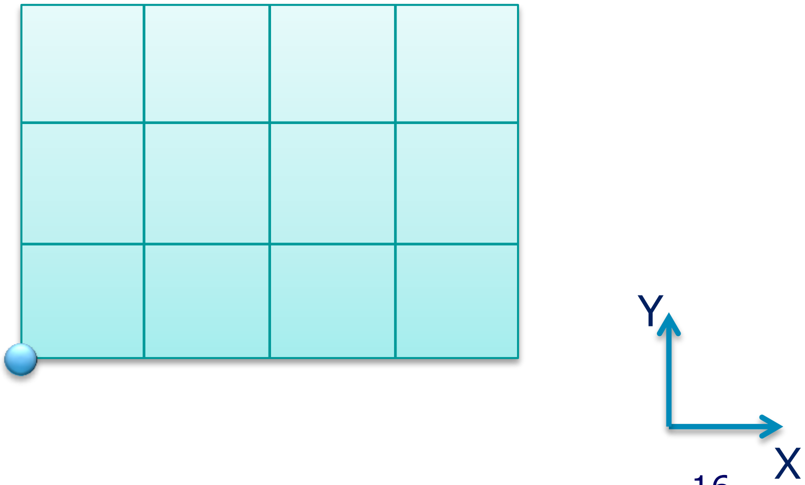
Cell
Address = f(i, j)



Implementation: ideas

- Conceptual data structures

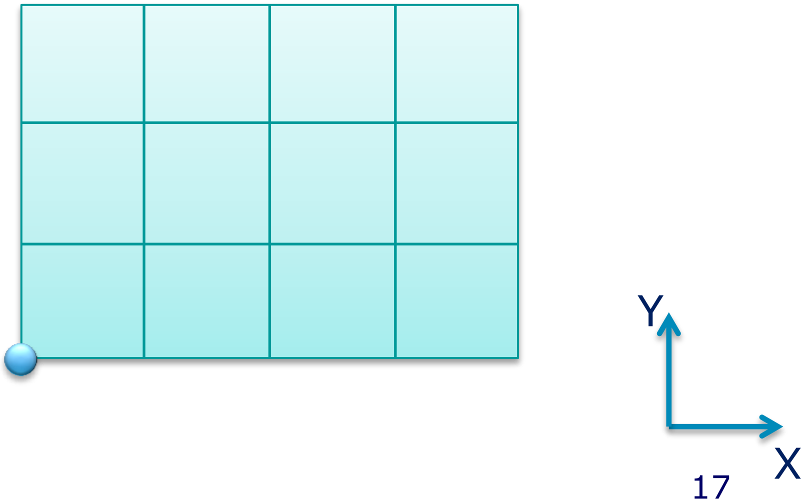
```
Grid2D
{
  …
  Container<Cell> gridCells;
}

Cell
{
  Container<Object> gameObjects;
}
```



Implementation: array of cells

- A naïve UniformGrid data structure

```
NaiveUniformGrid2D
{
  …
  Array<Cell> gridCells;
}

Cell
{
  Container<Object> gameObjects;
}
```

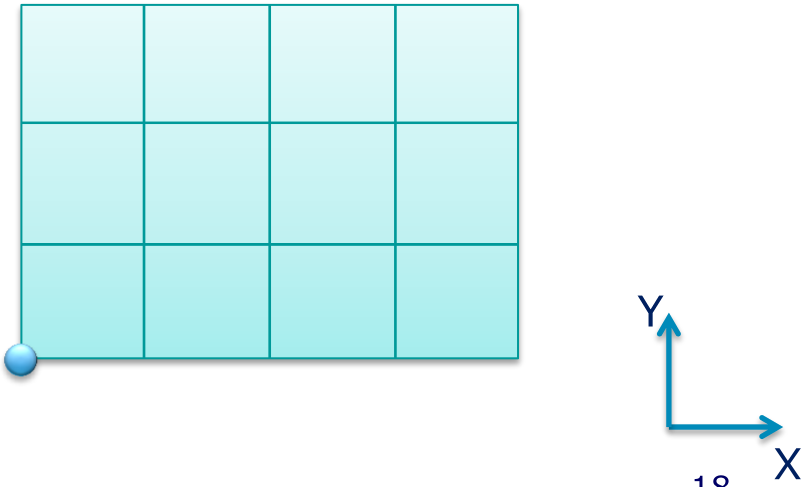


Implementation: array of cells

- Retrieving the cell at a point in space

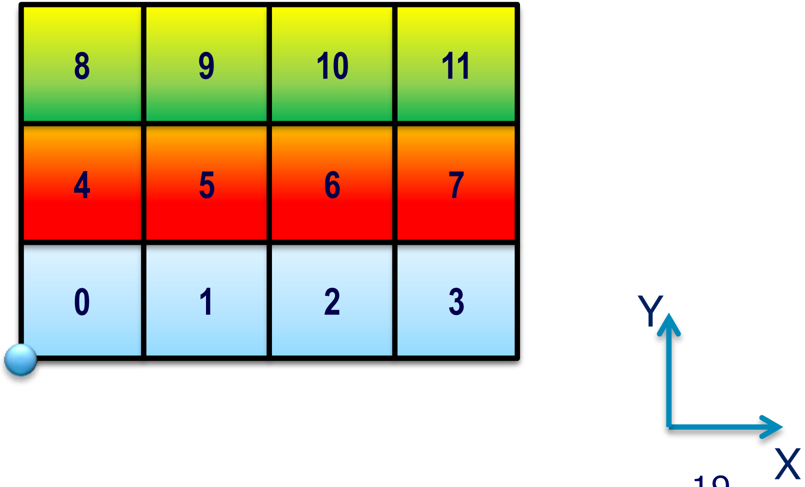
0	1	2	3	4	5	6	7	8	9	10	11

`gridCells array`



Implementation: array of cells

- Retrieving the cell at a point in space



8	9	10	11
4	5	6	7
0	1	2	3

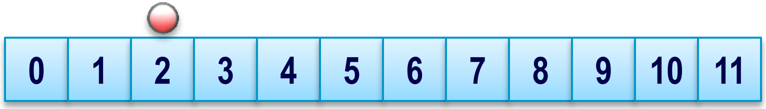
0	1	2	3	4	5	6	7	8	9	10	11

gridCells array

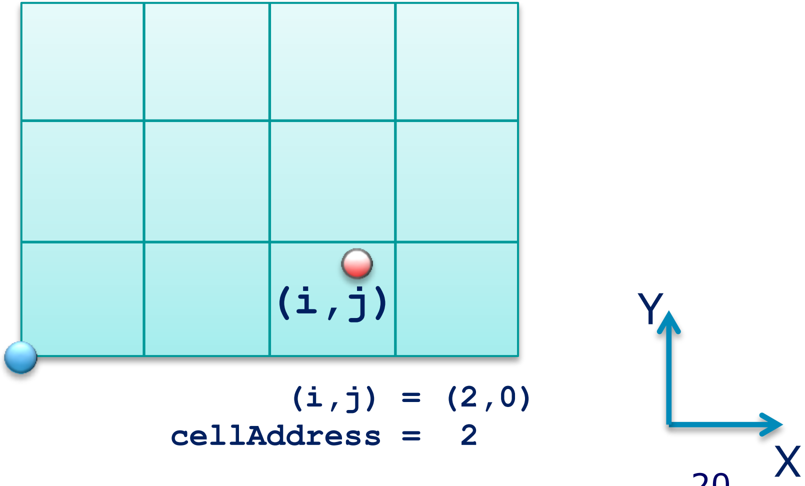
Implementation: array of cells

- Retrieving the cell at a point in space

```
const int X = 0, Y = 1;
int getCellIndex(int d, Vector2 pt) { return (int)(floor((p[d] - origin[d])/cellSize[d]));}

int getCellAddress(Vector2 pt)
{
  int i = getCellIndex(X, pt);
  int j = getCellIndex(Y, pt);
   return (numX * j) + i;
}
```



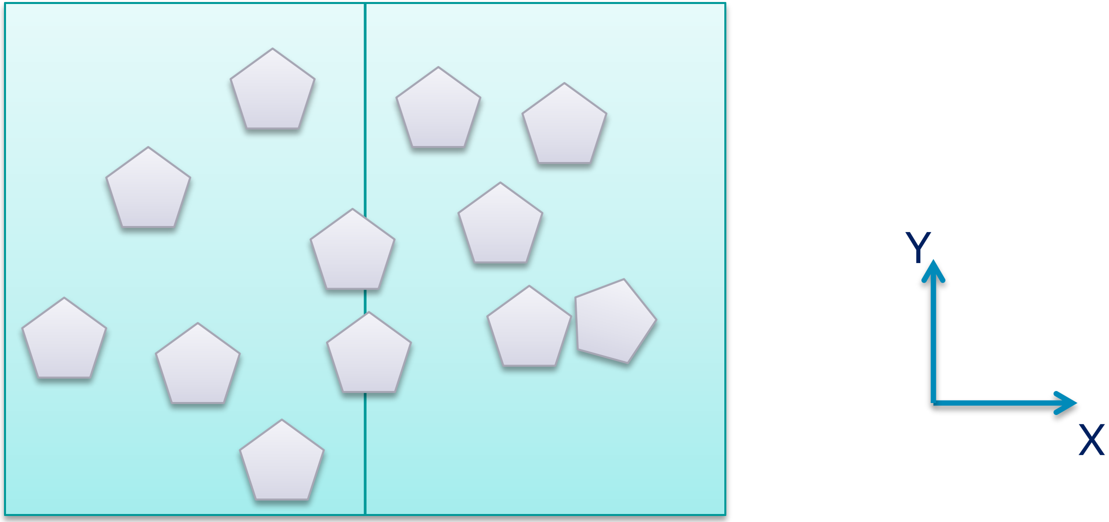
gridCells array



(i,j) = (2,0)
cellAddress = 2

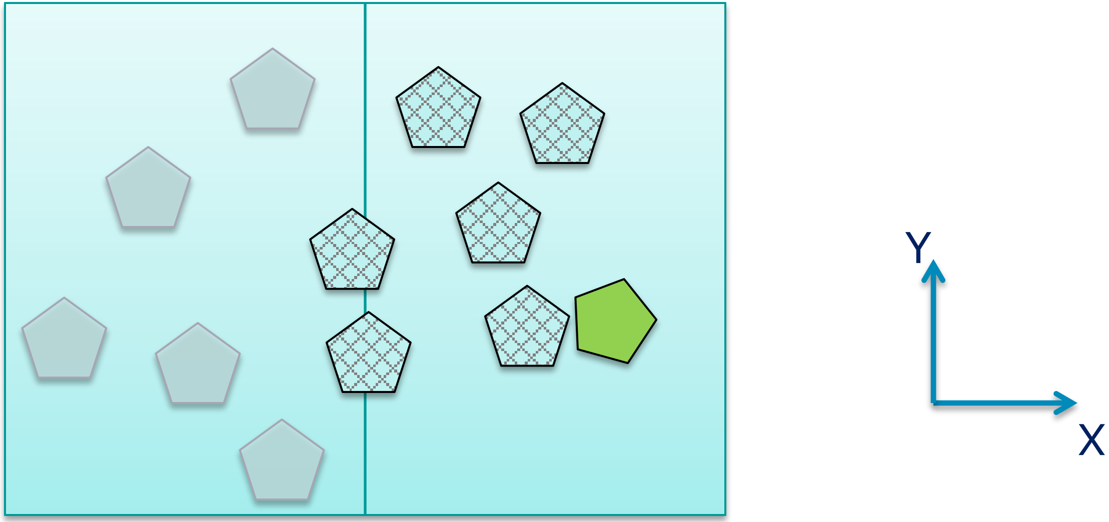
Grid-size selection strategies

- What size should we choose for the grid cells?



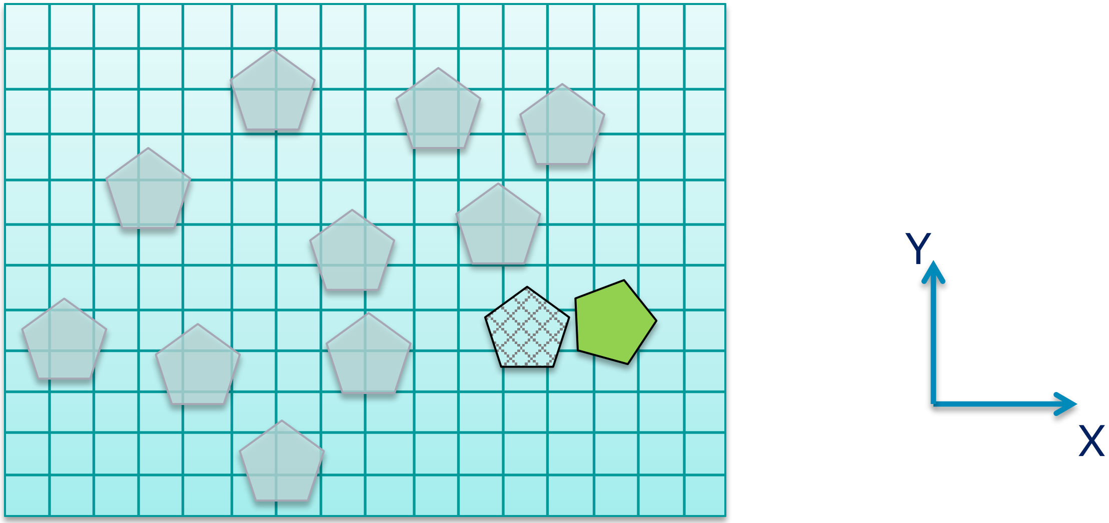
Grid-size selection strategies

- What size should we choose for the grid cells?



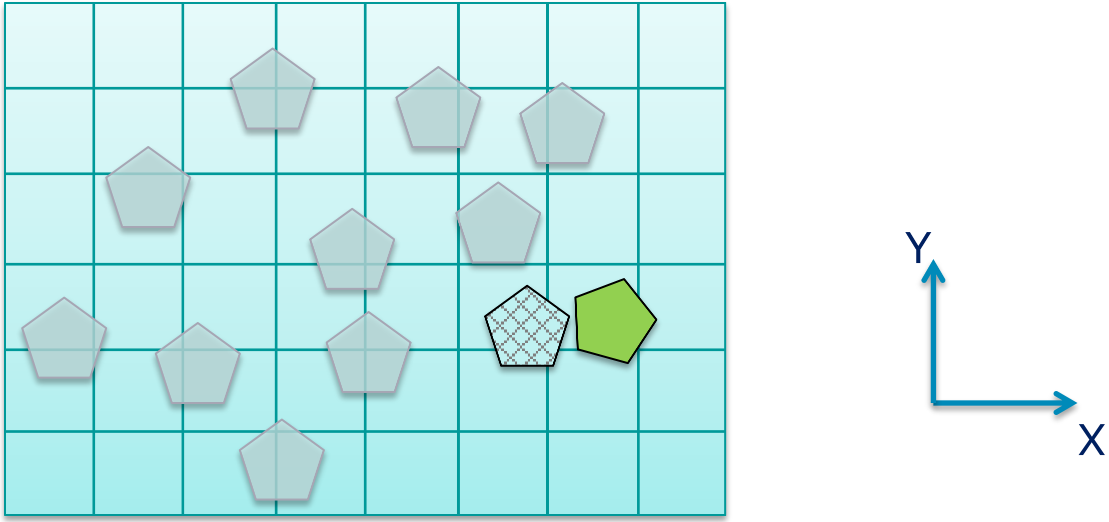
Grid-size selection strategies

- What size should we choose for the grid cells?



