Functional Programming is Everywhere

Ilya Sergey

ilyasergey.net

Yale-NUS College

NUS National University of Singapore

zilliqa

PLMW @ ICFP 2019

About myself

MSc	Saint Petersburg State University, 2008
PhD	KU Leuven, 2008-2012
Currently	Associate Professor (tenure-track) at Yale-NUS College & NUS
Previously	Lecturer → Associate Professor at University College London Postdoc at IMDEA Software Institute Software Engineer at JetBrains

Functional programmer since 2005

Functional programmer since 2005

OCaml

Groovy

Scala

2005 2006 2007 2008 2010 2011

Functional Languages

OCaml

Scala

Groovy

JS

Java

The Essence of Functional Languages

- Higher-order functions and closures
- Types and Type Inference
- Polymorphism
- Laziness
- Point-free style
- Combinator Libraries
- Purely functional data structures
- Algebraic Data Types
- Pattern Matching
- Folds
- Continuations and CPS
- Structural Recursion
- Type Classes
- Monads

Check out this year's ICFP program…

This Talk

Functional Programming

This Talk

Functional Programming

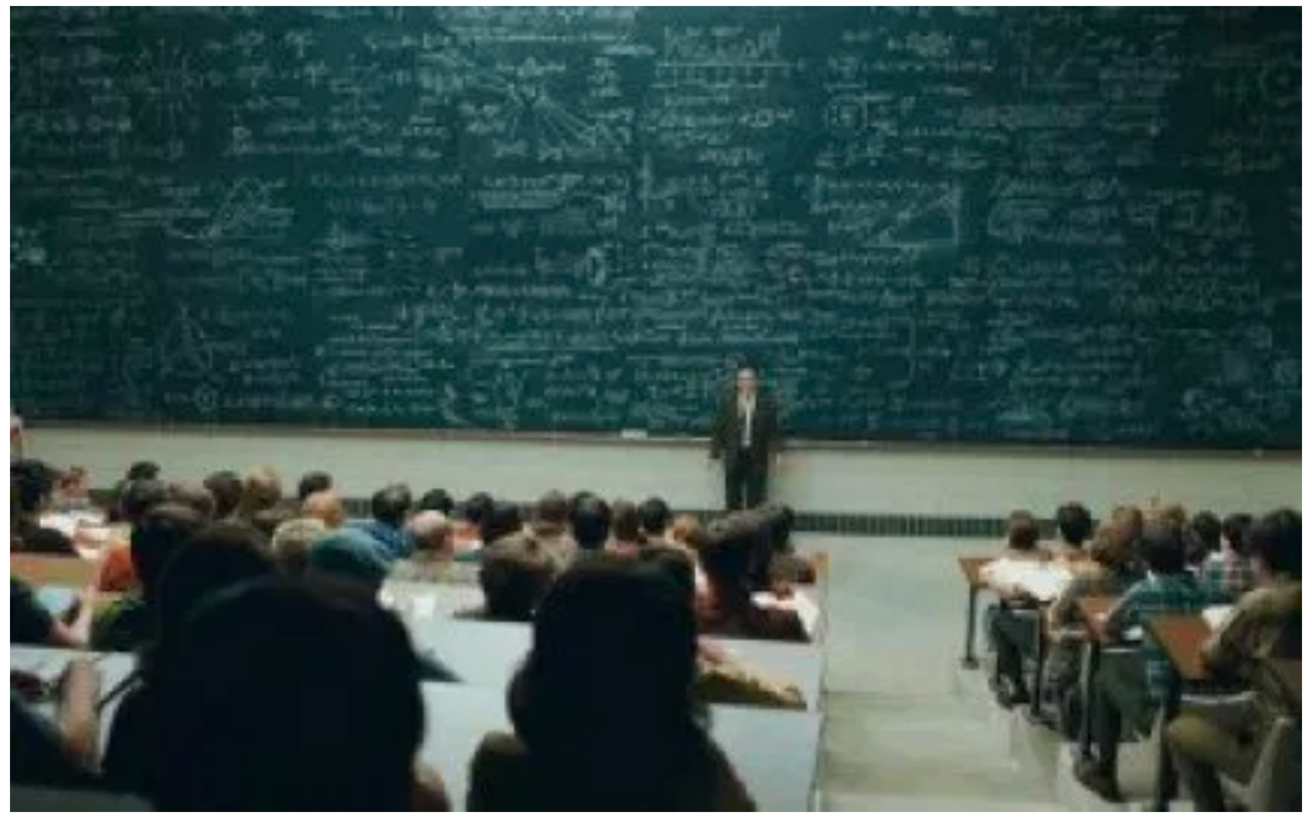
This Talk

Functional Programming Ideas

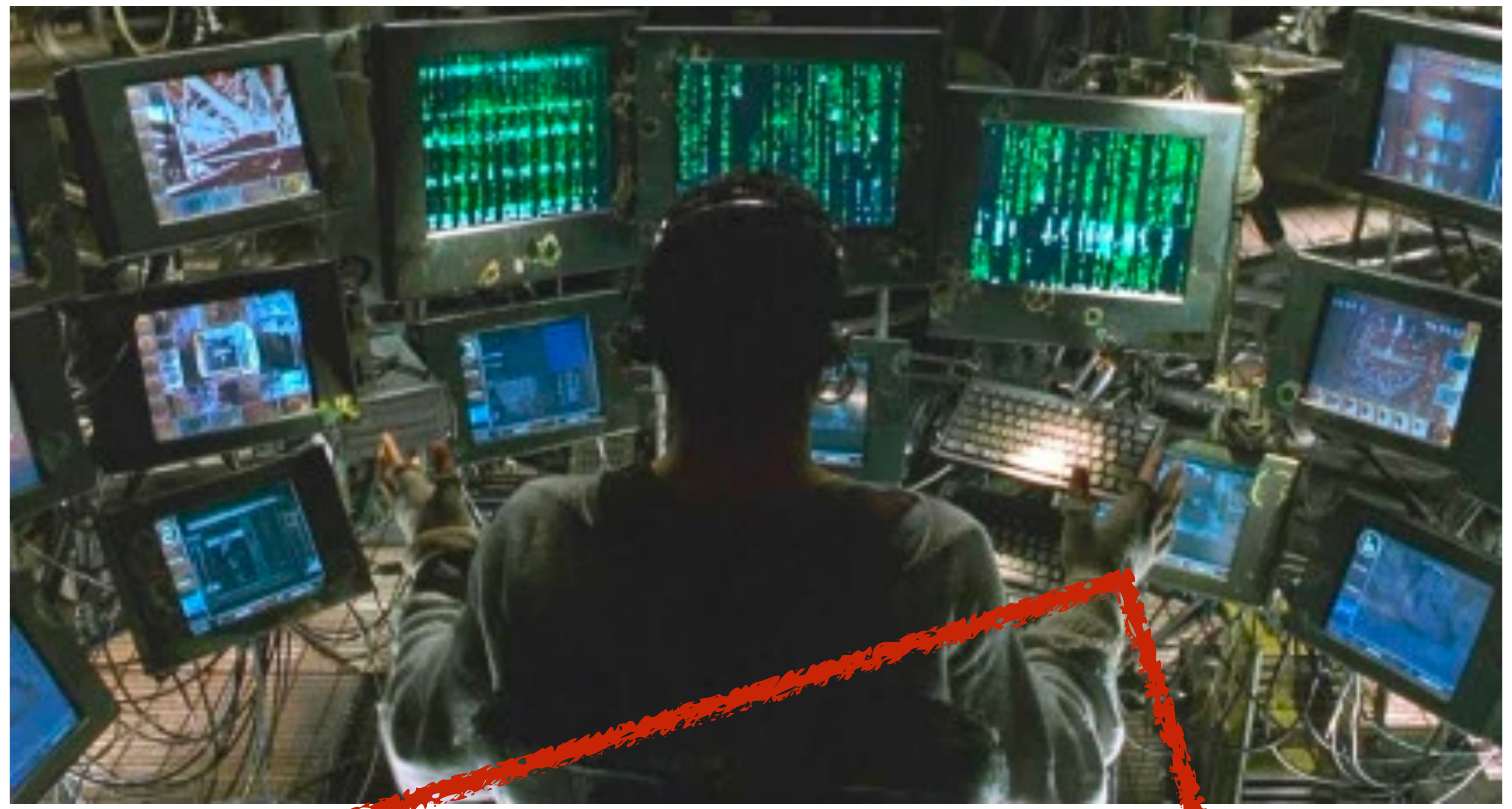
Functional Programming Ideas in…

Research

Teaching



Software Engineering



Disclaimer:
personal experience

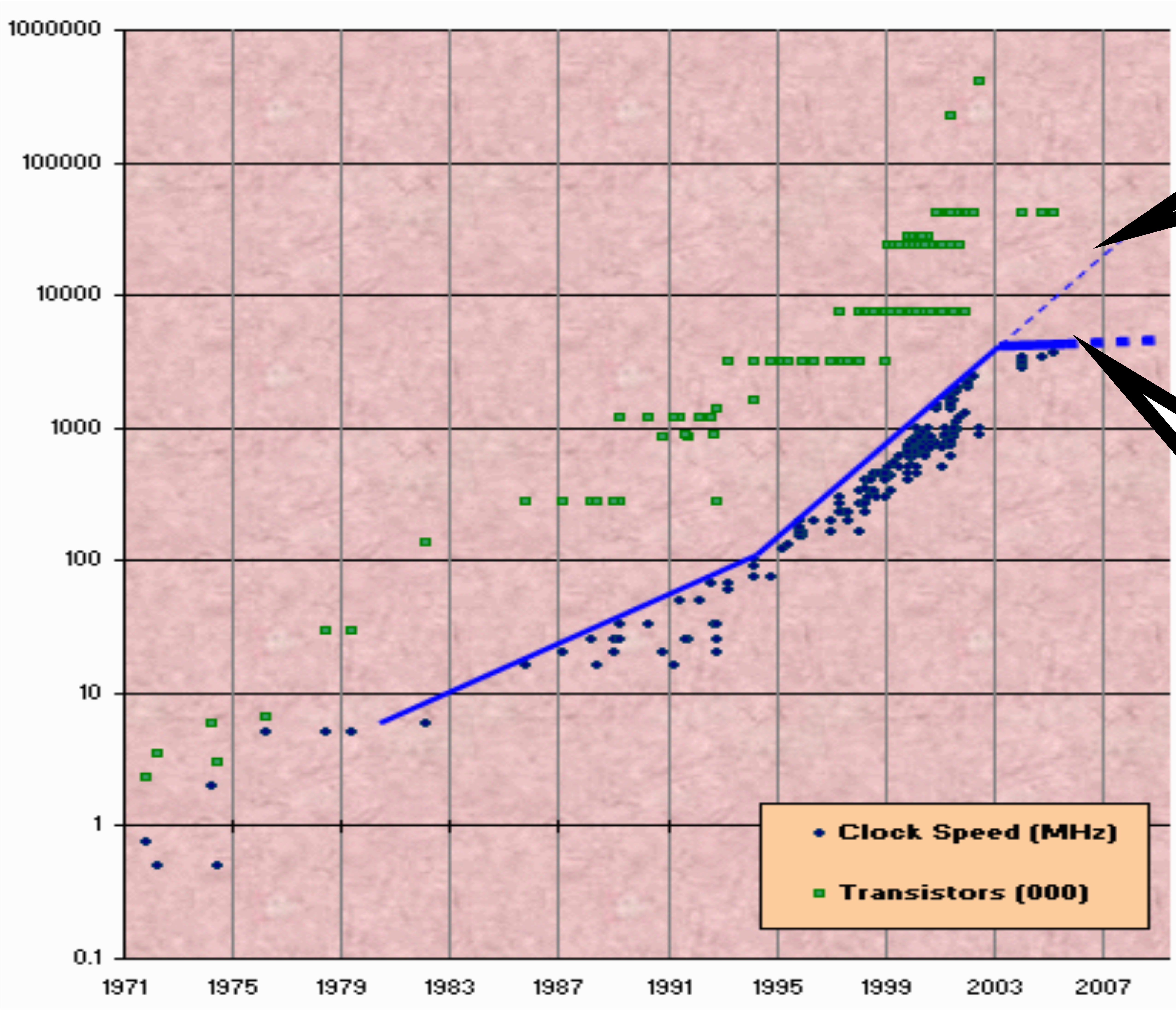
Functional Programming Ideas in…

Research

Teaching

Software Engineering

Moore’s Law

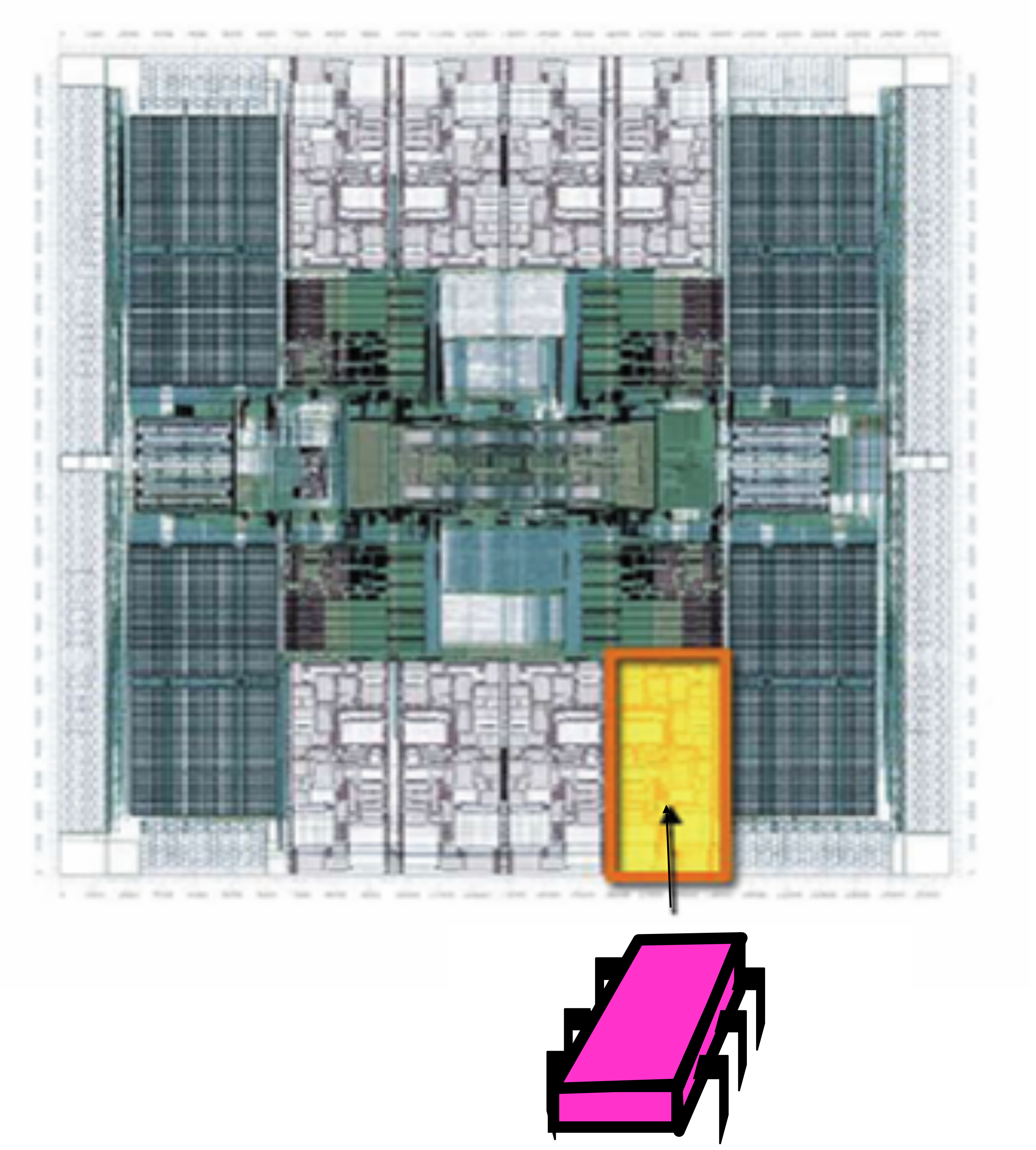


