Principles of Computer Game Design and Implementation

Lecture 14

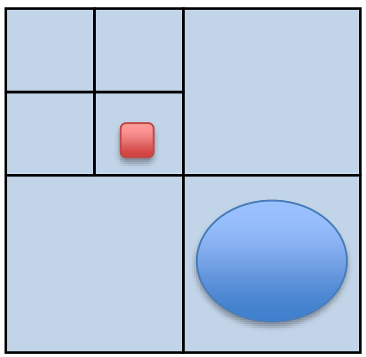
We already knew

- Collision detection – high-level view
  - Uniform grid

Outline for today

- Collision detection – high level view
  - Other data structures

Non-Uniform Grids



Idea: choose the cell size depending on what is put there

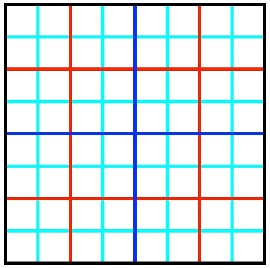
- Ideal for static objects

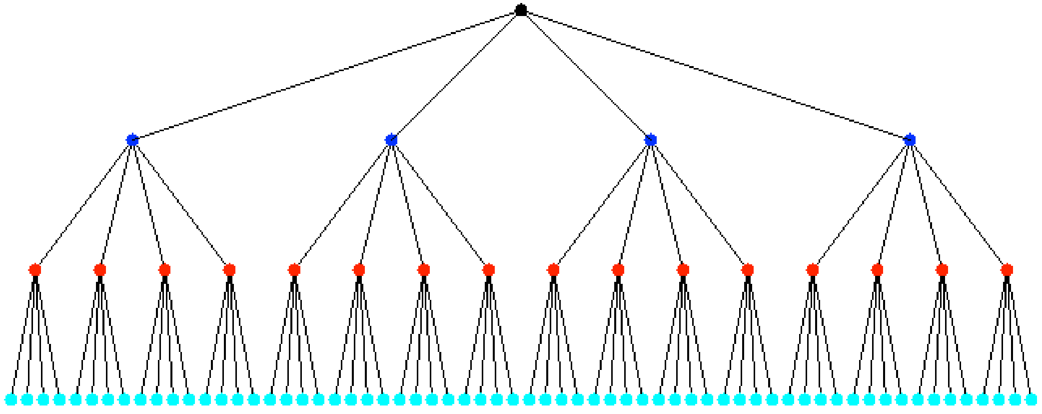
- Locating objects becomes harder
- Cannot use coordinates to identify cells
- Use *trees* and *navigate* them to locate the cell.

Quad- and Octrees

Quadtree: 2D space partitioning

- Divide the 2D plane into 4 (equal size) quadrants
  - Recursively subdivide the quadrants
  - Until a termination condition is met