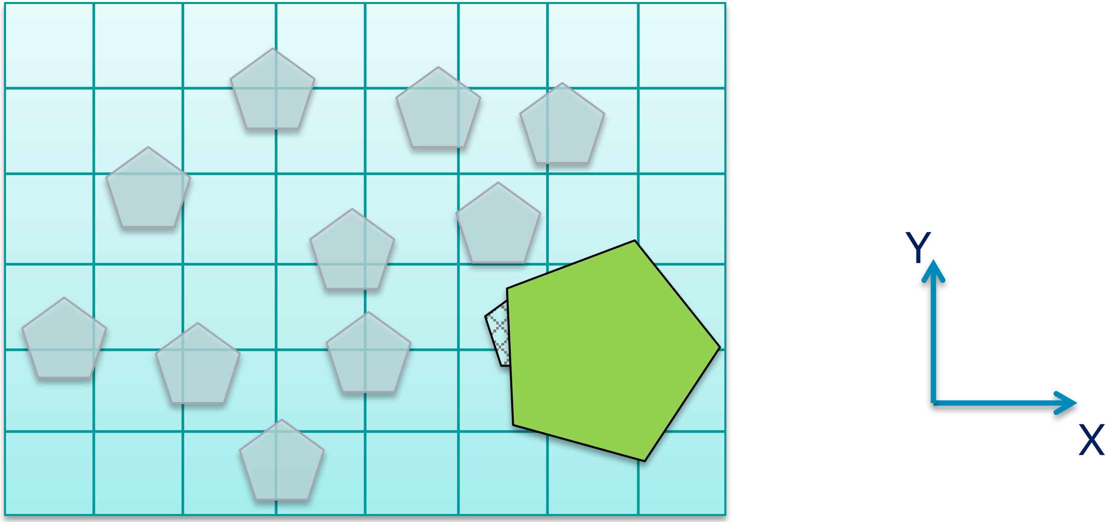
Grid-size selection strategies

- Optimum size ~ max object size + ε



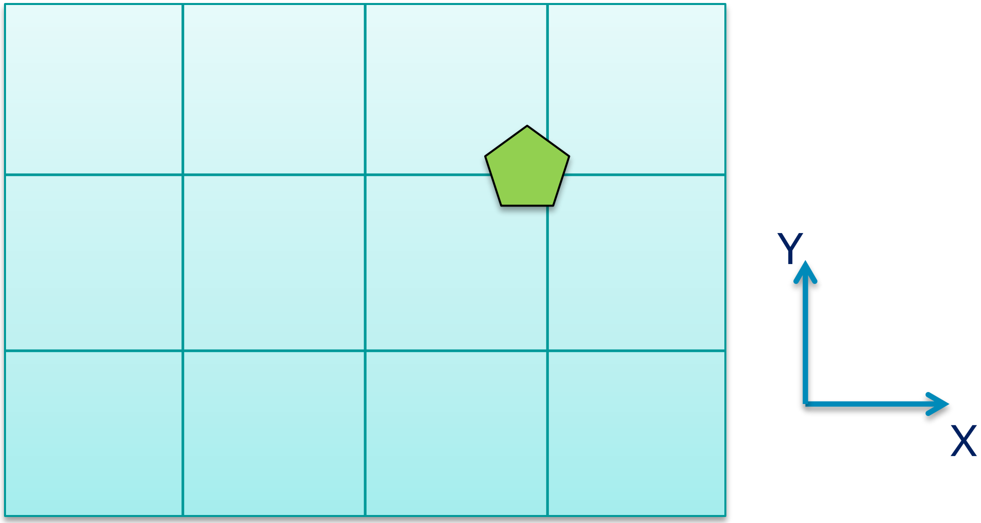
Grid-size selection strategies

- What if object size varies significantly?



Populating the grid

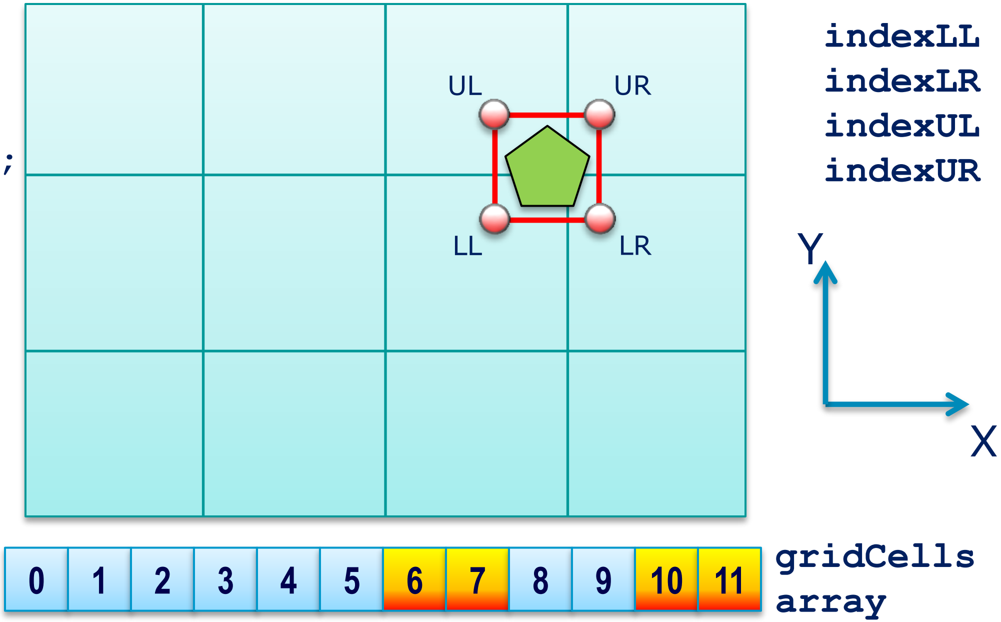
- Inserting an object into the grid



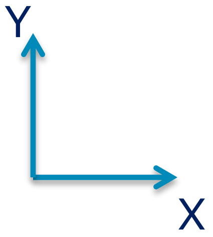
Populating the grid

- Insert into every overlapped cell

```
void addObject(Object obj)
{
  pt = obj.minAABBPoint();
  addrLL = getCellAddress(pt);
  addrLR = addrLL + 1;
  addrUL = addrLL + numX;
  addrUR = addrUL + 1;
  gridCells[addrLL].add(obj);
  gridCells[addrLR].add(obj);
  gridCells[addrUL].add(obj);
  gridCells[addrUR].add(obj);
}
```



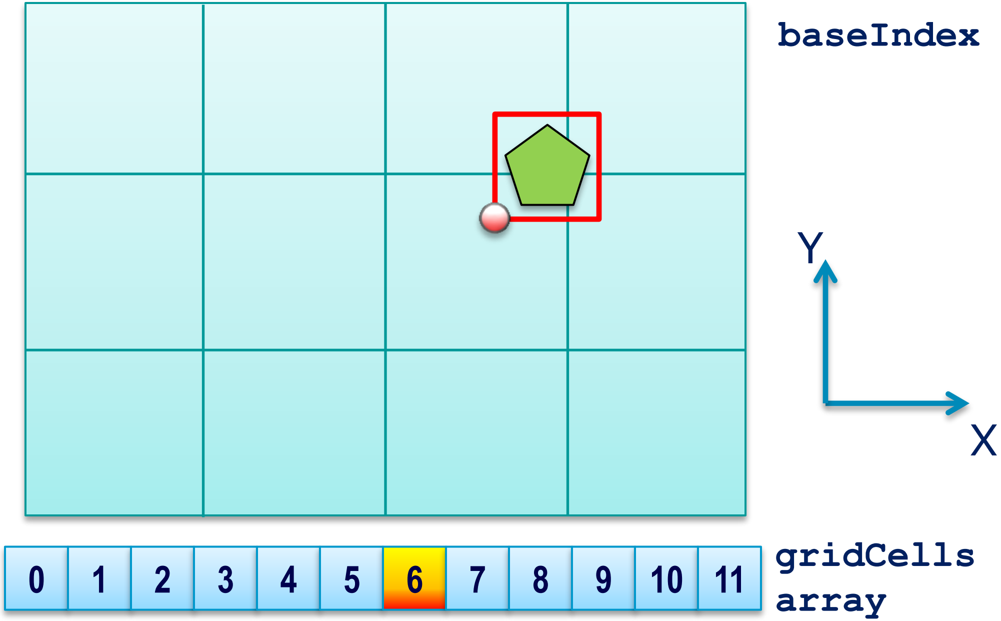
```
indexLL = (2,1)
indexLR = (3,1)
indexUL = (2,2)
indexUR = (3,2)
```



Populating the grid

- Inserting into one cell (others are implicit)

```
void addObject(Object obj)
{
  pt = obj.minAABBPoint();
  addr = getCellAddress(pt);
  gridCells[addr].add(obj);
}
```

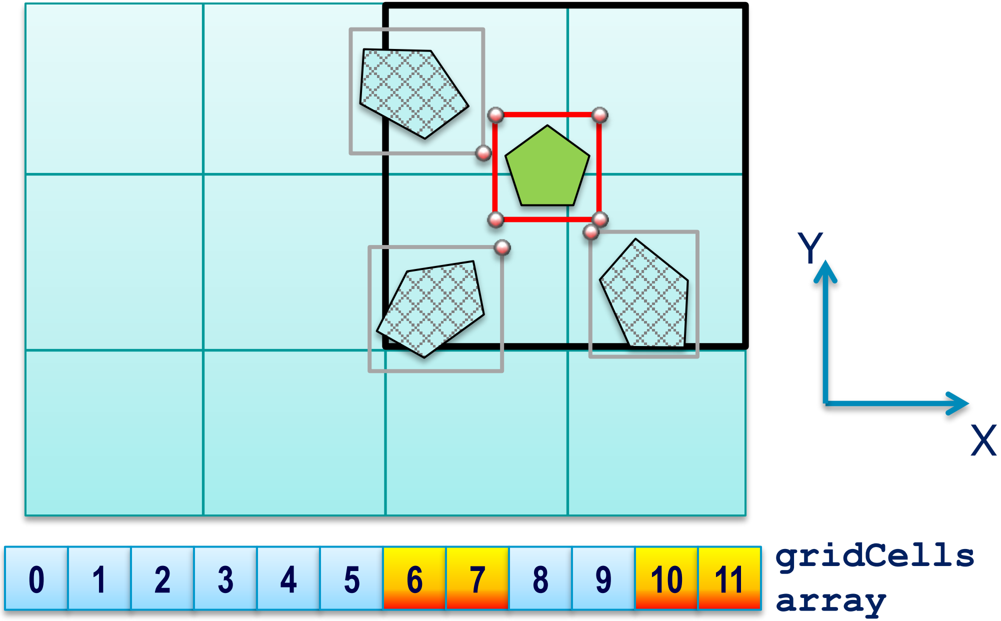


baseIndex = (2,0)

gridCells array

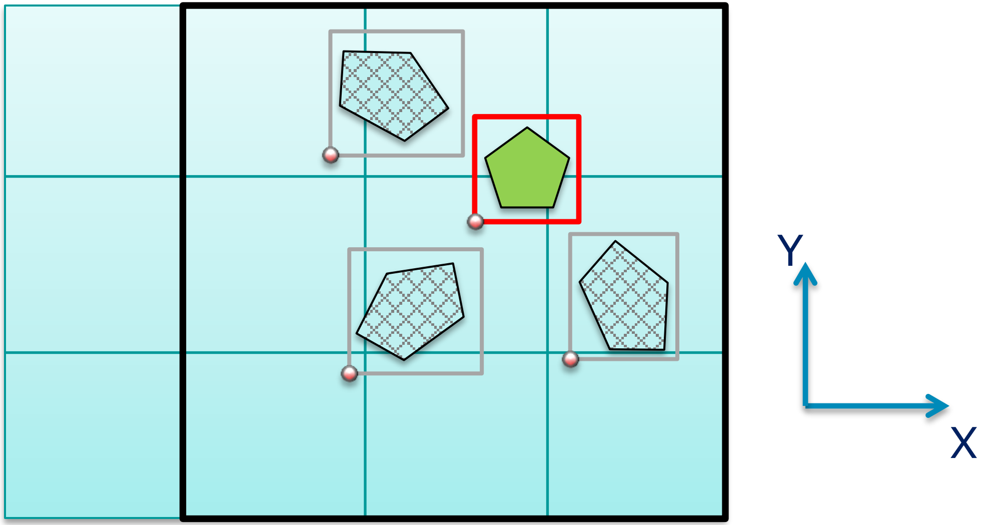
Pairwise testing: visit which cells?

- If insert objects into every overlapped cell



Pairwise testing: visit which cells?