Transistor count still rising

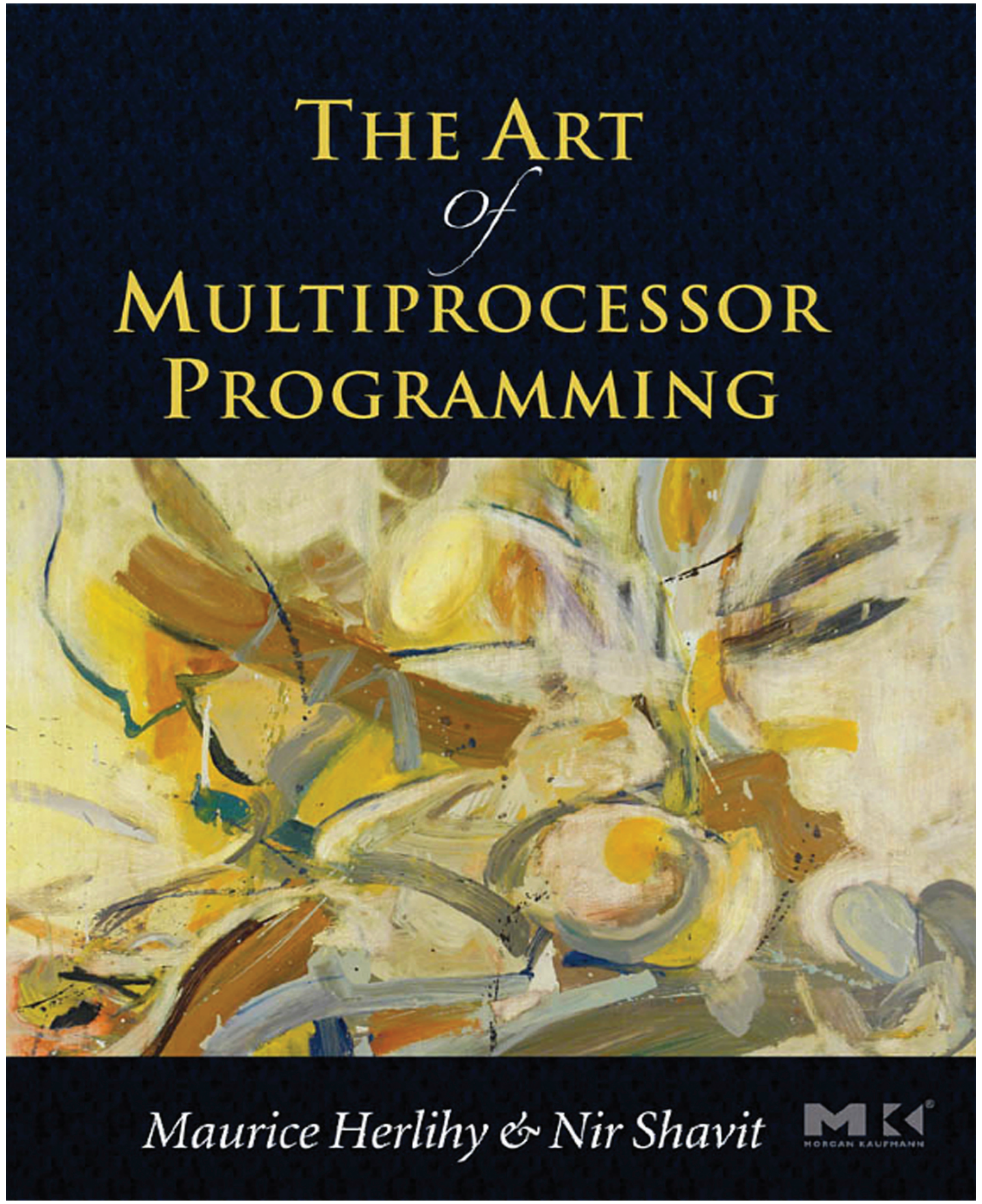
Clock speed flattening sharply

The Multicore Processor

All on the same chip



Sun T2000 Niagara


THE ART
of
MULTIPROCESSOR
PROGRAMMING
Maurice Herlihy & Nir Shavit

Specifications for Concurrent Data Structures

My research agenda since 2014

Reusable Specifications for Concurrent Data Structures

Abstract Specifications of a Stack

$\{ S = xs \}$ `push x` $\{ S' = x :: xs \}$

$\{ S = xs \}$ `pop()` $\{ \quad res = \bot \wedge S = Nil$
$\vee\ \exists x, xs'.\ res = x \wedge$
$xs = x :: xs' \wedge S' = xs' \}$

Suitable for sequential programming

Abstract Specifications of a Stack

$\{ S = xs \}$ `push x` $\{ S' = x :: xs \}$

