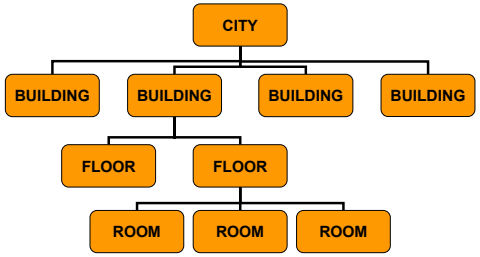
Space Subdivision

- Divides space into smaller regions
- May be hierarchical
- Can be used in 2D or 3D
- Only load/process/render relevant regions
- Several catagories:
  - Structural
  - Partitions
  - Bounding volumes

Structural subdivision

- World organised according to the natural structure and sub-structure of items in the scene



Structural subdivision (cont.)

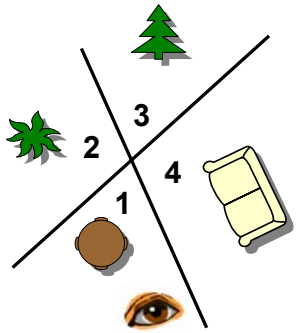
```
class City
{   // A city is a bunch of buildings
    vector<Building> buildings;
};
class Building
{   // A building has some floors
    vector<Floor> floors;
    City* city;
    BBox boundingbox;
};
class Floor
{   // A floor has a number of rooms
    vector<Room> rooms;
    Building* building;
    int floor_number;
};
class Room
{   // A room has some stuff in it, and access to other rooms
    Floor* floor;
    vector<Room*> adjacent_rooms;
    ...
};
```


Partition planes

- Space is subdivided by placing on or more partition planes
- A partition plane divides space into three:
  1. Region in front of the plane
  2. Region behind the plane
  3. Region on the plane
- Some subdivision schemes use only two regions:
  1. Region in front of the plane
  2. Region behind the plane
  - Region on the plane is considered to be either in front or behind (but be consistent!)

Partition planes

- Partition planes can define a depth order
  - Everything on the other side of a partition plane from the camera is further away than everything on the same side of the partition plane as the camera



Anything in {1,2} appears in front of anything in {3,4}

Anything in {1,4} appears in front of anything in {2,3}

1 is always in front of {2,3,4}

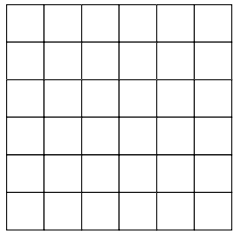
{2, 4} is always in front of 3

Nothing in 2 occludes anything in 4, and vice versa

Depth order: 1 (nearest), 2&4 (any order), 3 (furthest)

Regular grid

- Place axis-aligned partition planes at regular intervals
- Forms a regular grid over world
- Each grid square is a small block of the world
- Each block contains info about what is in the block
- Ideal for map-based worlds
- Easily implemented as a 2D array
- Usually the grid is square



Regular grid (cont.)

```
// A block contains whatever
class Block
{
        list<Monster> monsters;
        list<Model> objects;
        ...
};

// The world is a regular 2D array of blocks
Block world[WORLD_BLOCKS][WORLD_BLOCKS];
```


Regular grid (cont.)

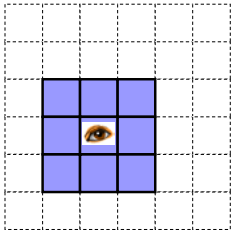
```
class Player
{
        float world_x, world_y;   // World position of player
        int block_x, block_y;     // Block coordinates
};

Player::Move(float dx, float dy)
{
        world_x += dx;
        block_x = (int)floor(world_x);
        world_y += dy;
        block_y = (int)floor(world_y);
}
```

Regular grid (cont.)

```
const int subbits = 8;

class Player
{
   unsigned int world_x, world_y; // World coordinates

   unsigned int block_x, block_y; // Block coordinates
   unsigned int sub_x, sub_y;     // Sub-block coordinates
};

Player::Move(int dx, int dy)
{
   world_x += dx;

   block_x = world_x >> subbits;  // Remove lower 8 bits
   sub_x = world_x & ((1 << subits) - 1);  // Mask out upper bits

   ... // Ditto for y
}
```


Regular grid (cont.)

