Physics based animation

Lecture 11 - Collision detection
Part 4 - Bounding Volume Hierarchies

Grégory Leplâtre

g.leplatre@napier.ac.uk, room D32
School of Computing
Edinburgh Napier University

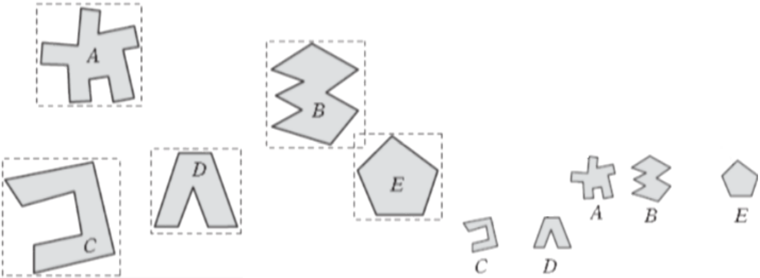
Semester 1 - 2016/2017

Plan

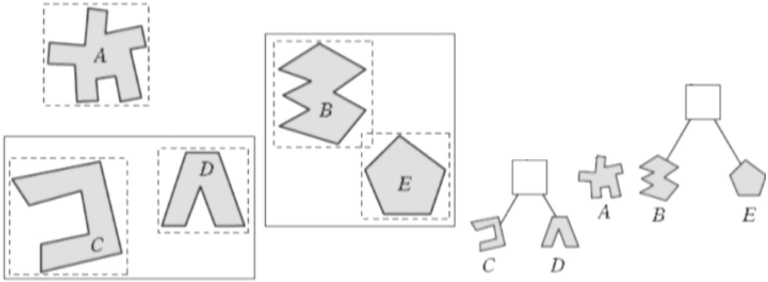
- Bounding volume hierarchy
- Overview of most common construction (top down) and traversal (DF) method

