LOOK
A Geometry-Centered Kernel for Geometric Computation

Stefan Funke   Kurt Mehlhorn

Max-Planck-Institut f. Informatik

Saarbruecken

Germany

Predicates vs Constructions

- standard floating-point filtering only works for predicate evaluations, as we assume availability of exact input data to the predicate
- current geokernels *always* perform constructions exactly
  - coordinate-centered view
  - overkill, if exact coordinates are never required
  - valuable geometric information is lost
- ideally we should only construct an object 'exactly' if it is really needed in the evaluation of some predicate
- concentrate on the geometry instead of on the coordinates
- exploit geometric information in the evaluation of predicates

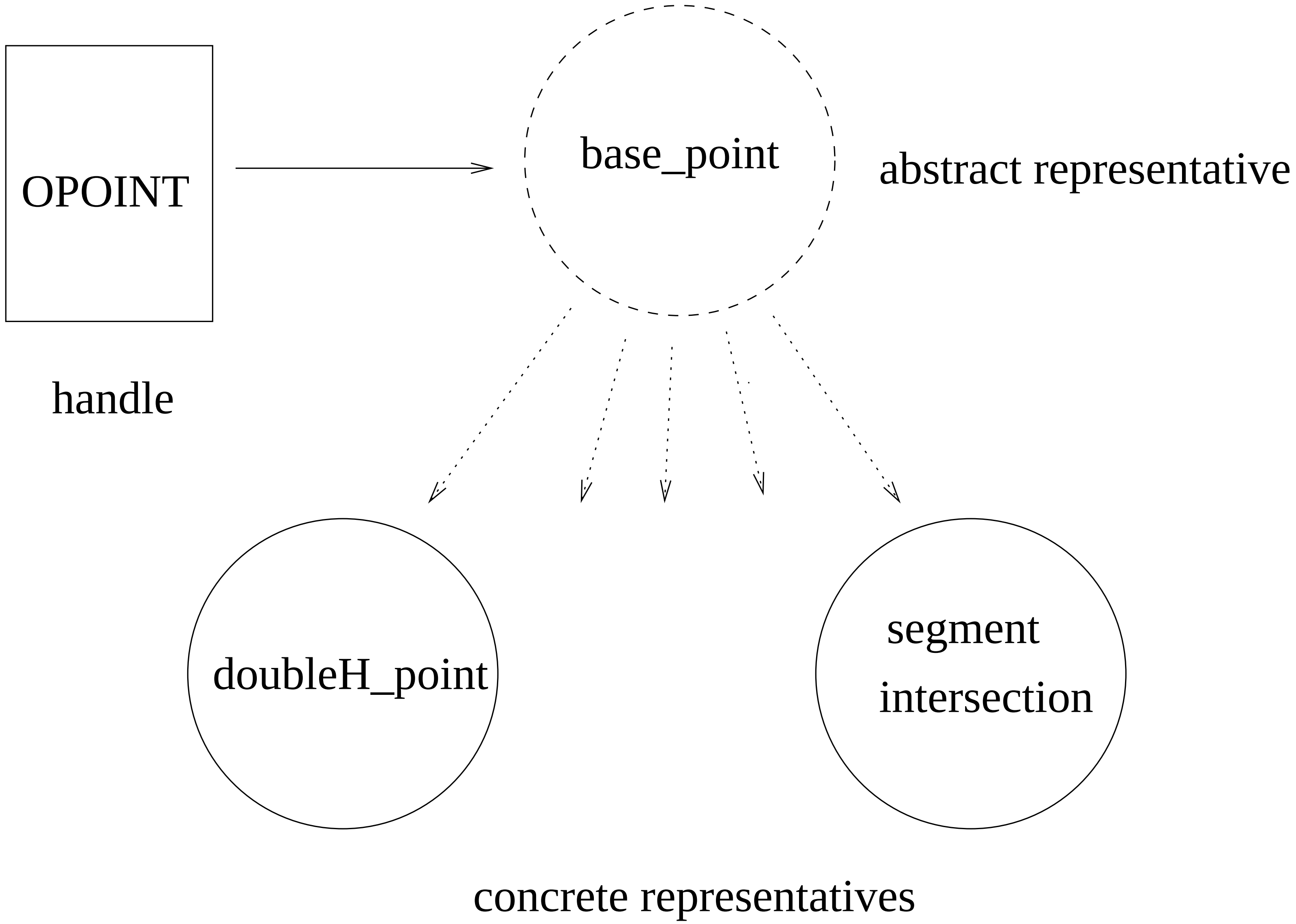
Lazy Constructions

- when constructing geometric objects
  - do not compute their exact numerical representation but just a floating-point approximation with error bounds
