Kinetic BV Hierarchies and Collision Detection

Gabriel Zachmann
Clausthal University, Germany
zach@tu-clausthal.de


Bounding Volume Hierarchies

- BVHs are standard DS for collision detection ...
- ... but not just for collision detection!
  - Can be applied to ray-tracing, occlusion culling, etc.

- Pre-processed hierarchy becomes invalid when object deforms
  → BVH must be rebuilt or updated after deformations
Brute Force Update and Variants

- Problems of Brute-Force Updates:
  - Many update operations
  - No use of temporal coherence

- Other approaches:
  - Hybrid updates [van den Bergen, 1998]
  - Inflation and lazy updates of the BVs [Mezger et al. 2003]
  - Restriction of deformation schemes [James and Pai, 2004]
  - Intrinsic collision test on the GPU [Wong and Baciu 2005]
  - Chromatic decompositions [Govindaraju et al. 2005]
  - BVH reconstruction [Saarbrücken, 2006]

Our Approach

- Observation:
  - Motion in the physical world is normally continuous
  - Changes in the combinatorial structure of the BVHs occur only at discrete time points

⇒ We store only the combinatorial structure of the BVH and use an event-based approach for updating (kinetization)
Intro                      Kinetic BVHs                   Kinetic Continuous Coll Det
Kinetic Ray Tracing                    Conclusion

Advantages

- Fewer update operations
- Valid BVHs at every point in time
- Independent of query sampling frequency
- Can handle all kinds of objects
  - polygon soups (point clouds, and NURBS models)
- Can handle insertions/deletions during run-time
- Can handle all kinds of animated deformations
  - Only a flightplan is required for every vertex
  - These flightplans may change during simulation
Recap: Kinetic Data Structures

- **KDS** are a framework for designing and analyzing algorithms for objects in motion [Basch et al. 1997]
- KDS framework leads to event-based algorithms that "sample" the state of parts of a system only as often as necessary for a special task (e.g. a bounding box)

KDS terminology

- The task is called the **attribute**
- A KDS consists of set of **certificates**
- Certificate failures are called **events** → event queue
- If the attribute changes at the time of an event, the event is called **external**, otherwise **internal**
Quality of a KDS

- A KDS is **compact**, if it requires only little space
  - Space = number of certificates
  - Little = should not exceed static data structure by "too much"
- A KDS is **responsive**, if we can update it quickly in case of a certificate failure
  - Depends on number of certificates
  - Quickly = e.g. $O(\log(\text{number of certificates}))$
- A KDS is **local**, if one object is not involved in too many events
  - Depends on number of certificates
  - Not too many = e.g. $O(\log(\text{number of certificates}))$
- A KDS is **efficient**, if the ratio of internal events to external events is reasonable
  - Desirable is: $O(1)$ or less

Kinetic AABB Tree

- Kinetization of the AABB tree
- Pre-processing: Build the tree by any algorithm suitable for static AABB trees
  - For a theoretical analysis, it is only required that the height of the BVH is logarithmic
- Store with every node the indices of those points that determine the BV (the "realizing points")
- Initialize the event queue
The Events

- Leaf Event:
- Inner node event:
  - Is determined by only 2 points of its 2 children

- Flightplan Update Event

The Simulation Loop

```python
while simulation runs:
    determine time \( t \) of next frame
    \( e \leftarrow \text{min event in event queue} \)
    \( \text{while } e.t.timestamp < t \)
    process event \( e \)
    \( e \leftarrow \text{min event in event queue} \)
    check for collisions (or cast ray, or ...)
    render scene
```
Event Handling at Run-Time

- **Leaf Events:**
  - At the bottom of the AABB:
  - Propagation through the tree:

Analysis

- **Theorem 1:**
  Given a 2-manifold mesh with $n$ vertices.
  The kinetic AABB tree is **compact** (num. certificates $= O(n)$),
  **local** ($O(\log n)$), **responsive** (one event $= O(\log n)$) and **efficient**.
  Furthermore, the kinetic AABB tree is a valid BVH at every point in time.

- **Theorem 2:**
  Given $n$ vertices, we assume that each pair of flightplans intersect at most $s$ times.
  Then, the total number of events is in nearly $O(n \log n)$.

- **Remark:** this bound is independent of the query frequency.
The Kinetic Boxtree

- Problem: the kinetic AABB tree needs up to 6 events for every BV
- Idea: a kinetic BoxTree
  - A.k.a.: bounding interval hierarchy (BIH), SKD tree
  - Property: combination of k-d tree and AABB
  - Advantage: uses less memory than the kinetic AABB tree
  - Only 1 "splitting plane" per BV → only 1 event per BV

Event Computation

- 1D example:
In 3D:

Analysis:
The kinetic BoxTree is compact, local and efficient. But not responsive.

