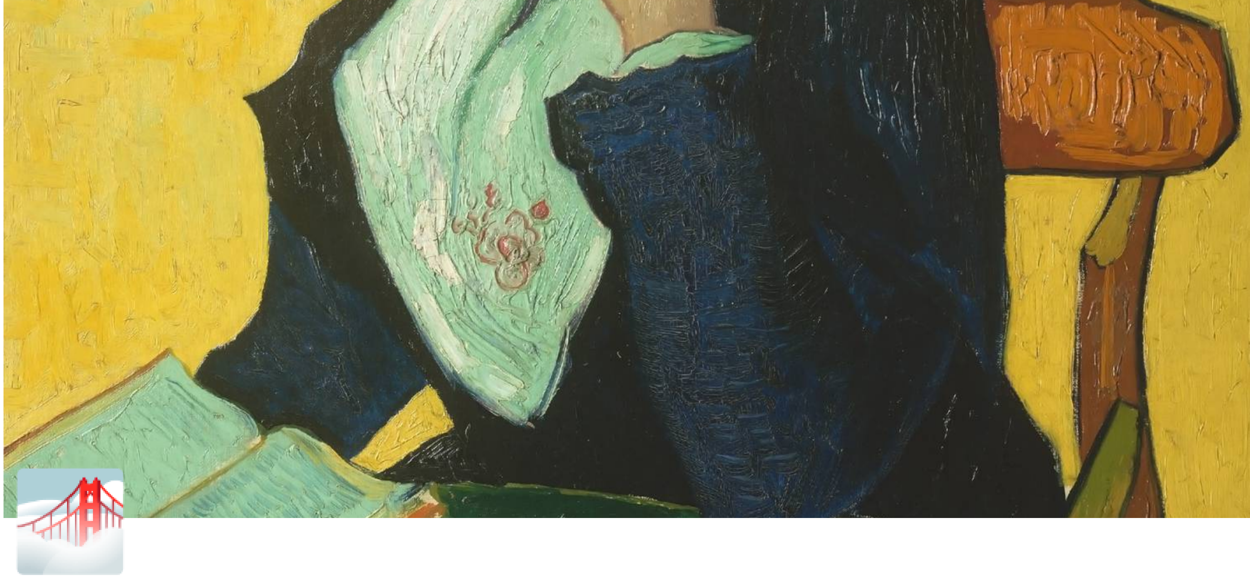
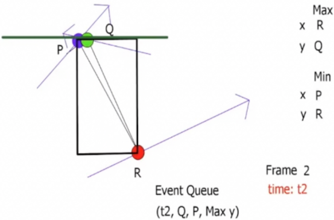


Kinetic Data Structures

Given a number of objects and a **Flight Path**, i.e. an algebraic function defining the position of an object in time, for each object, the purpouse of a Kinetic Data structure is to define an **attribute**, like a kd-tree, a bounding box, a BSP Tree and so on.

everything that describes the attribute except concrete coordinates is called a **Combinatorial structure.** for example if we are trying to compute the bounding box of a set of point in space we can take the pointer of the points representing maximum and minimum coordinates for the bounding box



when something changes in the combinatorial structure we obtain a **Combinatorial change.** as example, the points exchanging the status of “more far away**”**

we can define a **Certificate,** being a simple **geometric relation, or predicate**, involving a few of the objects contained in the Kinetic data structure, in practice a condition to be respected.

if in a specific point in time one of the certificates fails, meaning that the condition is not respected anymore, we obtain an **Event**. In this case we have to do something. events are divided in:

- **External Events:** something changes in the combinatorial structure
- **Internal events:** The certificates of a combinatorial structure changes

finally we can say that a **Kinetic Data Structure** for a geometric attribute is a set of certificates that is true whenever the combinatorial structure of the attribute is valid and a set of rules and algorithms for repairing the attributes and the set of certificates in case of an event

Main loop

```
initialize the attribute for the input objects
initialize the set of certificates
compute all events (failure times) of all certificates
    (usually only up to some time in the future)
initialize the p-queue for all events, sorted by failure time
loop forever:
  do computations using KDS ...
  update time tnew := told + deltat
  while timestamp (front event in queue) <= tnew:
    pop front event from the event queue
    if external event:
      change the attribute
    update the set of certificates:
        some failure times of later events might change
        some certificates may need to be deleted
        maybe, some new certificates need to be created
```

Performance measures

Responsiveness:

A KDS is **responsive**, if the cost to update the set of certificates and the attribute in case of an event is “*small*” Usually, "*small*" means $O(\log^S n)$ or $O(n^\epsilon)$

Efficiency:

A KDS is efficient, if the ratio of $\frac{\#(total\ events)}{\#(external\ events)}$ is small, I.e., the #(internal events), i.e. the attribute's combinatorial structure does not change, is small, I.e., the #events is comparable to the #(attribute changes) over time

Compactness:

A KDS is compact, if the number of certificates is close to linear in the number of input object

Locality:

A KDS is local, if all objects participate only in a small number of certificates one advantage: if an object changes its flight path, then the cost for updating all events affected by it is not too high

Kinetic bounding volume hierarchies

Unlike traditional bounding volume hierarchies (BVH), which are static and designed for stationary objects, KBVHs handle objects that change their position or shape over time.

In a KBVH, the bounding volumes of objects are organized in a hierarchical manner to facilitate efficient collision detection and spatial queries. The hierarchy is typically constructed using a top-down approach, where a recursive partitioning process subdivides the space into smaller regions and assigns bounding volumes to each partition. This process continues until each bounding volume contains a manageable number of objects or reaches a specified depth limit.