- If insert objects only into one key cell

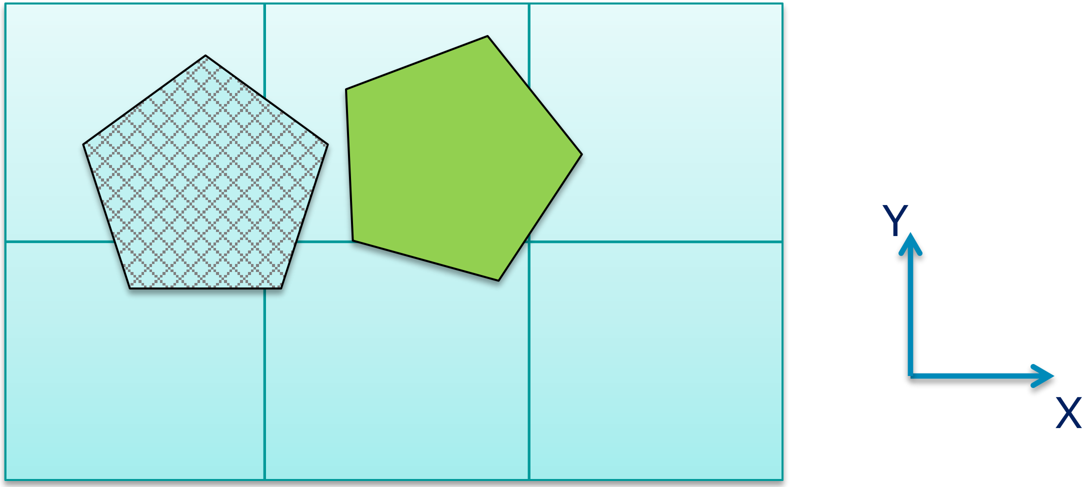


0	1	2	3	4	5	6	7	8	9	10	11

gridCells
array

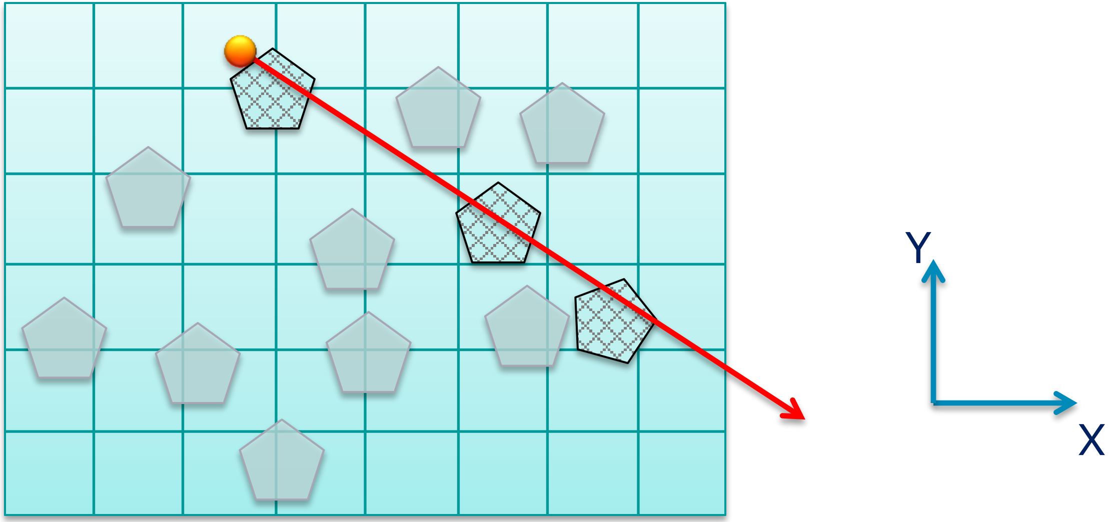
Avoiding duplicate tests

- Bitfield, time stamping…



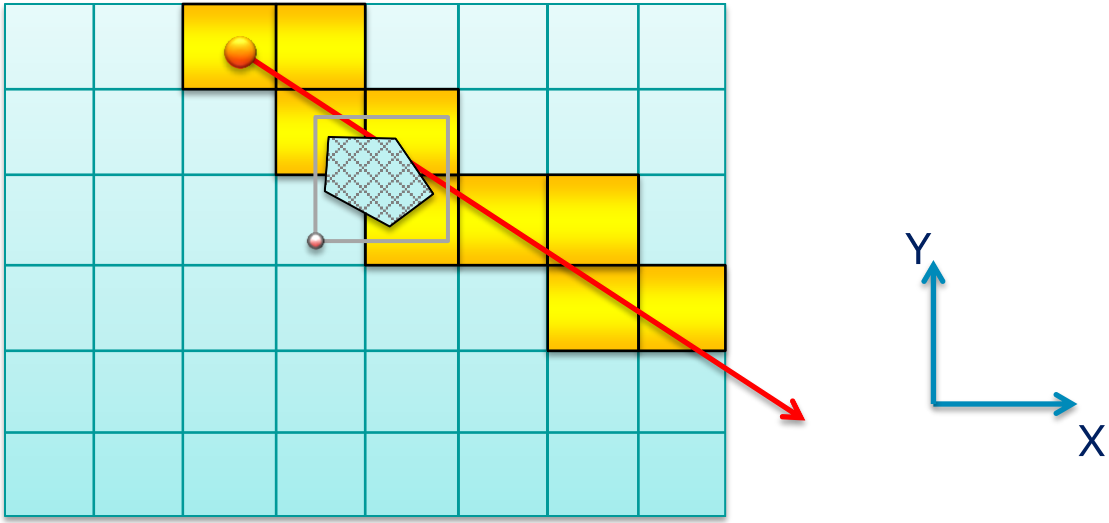
Ray intersection/line of sight tests

- Find all objects that intersect a ray



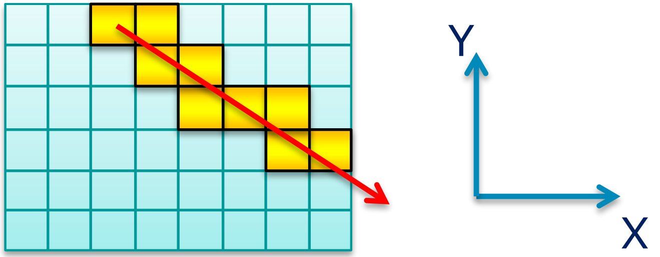
Ray intersection/line of sight tests

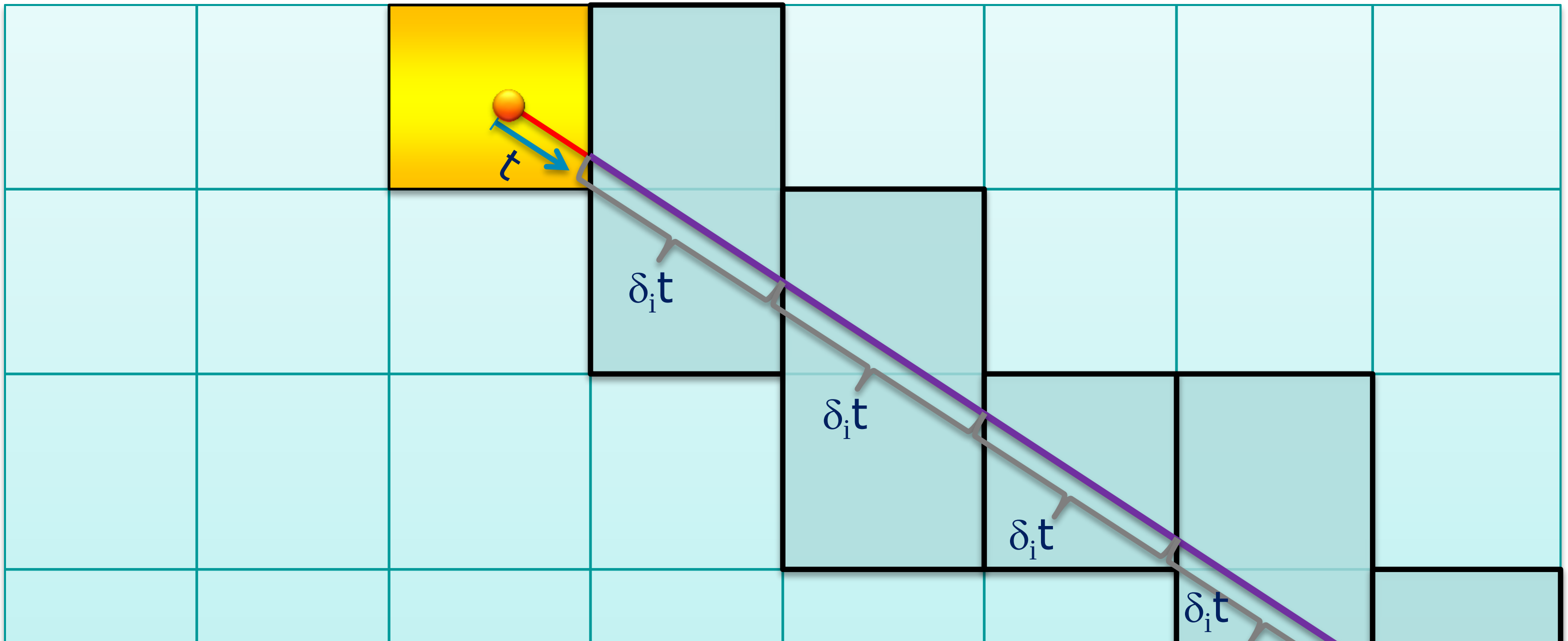
- Find all objects that intersect a ray



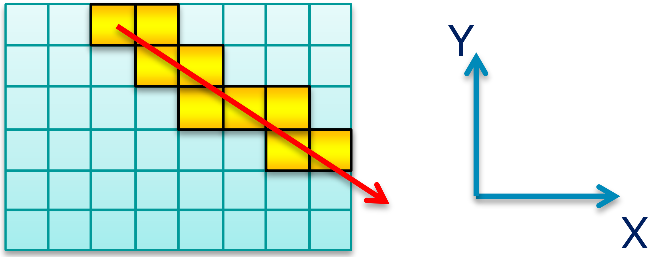
Ray intersection

- Walking along the ray

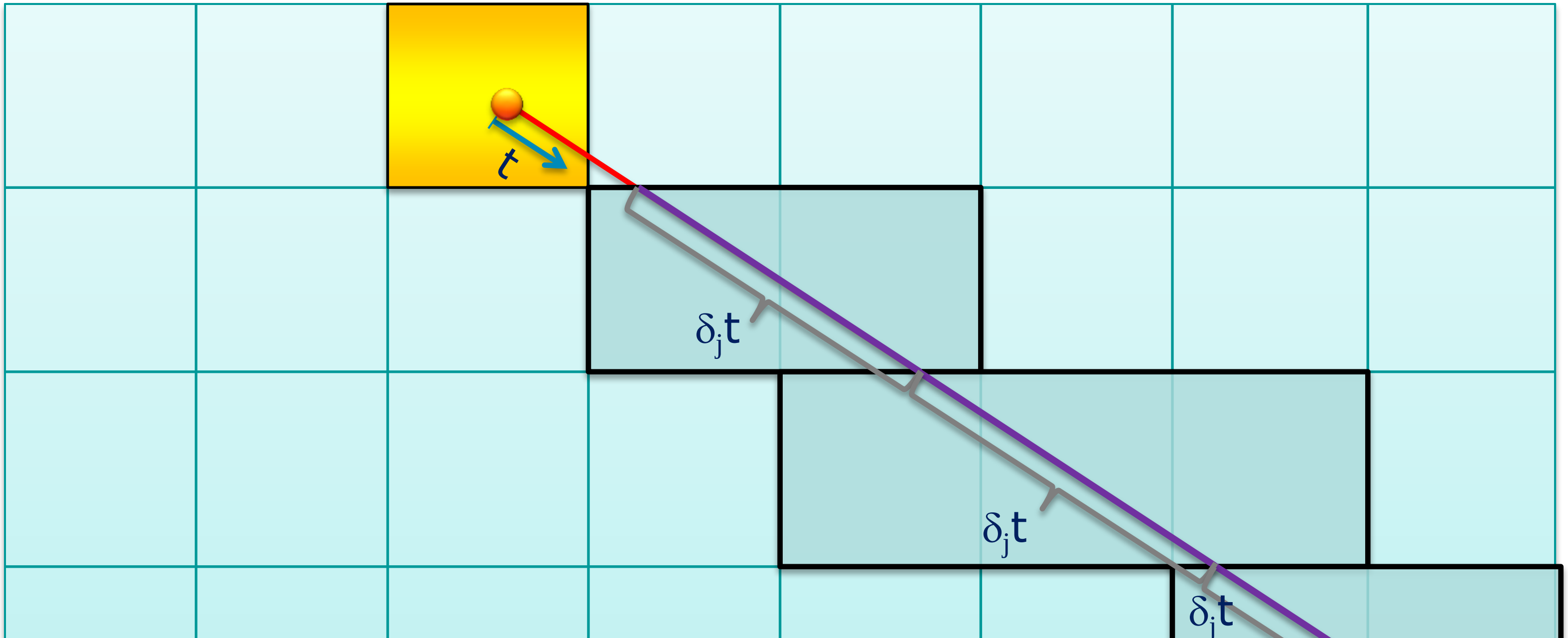




Ray intersection

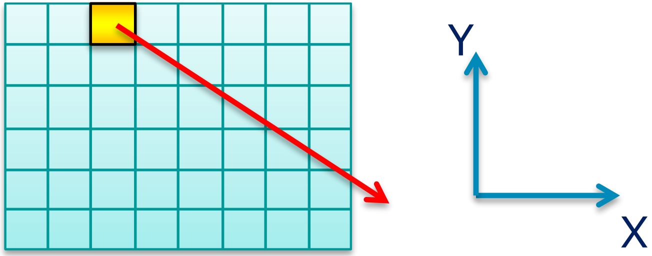


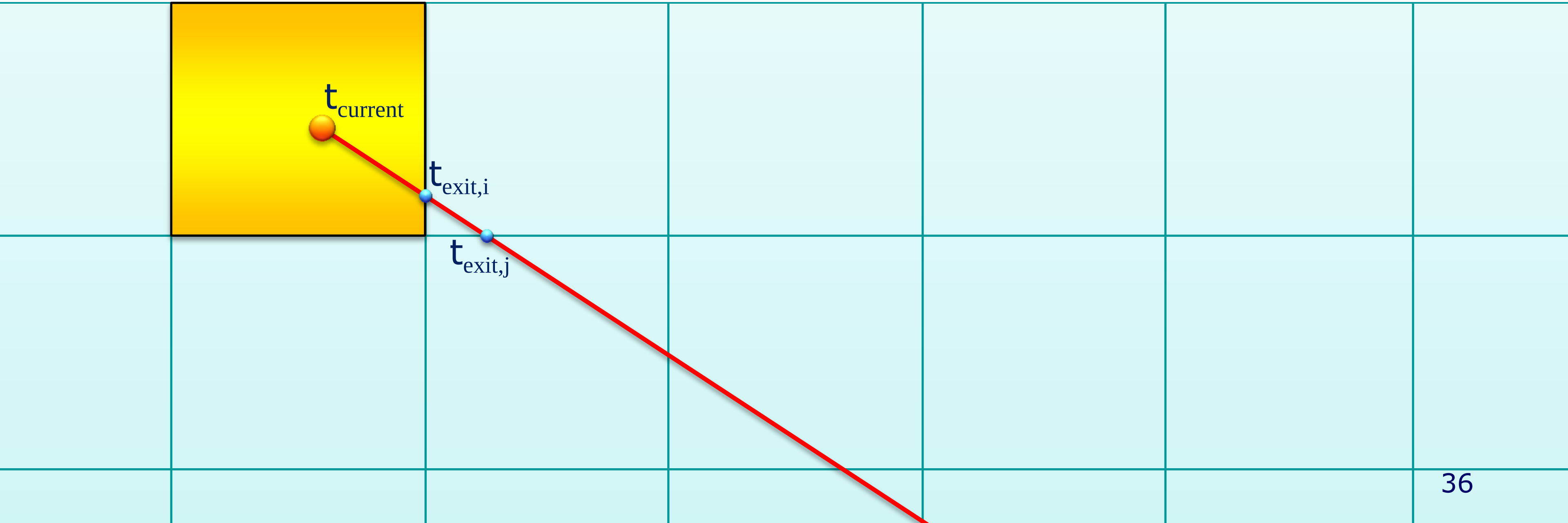
- Walking along the ray



Ray intersection

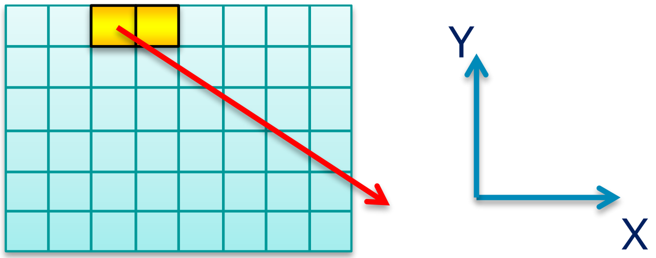
- Walking along the ray

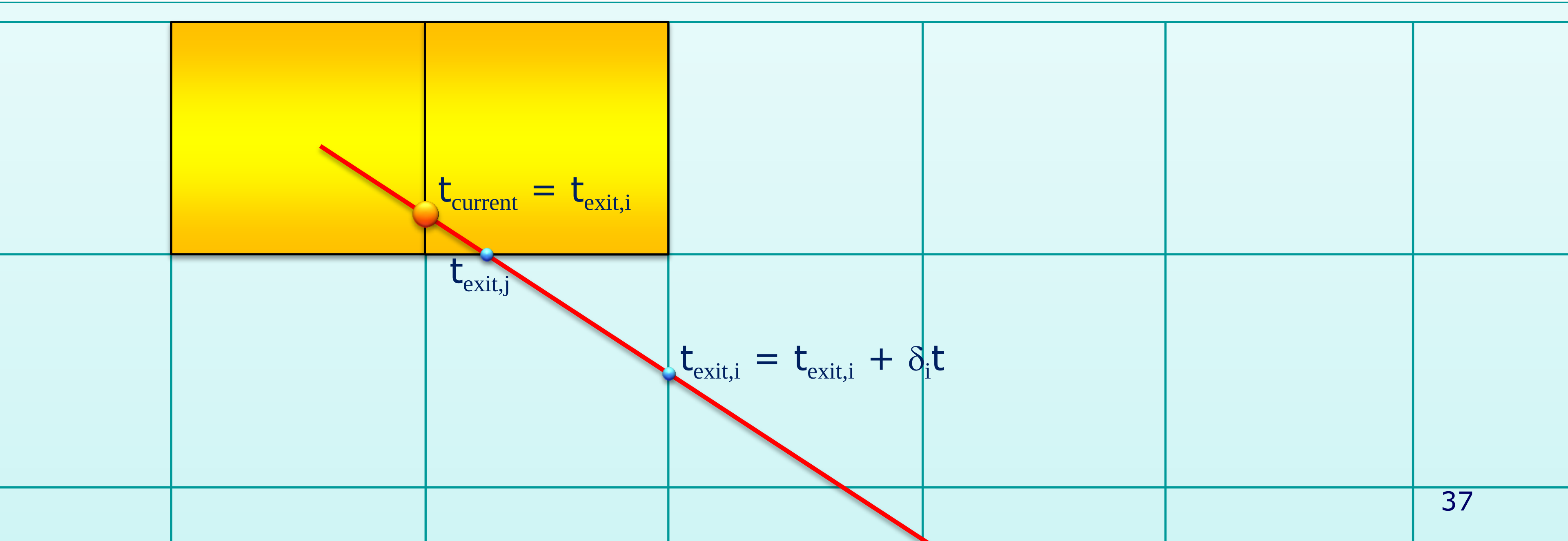




Ray intersection

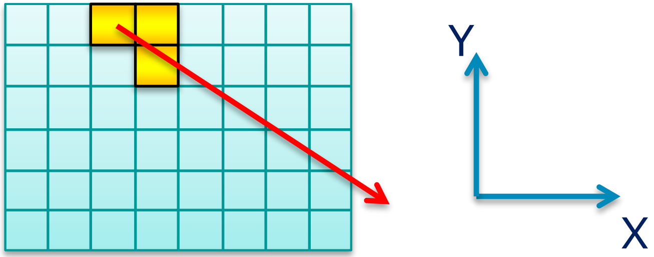
- Walking along the ray

