# ACM/CS 114
# Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

# A more careful look at contact detection

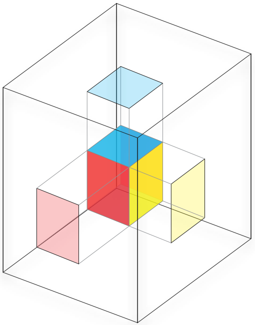
- consider a collection of tetrahedral meshes that model bodies in relative motion with triangular meshes as boundaries
- during the simulation, the bodies may come in contact
  - with each other or themselves
  - *contact events* consist of intersections among nodes, edges or faces
  - unless the mechanics is informed of the contact events, the objects will inter-penetrate
  - contact *detection* involves isolating the pairs of topological entities from each boundary that have intersected, whereas contact *resolution* refers to the calculation of appropriate restoring forces on the bodies
- the typical simulation update step proceeds along the following lines
  1. define the contact surfaces at time $t$
  2. predict the location of the nodes at a later time $t + \Delta t$ by integrating the equations of motion
  3. search for potential contact events among nodes, edges and faces to identify the entities that come in contact
  4. correct the future location of the nodes by applying forces that tend to remove the overlap

# Contact search

- the contact event identification in Step 3 above has the potential to dominate the calculation
  - given a candidate pair, the intersection logic involves very expensive geometrical calculations
  - naïve algorithms are $\mathcal{O}(n^2)$ in the number of topological entities on the boundary, prohibitively expensive even for moderate size calculations
- hence, a more sensible strategy is to break up the contact search into two separate steps
  - use a specialized data structure that encodes the location of mesh nodes and build a relatively fast algorithm to narrow down the candidates to a small number
  - perform the detailed calculations on the reduced set
- typically, the fast searches are implemented using *orthogonal range queries* that identify points whose coördinates fall within a given box
  - build a bounding box that contains the initial and final position of a given surface element, perhaps in some reduced form
  - form the *capture box* by enlarging the bounding box to account for the motion of the nodes
  - query the data structure for nodes that fall within the capture box

# Orthogonal range queries

- an orthogonal range query is a generalization of the interval test to higher dimensions
  - given a number $p$ and an interval $[a, b)$, return `true` if the number falls within the interval, otherwise return `false`
  - extend by performing a test for each coördinate: does the point $p$ fall within a given box?
- there is a variety of data structures that are *a priori* well suited to this problem
  - however, the problem context establishes some crucial constraints



- we will classify algorithms according to the following metrics
  - $b(N)$: the time it takes to initially populate the data structure with $N$ points
  - $r(N)$: the complexity of rebuilding or update the data structure
  - $s(N)$: the amount of storage required
  - $q(N, n)$ and $\bar{q}(N, n)$: the (average) time required to perform a query if there are $n$ points in the given range
- also, we'll start out in one dimension and generalize

# Sequential scan

- the simplest approach is to look at each record and determine whether it falls in the range
  - this algorithm is trivial to implement and requires no extra storage
  - the performance is acceptable for sufficiently small $N$, or if most of the records fall in the query interval

$$\begin{array}{lclclcl} b_{\mathrm{SCAN}} & = & \mathcal{O}(N) & , & r_{\mathrm{SCAN}} & = & 0 \\ s_{\mathrm{SCAN}} & = & \mathcal{O}(N) & , & q_{\mathrm{SCAN}} & = & \mathcal{O}(N) \end{array}$$

**Algorithm 1**: RQ.SCAN(points, interval)

*candidates* $\leftarrow \emptyset$
**for** *point* **in** *points* **do**
  **if** *point* $\in$ *interval* **then**
    *candidates.insert(point)*
**return** *candidates*

# Binary search

- if the records are sorted, a binary search can locate any record with cost $\mathcal{O}(\log N)$, so in order to find all $p \in [a, b)$
  - find the first point that satisfies $p >= a$
  - and collect points in sequence while $p < b$
  - simple analysis yields

$$\begin{array}{rclcrcl} b_{\mathtt{BS}} & = & \mathcal{O}(N \log N) & , & r_{\mathtt{BS}} & = & \mathcal{O}(N) \\ s_{\mathtt{BS}} & = & \mathcal{O}(N) & , & q_{\mathtt{BS}} & = & \mathcal{O}(\log N + n) \end{array}$$

    since the records must be sorted initially, while rebuilding the data structure can be done in linear time since it is almost sorted