Test Scenes
Results

Shirt Scene (~ 100,000 triangles)

Total Time (incl. Collision Detection Time)

Total num triangles
Kinetic Separation Lists for Continuous Coll. Det.

- Continuous collision detection = determine earliest time of contact
- Conventional approach: swept volumes

**Problems:**
- Same as for static BVH updates
- Swept volumes are too large

**Idea:**
1. Maintain the list of nodes of the bounding-volume test tree (BVTT) where the simultaneous traversal stopped
2. Kinetize this list → "kinetic separation list"

**Toy example:**

**Advantages:**
- Continuous collision detection is reduced to the discrete problem of determining changes in the separation list
- Collisions are automatically reported in the correct order
- Inter-object and self-collision detection
Initialization of the Kinetic Separation List (KSL)

- Traverse the 2 BVHs of the 2 objects as usual
  - For self collision detection: Test object against itself
- Compute separation list and initial events:
  - Separation list = set of pairs of nodes
  - Pair of nodes \( \in \) separation list \( \iff \)
    - BVs have been reached by traversal AND
    - BVs do not overlap OR nodes are leaves
- Example:

The Events

- Pair of BVs is in the KSL, will overlap at time \( t \):
- Pair of parent BVs does overlap, will cease to overlap at time \( t \):
Event Handling During Run-Time

- BVs begin to overlap at time $t$:

- Parent BVs will cease to overlap at time $t$:
- Topology of BV changes, i.e., extent is realized by other vertices:

![Diagram](image)

- **Analysis**
  - Worst case:
    - Theorem 1:
      In the worst case, our kinetic separation list is local ($O(n)$), responsive ($O(1)$), efficient, and, arguably, compact ($O(n^2)$).
    - Theorem 2:
      In the average case, our kinetic separation list is local ($O(1)$), responsive ($O(1)$), efficient, and compact ($O(n)$).
Results

- Time for updates and collision check:

- Self Collision:

- Two animated objs:
Kinetic Ray Tracing of Animated Scenes

- Work in Progress
- Current challenge: animated scenes
- Current approaches: re-build acceleration data structure
  - kd-tree, grid

Overview of Our Approach

- Idea: Kinetic Grid of Frusta
  - Maintain sorted list of polygons for each pixel frustum
  - Events:
    - swap pair of polygons
    - delete / insert polygon
  - Problem: Ordering polygons in a frustum (e.g. by kinetic ray casting) leads to high-order polynomials and "ugly" events
  - Track intersection points of polygons with the frusta
- Two Tasks:
  - Event-based polygon tracking
  - Kinetic sorting within frusta
Event-Based Polygon-Tracking

Kinetic Sorting within a Frustum
Advantages

- Independent of rendering frequency
  - E.g., rendering in slow motion without extra cost for updates or intersection tests
- Antialiasing for free
- No complicated mix of different data structures for dynamic and static objects
  - Static objects simply emit no events
- Can handle insertions/deletions during run-time
- Can handle all kinds of animated deformations
  - Only a flightplan is required for every vertex
  - These flightplans may change during simulation

Analysis

- Assumptions:
  - Num intersection between polygons and frusta is bounded
  - Num intersection of flightplans and frusta borders is bounded
- Event-based polygon tracking:
  - Compactness and total number of events: $O(n)$
  - Locality and Responsiveness: $O(1)$
- Kinetic sorting within frusta:
  - Compactness: $O(n)$
  - Locality and Responsiveness:
    - $O(\log n)$ (kinetic tournament) or $O(1)$ (kinetic heap)
    - Num. events: $O(n \log n)$ (kin. tournament) or $O(n \log^2 n)$ (kin. heap)
Conclusions

- Novel, event-based data structures (KDS) for
  - Updating BVHs
  - Continuous collision detection and self-collision detection
  - Ray tracing
- BVH update is in $O(n \log n)$
  - Dito for Kinetic Grid of Frusta
- Uncoupling of data structure updating from query frequency
  - Computational effort for coll.det. independent of num. in-betweens
  - Dito for ray casting of primary rays
- Coll. det. is up to 50 times faster than swept volume approach in practically relevant scenarios

Future Work

- Use our kinetic data structures also for other kinds of primitives like NURBS
- Extend to scenarios with unknown flightplans
- Kinetic Light-buffers (for shadow rays)
- Improve performance of kinetic sorting within frusta by using *kinetic heaters* or *kinetic hangers*
- Integrate other kinetic data structures (like the kinetic AABB-Tree) for secondary rays
- Improve quality of anti-aliasing
- Parallelization