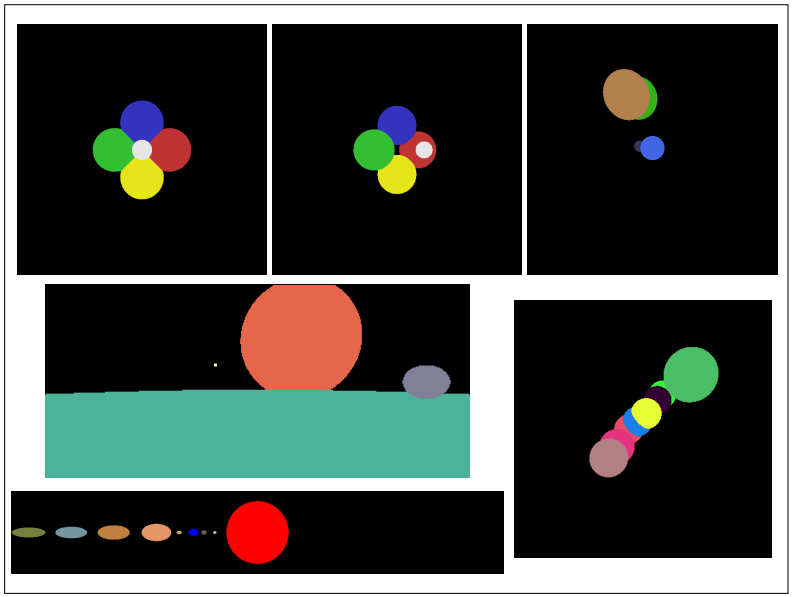
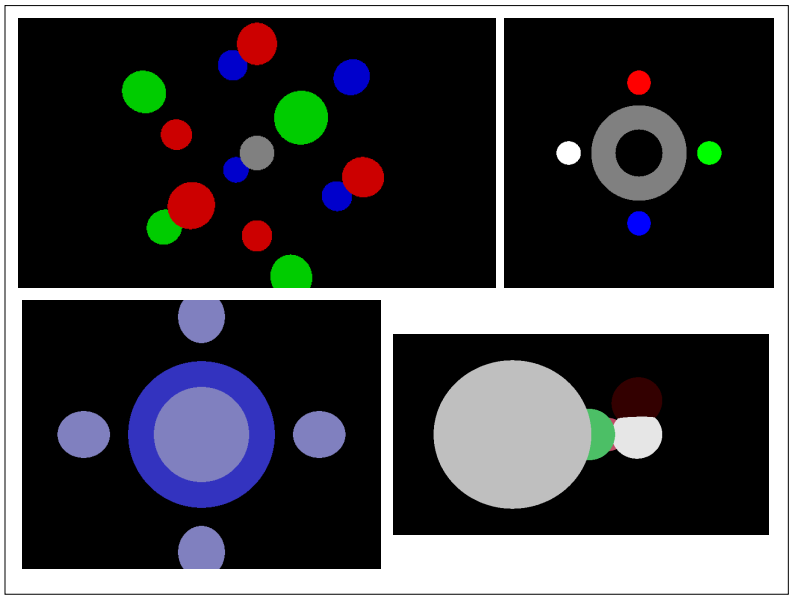
CSC 433/533
Computer Graphics