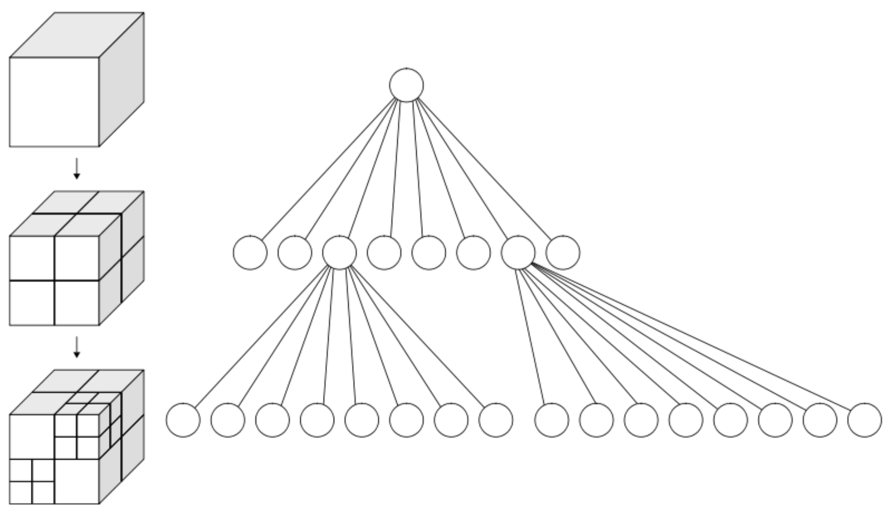




Quad- and Octrees

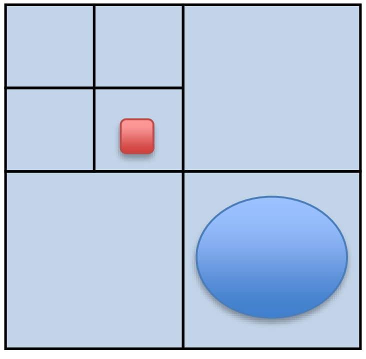
Octree: 3D space partitioning

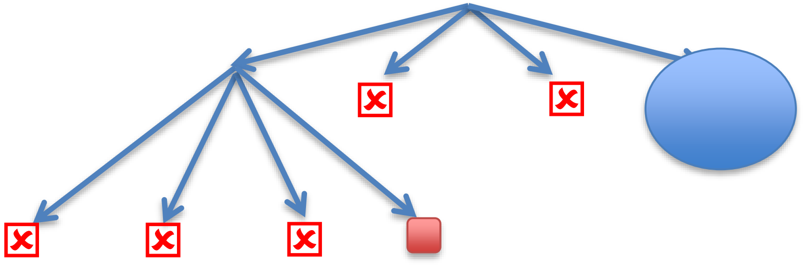
- Divide the 3D volume into 8 (equal size) parts
  - Recursively subdivide the parts
  - Until a termination condition is met



Termination Conditions

- Max level reached
- Cell size is small enough
- Number of objects in any sell is small

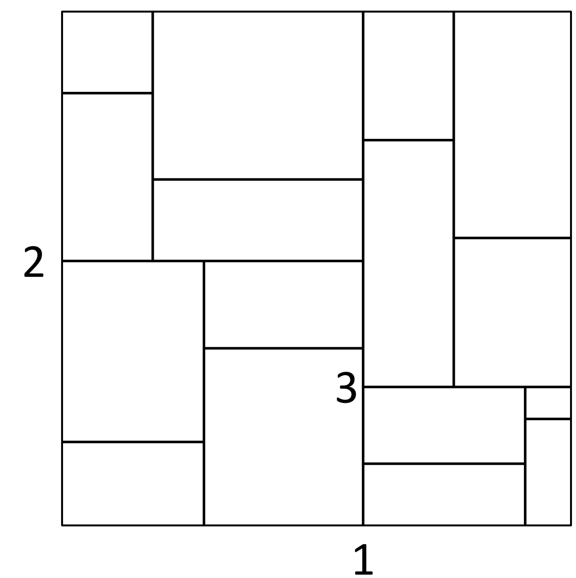


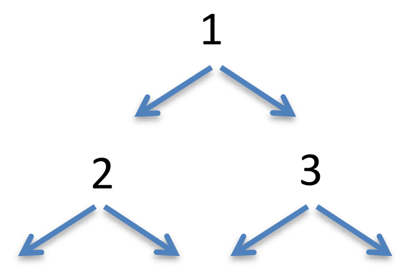


k-d Trees

k-dimensional trees

- 2-dimentional *k*-d tree
  - Divide the 2D volume into 2 parts vertically
    - Divide each half into 2 parts horizontally
      - Divide each half into 2 parts vertically
        - » Divide each half into 2 parts horizontally
          - Divide each half ….



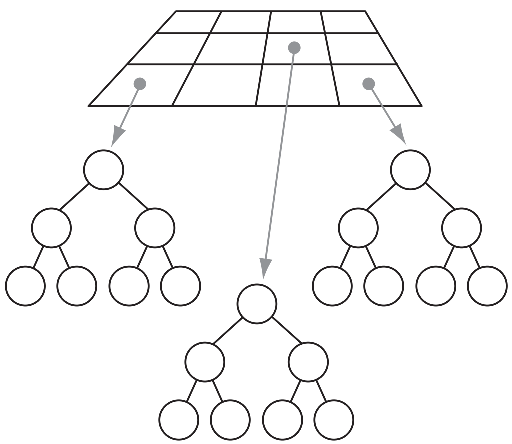


k-d Trees vs (Quad-) Octrees

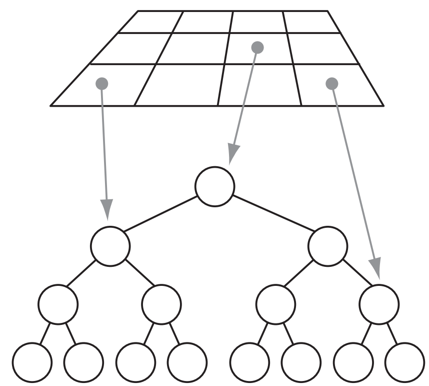
- For collision detection *k*-d trees can be used where (quad-) octrees are used
- *k*-d Trees give more flexibility
- *k*-d Trees support other functions
  - Location of points
  - Closest neighbour
- *k*-d Trees require more computational resources