$\{ S = xs \}$ `pop()` $\{ \; res = \bot \wedge S = Nil$
$\vee \; \exists x, xs'. \; res = x \wedge$
$xs = x :: xs' \wedge S' = xs' \}$

Breaks composition in the presence of thread interference.

$\{ S = \mathrm{Nil} \}$

```
y := pop();
```

$\{ y = ??? \}$

$\{ S = Nil \}$

```
y := pop();   ||   push 1;
                   push 2;
```

$\{ y \in \{\perp\} \cup \{1, 2\} \}$

$\{ S = \mathrm{Nil} \}$

```
y := pop();   ||   push 1;   ||   push 3;
                   push 2;
```

$\{ y \in \{\perp\} \cup \{1, 2, 3\} \}$

No proof reuse
(not thread-modular)

A reusable specification for pop?

$\{ S = Nil \}$

```
y := pop();
```

$\{ y = ??? \}$

The Essence of Functional Languages

- Higher-order functions and closures
- Types and Type Inference
- Polymorphism
- Laziness
- Point-free style
- Combinator Libraries
- Purely functional data structures
- Algebraic Data Types
- Pattern Matching
- Folds
- Continuations and CPS
- Structural Recursion
- Type Classes
- Monads

The Essence of Functional Languages

- Higher-order functions and closures
- Types and Type Inference
- Polymorphism
- Laziness
- Point-free style
- Combinator Libraries
- Purely functional data structures

- Algebraic Data Types
- Pattern Matching
- Folds
- Continuations and CPS
- Type Classes
- Monads

Enablers for Modular Development

Idea: Interference-*Parameterised* Specifications

Capture the effect of *self-thread*,
parametrise over the effect of *others*.

(aka *Subjective specifications*)