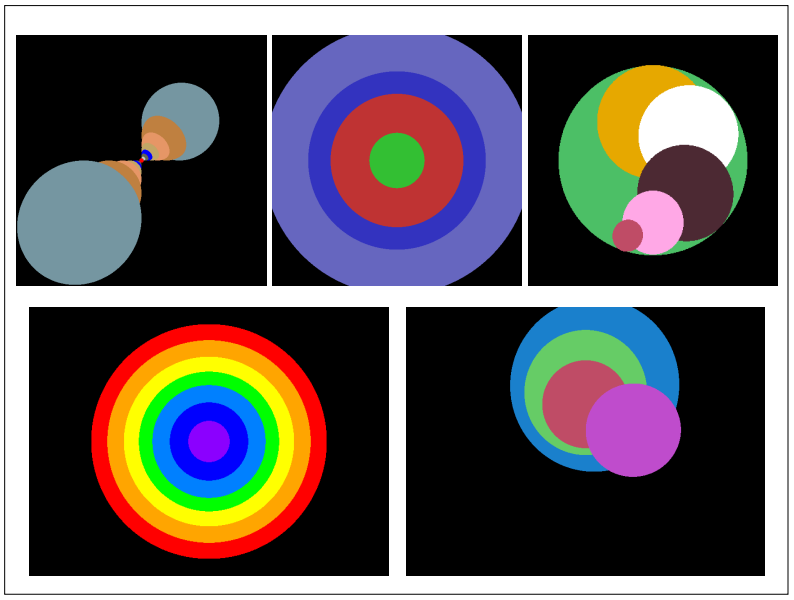
Alon Efrat
Credit: Joshua Levine

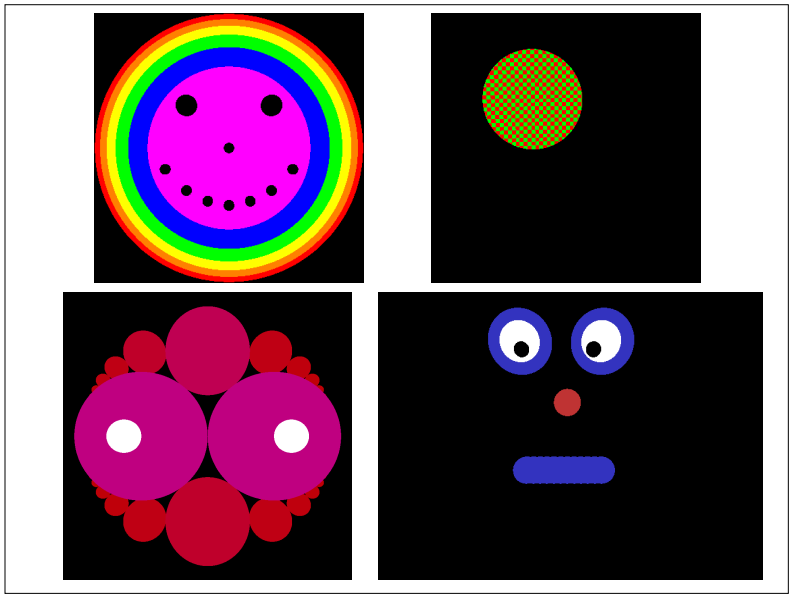
Lecture 15
Acceleration Structures

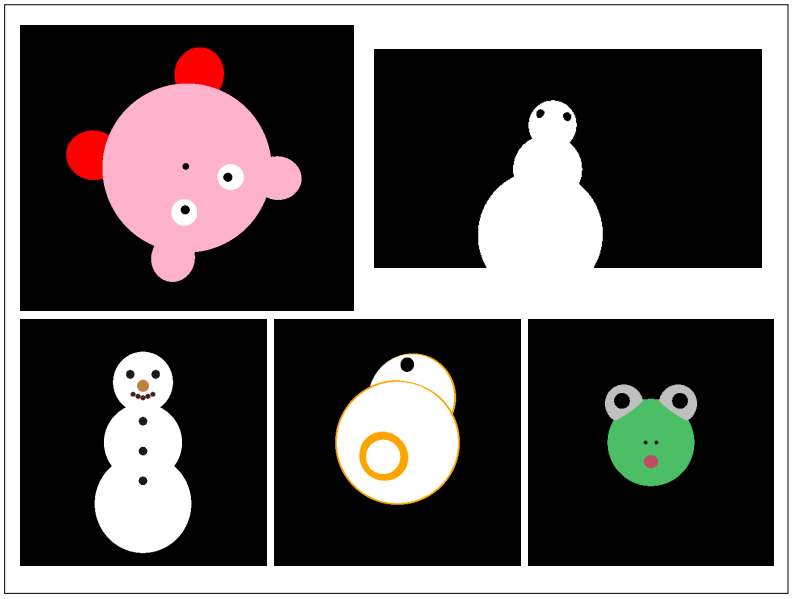
Assignment 03
Scenes







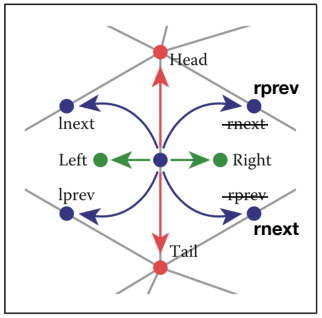




Mesh Data Structures and Queries

Winged-Edge Structure

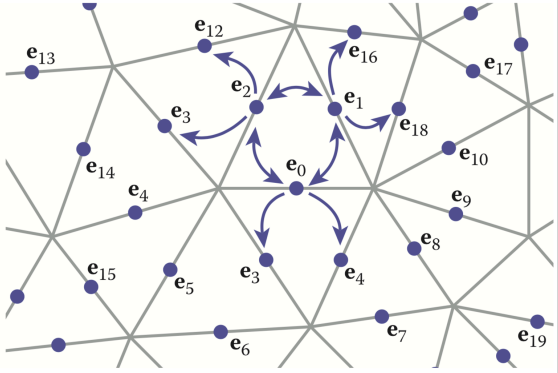
- Widely used mesh structure that focuses on edges instead of triangles
- Edges store pointers to:
  - Head/Tail vertices
  - Left/Right triangles
  - Left/Right “next” edges
  - Left/Right “previous” edges
- Each vertex/triangle stores one pointer to some edge



Winged-Edge Structure

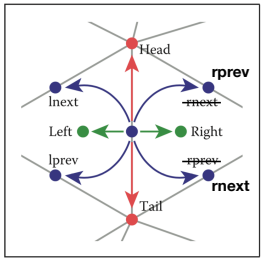
Winged-edge table

	ln	rp	lp	rn
[0]	1	4	2	3
[1]	18	0	16	2
[2]	12	1	3	0
	⋮			



Winged-Edge Structure

```
Edge {
  lprev, lnext, rprev, rnext; //Edge
  head, tail;                 //Vertex
  left, right;                //Face
}

Face {
  ...
  e;  //Edge
}

Vertex {
  ...
  e;  //Edge
}
```