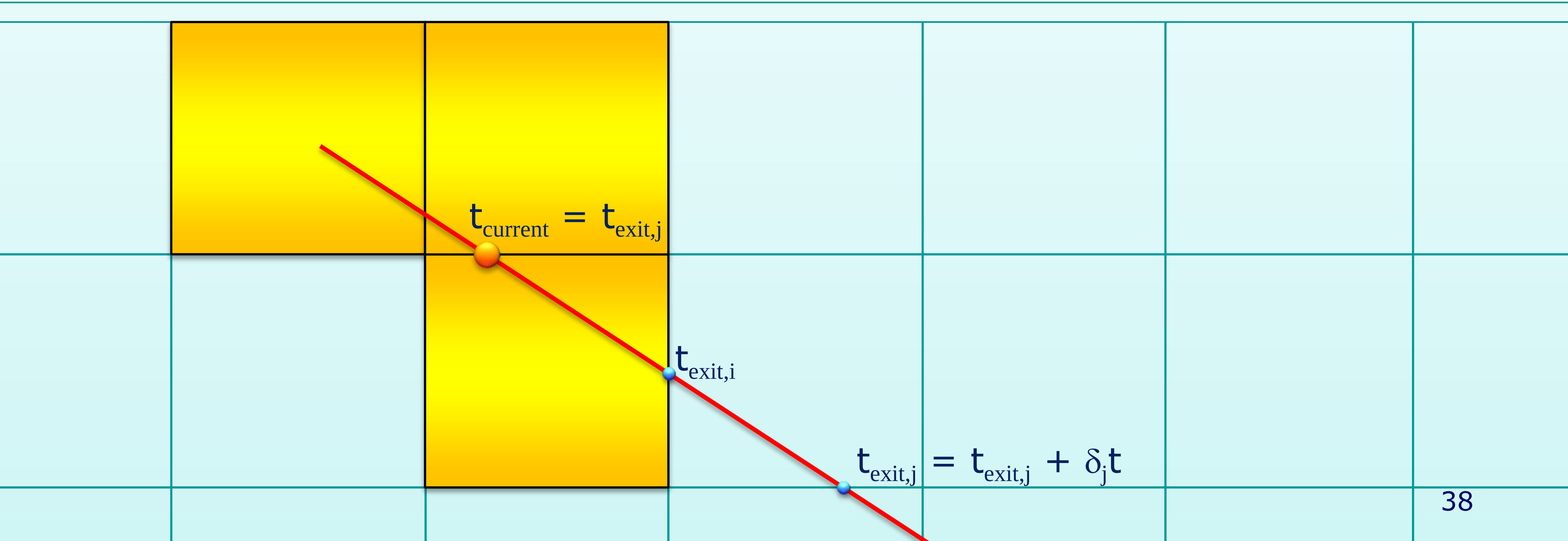



$t_{current} = t_{exit,i}$
$t_{exit,j}$
$t_{exit,i} = t_{exit,i} + \delta_i t$

Ray intersection

- Walking along the ray





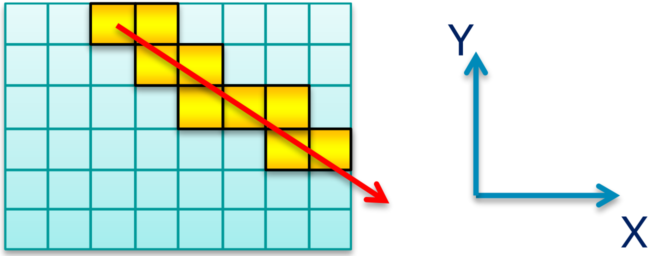
$t_{current} = t_{exit,j}$

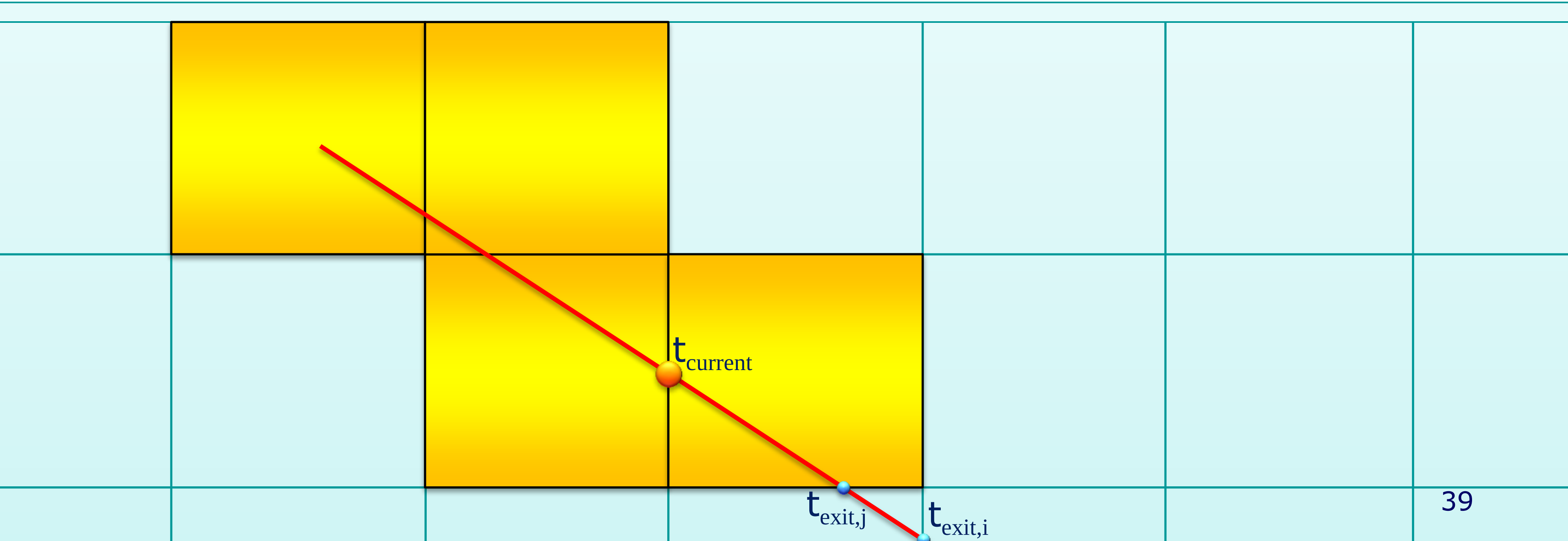
$t_{exit,i}$

$t_{exit,j} = t_{exit,j} + \delta_j t$

Ray intersection

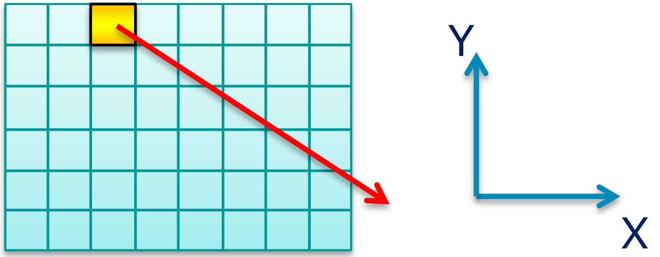
- Walking along the ray

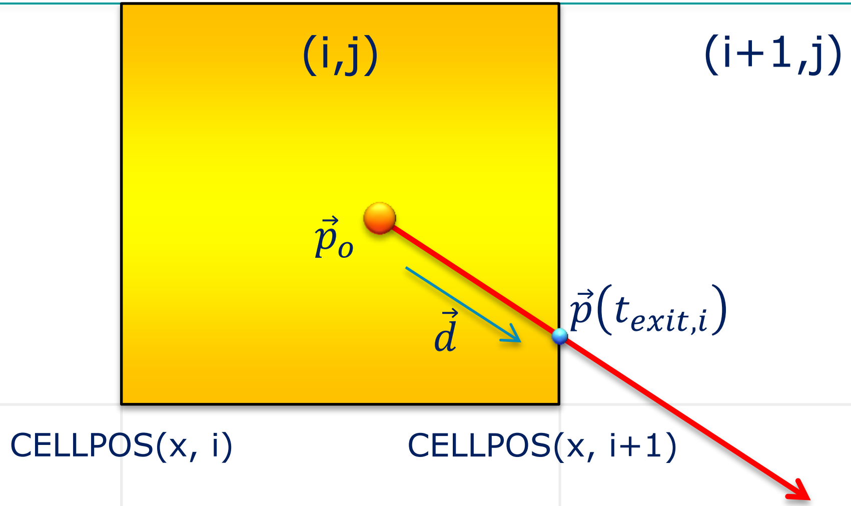




Ray intersection

- Initializing the walk



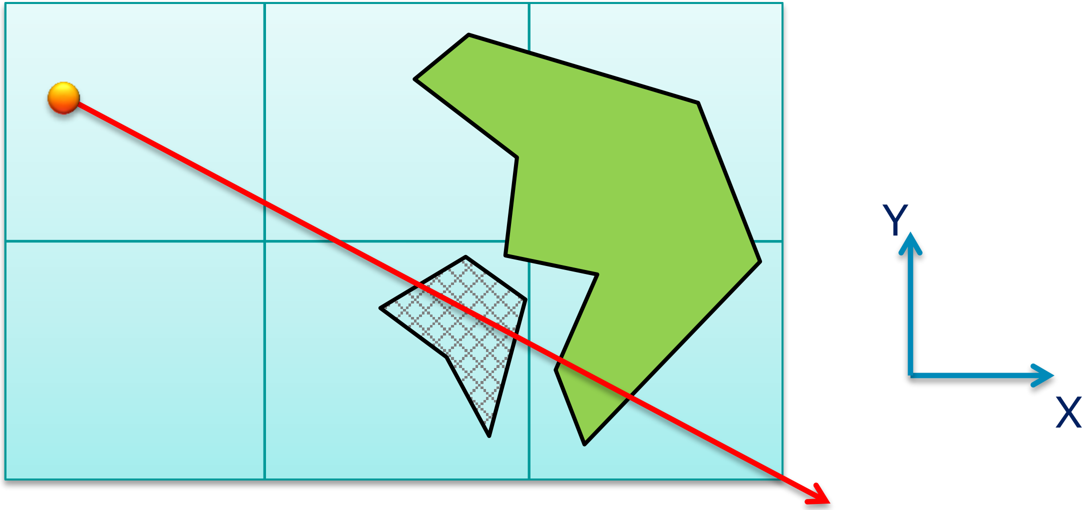


$$\vec{p}(t) = \vec{p}_o + t\vec{d}$$
$$\vec{p}(t_{exit,i}) = \vec{p}_o + t_{exit,i}\vec{d}$$
$$\vec{p}(t_{exit,i}).x = \vec{p}_o.x + t_{exit,i}\vec{d}.x$$
$$i = getCellIndex(X, \vec{p}_o.x)$$
$$\vec{p}(t_{exit,i}).x = getCellPos(X, i+1)$$
$$t_{exit,i} = (getCellPos(X, i+1) - \vec{p}_o.x)/\vec{d}.x$$
$$\delta_i \mathrm{t} = \mathrm{cellSize.x}/\vec{d}.x$$

```
float getCellPos(int d, int index) { return (((float)index) * cellSize[d]) + origin[d]; }
```