- Only process blocks near the player:
  - Geometry upload
  - Monster AI
- Far-away blocks can be removed
- Incremental loading as player moves
- Limit memory usage



Regular grid (cont.)

```
Render(Player& player, int range)
{
        int bx = player.block_x;
        int by = player.block_y;
        for(int y = by-range; y <= by+range; y++)
        {
                for(int x = bx-range; x <= bx+range; x++)
                {
                        if(IsInWorld(x, y))
                        {
                                Block& block = FetchBlock(x, y);
                                block.Render();
                        }
                }
        }
}
```


Regular grid (cont.)

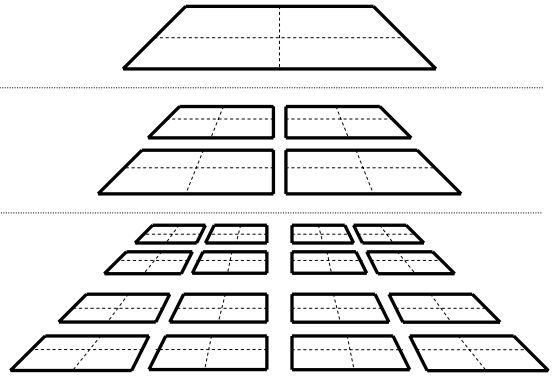
```
Block& FetchBlock(int x, int y)
{
   if(!BlockIsInCache(x, y))
   {
        if(CacheIsFull())
                RemoveLeastUsedBlockFromCache();
        Block block = ReadBlockFromDisk(x, y);
        AddBlockToCache(block);
   }

   return BlockFromCache(x, y);
}
```

Quadtrees

- Regular grid in a hierarchy
- Divides world into four blocks using two axis-aligned partition planes
- Divides each block into four sub-blocks
- … and so on
- Each block is linked to its for sub-blocks
  - Forms a tree with four children per node

Quadtrees (cont.)

- Level 0
- Level 1
- Level 2
- …



Quadtrees (cont.)

```
class Block
{
        Block* parent;
        Block* children[4];
        BBox boundingbox;
        ...
};

// The world is the top-level block
Block world;
```


Quadtrees (cont.)

- Number of blocks at a level:

  $n(level) = 4^{level}$

- Total number of blocks for all levels:

  $N(level) = 4^0 + 4^1 + 4^2 + \ldots + 4^{level}$

  $= (4^{(level+1)} - 1) / 3$

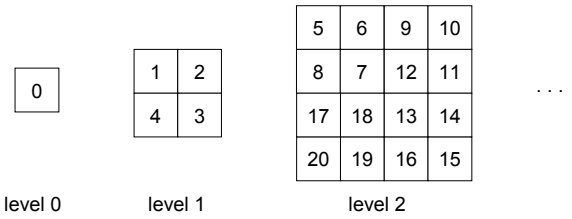
- So overhead for using hierarchy instead of regular grid is:

  $N(level) / n(level) < 4/3$

  No more than 33% extra blocks

Quadtrees (cont.)

- Storing a quadtree without pointers
  - Can store as a single array
  - Enumerate the blocks as follows:



Quadtrees (cont.)