---

**Algorithm 2**: RQ.BS(points, interval=(a,b))

---

$candidates \leftarrow \emptyset$

$iterator \leftarrow$ BINARY-SEARCH-LOWER-BOUND$(points, a)$

**while** $\star iterator \leq b$ **do**

    **if** $point \in interval$ **then**

        $candidates.insert(point)$

**return** $candidates$

---

# Tricks with trees

- alternatively, we can store the points at the leaves of a binary tree data structure
  - each internal tree node acts has a *discriminator* that splits the data set into two subsets
  - numbers less than the discriminator go to the left branch, the rest to the right
  - once the population drops below some threshold, create a leaf node to hold the points
- two sensible choices for the discriminator are
  - the midpoint of the interval: yields a recursive subdivision of the interval
    - also known as interval trees or *orthotrees*
    - quadtrees in two dimensions, octrees in three
  - the median of the data set: partitions the data in subsets of equal size
    - *kd* trees

# Creating a binary tree

**Algorithm 3**: TREE.MAKE(points)

```
if length[points] < tree.leafSize then
    leaf ← tree.newLeaf()
    leaf.insert(points)
    return leaf
else
    branch ← tree.newBranch()
    select branch discriminator
    left ← {x ∈ points : x < discriminator}
    branch.left ← tree.make(left)
    right ← {x ∈ points : x >= discriminator}
    branch.right ← tree.make(right)
    return branch
```

# Querying a binary tree

**Algorithm 4**: RQ.TREE(tree, interval=(a,b))

```
if tree is leaf then
    return RQ.SCAN(tree.points, interval)
else
    candidates ← ∅
    if tree.discriminator ≥ a then
        candidates ← candidates + RQ.TREE(tree.left, interval)
    if tree.discriminator < b then
        candidates ← candidates + RQ.TREE(tree.right, interval)
    return candidates
```

# Performance of binary trees

- for midpoint splitting, the depth $D$ of the tree depends on the point distribution

$$\begin{aligned} b_{\texttt{ORTHO}} &= \mathcal{O}((D+1)N) & r_{\texttt{ORTHO}} &= \mathcal{O}((D+1)N) & & \\ s_{\texttt{ORTHO}} &= \mathcal{O}((D+1)N) & q_{\texttt{ORTHO}} &= \mathcal{O}(N) & \bar{q}_{\texttt{ORTHO}} &= \mathcal{O}(D+n) \end{aligned}$$

- for median splitting the depth of the tree depends only on the number of records

$$\begin{aligned} b_{\texttt{KD}} &= \mathcal{O}(N \log N) & r_{\texttt{KD}} &= \mathcal{O}(N) \\ s_{\texttt{KD}} &= \mathcal{O}(N) & q_{\texttt{KD}} &= \mathcal{O}(n + \log N) \end{aligned}$$

# Binning

- another strategy is to partition the interval $[a, b)$ into $M$ cells of width

$$\delta := x_{m+1} - x_m = \frac{b - a}{M}$$

  - the $m^{\text{th}}$ cell $C_m$ holds points in the interval $[x_m, x_{m+1})$
  - the point container then becomes an array of $M$ point containers
  - and the array index for a point $p$ is obtained through

$$i = \lfloor \frac{p - a}{\delta} \rfloor$$

- the process of putting the points in the container is known as a *cell sort*
- cell arrays can be *dense* or *sparse*, trading computational complexity for storage
- they are optimal when properly tuned

$$\begin{array}{lll} b_{\texttt{CELL}} = \mathcal{O}(N+M) & r_{\texttt{CELL}} = \mathcal{O}(N+M) & \\ s_{\texttt{CELL}} = \mathcal{O}(N+M) & q_{\texttt{CELL}} = \mathcal{O}(J+n) & \bar{q}_{\texttt{CELL}} = \mathcal{O}(n) \end{array}$$

where $J$ is the number of cells that overlap the query interval

# Querying a cell array

**Algorithm 5**: RQ.CELL(cells, interval=(a,b))

```
candidates ← ∅
i ← index(cells, a)
j ← index(cells, b)
for point in cells[i] do
    if point ≥ a then
        candidates.insert(point)
for k in [i + 1..j − 1] do
    if point in cells[k] then
        candidates.insert(point)
for point in cells[j] do
    if point < b then
        candidates.insert(point)
return candidates
```

# Generalizing to higher dimensions

- 