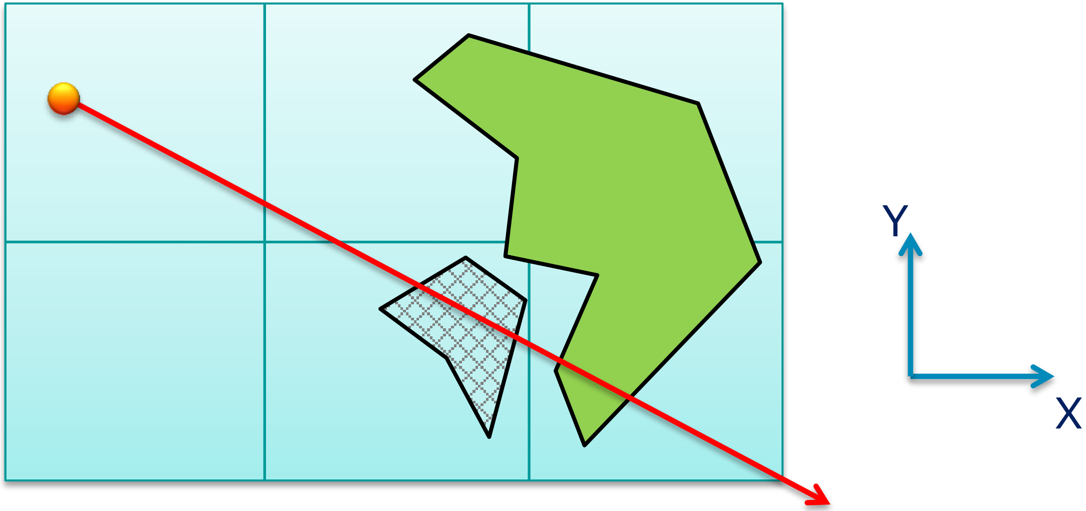
Avoiding duplicate tests

- Time stamping easier than with pairwise tests



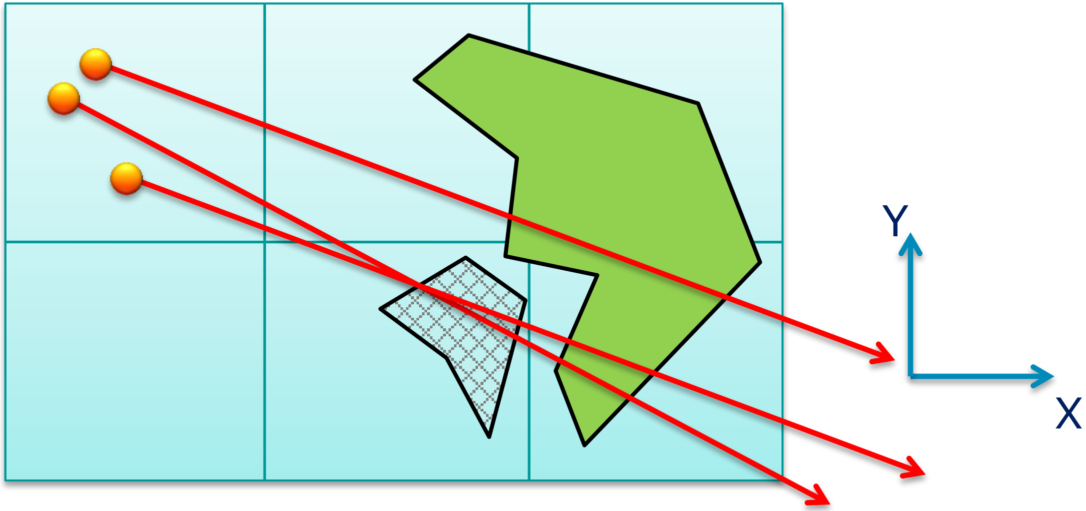
Avoiding duplicate tests

- May find an intersection in a different cell



Avoiding duplicate tests

- Batch ray tests as optimization strategy

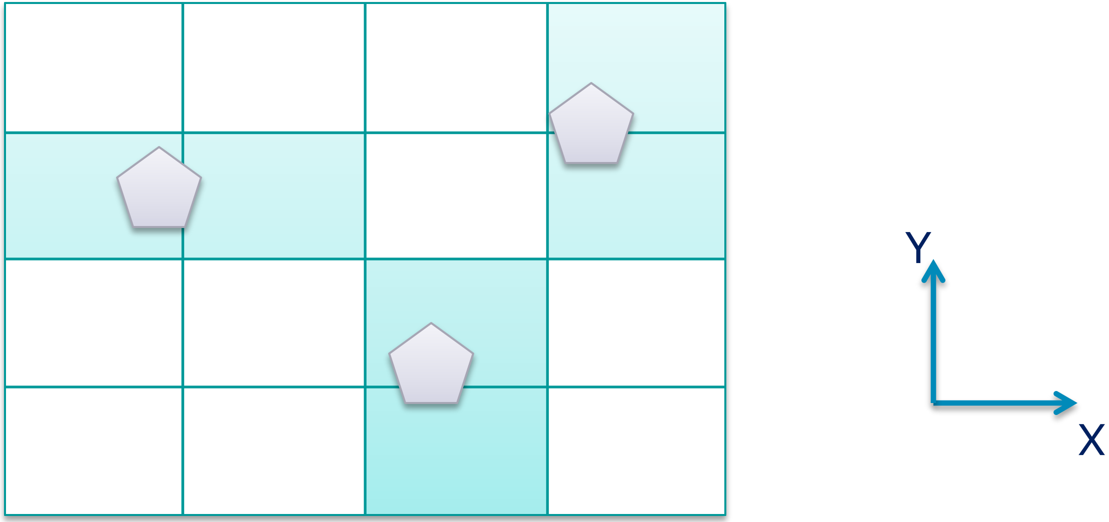


Back to array of cells

- Does anyone see a problem with this naïve approach?
  - Most cells are likely empty
  - Doesn’t scale well due in part to large memory requirements
- For these reasons, this naïve array of cells approach is often a bad choice in practice

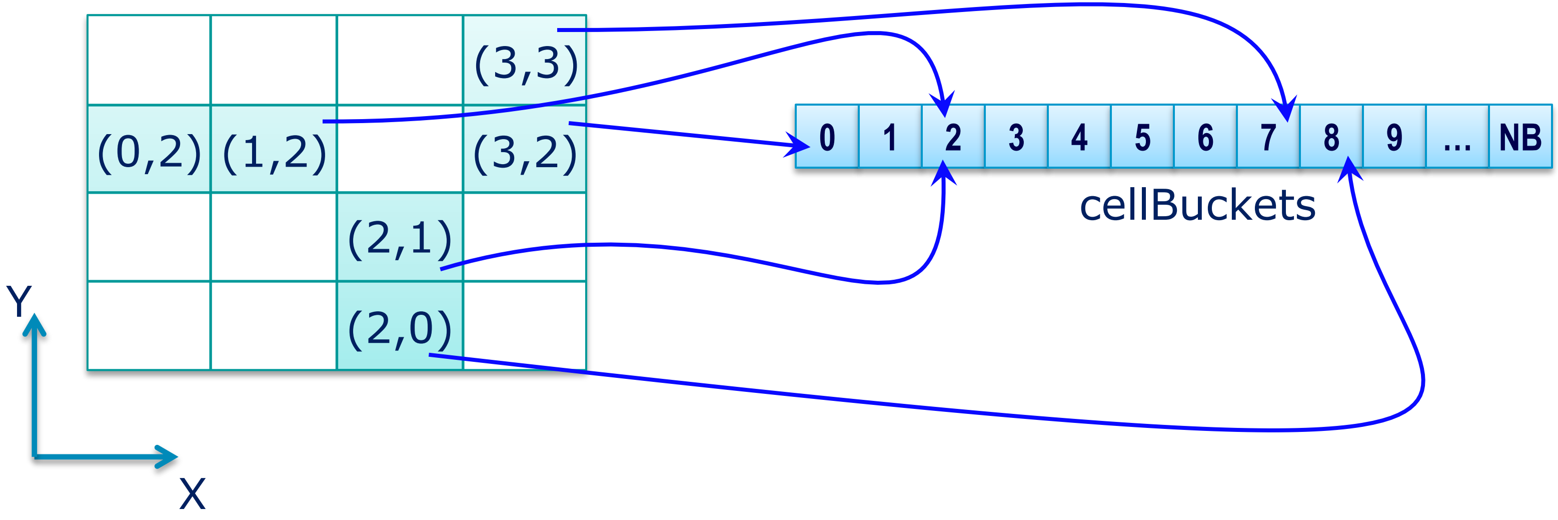
Implementation: spatial hash

- Consider the following grid



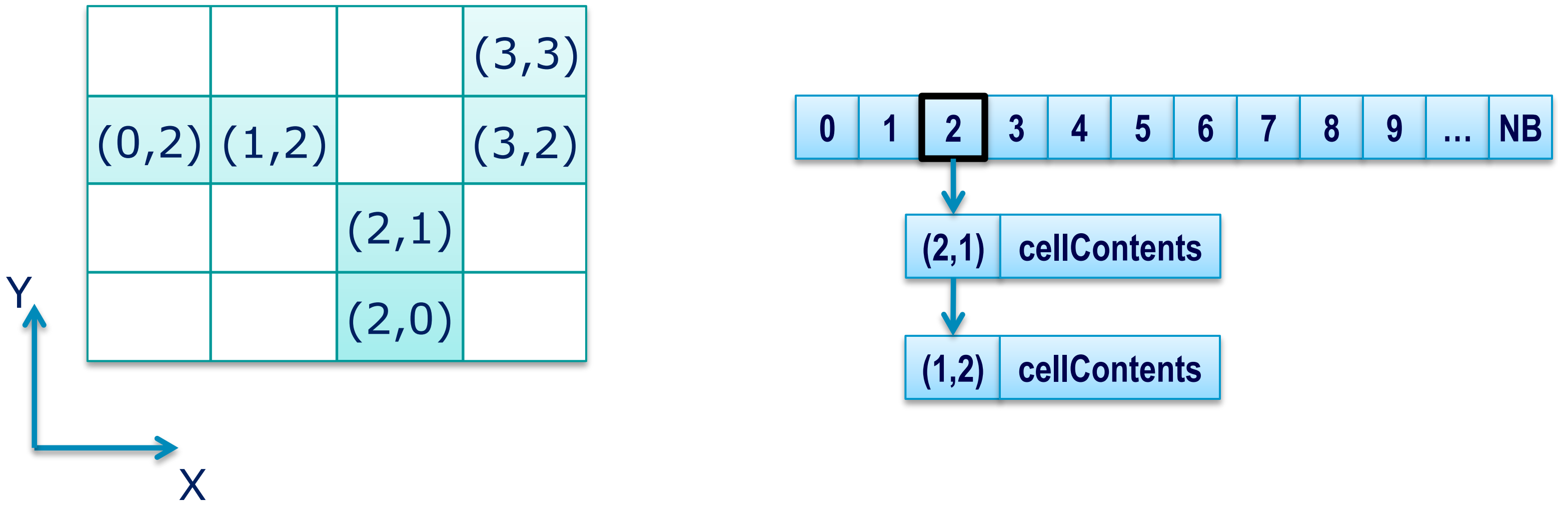
Implementation: spatial hash

- A multiplicative hash based on cell indices assigns each cell to a bucket



Implementation: spatial hash

- Each bucket contains a list of cells



Implementation: spatial hash

- Spatial hash grid data structures

```
Bucket
{
  Container<BucketRecord> records;
}

BucketRecord
{
  IntVector2 cellIndex;
  Cell cellContents;
}
```

```
SpatialHashGrid2D
{
  …
  Array<Bucket> cellBuckets;
}
```

Implementation: spatial hash

- Spatial hash grid

```
int getCellIndex(int d, Vector2 pt) { return (int)(floor(p[d]/cellSize[d])); }
float getCellPos(int d, int index) { return ((float)index) * cellSize[d]; }

int prime1 = 0xAB1D261;
int prime2 = 0x16447CD5;

int bucketAddress = (prime1 * i + prime2 * j) % numBuckets;
```

Implementation: spatial hash

- Spatial hash grid

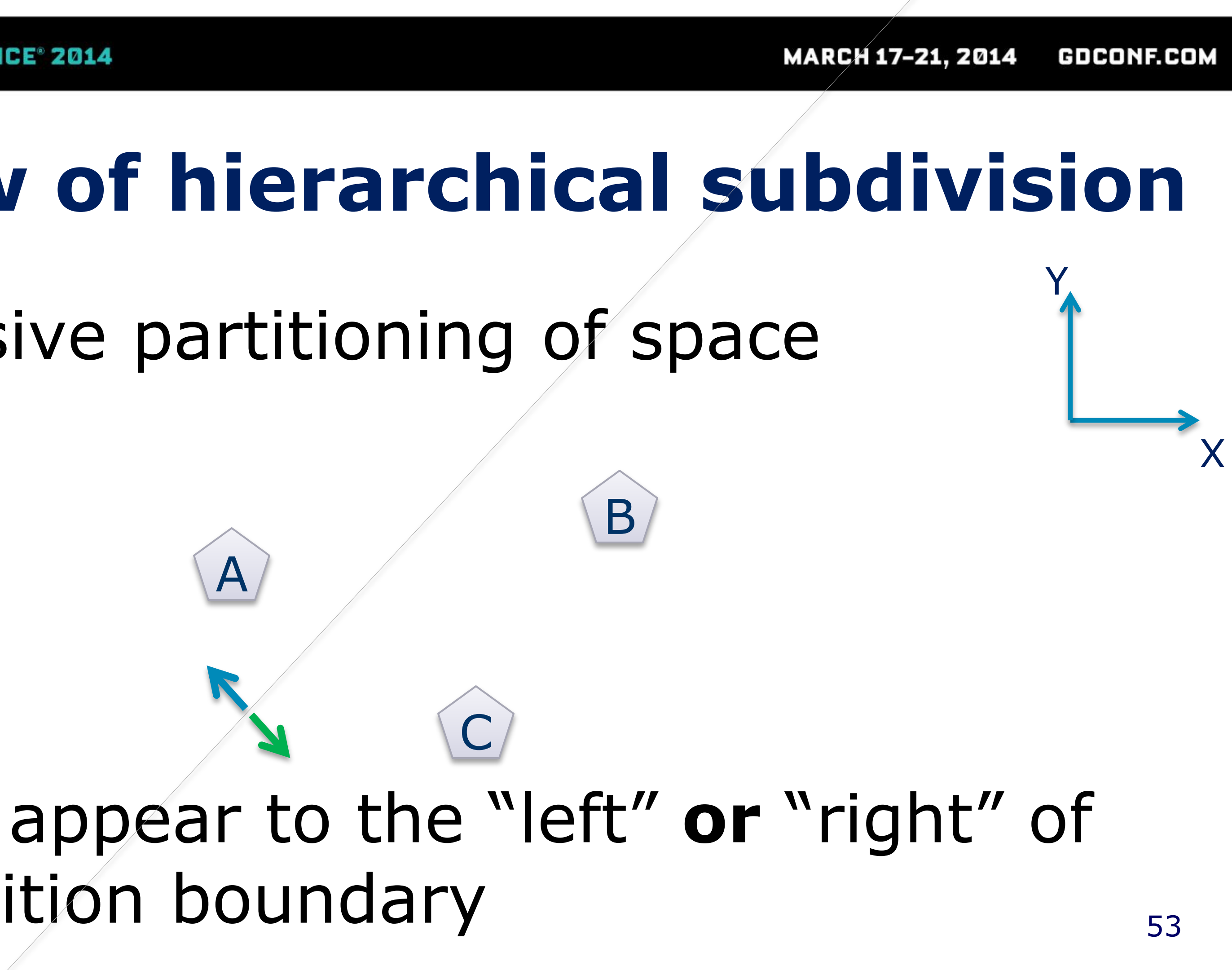
```
Cell getCell(Vector2 pt)
{
  int bucketAddress = getBucketAddress(pt);
  IntVec2 index = { getCellIndex(X, pt), getCellIndex(X, pt) };
  if (!cellBuckets[bucketIndex].contains(bucketAddress))
  {
    cellBuckets[bucketIndex].insert(new BucketRecord({
                                    cellIndex = index,
                                    cellContents = new Cell}));
  }
  return record.recordAt(bucketAddress);
}
```