- Block index of first block in level:
  start(level) = N(level) – n(level)
  = $(4^{level} - 1) / 3$
- Block index of child block c={0,1,2,3} of block b:
  child(b, c) = start(level+1) + (b – start(level))*4 + c
  = $(4^{(level+1)} - 1) / 3 + 4\ b - 4\ (4^{level} - 1) / 3 + c$
  = $4\ b + (4^{(level+1)} - 1) / 3 - (4^{(level+1)} - 4) / 3 + c$
  = 4 b + 1 + c
- Block index of parent of block b:
  parent(b) = (b – 1) / 4 (integer division)
  Block b is child (b – 1) % 4 of parent

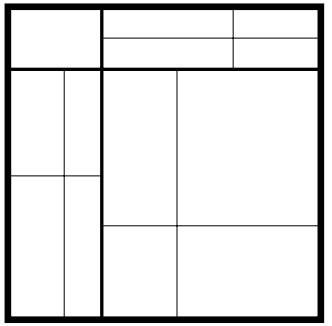
Quadtrees (cont.)

- Level number of block b:
  - Not fast but simple way
    level(b) = floor($(3\ (b + 1))^{1/4}$)
  - The "at least we're avoiding floats" way:
    - Need to find largest power of 4 <= 3 (b + 1)

```
b' = 3 (b + 1)
     if(b' & (3 << 30))       level(b) = 15
else if(b' & (3 << 28))       level(b) = 14
else if(b' & (3 << 26))       level(b) = 13
. . .
else if(b' & (3 << 4))        level(b) = 2
else if(b' & (3 << 2))        level(b) = 1
else                          level(b) = 0
// b' is never 0
```


Quadtrees (cont.)

- Subdivision does not always need to be done in the middle
  - Split according to some criteria, e.g. same number of objects in each quadrant
- Subdivision does not always need to be done to the same level
  - E.g. stop when number of objects in quadrant less than threshold



Quadtrees (cont.)

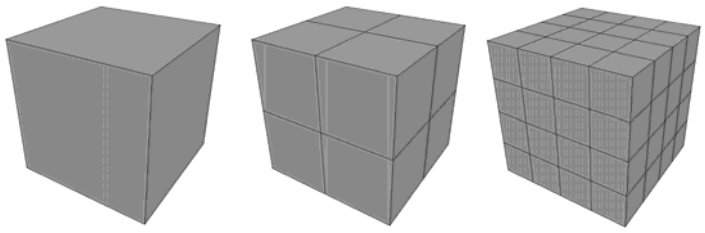
```
class Block
{
    Pos bbox[2];                        // Bounding box (x0,y0)->(x1,y1) of block
    Block* parent;
    Block* children[4];
};

Block* MakeBlock(float x0, float y0, float x1, float y1, Block *parent)
{
    Block* block = new Block();
    block->bbox[0].x = x0; block->bbox[0].y = y0;
    block->bbox[1].x = x1; block->bbox[1].y = y1;
    block->parent = parent;

    Pos split;
    if(NeedToSubdivide(block, split))   // Determine if block needs to be subdivided
    {                                   // and if so, where to split
          block->children[0] = MakeBlock(x0, y0, split.x, split.y, block);
          block->children[1] = MakeBlock(split.x, y0, x1, split.y, block);
          block->children[2] = MakeBlock(split.x, split.y, x1, y1, block);
          block->children[3] = MakeBlock(x0, split.y, split.x, y1, block);
    }

    return block;
}
```


Quadtrees (cont.)

```
// Find the block that a given point is in
Block* FindBlock(Block* block, Pos p)
{
    for(int c = 0; c < 4; c++)
    {
          if(block->children[c])
                if(IsInBBox(p, block->children[c]->bbox))
                      return FindBlock(block->children[c], p);
    }

    return block;
}

bool IsInBBox(Pos p, Pos bbox[2])
{
    return (bbox[0].x <= p.x) && (p.x < bbox[1].x) &&
           (bbox[0].y <= p.y) && (p.y < bbox[1].y);
}
```