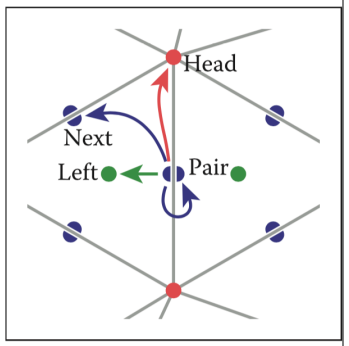

```
EdgesOfVertex(v) {
  e = v.e;
  do {
    if (e.tail == v) {
      e = e.lprev;
    } else {
      e = e.rprev;
    }
  } while (e != v.e);
}
```

Winged-Edge Storage Requirements

- Vertex data: 3 floats for position, 1 int for edge reference
  - 4*4 = $16n_v$ bytes
- Face data: 1 int for edge reference
  - 4*1 = $4n_t = 8n_v$ bytes.
- Edge data, 8 ints for references
  - $n_e \approx 3n_v$
  - 8*4*3 = $96n_v$ bytes.
- In total, $120n_v$ bytes.

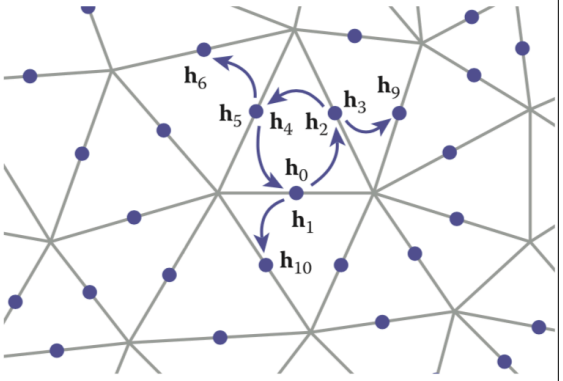
Half-Edge Structure

- Simplifies winged-edge, removes awkwardness of checking which way edges are oriented
- Each **half-edge** store pointers to:
  - Head vertex
  - Left triangle
  - Left "next" edge
  - The opposite "pair" half-edge
- Each vertex/triangle stores one pointer to a half-edge



Half-Edge Structure

	Pair	Next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



Half-Edge Structure

```
HEdge {
  pair, next;  //HEdge
  v;           //Vertex
  f;           //Face
};
```

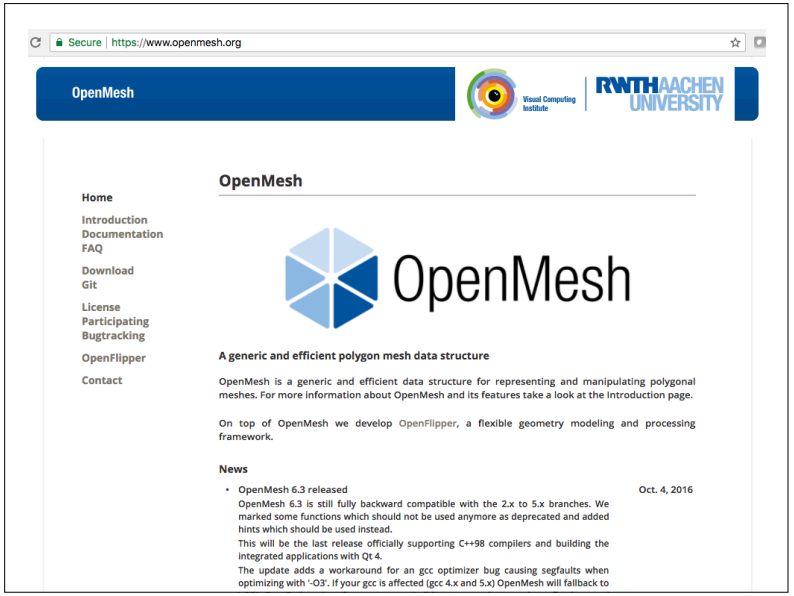
```
EdgesOfVertex(v) {
  h = v.h;
  do {
    h = h.next.pair;
  } while (h != v.h);
}
```

```
EdgesOfVertex(v) {
  e = v.e;
  do {
    if (e.tail == v) {
      e = e.lprev;
    } else {
      e = e.rprev;
    }
  } while (e != v.e);
}
```

Winged-Edge Implementation

Half-Edge Storage Requirements

- Vertex data: 3 floats for position, 1 int for edge reference
  - $4*4 = 16n_v$ bytes
- Face data: 1 int for edge reference
  - $4*1 = 4*n_t = 8n_v$ bytes.
- Edge data, 4 ints for references, but store a pair of half edges for each edge
  - $n_h \approx 6n_v$
  - $8*4*6 = 96n_v$ bytes.
- In total, $120n_v$ bytes.



Spatial Data Structures to Accelerate Queries

Challenge: Ray Tracing Millions of Triangles

- In a scene with meshed objects, most rays will only hit a very small number of triangles
- How to quickly find the triangles without having to compute triangle-ray intersection on each?