Art history moment

Tree-based Spatial Subdivision

Overview of hierarchical subdivision

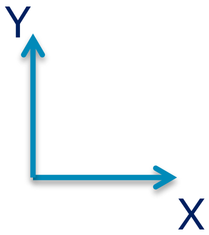
- A recursive partitioning of space

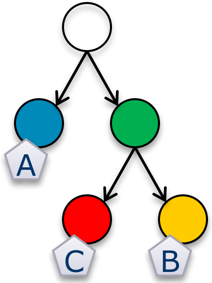


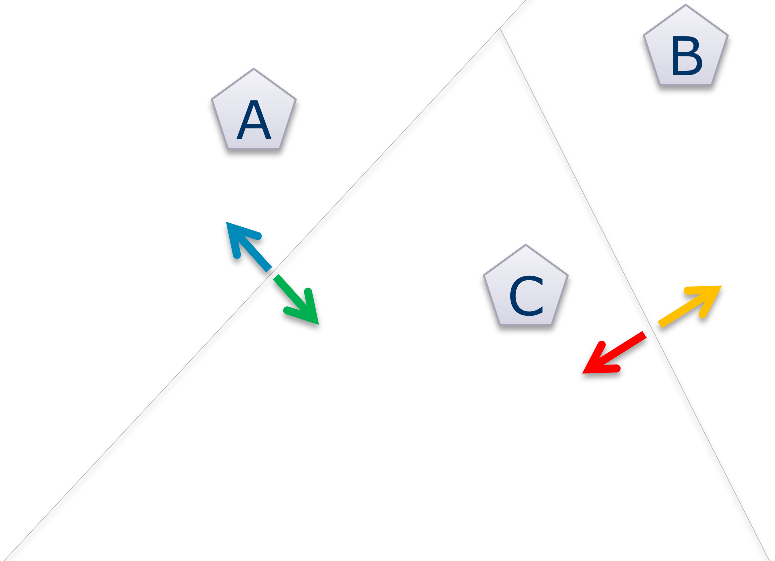
- Objects appear to the "left" **or** "right" of the partition boundary

Overview of hierarchical subdivision

- Notice that we can represent this as a binary tree

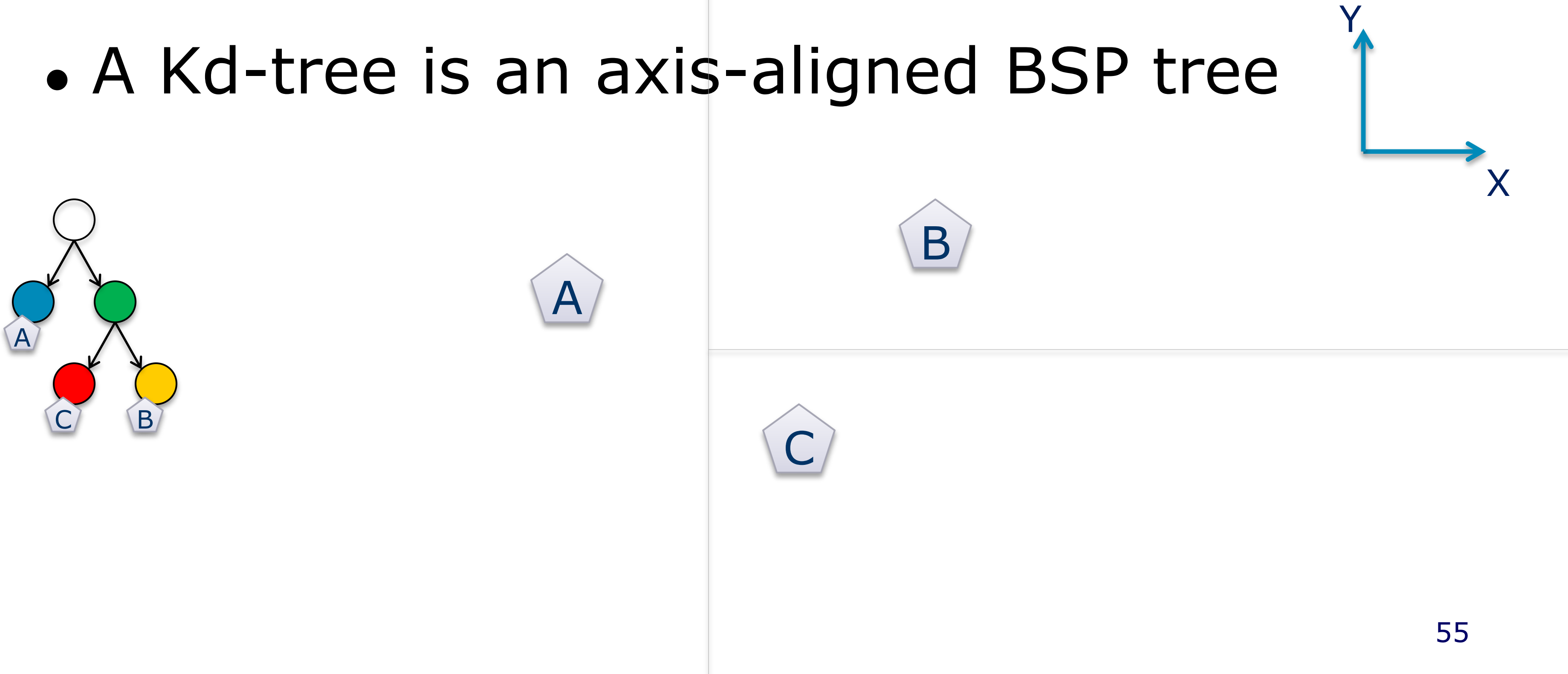






Implementation: Kd-tree

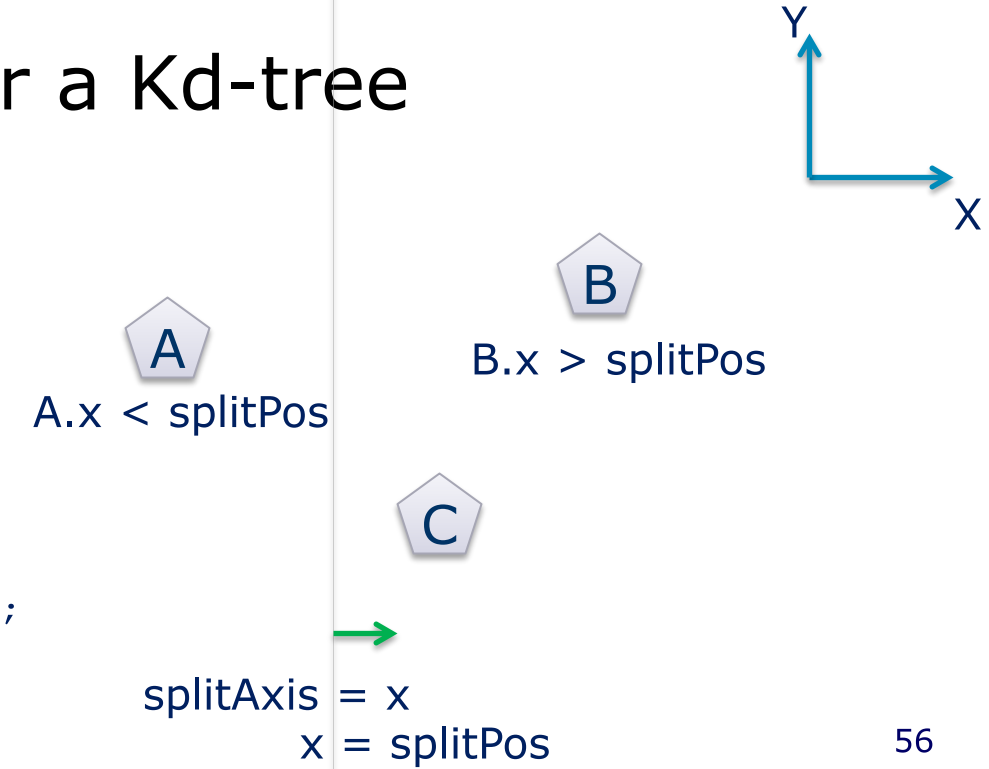
- A Kd-tree is an axis-aligned BSP tree



Implementation: Kd-tree

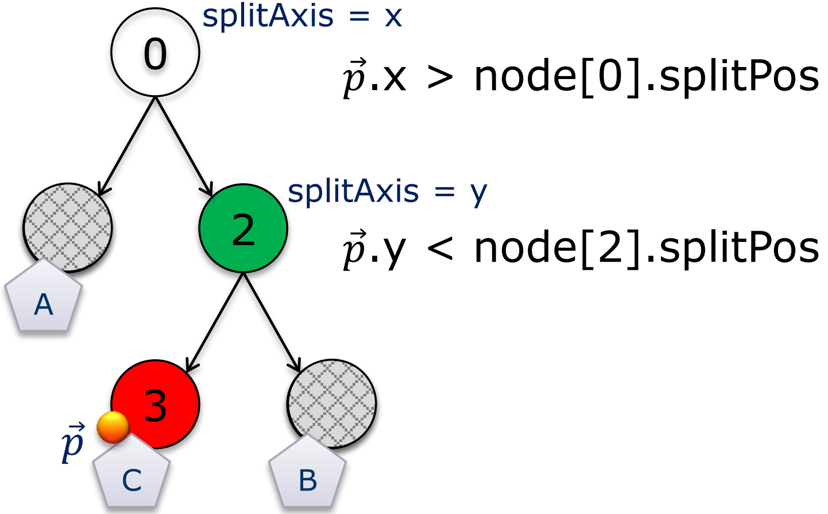
- Data structures for a Kd-tree

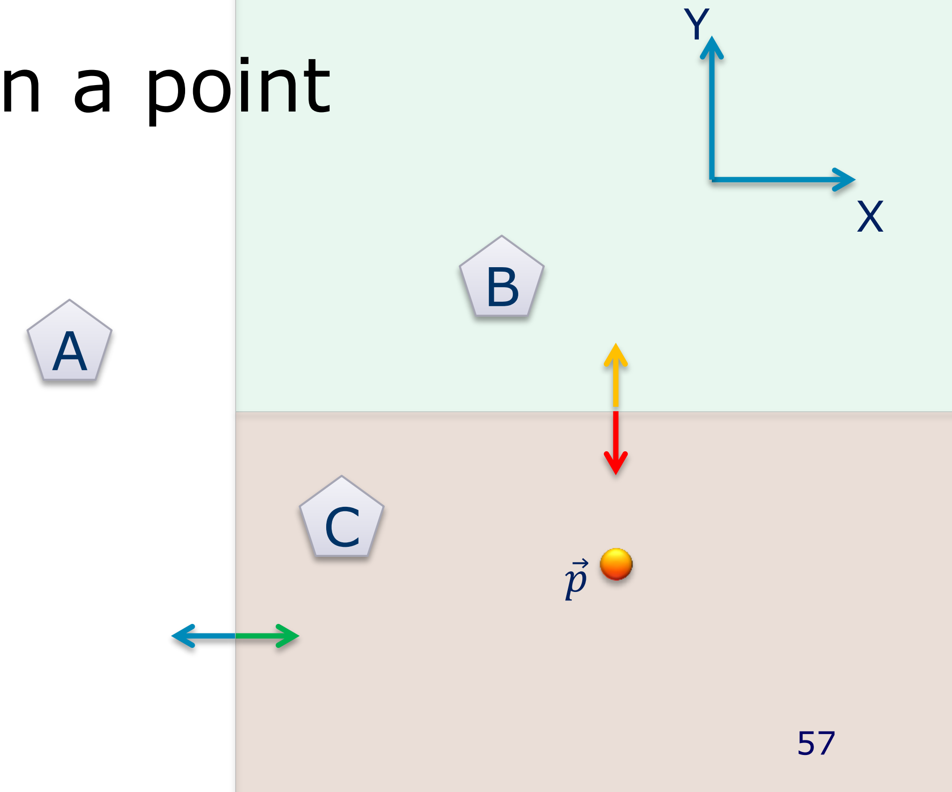
```
KdTree { KdNode rootNode; }

KdNode
{
  int nodeType;
  int splitAxis;
  float splitPos;
  union
  {
    KdNode *childNodes;
    Container<Object> gameObjects;
  }
}
```



Implementation: Kd-tree

- Locating a node given a point


splitAxis = x
0
$\vec{p}$.x > node[0].splitPos
splitAxis = y
2
$\vec{p}$.y < node[2].splitPos
A
3
$\vec{p}$
C
B