  - store the geometric constructions and the objects involved to allow exact arithmetic evaluation if needed later on
- make this mechanism transparent such that predicates do not have to distinguish between constructed points and points of the input data

Features of LOOK

- C++ datatypes for basic geometric objects: OPOINT, OLINE, OSEGMENT, ...
- objects can be initialized with input data or constructed using geometric operations like intersection, center of a triangle, intersection of two circles, etc.
- predicates work on these objects, irrespectively of whether they come from input data or a geometric construction
- for exact evaluation of predicates, the 'weakest' arithmetic is used that is sufficient to get a correct result (= multi-stage filtering)

Implementation Details (1)
Reference Counting and Handles



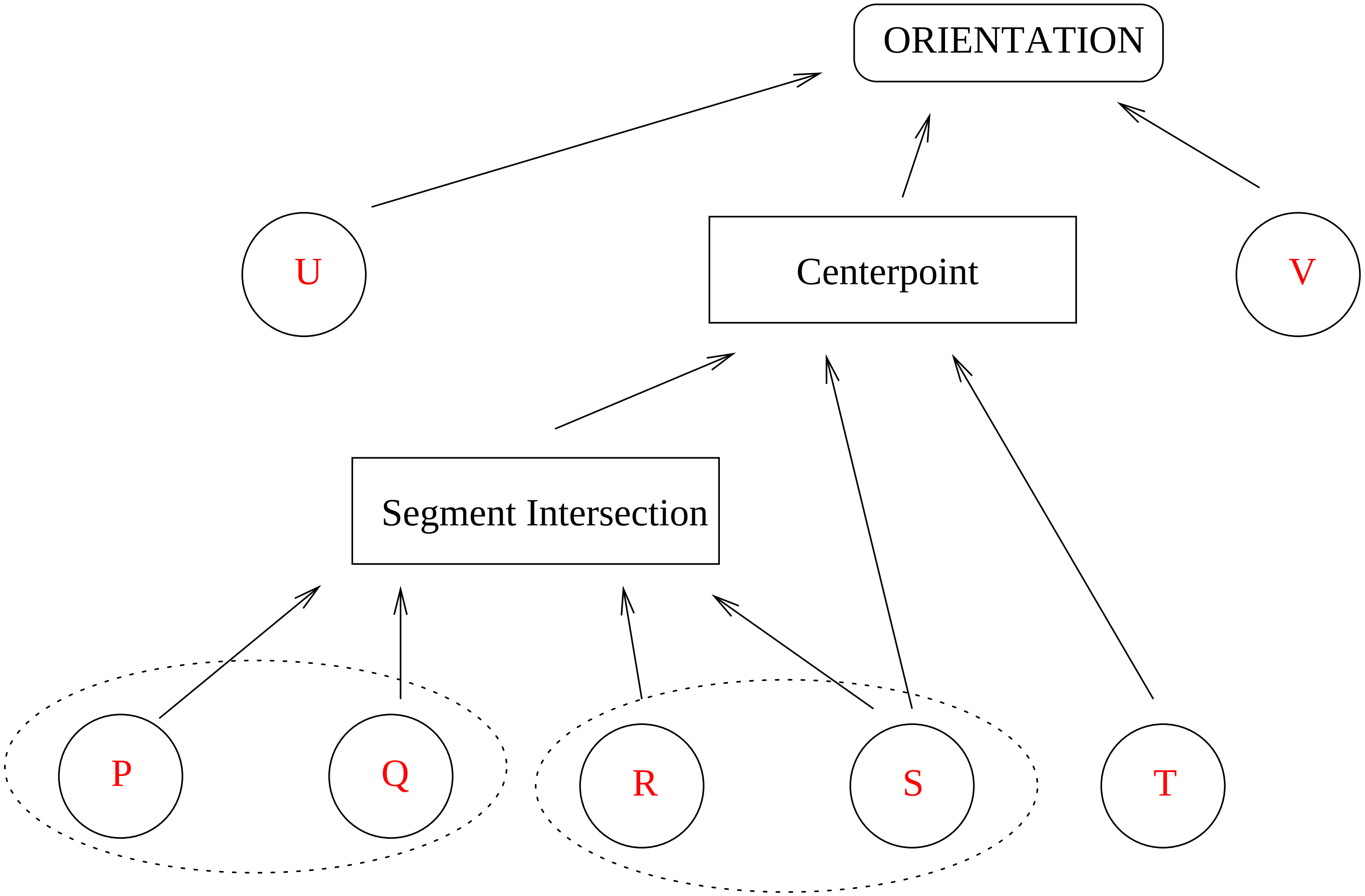
Implementation Details (2)
Floating-Point filters