Recall: Generic Shapes

- Helpful to consider all types of objects from an abstract parent class of surfaces:

```
class Surface {
  ...
  intersect(eye, dir) {
    return {
      "t": t_min,
      "normal": undefined,
      "hit": false,
    };
  }
};
```

Ray to be intersected

Information about first intersection

Was there an intersection?

Generic Surfaces for Triangle Meshes

- How to deploy the same structure for a Triangle Mesh

```
class TriangleMesh extends Surface {
  ...
  tri_list = [];  //list of triangles
  ...
  intersect(eye, dir) {
    let hitrec = super.intersect(eye, dir);
    tri_list.forEach( function(tri) {
      let tri_hitrec = tri.intersect(eye, dir);
      if (tri_hitrec.hit) {
        hitrec = tri_hitrec;
      }
    });
    return hitrec;
  }
  ...
};
```

This loop is slow if there are lots of triangles

```
class Triangle extends Surface {
  ...
  intersect(eye, dir);
  ...
};
```

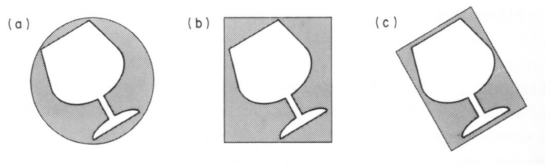
Solution: Use Spatial Data Structures To Organize Objects in Space

- Two approaches:
  - **Object Partitioning**: group objects into disjoint sets, but multiple sets might overlap
  - **Space Partitioning**: divide scene into regions of non-overlapping space, but each region might store multiple objects
- Both approaches relevant to ray tracing, but also generally helpful for locating objects in many computer graphics applications

Object Partitioning

Bounding Volumes

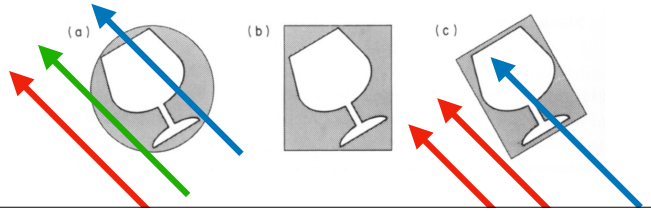
- Idea: Replace many repeated queries with a simpler query that can avoid the more expensive operation
- For ray tracing, if the ray doesn't hit the bounding volume, then it doesn't hit the object



[Glassner 89, Fig 4.5]

Bounding Volumes

- Cost analysis:
  - **True Negative** (neither hits bounding volume, nor surface) is cheaper
  - **True Positive** (hits both bounding volume and surface) is more expensive
  - **False Positive** (hits bounding volume, misses surface) is more expensive
- Worth doing if the hit test for the bounding volume is relatively cheap compared with the hit test for the surface



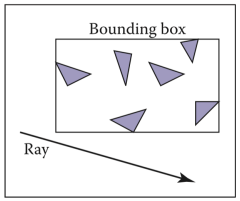
[Glassner 89, Fig 4.5]

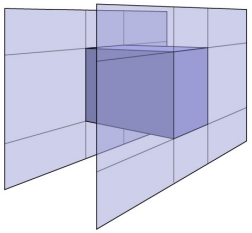
Axis-Aligned Bounding Boxes (AABBs)

- A variety of possible "simple" shapes could be used, but axis-aligned bounding boxes are a nice trade off:
  - Very simple intersection check
  - Not necessarily the tightest fit, but often a bit tighter than bounding spheres and often less work to compute than other primitives
  - Simply compute the min/max values in x, y, and z of the surface.

Intersecting AABBs

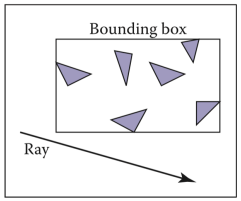
- Key idea: instead of doing an intersection with the faces of the cube, one could instead use range queries on the slabs.

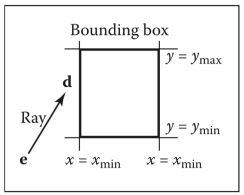




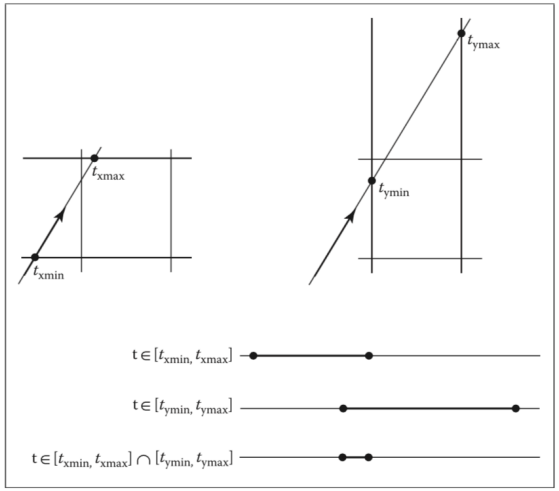
Intersecting AABBs

- Key idea: instead of doing an intersection with the faces of the cube, one could instead use range queries on the slabs.
- Let the AABB (in two-dimensions) be defined as all points $(x,y) \in [x_{min},x_{max}] \times [y_{min},y_{max}]$
- Given a ray $(x_e,y_e) + t(x_d,y_d)$, solve for t values where each plane is crossed
  - e.g. $t_{xmin} = (x_{min} - x_e) / x_d$