Grid vs Trees

- Grid is faster
- Trees are more accurate
- Combinations can be used



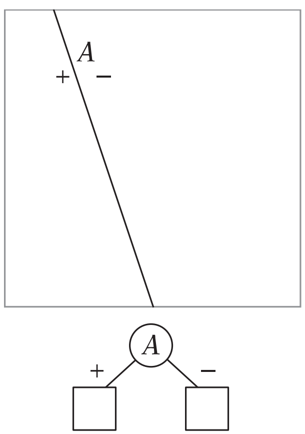
Cell to tree

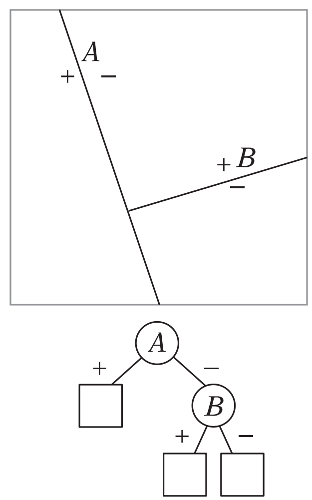


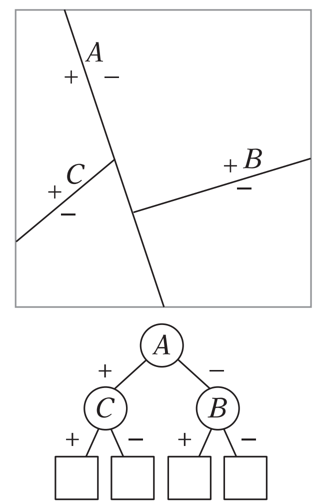
Grid to tree

Binary Space Partitioning

- BSP tree: recursively partition tree w.r.t. *arbitrary* dividing planes





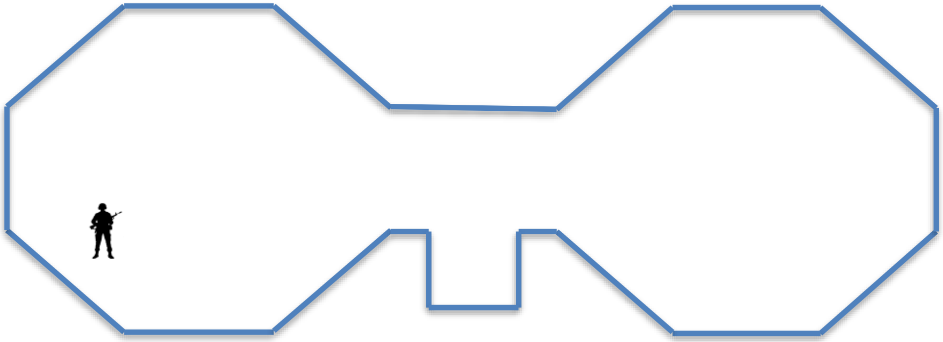


How To Partition

- Depend on the task
  - Originally for hidden-surface removal optimisation
  - Used in ray tracing
  - Used where octrees or *k*-d trees are used
- In many cases are *precomputed* in advance
  - DOOM, Quake,… for collision detection (among other things)

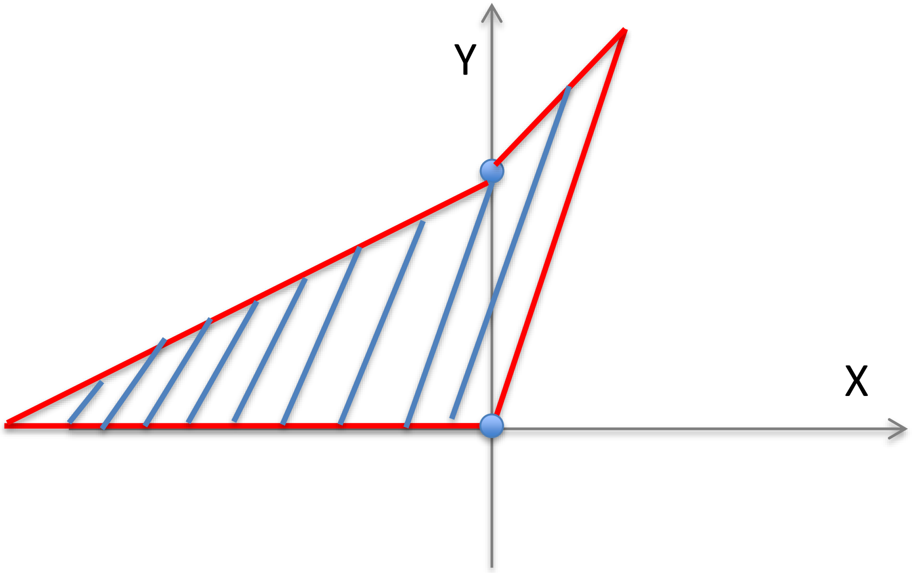
Solid-Leaf BSP Trees

- Build to represent “solid volume” occupied by the geometry
  - How to keep our hero in the room?