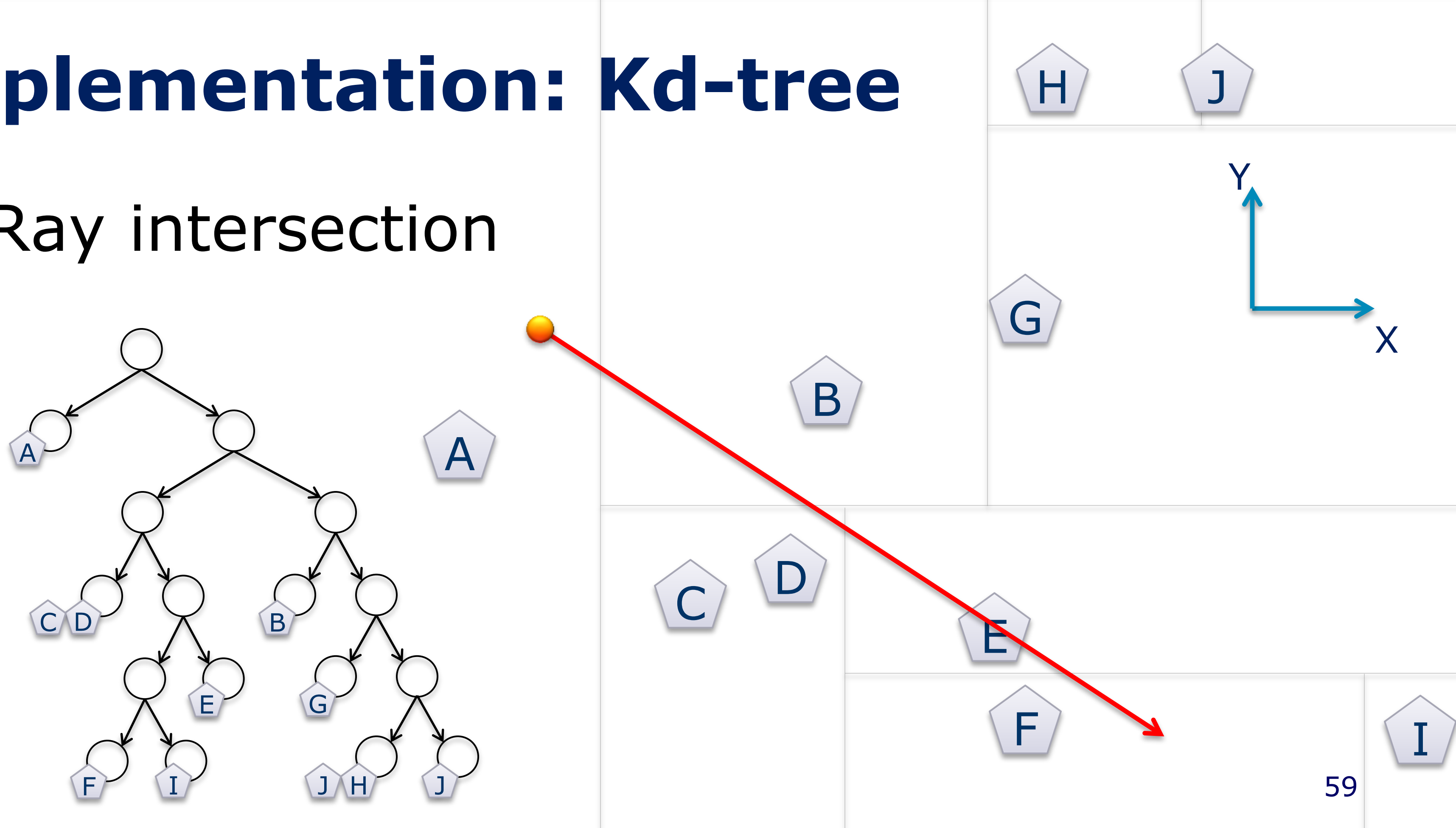
Implementation: Kd-tree

- Locating a node given a point

```
KdNode findNode(Vector2 pt)
{
  currentNode = rootNode;
  while (currentNode.hasChildren)
  {
    if (pt[currentNode.splitAxis] <= currentNode.splitPos)
      currentNode = currentNode.childNodes[0];
    else
      currentNode = currentNode.childNodes[1];
  }

  return currentNode;
}
```