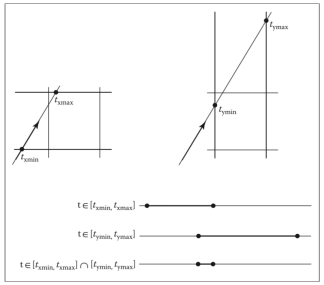




Intersecting AABBs



Intersecting AABBs



- High-level idea of the code:

```
t_xmin = (x_min - x_e) / x_d,
t_xmax = (x_max - x_e) / x_d;
t_ymin = (y_min - y_e) / y_d;
t_ymax = (y_max - y_e) / y_d;
if (t_xmin > t_ymax) or (t_ymin > t_xmax) {
  return false;
} else {
  return true;
}
```

- Code for this can be a bit nuanced (see the book for avoiding the divide by zero if `x_d, y_d == 0` and for handling the case where `x_d, y_d < 0`)
- 3D case is the same, except finding 3 intersecting intervals

Implementing Bounded Volumes

- Two modules:
  - A subclass of `Surface` called `AABB` that does the ray-box intersection
  - A subclass of `Surface` called `BoundedSurface` that keeps track of the `AABB` of the complex surface inside of it
- In `BoundedSurface.intersect()` first call the AABB's intersection check.

Generic Surfaces for AABBs

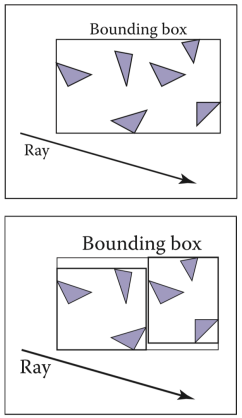
- How to use generics for AABBs

```
class AABB extends Surface {
  bbmin, bbmax;  //Vec3's
  ...
  intersect(eye, dir) {
    //check bounds on intersection relative to bbmin, bbmax
  }
  ...
};

class BoundedSurface extends Surface {
  bounding_box;  //an AABB
  ...
  intersect(eye, dir) {
    let hitrec = super.intersect(eye, dir);
    if (bounding_box.intersect(eye, dir).hit) {
      //do more expensive intersection with rest of the surface
      //update hitrec;
    }
    return hitrec;
  }
  ...
};
```

Hierarchical Bounding Boxes

Complex Objects Might Not Be Easily Modeled with a Single Box

- In particular, a single box might not fit well
- Challenges: How many boxes to use? Can we avoid checking all of them?
- Solution: Use a hierarchy of many boxes



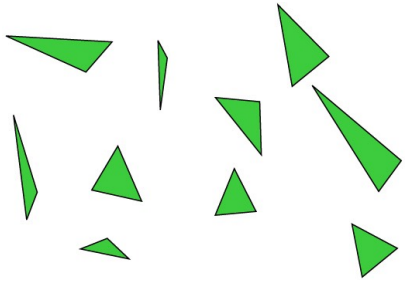
Using a Hierarchy of Boxes

- Typically, we start top-down
- First, build a bounding box for the entire scene
  - Next, split the elements of the scene and create separate bounding boxes for each group
  - Repeatedly split each sub-box until there are only a few elements in each box

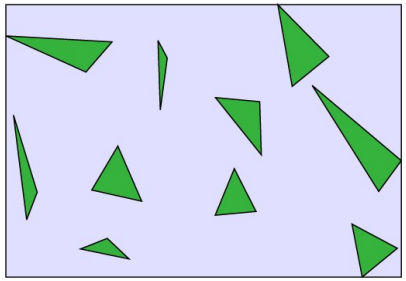
Bounding Volume Hierachies (BVHs)

- Splitting up elements of the scene into boxes can be challenging to do efficiently.
  - In practice, we partition along a different axis and rotate through the three axes to do the splits
- Key point: elements must be completely contained within boxes, but boxes may (usually) overlap

Constructing BVHs

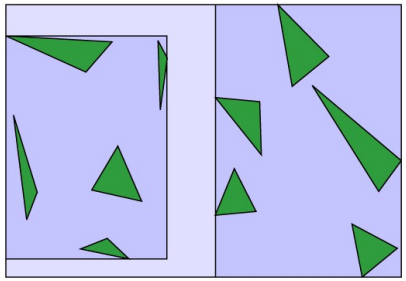


Constructing BVHs



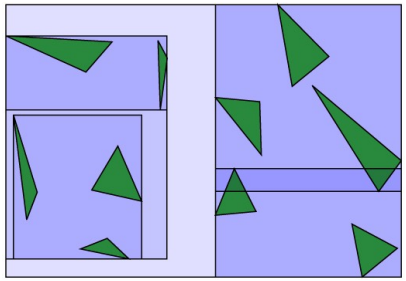
Tree Depth 0

Constructing BVHs



Tree Depth 1 (split x-axis)