predicate evaluation:

- 1st stage: semi-static filters on cartesian coordinates automatically generated by EXPCOMP
- 2nd stage: interval filters (fully-dynamic) on cartesian coordinates
- 3rd stage: interval filters on homogenous coordinates
- 4th stage: exact arithmetic evaluation: depending on type of objects involved either using exact LEDA integer or real arithmetic

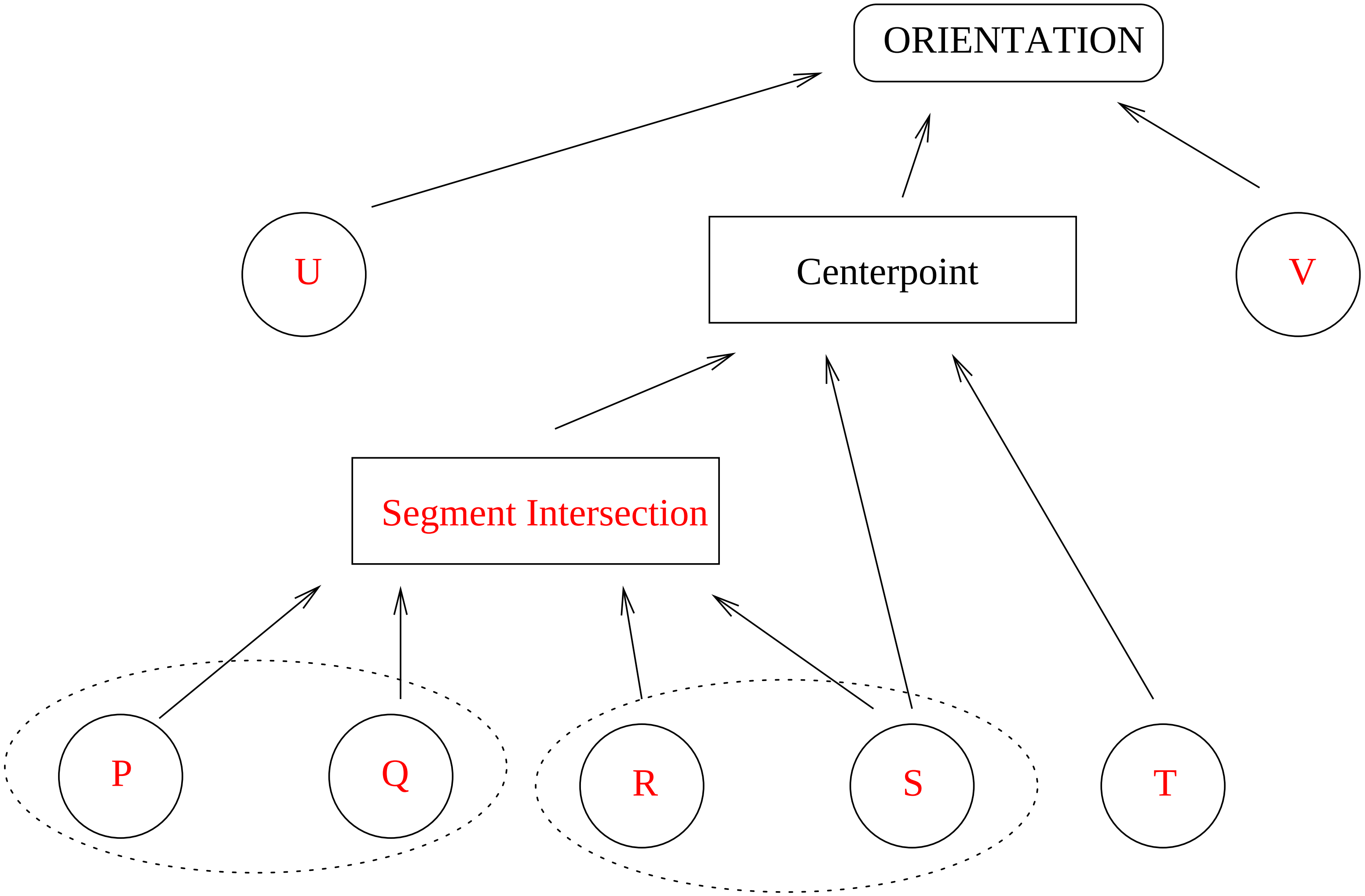
constructions are always performed lazily, only cartesian approximations are computed on instantiation, other representations (LEDA integer and real) only on demand