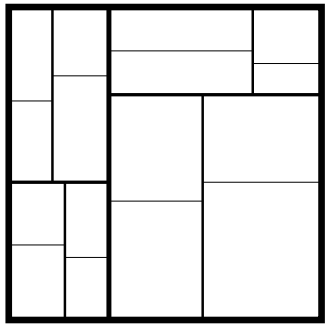

Octrees

- Like quadtrees, extended to 3D
- Subdivides a volume
- Divides a cube into 8 sub-cubes using three axis-aligned partition planes



kD-Trees

- A hierarchical binary 2D subdivision
- Splits world by alternating between axis-aligned partition planes
  - E.g. first split along Y axis, then X axis, then Y axis, etc.
- Obvious extension to 3D

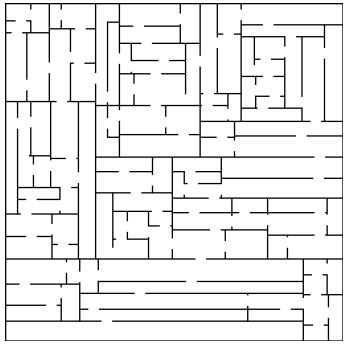


kD-Trees (cont.)

```
class Block
{
     Pos bbox[2];                    // Bounding box (x0,y0)->(x1,y1) of block
     Block* parent;
     Block* children[2];
};


Block* MakeBlock(float x0, float y0, float x1, float y1, Block *parent, int direction)
{
     Block* block = new Block();
     block->bbox[0].x = x0; block->bbox[0].y = y0;
     block->bbox[1].x = x1; block->bbox[1].y = y1;
     block->parent = parent;

     Pos split;
     if(NeedToSubdivide(block, direction, split))
     {
            if(direction == X_AXIS)
            {
                        block->children[0] = MakeBlock(x0, y0, x1, split.y, block, Y_AXIS);
                        block->children[1] = MakeBlock(x0, split.y, x1, y1, block, Y_AXIS);
            }
            else if(direction == Y_AXIS)
            {
                        block->children[0] = MakeBlock(x0, y0, split.x, y1, block, X_AXIS);
                        block->children[1] = MakeBlock(split.x, y0, x1, y1, block, X_AXIS);
            }
     }
     return block;
}
```


kD-Trees (cont.)

- Making a random maze with kD-trees
  - Place wall horizontally or vertically
  - Make door in wall
  - Repeat for sub-blocks
- Only place wall if it begins and ends at a wall
- Guarantees that there is exactly one path from any room to any other room



BSP trees

- Binary Space Partition
- Hierarchical, like quadtrees and kD-trees
- Splits a volume into two sub-volumes by plane
- Split each sub-volume into two
- … and so on
- Generalisation of kD-trees, as the plane can be arbitrary, not just axis-aligned

BSP trees (cont.)

- BSP tree used primarily to organise polygons
  - Partition plane for a tree node chosen to be the plane of a polygon
  - Add polygons on the plane to the node
  - Add polygons in front of the plane to one child
  - Add polygons behind the plane to the other child
  - Split polygons which fall across the plane
- Partition plane does not *need* to be co-incident with a polygon plane

BSP trees (cont.)

```
struct BSPnode
{
   Plane plane;                  // The partition plane
   list polygons;                // Polygons on the plane
   BSPnode *front, *back;        // Links to the two children
};

void BuildTree(BSPnode *node, list polygons)
{
   Polygon *poly = polygons.GetPolygon();  // Take poly from list
   node->plane = plane->GetPlane();        // Get the plane of poly
   node->polygons.AddPolygon(poly);        // Add poly to node

   list backpolys, frontpolys;
```


BSP trees (cont.)