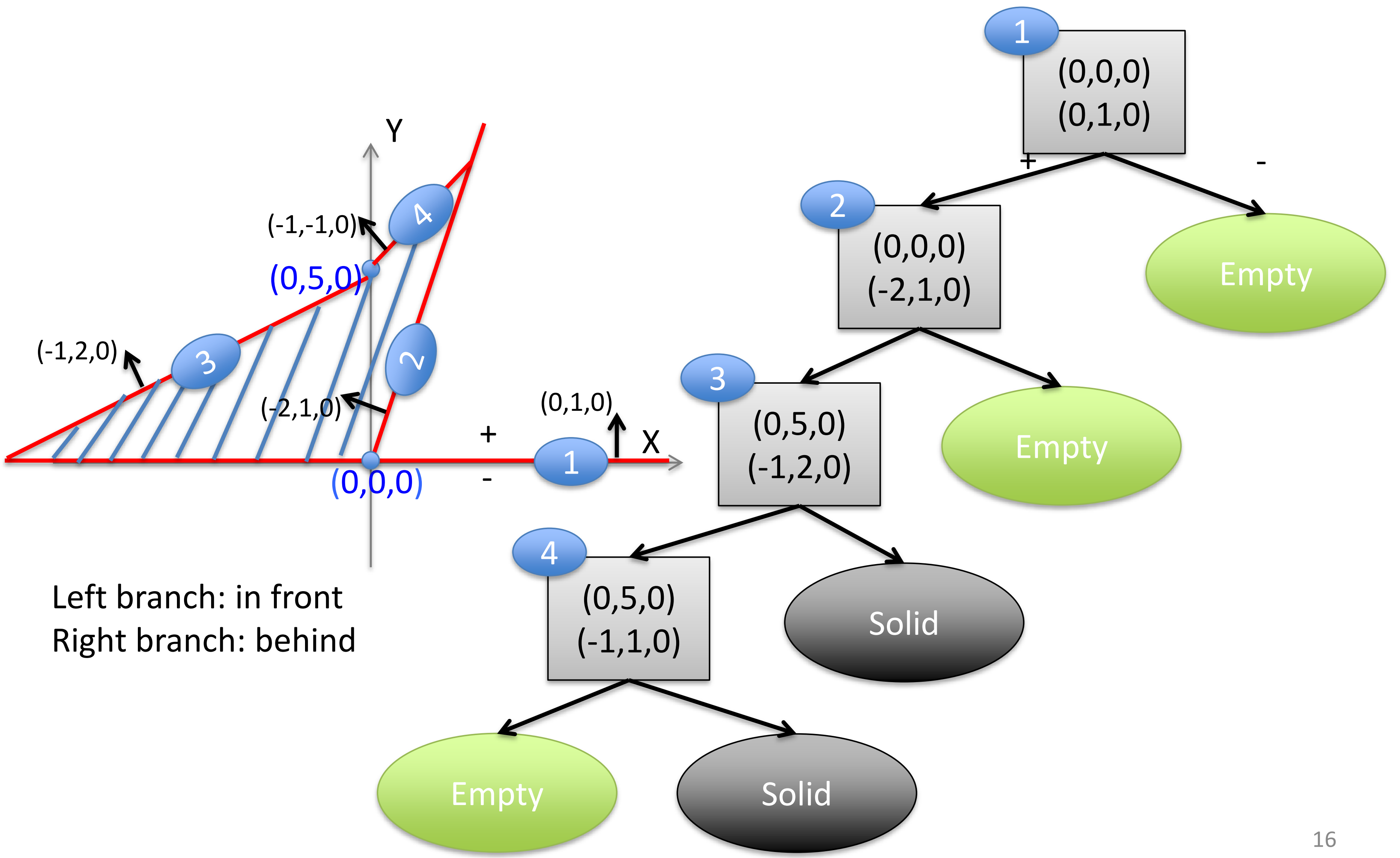


Floor plan

Space Partitioning



Space Partitioning



Left branch: in front
Right branch: behind

BSP Code (1)

```
class Plain {
  private Vector3f myPosition, myDirection;
  public Plain(Vector3f position, Vector3f direction) {
    myPosition = position;
    myDirection = direction;
  }

  public boolean isInFront(Vector3f pos) {
    if(pos.subtract(myPosition).dot(myDirection)>0) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Does not take the boundary into account

BSP Code (2)