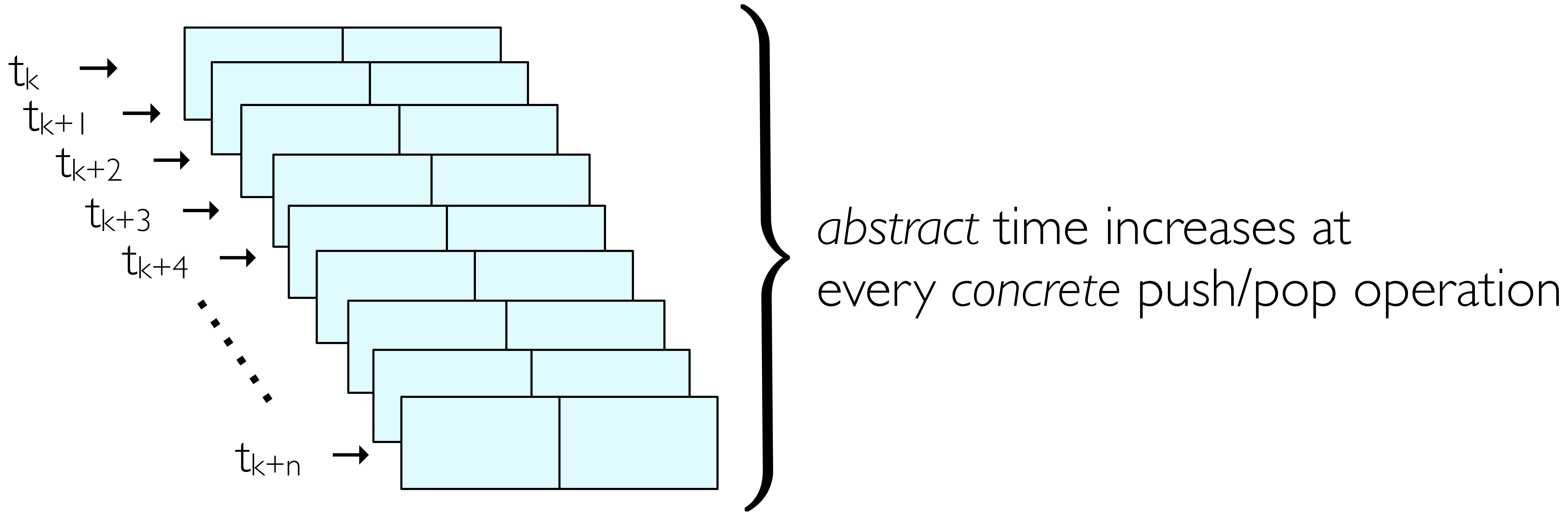
Atomic stack specifications

$\{ S = xs \}$ **`push x`** $\{ S' = x :: xs \}$

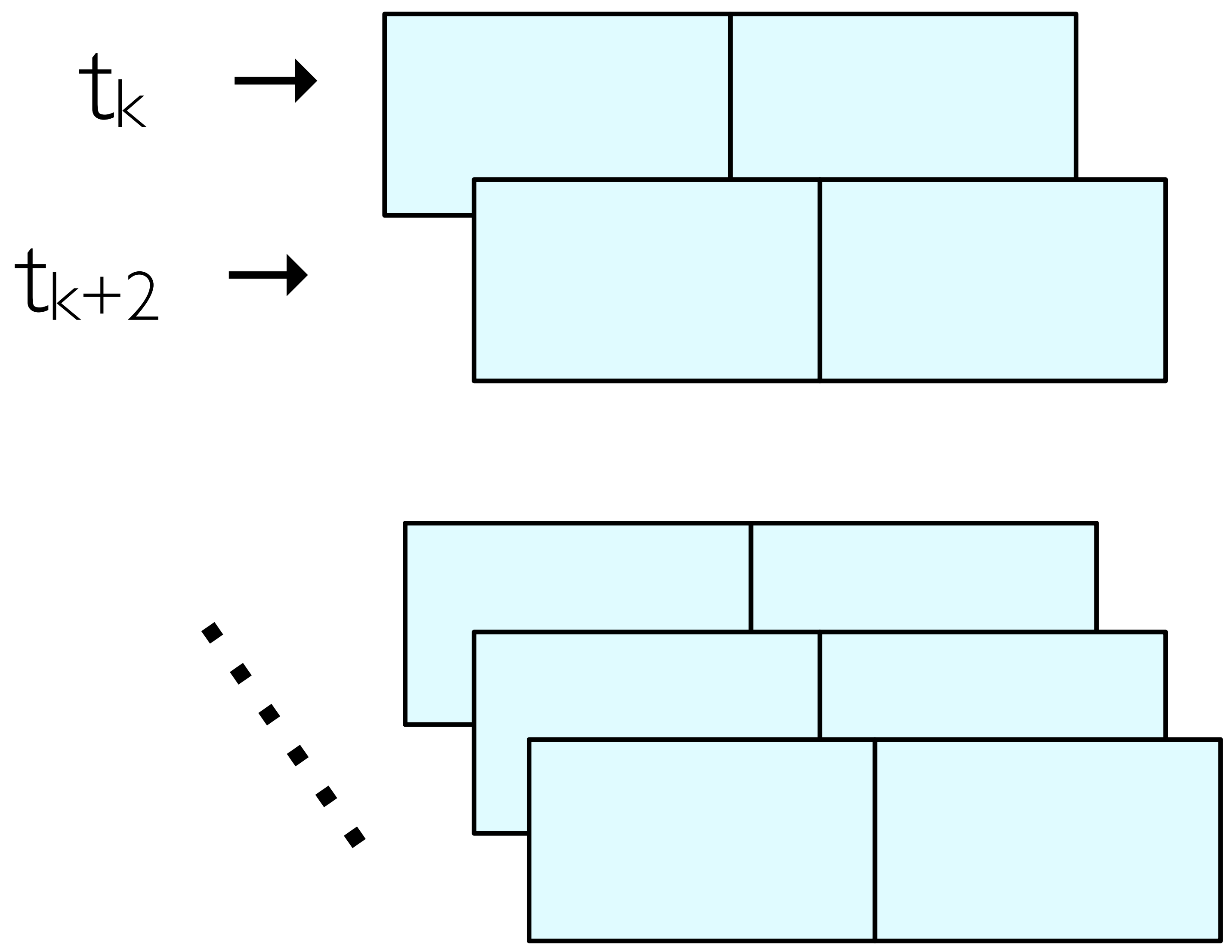
Atomic stack specifications

$t_k \rightarrow$	xs	$x :: xs$

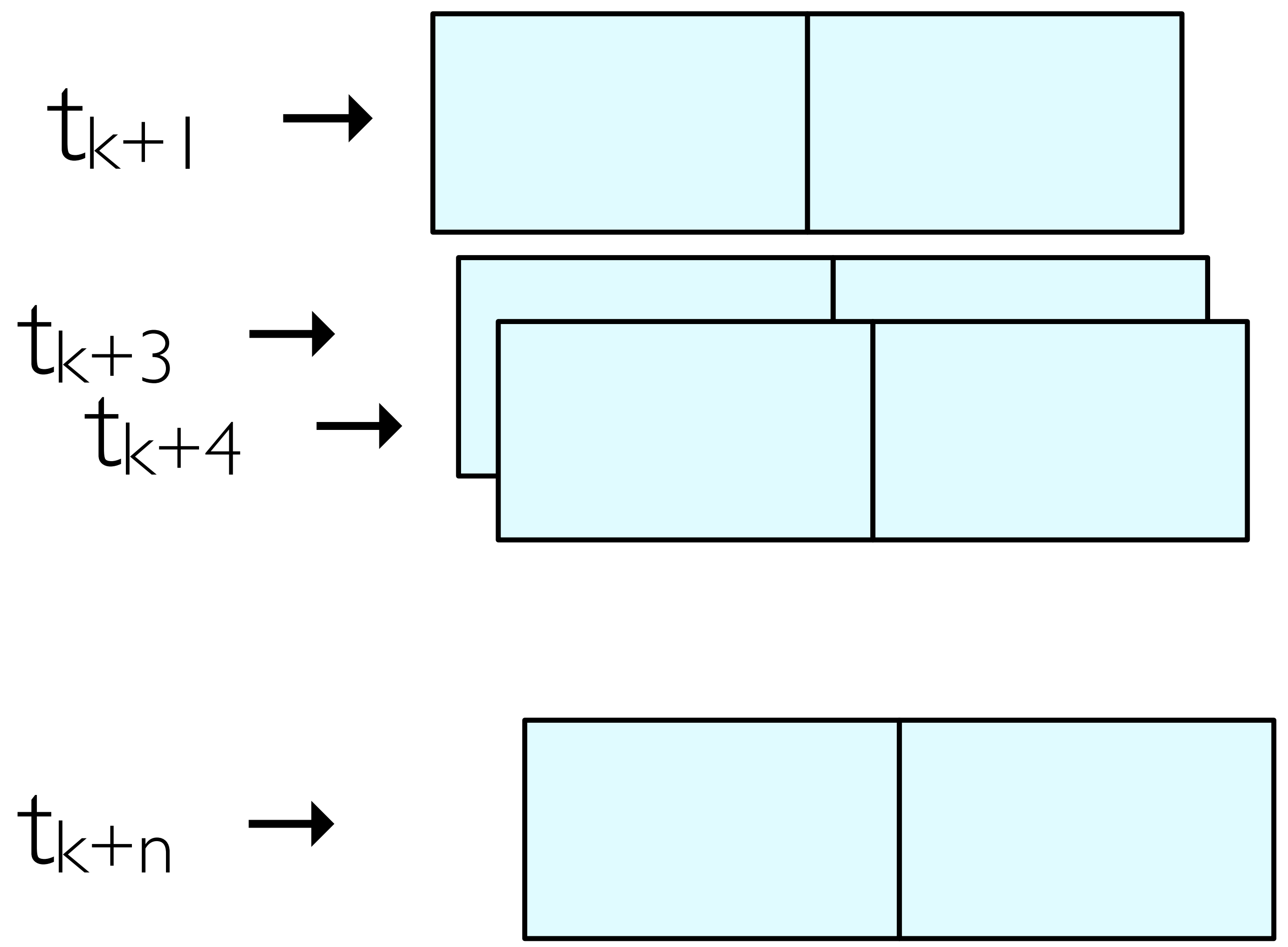
“abstract timestamp”


t_k
t_{k+1}
t_{k+2}
t_{k+3}
t_{k+4}
t_{k+n}
abstract time increases at
every *concrete* push/pop operation

Changes by *this* thread



Changes by *other* threads



Subjective stack specifications

Sergey, Nanevski, Banerjee [ESOP'15]

- H_s — "ghost history" of my **pushes/pops** to the stack
- H_o — "ghost history" of **pushes/pops** by all other threads

$\{ H_s = \varnothing \}$

λH_o . **pick** (*pushed*(H_o))

```
y := pop();
```

$\{ y = \bot \vee y = v, \text{ where } v \in \underbrace{pushed(H_o)} \}$

what I popped depends
on what the others have pushed

$\{ S = Nil \}$

`y := pop();`	`push 1;` `push 2;`	`push 3;`

$\{ H_s = \varnothing \}$

```
push 1;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \}$

```
push 2;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \oplus t_2 \mapsto (ys, \mathbf{2}::ys) \}$

$$\{ H_s = \varnothing \}$$

```
push 1;
```

$$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \}$$

```
push 2;
```

$$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \oplus t_2 \mapsto (ys, \mathbf{2}::ys) \}$$

$\{ H_s = \varnothing \}$

```
push 1;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \}$

```
push 2;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \oplus t_2 \mapsto (ys, \mathbf{2}::ys) \}$

$\|$

$\{ H_s = \varnothing \}$

```
push 3;
```

$\{ H_s = t_3 \mapsto (zs, \mathbf{3}::zs) \}$

$\{ H_s = \varnothing \}$

```
y := pop();
```

$\{ y \in \{\bot\} \cup \textit{pushed}(H_o) \}$

$\{ H_s = \varnothing \}$

```
push 1;

push 2;
```

$\{ H_s = t_1 \mapsto (xs, \mathbf{1}::xs) \oplus$
$t_2 \mapsto (ys, \mathbf{2}::ys) \}$