Lazy Evaluation Strategies



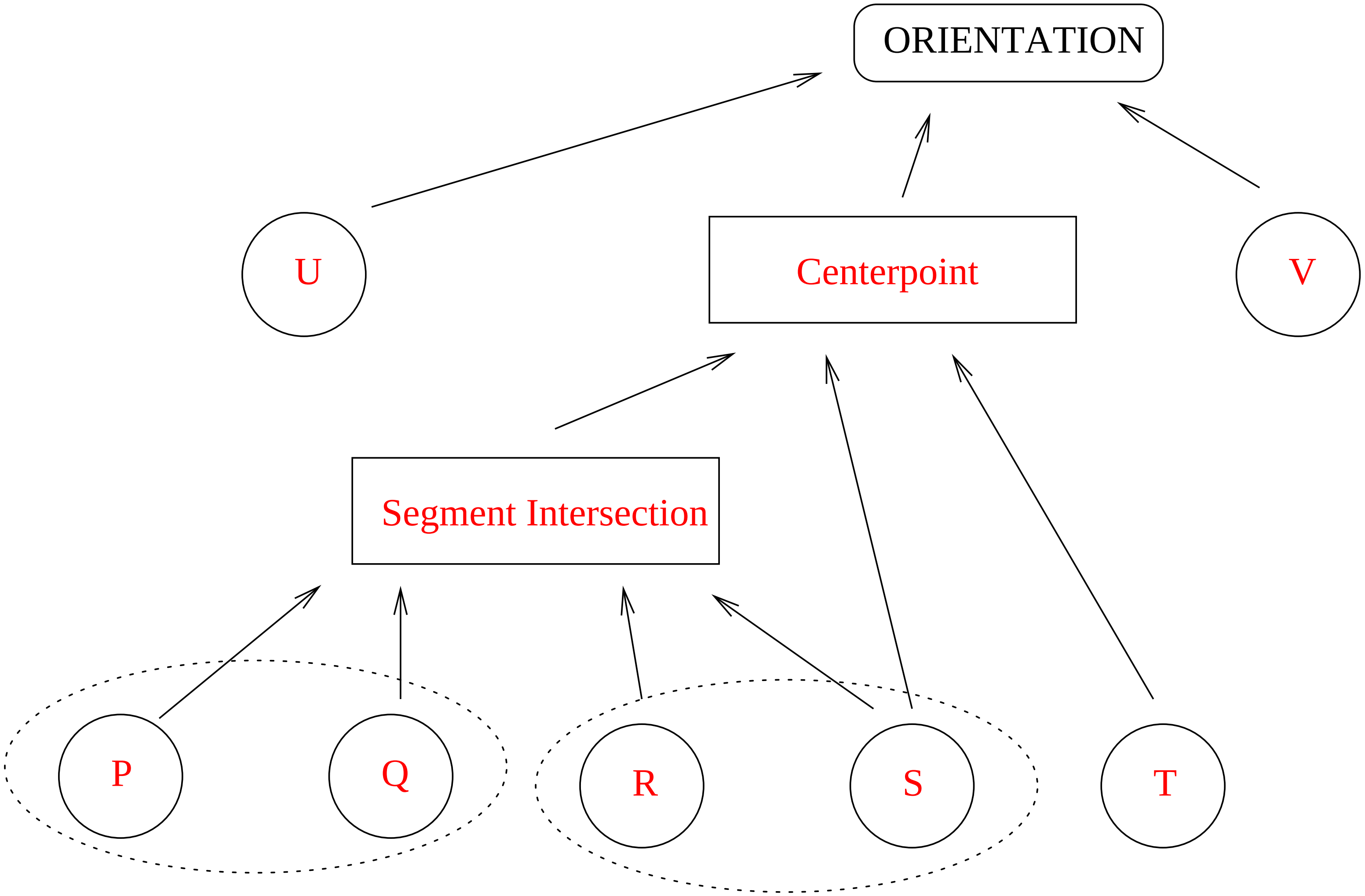
- progressive exact evaluation

Lazy Evaluation Strategies



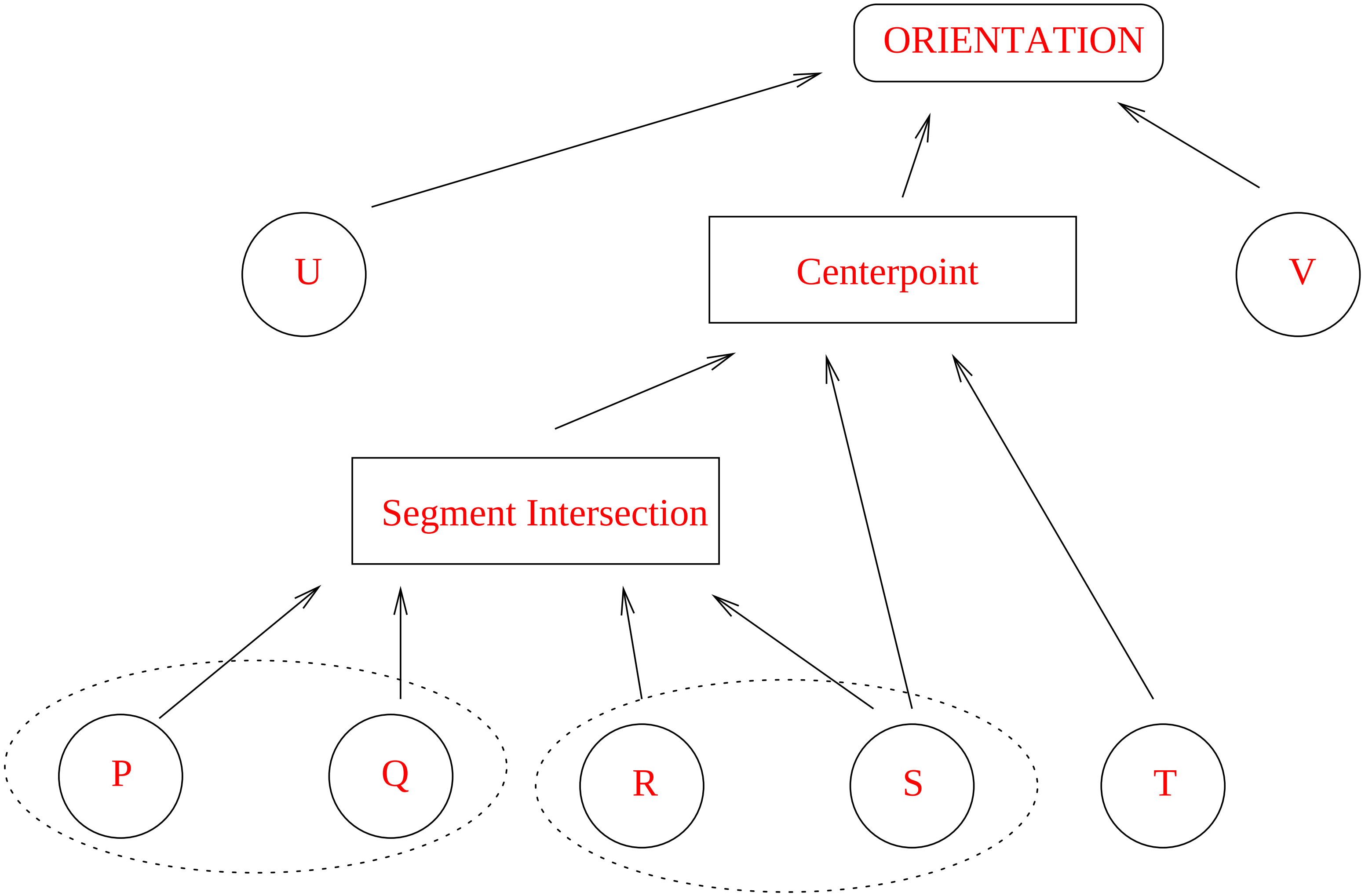
- progressive exact evaluation

Lazy Evaluation Strategies



- progressive exact evaluation

Lazy Evaluation Strategies



- progressive exact evaluation
- conservative memory management

Programming with LOOK

three scenarios

- no new constructions/predicates $\Rightarrow$ similar to existing geokernels like LEDA or CGAL but making full use of all features mentioned above