```
enum NodeType {solid, empty, internal};

class BSPTree {
    NodeType myType;
    Plain myPlain;
    BSPTree myInfront, myBehind;
    public BSPTree(NodeType t) {
    // if((t != NodeType.empty) || (t != NodeType.solid))
    //                               throw new Exception();
      myType = t;
      myPlain = null;
      myInfront = null;
      myBehind = null;
    }
```

BSP Code (3)

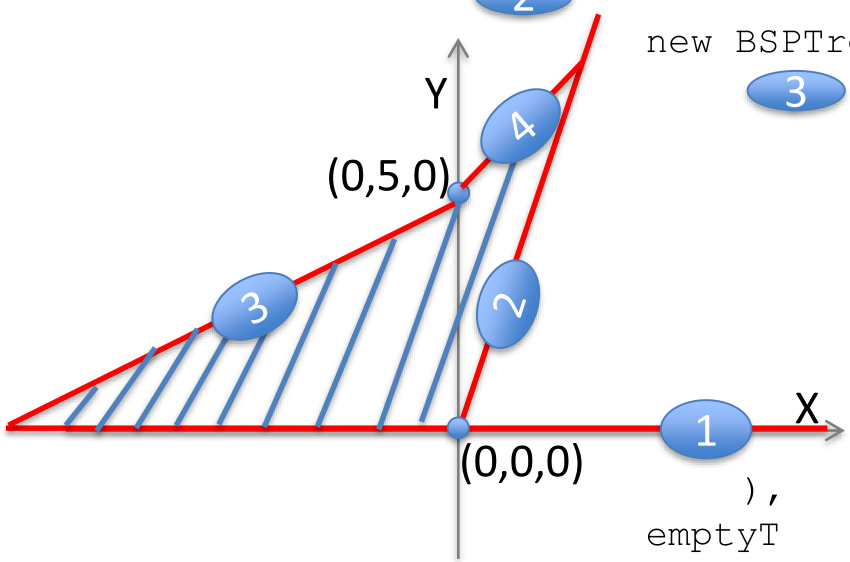
```
public BSPTree(Plain p, BSPTree infront, BSPTree behind) {
  myPlain = p;
  myType = NodeType.internal;
  myInfront = infront;
  myBehind = behind;
}
```

BSP Code (4)

```
public boolean isSolid(Vector3f pos) {
  if(myType == NodeType.solid) {
    return true;
  }
  if(myType == NodeType.empty) {
    return false;
  }
  if(myPlain.isInFront(pos)) {
    return myInfront.isSolid(pos);
  }
  else {
    return myBehind.isSolid(pos);
  }
}
```

BSP Code (5)

```
BSPTree solidT = new BSPTree(NodeType.solid);
BSPTree emptyT = new BSPTree(NodeType.empty)
t = new BSPTree(new Plain(new Vector3f(0,0,0), new Vector3f(0,1,0)),
                new BSPTree(new Plain(new Vector3f(0,0,0),
                                      new Vector3f(-2,1,0)),
                      new BSPTree(new Plain(new Vector3f(0,5,0),
                                            new Vector3f(-1,2,0)),
                            new BSPTree(new Plain(new
                                            Vector3f(0,5,0),
                                               new
                                            Vector3f(-1,1,0)),
                                  emptyT,
                                  solidT
                            ),
                            solidT
                      ),
                      emptyT
                ),
                emptyT
);
```



BSP Code (6)

```
private AnalogListener analogListener = new
AnalogListener() {
  public void onAnalog(String name,
           float value, float tpf){
    if(name.equals("Move right")){
      Vector3f newPos =
(ball.getLocalTranslation().add(Vector3f.UN
IT_X.mult(10*tpf)));
      if(t.isSolid(newPos)){
         ball.setLocalTranslation(newPos);
      }
  }
```

Conclusion

- Hierarchical data structures help on both mid- and high-level collision detection
- About 10% of console memory is spent on collision detection data structures
- Collision detection is easy when the number of entities is small, but becomes a challenge when the number grows.