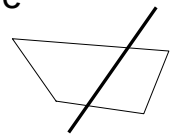
```
while(poly = polygons.GetPolygon())     // Go through remaining polys
{
      int type = ClassifyPoly(poly, plane);
      if(type == COINCIDENT)            // Poly on plane
               node->polgons.AddPolygon(poly);
      else if(type == BEHIND)           // Poly behind plane
               backpolys.AddPolygon(poly);
      else if(type == INFRONT)          // Poly in front of plane
               frontpolys.AddPolygon(poly);
      else if(type == SPANNING)         // Poly across plane
      {
               Polygon *frontpiece, *backpiece;
               SplitPolygon(poly, plane, frontpiece, backpiece);
               backpolys.AddPolygon(backpiece);
               frontpolys.AddPolygon(frontpiece);
      }
}
```


BSP trees (cont.)

```
   if(!frontpolys.Empty())
   {
         node->front = new BSPnode();
         BuildTree(node->front, frontpolys);
   }
   if(!backpolys.Empty())
   {
         node->back = new BSPnode();
         BuildTree(node->back, backpolys);
   }
}
```


BSP trees (cont.)

- Splitting a polygon by a plane
  - Convex polygons: easy
    - Always creates 2 convex polys



  - Concave polygons: hard
    - Can create any number of concave polys
    - Avoid using concave polygons

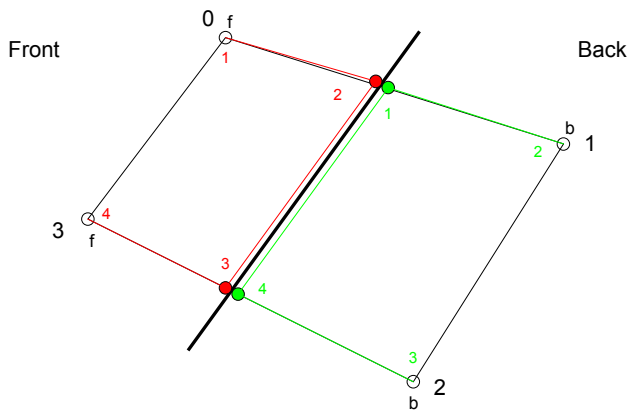
BSP trees (cont.)

```
void SplitPolygon(Polygon *poly, Plane plane, Polygon *&front, Polygon *& back)
{
    int numv = poly->NumVertices()
    Point p1, p2;
    float side1, side2;

    front = new Polygon(); back = new Polygon();
    p1 = poly->Vertex(numv-1);
    side1 = ClassifyPoint(point1, plane);
    for(int v = 0; v < numv; v++)
    {
        p2 = poly->Vertex(v);
        side2 = ClassifyPoint(p2, plane);
        if(side2 > 0)
        {       // Point p2 in front of plane
                if(side1 < 0)
                {       // Edge crossed from back to front of plane
                        Point intersect = EdgePlaneIntersect(p1, p2, plane);
                        front->AddPoint(intersect);
                        back->AddPoint(intersect);
                }
                front->AddPoint(p2);
        }
```


BSP trees (cont.)

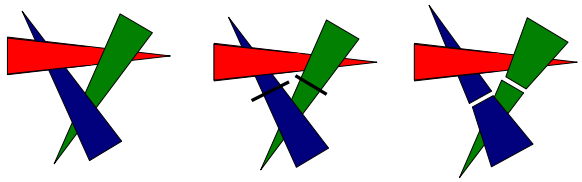
```
        else if(side2 < 0)
        {       // Point p2 is behind plane
                if(side1 > 0)
                {       // Edge crossed from front to back of plane
                        Point intersect = EdgePlaneIntersect(p1, p2, plane);
                        front->AddPoint(intersect);
                        back->AddPoint(intersect);
                }
                back->AddPoint(p2);
        }

        else
        {       // Point p2 is on plane
                front->AddPoint(p2);
                back->AddPoint(p2);
        }
        p1 = p2;
        side1 = side2;
    }
}
```


BSP trees (cont.)



BSP trees (cont.)

- The good thing about BSP trees:
  - Depth ordering without ambiguity
    - Any polygon in the tree is either strictly in front of or behind any other polygon with respect to a partition plane (or co-planar)



BSP trees (cont.)