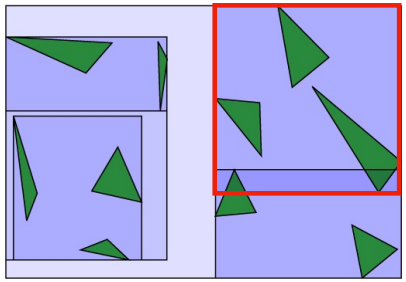
Constructing BVHs



Tree Depth 2
(split y-axis)

Constructing BVHs

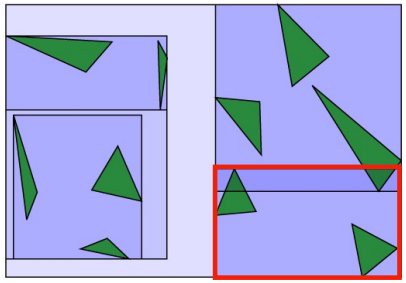
Overlapping
Boxes!



Tree Depth 2

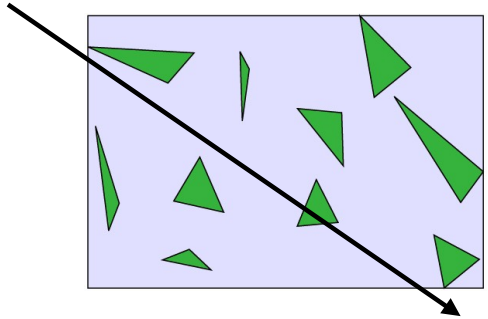
Constructing BVHs

Overlapping
Boxes!