- new predicates $\Rightarrow$ generic way to modify predicates to make use of lazy evaluation techniques, also using EXPCOMP, if necessary
- new constructions $\Rightarrow$ lazy-evaluation mechanism encapsulated in the base-classes, only the member functions actually computing the data have to be provided

Experimental Results(1)

LEDA RatKernel	CGAL<LOOK>	C<leda_real>	LEDA Fp
0.38	1.00	3.02	0.16

Table 1: ConvexHull of 50000 32bit integer points

	CGAL<LOOK>	C<leda_real>	C<double>
intersection	0.52	7.51	0.11
total	1.08	10.1	0.17

Table 2: ConvexHull of the intersection points of 500 circles

Experimental Results(2)

Nesting	LEDA	LEDA RatKernel		LOOK	
Depth	Fp	no cart. FP	cart. FP	MaxMem	MinMem
1	0.53	0.55	0.53	1.34	1.51
2	0.52	1.22	1.11	1.89	2.20
3	0.56	73.1	1.84	2.34	3.21
4	0.53	836	103	4.16	6.61
5	0.55	8065	1009	24.23	21.60

Table 3: Iterated Voronoi computations of an integer point set

Conclusion

- big improvement for moderate nesting depths
- easy to use
- does not really help for large nesting depths $\Rightarrow$ algorithmic changes needed, e.g.:
  - rounding
  - algorithms with low-degree predicates
  - structural filtering