- The bad thing about BSP trees:
  - Creates a polygon soup, which makes efficient rendering difficult
  - Tree can be unbalanced
    - Find best polygon to place plane on, e.g. near centroid of all polygons, or with equal number of polygons in both children

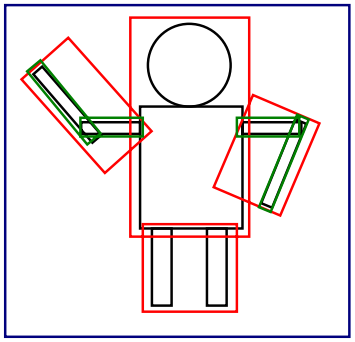
Bounding volumes

- Instead of partitioning by a plane, divide space by a volume
- Use a mathematically simple shape to define a boundary
  - Bounding boxes
  - Bounding spheres
- Subdivide by dividing the bounded volume by multiple smaller bounding volumes
- Typically used to subdivide objects, but can also be used to subdivide a world
- Optimal bounding volumes typically determined by pre-processing

Bounding volumes (cont.)

- Bounding boxes
  - Have a big box around everything
  - Subdivide into smaller boxes
  - The union of children does not need to be the same volume as the parent
  - Boxes may be axis-aligned (AABB), or rotated to give a tighter fit

Bounding volumes (cont.)



Bounding volumes (cont.)

```
class AABB                    // Axis-aligned bounding box
{
    bool isleaf;              // true if this is a leaf, false otherwise
    Pos box[2];               // Bounding box in world coordinates

    AABB* parent;
};

class AABBNode : public AABB  // A non-leaf node in the AABB tree
{
    list<AABB*> children;
};

class AABBLeaf : public AABB  // A leaf node in the AABB tree
{
    vector<Polygons> polygons;
};
```

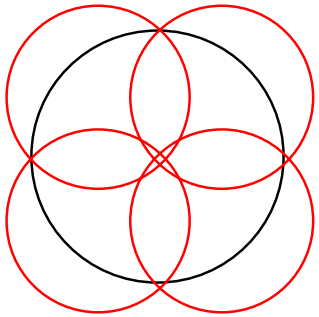

Bounding volumes (cont.)

```
class BoundingBox
{
   bool isleaf;
   Pos centre;          // Centre of box, in parent's coordinates
   float size[3];       // Size of box
   Matrix rot;          // Rotation around centre, in parent coords

   BoundingBox* parent;
};
```

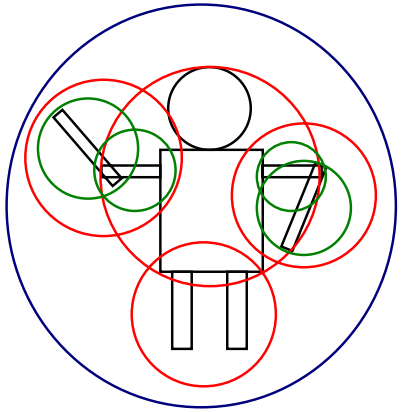

Bounding volumes (cont.)

- Spheres
  - Easy to compute if point is in a sphere

    |p – centre| < radius
  - Often used for fast collision detection
    - Sphere defines boundary of some structure
    - Sub-spheres define boundaries of sub-structures



Bounding volumes (cont.)



Bounding volumes (cont.)

```
class BoundingSphere
{
    bool isleaf;              // true if this is a leaf, false otherwise
    Pos centre;
    float radius;

    BoundingSphere* parent;
};

class BoundingSphereNode : public BoundingSphere
{
    list<BoundingSphere*> children;
};

class BoundingSphereLeaf : public BoundingSphere
{
    vector<Polygons> polygons;
};
```


Space Subdivision (cont.)

- Various space subdivisions can be mixed together
  - E.g., quadtree down to some minimum area size, then a regular grid in the leaf nodes
    - Fast processing over large areas (hierarchical)
    - Fast rendering over small areas (display list, vertex array)