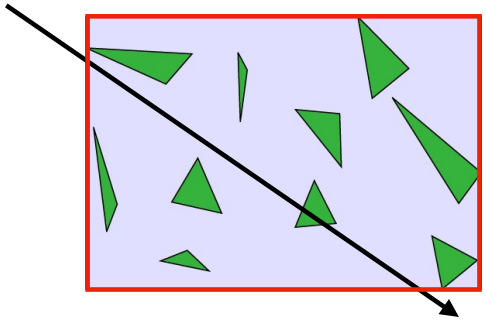
Tree Depth 2

Ray-BVH Intersection



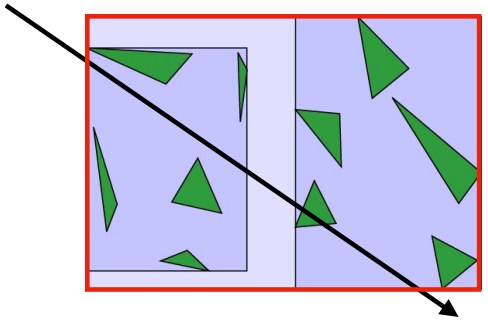
Tree Depth 0

Ray-BVH Intersection



Tree Depth 0

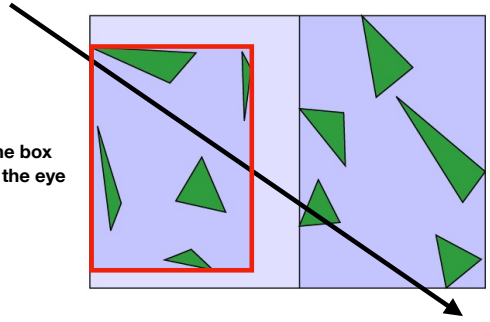
Ray-BVH Intersection



Tree Depth 1

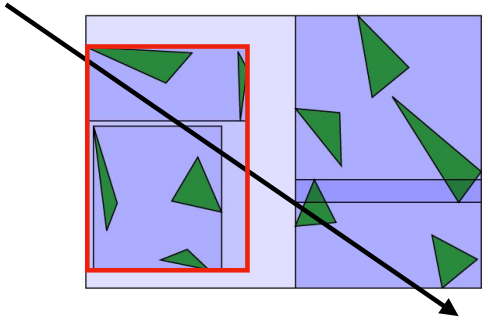
Ray-BVH Intersection

Select the box closest to the eye



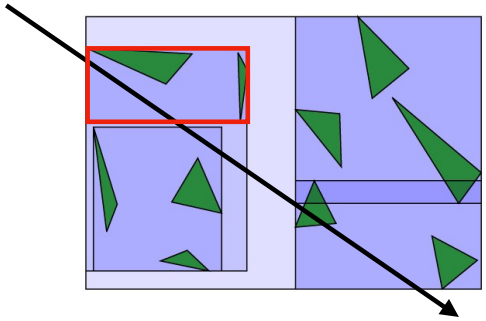
Tree Depth 1

Ray-BVH Intersection



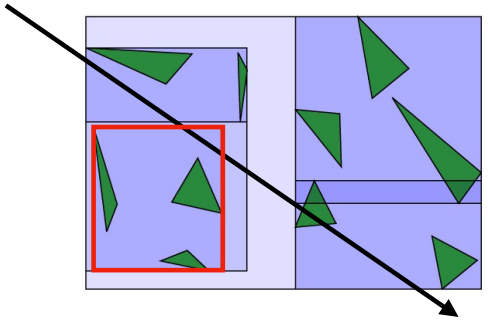
Tree Depth 2

Ray-BVH Intersection



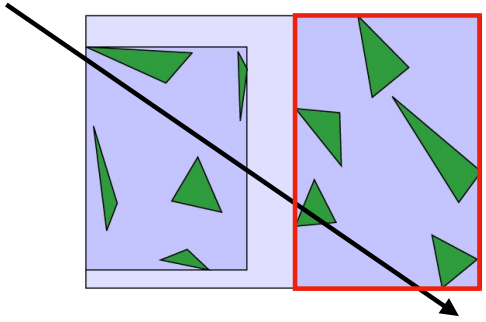
Tree Depth 2

Ray-BVH Intersection



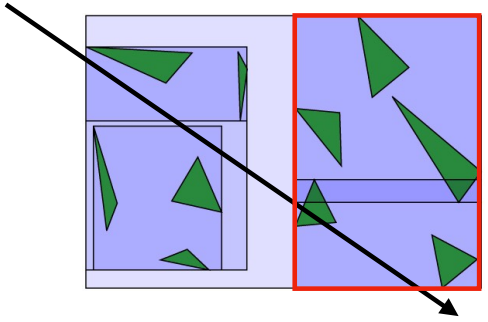
Tree Depth 2

Ray-BVH Intersection



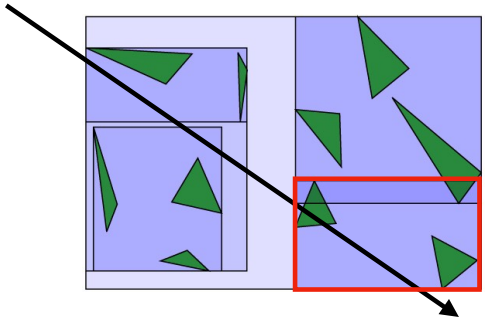
Tree Depth 1

Ray-BVH Intersection



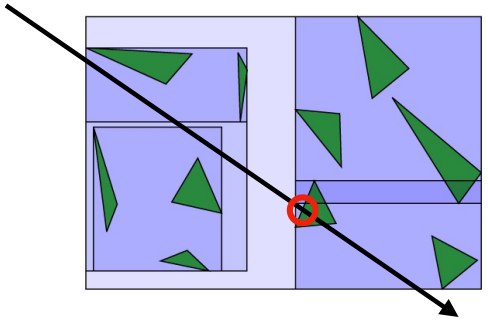
Tree Depth 2

Ray-BVH Intersection



Tree Depth 2

Ray-BVH Intersection



Tree Depth 2

Found hit!

Generic Surfaces for BVHs

- How to use generics for BVHs

```
class BVHnode extends Surface {
  bounding_box; //an AABB
  left, right;  //two BVHnode's
  ...
  intersect(eye, dir) {
    let hitrec = super.intersect(eye, dir);
    if (bounding_box.intersect(eye,dir).hit) {
      //if left, right children exist
        //recursive check for intersection and return result
      //if leaf
        //do more expensive intersection with surface elements
      //update hitrec
    }
    return hitrec;
  }
  ...
};
```

Generic Surfaces for BVHs

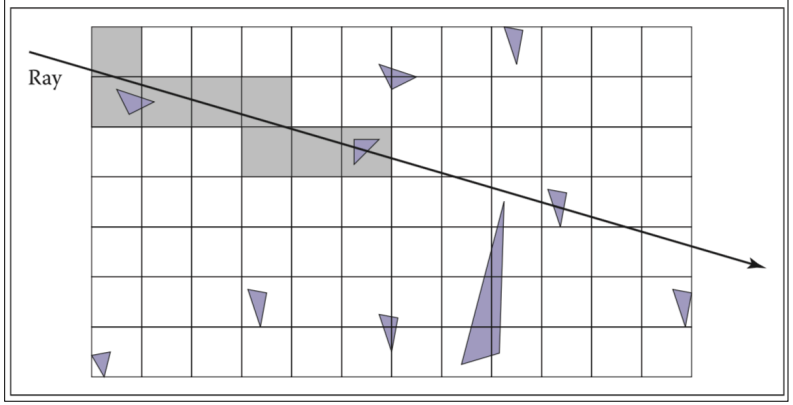
- How to use generics for BVHs

```
class BoundedSurface extends Surface {
  bvh_root; //a BVHnode
  ...
  intersect(eye, dir) {
    let hitrec = super.intersect(eye, dir);
    if (bvh_root.intersect(eye, dir).hit) {
      //result might also contain pointer to leaf node
      //in addition to surface elements
      //update hitrec
    }
    return hitrec;
  }
  ...
};
```