The main challenge in constructing a KBVH lies in efficiently updating the hierarchy as the objects move. When an object changes its position or shape, the hierarchy needs to be modified to reflect the new spatial relationships.

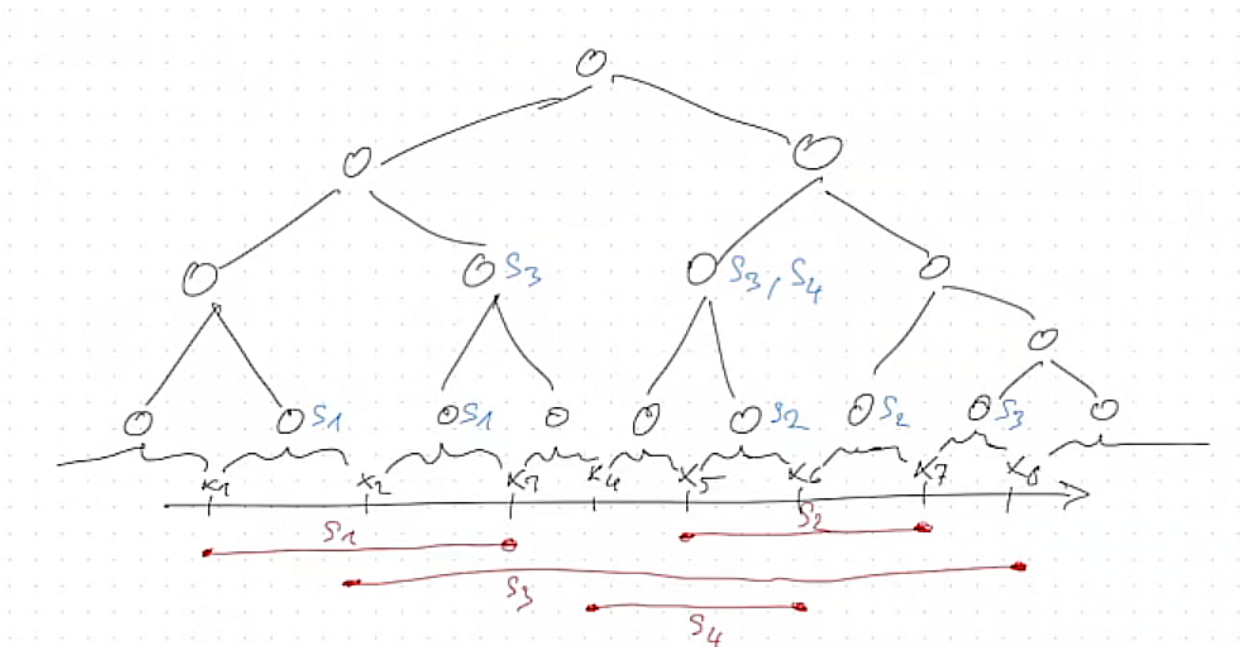
One common approach to updating a KBVH is to use a combination of spatial and temporal coherence. Spatial coherence takes advantage of the fact that nearby objects tend to remain close to each other, reducing the amount of tree reconstruction required. Temporal coherence exploits the fact that object motion is often smooth and predictable over short time intervals, allowing for incremental updates that reuse parts of the existing hierarchy.

Kinetic segment tree

Segment tree

a balanced binary tree over all the **elementary intervals** (see stabbing query problem, the points where ranges begins or ends) where the leaves contains the elementary

intervals and the inner nodes contains the space from interval to interval. more each node store a set of all the segments that are contained in its children's intervals



Construction

```
def buildSegmentTree():
  sort the endpoints in x
  construct skeleton tree over 2n+1 leaves
  for all nodes v bottom up:
    compute interval of v
  for all segments s in S:
    shift s through the tree
```

stabbing query

```
def query(v, q):
  if v is leaf:
    return
  if q in int(v1)
    query(v1, q)
  else:
    query(v1, q)
```

complexity: $O(n \log n)$ space and $O(k + \log n)$ time, given k output segments

adding the kinetic part

to make a kinetic segment tree we augment the standard one to store the endpoints in an array $R[0,...,2n-1]$ and we denote the segment $s_i = (a_i, b_i), 0 \leq a_i, b_i \leq 2n-1 \in \mathbb{N}$ are indicies into the array R of the segments, called **ranks**, so a segment whould be $s_i = [R[a_i], R[b_i]]$. we will also maintain a list $\mathcal{L}(s) = \{\nu | s \in S(\nu)\}$ of pointers to the nodes it is stored, “**fragment list**” and an array A with pointers to ν_i leaves for the elementary intervals $(i, i+1)$. the certificates are $R[i] < R[i+1]$, when this is violated it means that 2 endpoints has swapped their order so our intervals becomes $s = (i, j) \rightarrow s = (i, j+1)$

```
def update()
  init v = v_j #leaf for EI(j, j+1)
  m= sibling of v
  while s in S(m):
    delete s from S(m)
    v = parent(v)
    m = sibling of v
  add s to S(v)
```

Lemma

given a set s of n moving segments $\subseteq \mathbb{R}$ there is a KST with $O(n \log n)$ size, which can be modified in time $O(1)$ in case of a certificate failure. the worst case update time is $O(\log n)$ and the KST is local and efficient

proof

expected time: $h \leq h(j)$ is the height of the highest node ν such that $(j, j+1)$ is the right most end of $int(\nu)$

let’s claim $\bar{h} = \frac{1}{2n} \sum_{j=0}^{2n} h(j)$, this is a $O(1)$ operation, so the running time follows directly

cool material

https://cp-algorithms.com/data_structures/segment_tree.html#:~:text=A Segment Tree is a,quick modification of the array.

https://www.geeksforgeeks.org/segment-tree-sum-of-given-range/