Outline

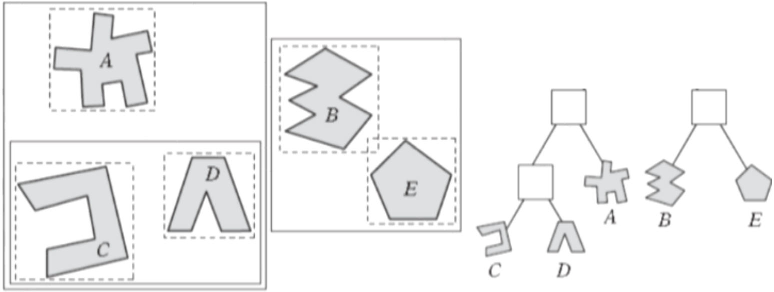
Introduction



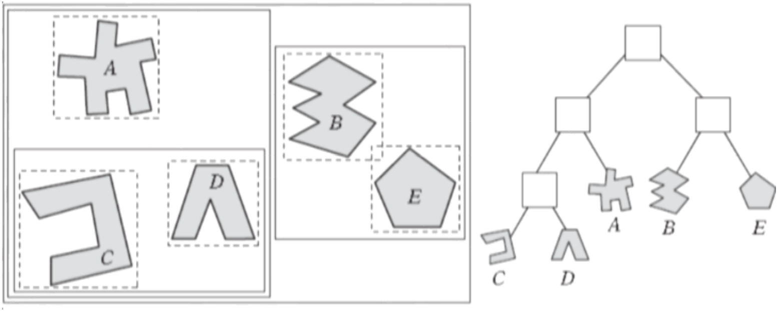
Introduction



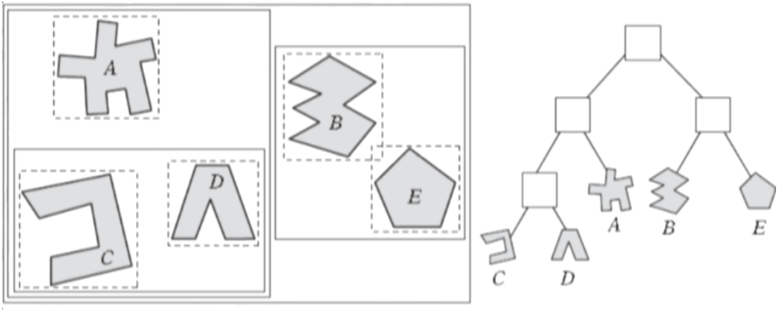
Introduction



Introduction



Introduction



main idea

If two nodes don’t intersect, their children can’t intersect

Desired characteristics

- The nodes given in any branch should be **near each other**

Desired characteristics

- The nodes given in any branch should be **near each other**
- Each node in the hierarchy should be of **minimal volume**

Desired characteristics

- The nodes given in any branch should be **near each other**
- Each node in the hierarchy should be of **minimal volume**
- Pruning a node **near the root** of the tree should remove more objects from further consideration than removal of a **deeper node**.

Desired characteristics

- The nodes given in any branch should be **near each other**
- Each node in the hierarchy should be of **minimal volume**
- Pruning a node **near the root** of the tree should remove more objects from further consideration than removal of a **deeper node**.
- The volume of overlap of sibling nodes should be minimal

Desired characteristics

- The nodes given in any branch should be **near each other**
- Each node in the hierarchy should be of **minimal volume**
- Pruning a node **near the root** of the tree should remove more objects from further consideration than removal of a **deeper node**.
- The volume of overlap of sibling nodes should be minimal
- The hierarchy should be balanced with respect to both its node structure and its content.

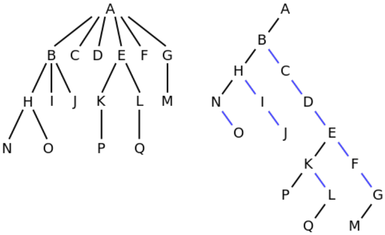
Desired characteristics

- The nodes given in any branch should be **near each other**
- Each node in the hierarchy should be of **minimal volume**
- Pruning a node **near the root** of the tree should remove more objects from further consideration than removal of a **deeper node**.
- The volume of overlap of sibling nodes should be minimal
- The hierarchy should be balanced with respect to both its node structure and its content.
- Worst case time for query should not exceed the average query time much

Desired characteristics

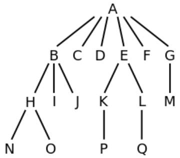
- The nodes given in any branch should be **near each other**
- Each node in the hierarchy should be of **minimal volume**
- Pruning a node **near the root** of the tree should remove more objects from further consideration than removal of a **deeper node**.
- The volume of overlap of sibling nodes should be minimal
- The hierarchy should be balanced with respect to both its node structure and its content.
- Worst case time for query should not exceed the average query time much

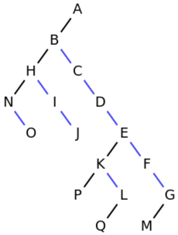
Tree degree



- What type of tree is best?

Tree degree



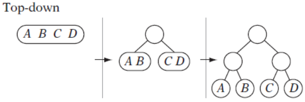


- What type of tree is best?
- No definite answer, but binary trees commonly used:
  - Easy to create and traverse
  - Top down creation only requires a single splitting plane for each node

Outline

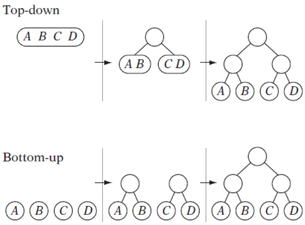
Building strategies

Building strategies



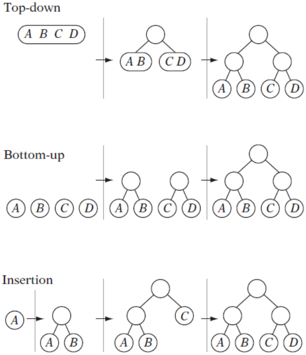
- **Top-down**: partitioning input set into two (or more) subsets, introduce bounds, and recursing over bounded subsets (typically does not produce best trees, but is easy to implement and hence popular).