Spatial Partitioning

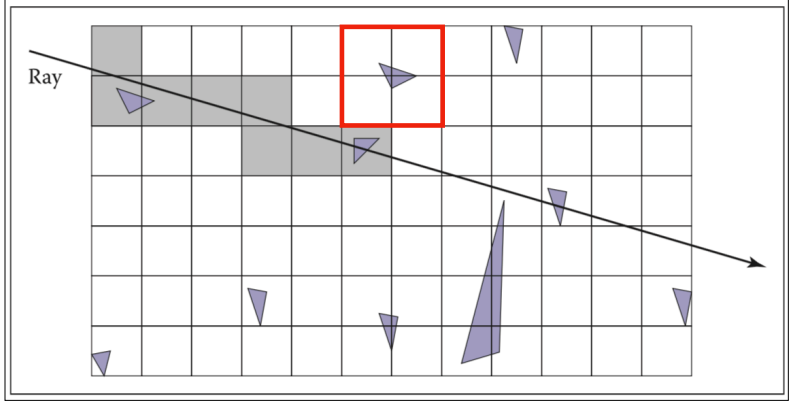
Subdivision w/ Regular Grids

- Instead of partitioning the objects, partition space
- Simplest approach: uniform grid of cells



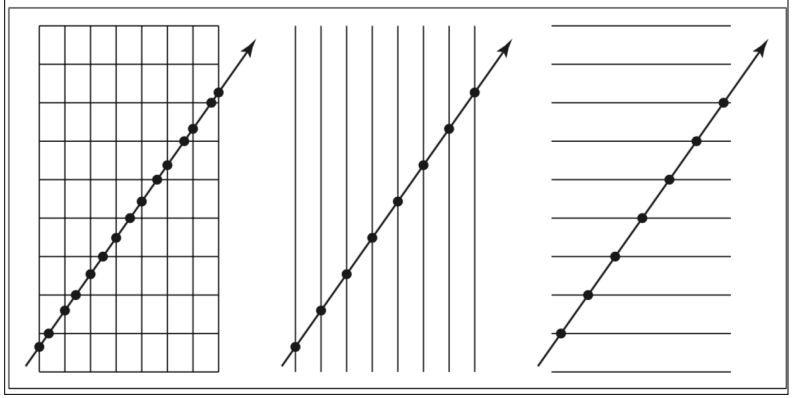
Subdivision w/ Regular Grids

- Multiple cells can span the same object



Regular Grid Traversal

- Cells that are accessed followed a regular pattern, based on direction of ray



Generic Surfaces for Grids

- How to use generics for Grids

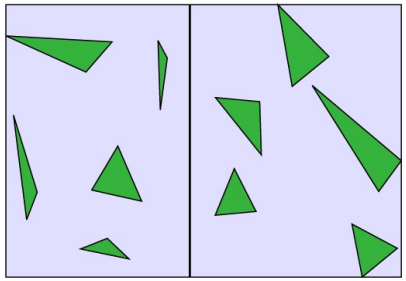
```
class GridCell : public Surface {
  boolean intersect(IntersectionRecord &result, Ray r) {
    //check if ray intersects any surfaces referenced by cell
  }
};

class BoundedSurface : public Surface {
  GridCell regular_grid[nz][ny][nx];
  ...
  boolean intersect(IntersectionRecord &result, Ray r) {
    //find first GridCell (i,j,k) hit based on hit of bounding box
    while( /* ray in bounds */ ) {
      if (!regular_grid[k][j][i].intersect(result, ray)) {
        //increment i, j, k, or some combination of them.
      }
    }
  }
  ...
};
```

Hierarchical Subdivision

- Similar idea to BVHs, but instead of partitioning objects hierarchically, we can partition space hierarchically.
- Most common case is to used axis-aligned partitioning
- Often, split in one dimension at each level, forming a binary space partitioning tree, or BSP tree.

k-d Trees

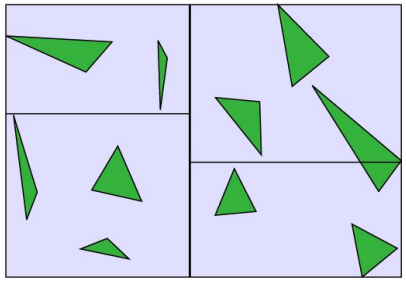
- One example of BSP trees



Tree Depth 1 (split in x)

k-d Trees

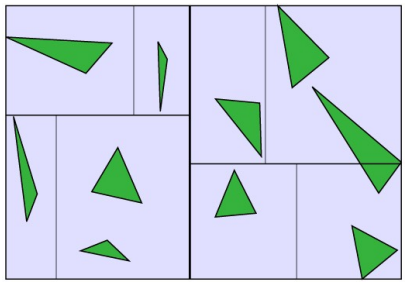
- One example of BSP trees



Tree Depth 2 (split in y)

k-d Trees

- One example of BSP trees



Tree Depth 3 (split in x)

Lec17 Required Reading

- FOCG, Ch. 5

Reminder: Assignment 04

Assigned: Wednesday, Oct. 9
Written Due: Monday, Oct. 23, 4:59:59 pm
Program Due: Wednesday, Oct. 28, 4:59:59 pm