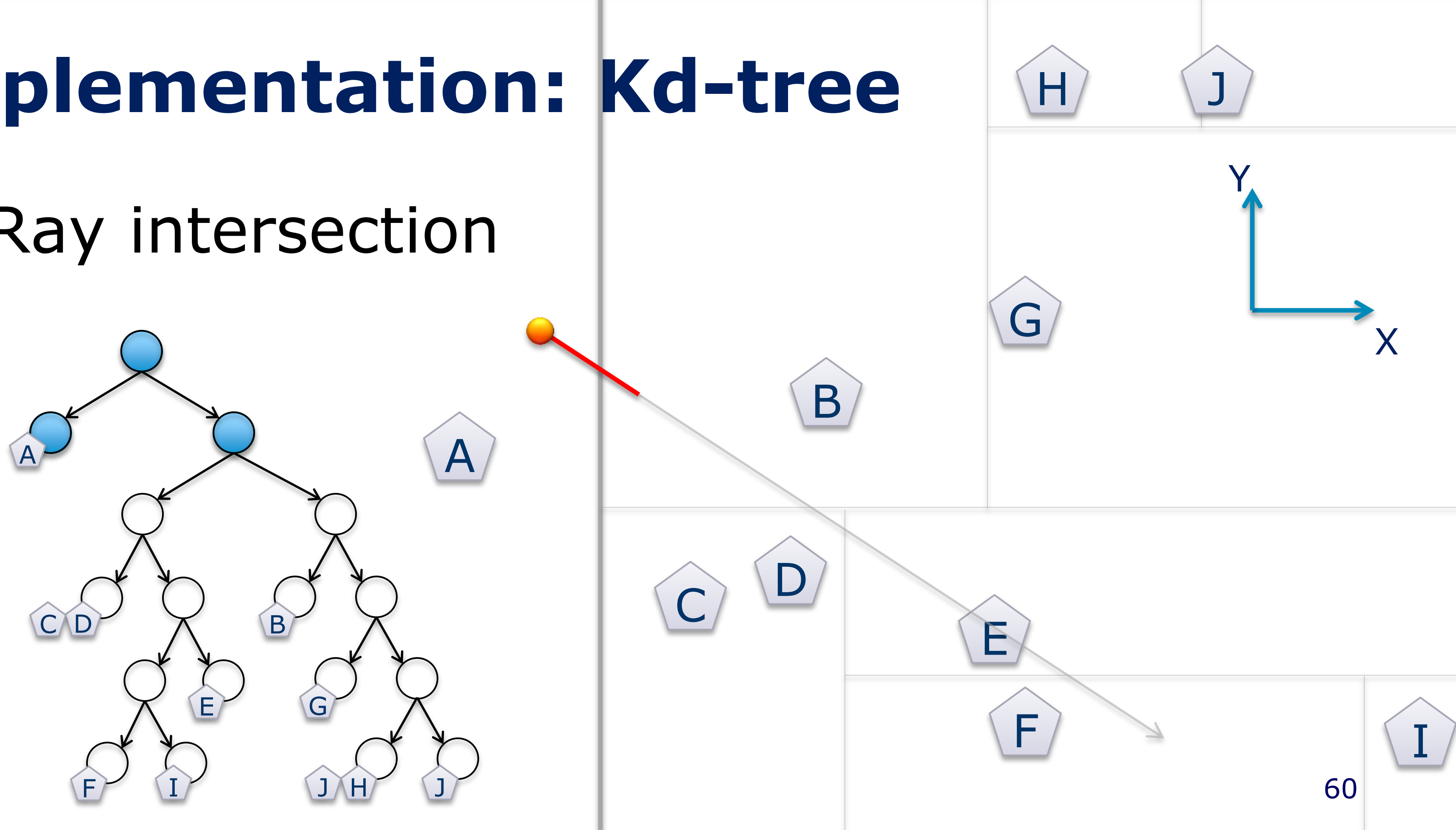
Implementation: Kd-tree

- Ray intersection



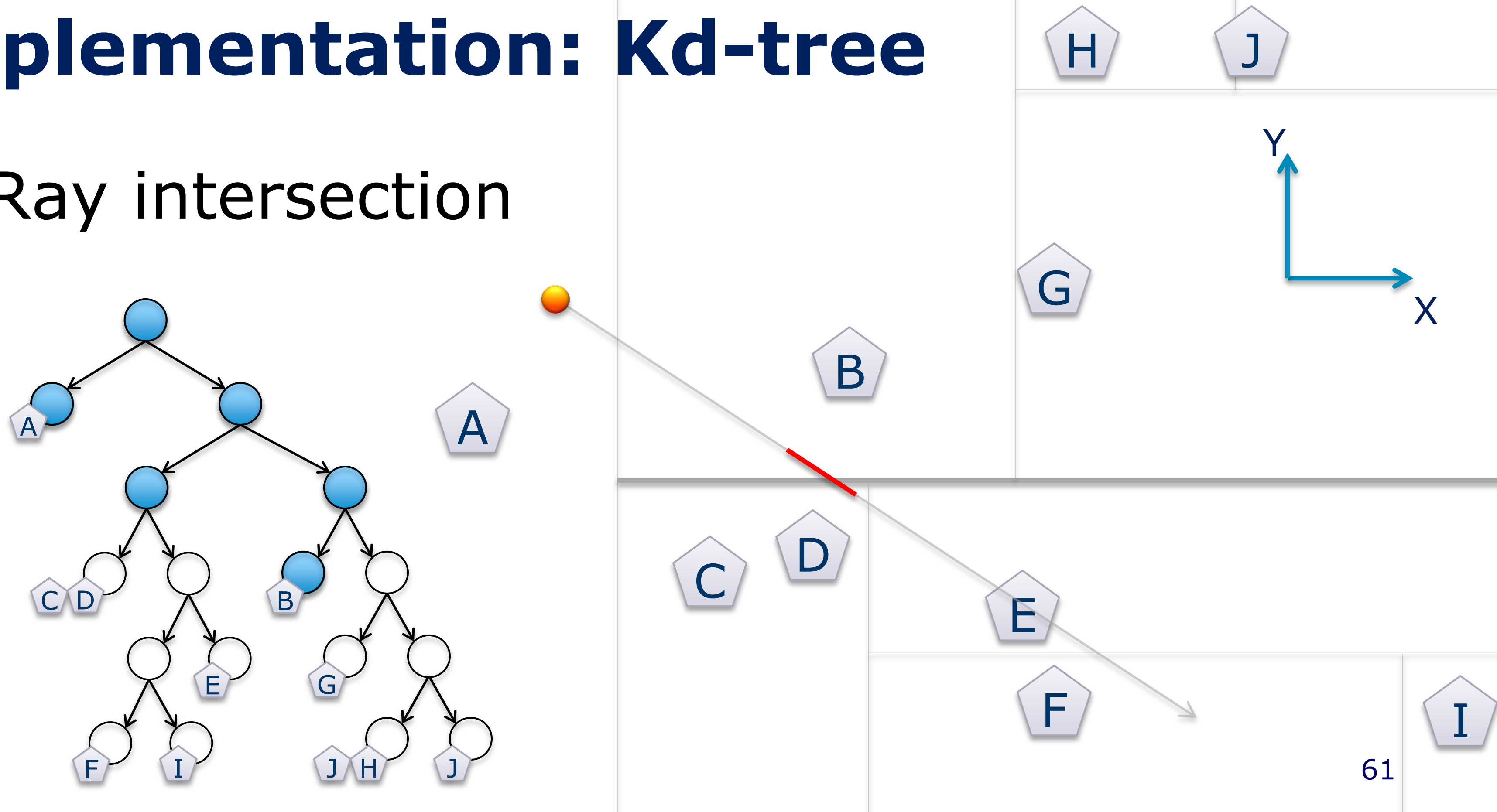
Implementation: Kd-tree

- Ray intersection



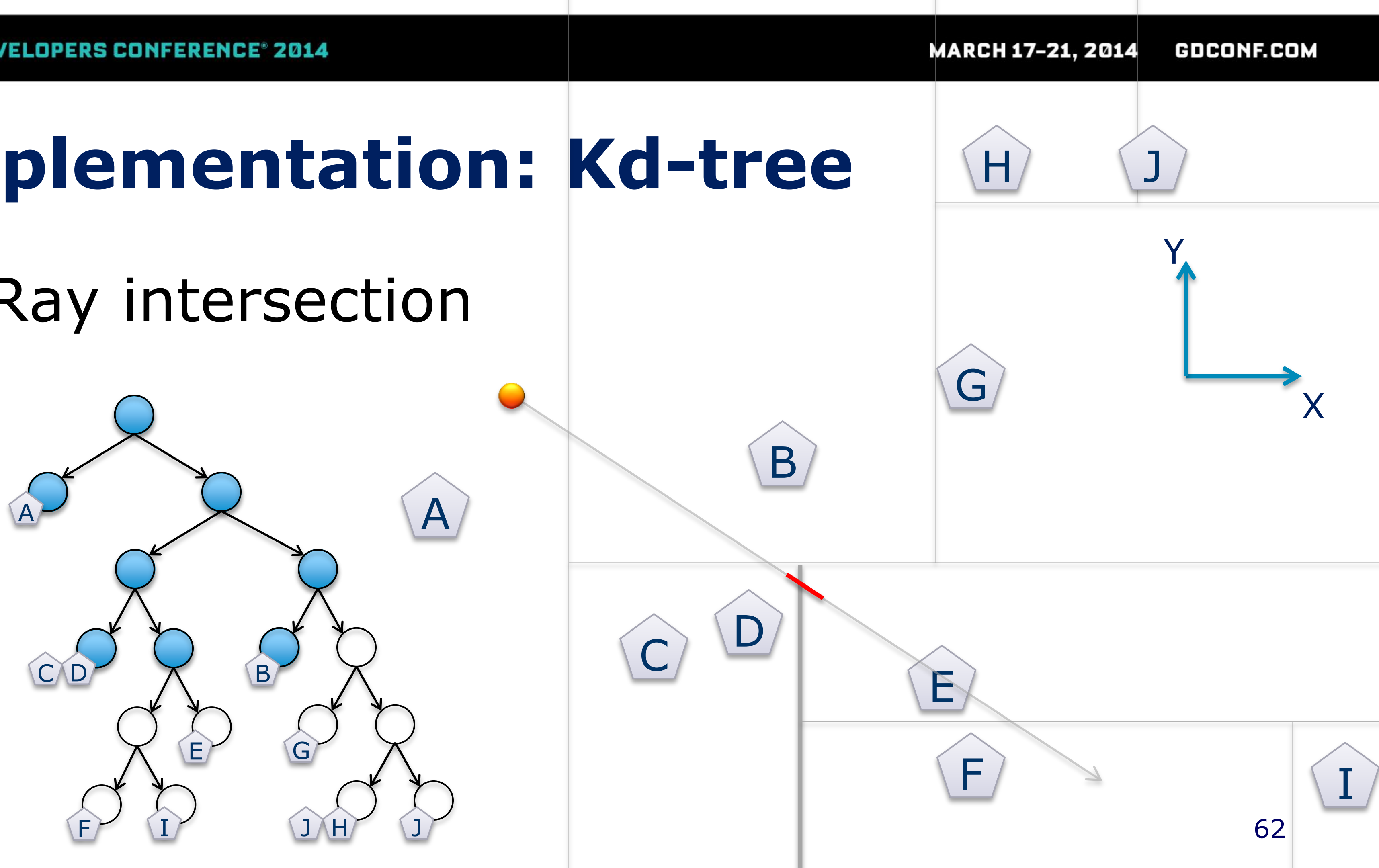
Implementation: Kd-tree

- Ray intersection



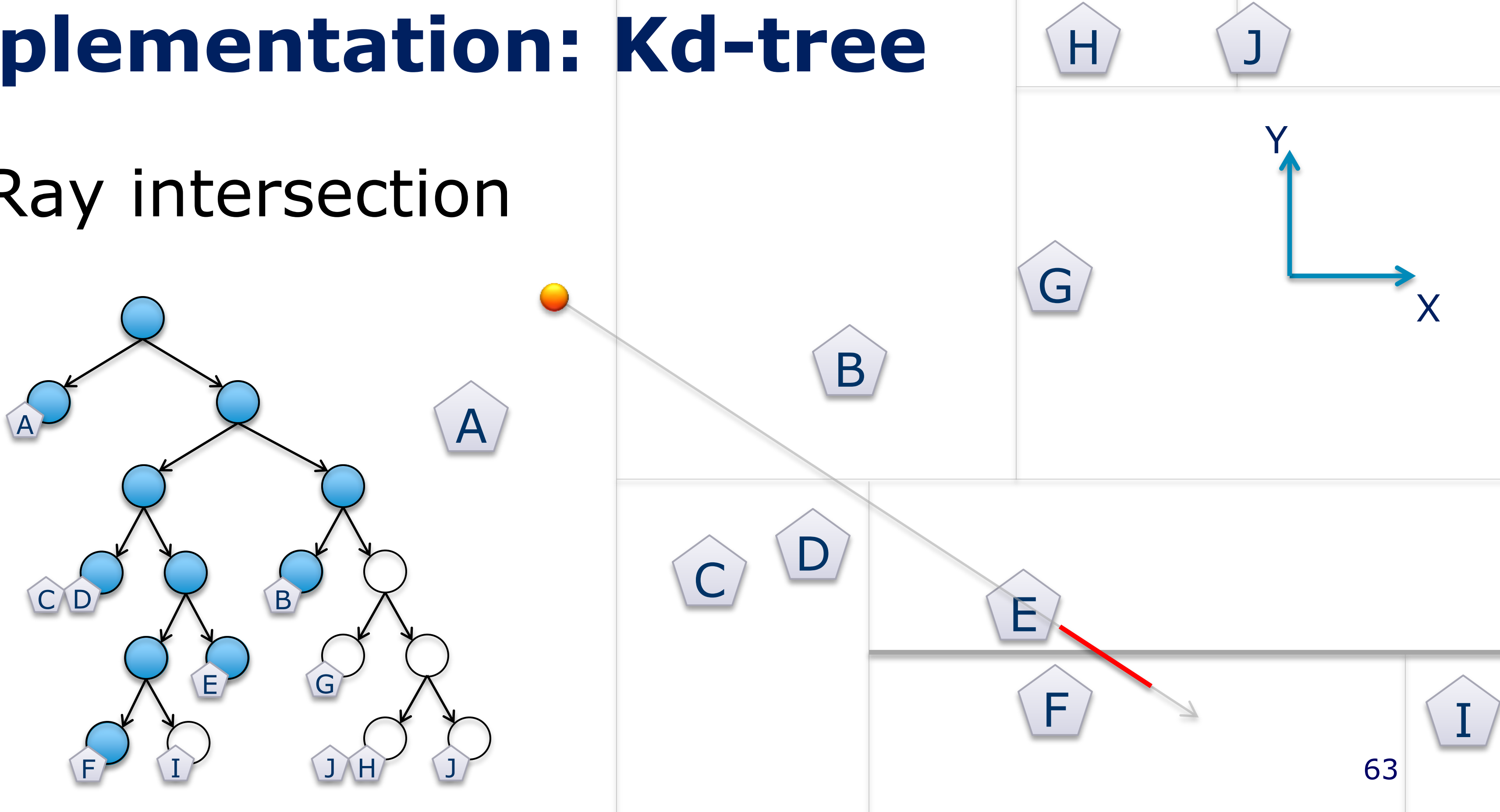
Implementation: Kd-tree

- Ray intersection



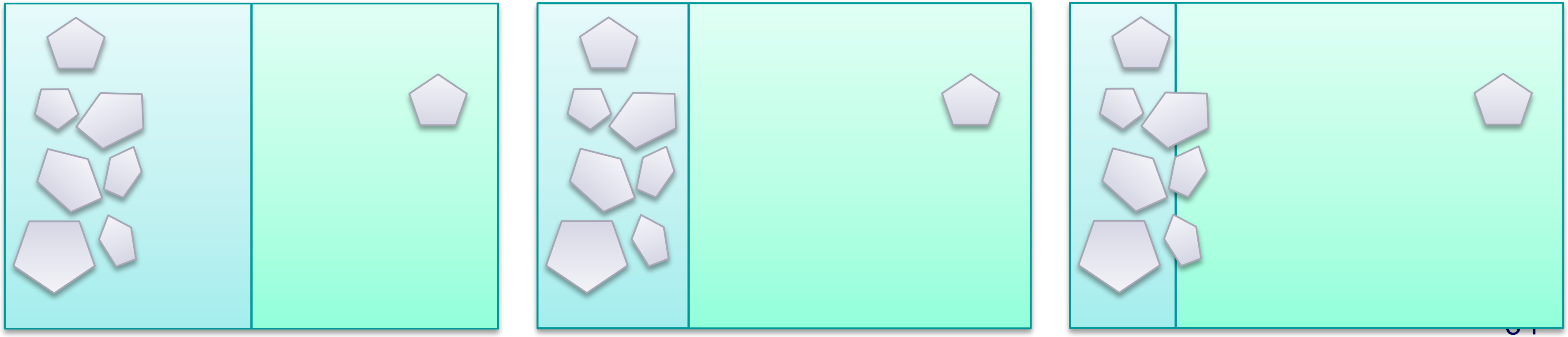
Implementation: Kd-tree

- Ray intersection



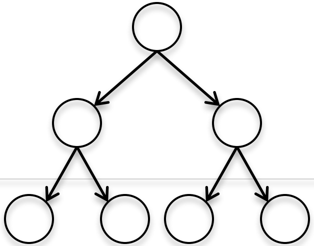
Implementation: Kd-tree

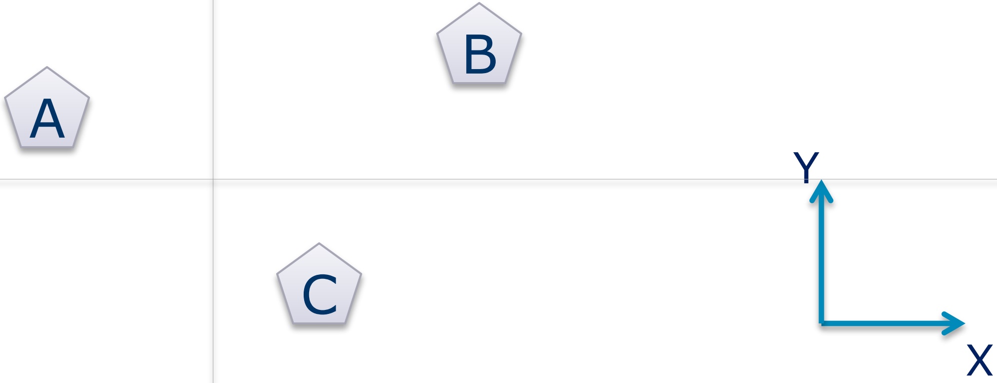
- Constructing a cost-optimized tree
  - Cost(cell) = Cost(traverse) + Probability(left hit)*Cost(left hit)+ Probability(right hit)*Cost(right hit)
  - Isolate complexity and seek large empty spaces
  - Deeply subdivided trees tend to be more efficient on modern hardware
  - Profile performance for your use case



Implementation: quadtree/octree

- If desired, a quadtree/octree can be implemented via a Kd-tree





Problems with hierarchical subdivision

- Not suitable for dynamic/moving objects

Memory cache considerations

- Typically 3-4 classes of system memory
    - L1 cache
    - L2 cache
    - L3 cache
    - Main memory
- Penalty to access to main memory w/cache miss
    - 50-200 clock cycles vs. 1-2 cycles for L1 cache hit
- Desirable to minimize occurrence of cache miss

Memory cache considerations

- Cache memory population
  - Cache lines on modern hardware are usually 32 or 64 bytes

Chipset/Processor	L1 Data Cache Line Size
Intel i7	64 bytes
Intel Atom	64 bytes
AMD Athlon 64	64 bytes
AMD Jaguar (Xbox One/PS4)	64 bytes
ARM Cortex A8	64 bytes
ARM Cortex A9	32 bytes

Cache considerations for grid

- Linked lists are bad. Real bad.
- Minimize structure size for cell bucket
  - Bucket record stores spatial index and pointer to cell. Cell data stored elsewhere
  - Closed hashing
  - Structure packing
  - Align buckets to cache-line boundaries
    - C++11 std::aligned_storage

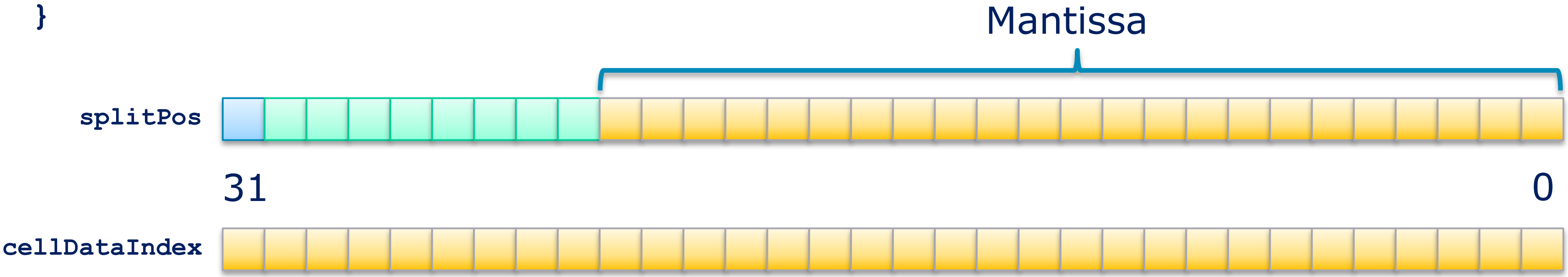
Cache considerations for Kd-tree

- With care and compromise, we can put a lot of tree into a single L1 cache line
  - Apply Christer Ericson’s bit packing approach
  - Cell data stored separate from tree itself
  - Binary heap data structure
  - Align structure to 64-byte boundary
  - A 64-byte cache line can store a fully subdivided 4 level Kd-tree
    - With 4 bytes left over to store sub-tree pointers

Cache considerations for Kd-tree

- Ericson's Compact KdNode

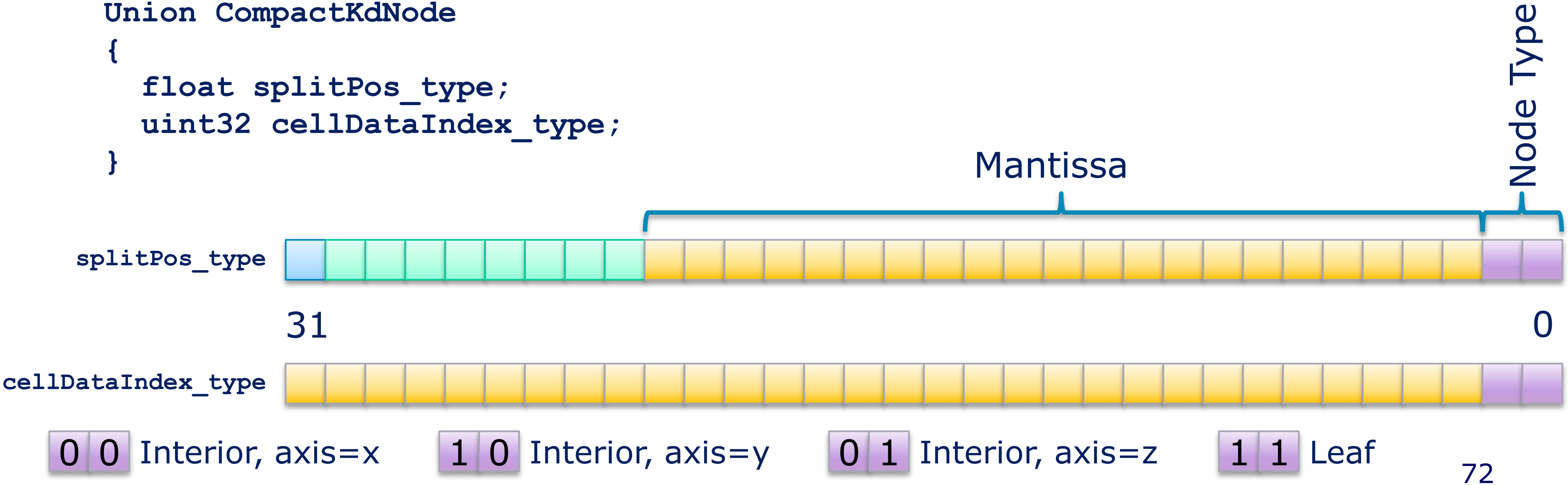
```
Union CompactKdNode
{
  float splitPos_type;
  uint32 cellDataIndex_type;
}
```



Cache considerations for Kd-tree

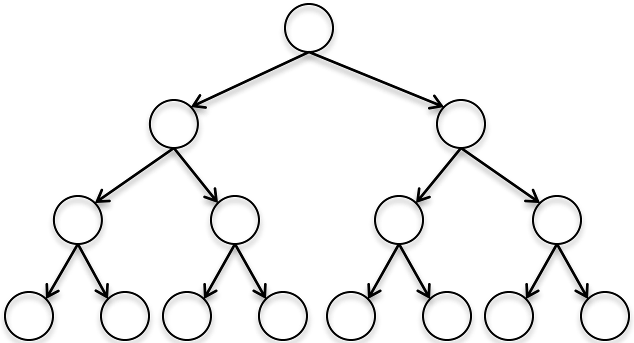
- Ericson’s Compact KdNode

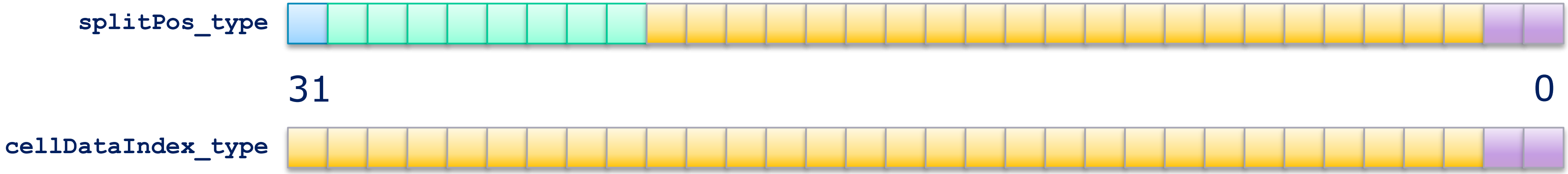
```
Union CompactKdNode
{
  float splitPos_type;
  uint32 cellDataIndex_type;
}
```



Cache considerations for Kd-tree

- Ericson’s Compact KdNode
  - 4 level Kd-tree = 15 nodes
  - 15 x 4 bytes = 60 bytes
  - 4 bytes left point to sub-trees





References

- Ericson, Christer, *Real-Time Collision Detection*, Morgan Kaufmann Publishers/Elsevier, 2005
- Hughes, John F., et al., *Computer Graphics Principles and Practice*, Third Edition, Addison-Wesley, 2014
- Pharr, Matt, and Greg Humphreys, *Physically-based Rendering*, Morgan Kaufmann Publishers/Elsevier, 2004
- Stoll, Gordon, "Ray Tracing Performance," SIGGRAPH 2005.
- Pascucci, Valerio, and Randall J. Frank, "Global Static Indexing for Real-time Exploration of Very Large Regular Grids," Supercomputing, ACM/IEEE 2001 Conference, 2001
- http://computinglife.wordpress.com/2008/11/20/why-do-hash-functions-use-prime-numbers/
- http://www.bigprimes.net/
- http://www.7-cpu.com

Any Questions?

Image is in the public domain. Painting date 1905. file source is http://commons.wikimedia.org/wiki/File:Hooiberg_by_emile_claus.jpg