Building strategies



- **Top-down**: partitioning input set into two (or more) subsets, introduce bounds, and recursing over bounded subsets (typically does not produce best trees, but is easy to implement and hence popular).
- **Bottom-up**: start with leaves of the tree, group two (or more) to form a bounding node, progressively grouping bounded volumes until a single node is reached (typically produces best quality BVH, but more difficult to implement).

Building strategies



- **Top-down**: partitioning input set into two (or more) subsets, introduce bounds, and recursing over bounded subsets (typically does not produce best trees, but is easy to implement and hence popular).
- **Bottom-up**: start with leaves of the tree, group two (or more) to form a bounding node, progressively grouping bounded volumes until a single node is reached (typically produces best quality BVH, but more difficult to implement).
- **Insertion**: incrementally insert an object into the tree so as to minimize some insertion cost measurement

Top down

Top down

```
void buildTopDownBVH(BVHNode treeNode, ArrayList objs) {
```

Top down

```
void buildTopDownBVH(BVHNode treeNode, ArrayList objs) {
  // create new node and add to tree
  Node newNode = new Node();
  treeNode.add(newNode);
```

Top down

```
void buildTopDownBVH(BVHNode treeNode, ArrayList objs) {
  // create new node and add to tree
  Node newNode = new Node();
  treeNode.add(newNode);
  // create a BV around object list
  newNode.BoundingVolume = ComputeBoundingVolume(objs);
```

Top down

```
void buildTopDownBVH(BVHNode treeNode, ArrayList objs) {
  // create new node and add to tree
  Node newNode = new Node();
  treeNode.add(newNode);
  // create a BV around object list
  newNode.BoundingVolume = ComputeBoundingVolume(objs);
  // store object list in node
  if( objects.Count <= MAX_OBJECTS_PER_LEAF ) {
    newNode.Type = LEAF;
    newNode. Objects = objs;
  }
```

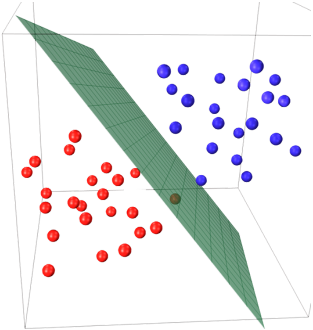
Top down

```
void buildTopDownBVH(BVHNode treeNode, ArrayList objs) {
  // create new node and add to tree
  Node newNode = new Node();
  treeNode.add(newNode);
  // create a BV around object list
  newNode.BoundingVolume = ComputeBoundingVolume(objs);
  // store object list in node
  if( objects.Count <= MAX_OBJECTS_PER_LEAF ) {
    newNode.Type = LEAF;
    newNode. Objects = objs;
  }
  else{
    // split object into two partitions
    newNode.Type = NODE;
    int splitIdx = RearrangeAndPartitionObjects(objs);
```

Top down

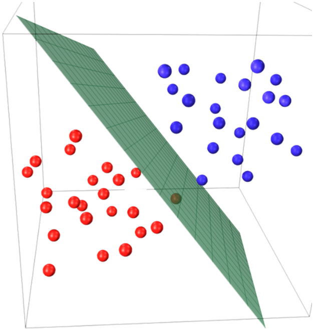
```
void buildTopDownBVH(BVHNode treeNode, ArrayList objs) {
  // create new node and add to tree
  Node newNode = new Node();
  treeNode.add(newNode);
  // create a BV around object list
  newNode.BoundingVolume = ComputeBoundingVolume(objs);
  // store object list in node
  if( objects.Count <= MAX_OBJECTS_PER_LEAF ) {
    newNode.Type = LEAF;
    newNode. Objects = objs;
  }
  else{
    // split object into two partitions
    newNode.Type = NODE;
    int splitIdx = RearrangeAndPartitionObjects(objs);
    // recursively build left and right branch
    BuildTopDownBVH(newNode.LeftTree, objs.Subset(0,
    splitIdx));
    BuildTopDownBVH(newNode.RightTree, objs.Subset(
    splitIdx,objs.Count);
  }
}
```

partitioning strategies



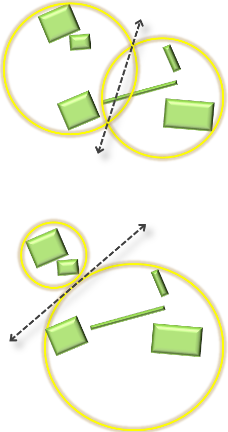
- Typical approach: **splitting hyperplane** (median cut algorithm)

partitioning strategies



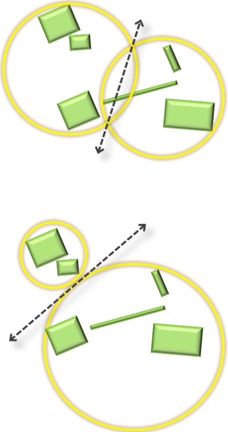
- Typical approach: **splitting hyperplane** (median cut algorithm)
  - Division in **two equal size sets**
  - In respect to a **projection axis**

Other partitioning strategies



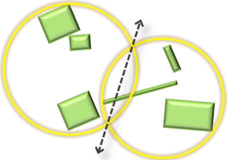
- Strategies derived from earlier desired features:

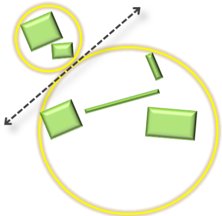
Other partitioning strategies



- Strategies derived from earlier desired features:
  - **Minimize** the sum of the volumes (or surface areas) of the **child volumes**.

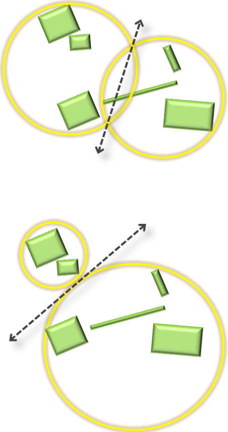
Other partitioning strategies





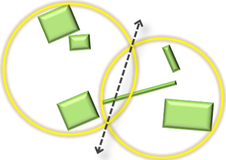
- Strategies derived from earlier desired features:
  - **Minimize** the sum of the volumes (or surface areas) of the **child volumes**.
  - **Minimize** the volume (surface area) of the **intersection of the child volumes**.

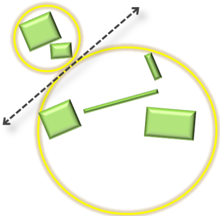
Other partitioning strategies



- Strategies derived from earlier desired features:
  - **Minimize** the sum of the volumes (or surface areas) of the **child volumes**.
  - **Minimize** the volume (surface area) of the **intersection of the child volumes**.
  - **Maximize the separation of child volumes**.

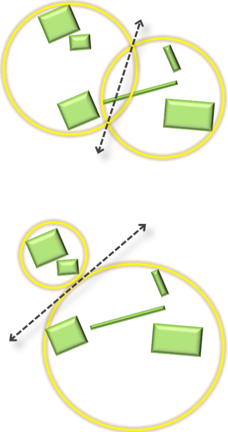
Other partitioning strategies





- Strategies derived from earlier desired features:
  - **Minimize** the sum of the volumes (or surface areas) of the **child volumes**.
  - **Minimize** the volume (surface area) of the **intersection of the child volumes**.
  - **Maximize the separation of child volumes**.
  - **Divide primitives equally** between the child volumes (known as a median-cut algorithm, and tends to produced balanced trees)

Other partitioning strategies



- Strategies derived from earlier desired features:
  - **Minimize** the sum of the volumes (or surface areas) of the **child volumes**.
  - **Minimize** the volume (surface area) of the **intersection of the child volumes**.
  - **Maximize the separation of child volumes**.
  - **Divide primitives equally** between the child volumes (known as a median-cut algorithm, and tends to produced balanced trees)
  - Combinations of above strategies.

Partitioning strategies - When to stop?

- The recursive partitioning stops (thereby forming a leaf node) when some termination condition is reached. This can include:
  - The node contains fewer than some k primitives.
  - The volume of the bounding volume falls below a cut-off limit.
  - The depth of the node has reached a predefined cut-off depth.

Partitioning strategies - When to stop?

- The recursive partitioning stops (thereby forming a leaf node) when some termination condition is reached. This can include:
  - The node contains fewer than some k primitives.
  - The volume of the bounding volume falls below a cut-off limit.
  - The depth of the node has reached a predefined cut-off depth.
- Partitioning might also fail because:
  - All primitives fall on one side of the split plane.
  - One or both child volumes end up with as many (or nearly as many) primitives as the parent volume.
  - Both child volumes are (almost) as large as the parent volume.

Partitioning strategies - When to stop?

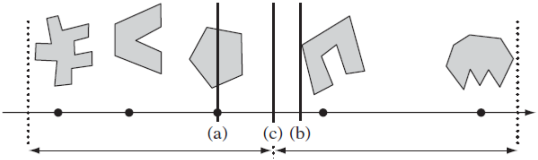
- The recursive partitioning stops (thereby forming a leaf node) when some termination condition is reached. This can include:
  - The node contains fewer than some k primitives.
  - The volume of the bounding volume falls below a cut-off limit.
  - The depth of the node has reached a predefined cut-off depth.
- Partitioning might also fail because:
  - All primitives fall on one side of the split plane.
  - One or both child volumes end up with as many (or nearly as many) primitives as the parent volume.
  - Both child volumes are (almost) as large as the parent volume.
- In the latter cases, it is reasonable to try other partitioning criteria before terminating the recursion.

Partitioning strategies - Choice of axis

- Local x, y, and z coordinate axes (easy to use, also form orthogonal set, i.e. good coverage).
- Axes from some aligned bounding volume (e.g. from a reference k-DOP).
- Axes of the parent bounding volume
- Axis along which variance is greatest (using the dimension with largest spread serves to minimise the size of the child volumes).
- Axis through the two most distant points (similar outcome to above)

Partitioning strategies - Choice of split point

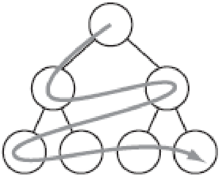
Partitioning strategies - Choice of split point



- **Object median**: splitting at the middle object (thereby evenly distributing the primitives and providing a balanced tree).
- **Object mean**: splitting at the mean of object coordinates (e.g. along the axis with greatest variance). Tends to give better results (smaller trees, queried quicker)
- **Spatial median**: Split space in two halves. Fast, but can lead to unblanced trees

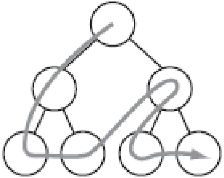
Outline

Tree traversal



- *Uninformed* methods:
  - **Breadth first**

Tree traversal

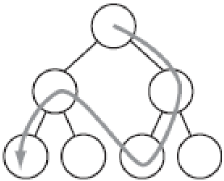


- *Uninformed* methods:
  - **Breadth first**
  - **Depth first**

Tree traversal

- *Uninformed* methods:
  - **Breadth first**
  - **Depth first**
- *Informed* search:

Tree traversal



- *Uninformed* methods:
  - **Breadth first**
  - **Depth first**
- *Informed* search:
  - **Best first**: Pick node that best meets a set of criteria.

Tree traversal

- *Uninformed* methods:
  - **Breadth first**
  - **Depth first**
- *Informed* search:
  - **Best first**: Pick node that best meets a set of criteria.
- For collision detection systems, DFS (enhanced by a simple heuristic - i.e. basically a best-first approach) is typically favoured over BFS.

Descent rules

Descent rules

- **Descend A before B.** Fully descend into the leaves of A before starting to descend into B. This will be very expensive if B's root bound fully contains A and/or if A contains lots of nodes.

Descent rules

- **Descend A before B.** Fully descend into the leaves of A before starting to descend into B. This will be very expensive if B's root bound fully contains A and/or if A contains lots of nodes.
- **Descend the larger volume.** Dynamically determine which BVH node is currently larger and descend into (avoiding the problems of Approach 1).

Descent rules

- **Descend A before B.** Fully descend into the leaves of A before starting to descend into B. This will be very expensive if B's root bound fully contains A and/or if A contains lots of nodes.
- **Descend the larger volume.** Dynamically determine which BVH node is currently larger and descend into (avoiding the problems of Approach 1).
- **Descend A and B simultaneously.** Similar to Approach 2 but without any cost evaluation overhead (but it may not prune space as efficiently).

Descent rules

- **Descend A before B.** Fully descend into the leaves of A before starting to descend into B. This will be very expensive if B's root bound fully contains A and/or if A contains lots of nodes.
- **Descend the larger volume.** Dynamically determine which BVH node is currently larger and descend into (avoiding the problems of Approach 1).
- **Descend A and B simultaneously.** Similar to Approach 2 but without any cost evaluation overhead (but it may not prune space as efficiently).
- **Descend A and B alternatively.**

Descent rules

- **Descend A before B.** Fully descend into the leaves of A before starting to descend into B. This will be very expensive if B's root bound fully contains A and/or if A contains lots of nodes.
- **Descend the larger volume.** Dynamically determine which BVH node is currently larger and descend into (avoiding the problems of Approach 1).
- **Descend A and B simultaneously.** Similar to Approach 2 but without any cost evaluation overhead (but it may not prune space as efficiently).
- **Descend A and B alternatively.**
- **Descend based on overlap.** Similar to Approach 2, priortise descent on degree of overlap between BVH regions.

Example: Informed DF Traversal

```
void BVHInformedDFS(CollisionResult r, BVHNode a, BVHNode
    b){
  // if the BV don't overlap, simply return
  if (!BVOverlap(a, b)) return;
```

Example: Informed DF Traversal

```
void BVHInformedDFS(CollisionResult r, BVHNode a, BVHNode
    b){
  // if the BV don't overlap, simply return
  if (!BVOverlap(a, b)) return;
  // if two leaf nodes found, then perform collision
    detection on primitives
  if (IsLeaf(a) && IsLeaf(b)) {
    CollidePrimitives(r, a, b);
```

Example: Informed DF Traversal

```
void BVHInformedDFS(CollisionResult r, BVHNode a, BVHNode
     b){
  // if the BV don't overlap, simply return
  if (!BVOverlap(a, b)) return;
  // if two leaf nodes found, then perform collision
    detection on primitives
  if (IsLeaf(a) && IsLeaf(b)) {
    CollidePrimitives(r, a, b);
  }
  // else descend A or B according to heuristic
  else {
    if (DescendA(a, b)) {
       BVHInformedDFS (r, a->left, b);
       BVHInformedDFS (r, a->right, b);
    } else {
       BVHInformedDFS (r, a, b->left);
       BVHInformedDFS (r, ,a, b->right);
    }
  }
}
```

Example: Informed DF Traversal

```
bool DescendA(BVHNode a, BVHNode b){
  // descend into the larger volume
   return IsLeaf(b) or (!IsLeaf(a) && (SizeOfBV(a) >=
    SizeOfBV(b)));
}
```

Outline

Summary

- Hierarchical representation allows the number of pairwise comparisons to be reduced

Summary

- Hierarchical representation allows the number of pairwise comparisons to be reduced
- Desired characteristics presented. Incidence on the structure of the hierarchy

Summary

- Hierarchical representation allows the number of pairwise comparisons to be reduced
- Desired characteristics presented. Incidence on the structure of the hierarchy
- Construction methods; top-down most common

Summary

- Hierarchical representation allows the number of pairwise comparisons to be reduced
- Desired characteristics presented. Incidence on the structure of the hierarchy
- Construction methods; top-down most common
- Traversal methods: DF with heurisitc

Summary

- Hierarchical representation allows the number of pairwise comparisons to be reduced
- Desired characteristics presented. Incidence on the structure of the hierarchy
- Construction methods; top-down most common
- Traversal methods: DF with heurisitc
- Technical considerations regarding tree encoding and traversal not addressed

References

- Ericson, C. (2004). Real-time collision detection. CRC Press.