$\{ H_s = \varnothing \}$

```
push 3;
```

$\{ H_s = t_3 \mapsto (zs, \mathbf{3}::zs) \}$

$\{ H_s = \varnothing \}$

```
y := pop();
```

$\{ y \in \{\bot\} \cup \underline{\mathit{pushed}}(H_o) \}$

||

$\{ H_s = \varnothing \}$

```
push 1;

push 2;
```

$\{ H_s = t_1 \mapsto (xs, \underline{\mathbf{1}}{::}xs) \oplus$
$t_2 \mapsto (ys, \underline{\mathbf{2}}{::}ys) \}$

||

$\{ H_s = \varnothing \}$

```
push 3;
```

$\{ H_s = t_3 \mapsto (zs, \underline{\mathbf{3}}{::}zs) \}$

$\{ H_s = \varnothing \}$

```
y := pop();
```

$\{ y \in \{\bot\} \cup \{\mathbf{1}, \mathbf{2}, \mathbf{3}\} \}$

||

$\{ H_s = \varnothing \}$

```
push 1;
push 2;
```

$\{ H_s = t_1 \mapsto (xs, \underline{\mathbf{1}}{::}xs) \oplus$
$t_2 \mapsto (ys, \underline{\mathbf{2}}{::}ys) \}$

||

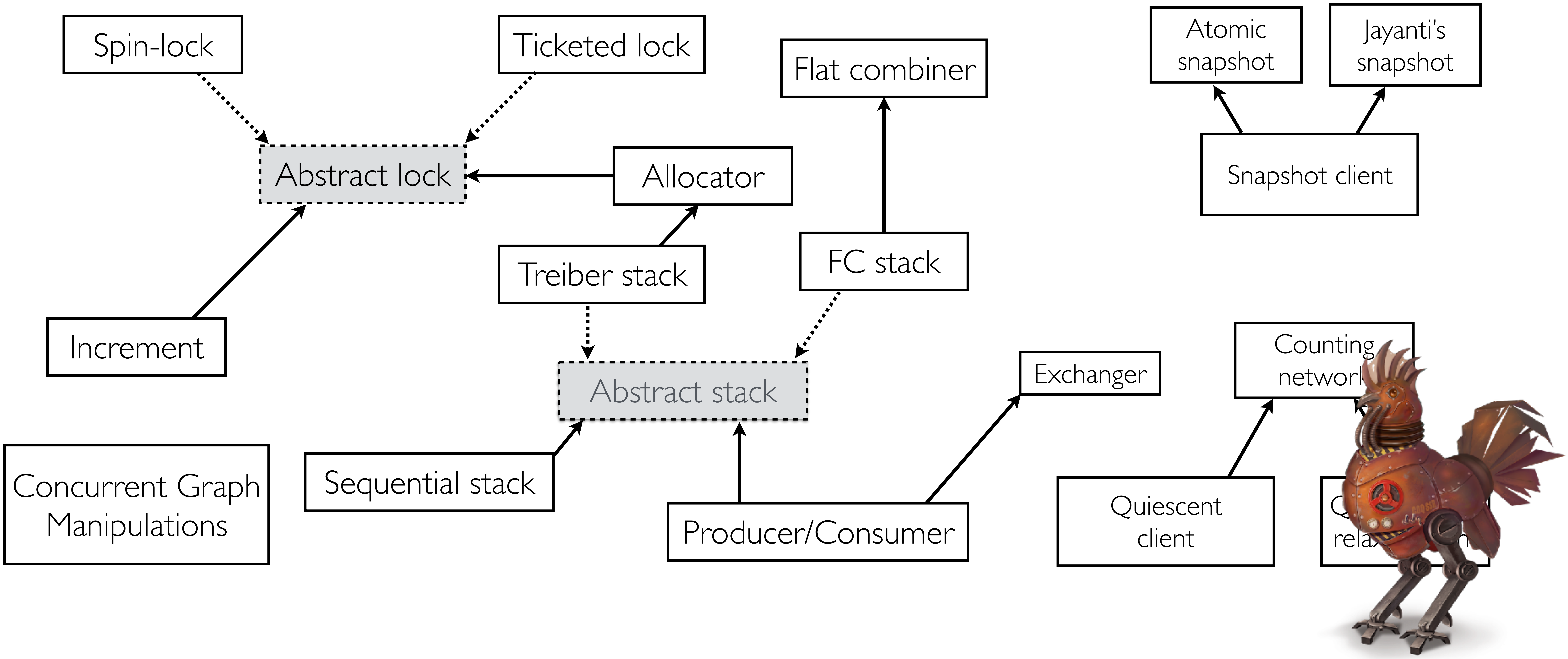
$\{ H_s = \varnothing \}$

```
push 3;
```

$\{ H_s = t_3 \mapsto (zs, \underline{\mathbf{3}}{::}zs) \}$

Payoff: Verified Concurrent Libraries

Sergey et al. [PLDI'15]



Functional Programming Ideas in…

Research

Teaching

Software Engineering

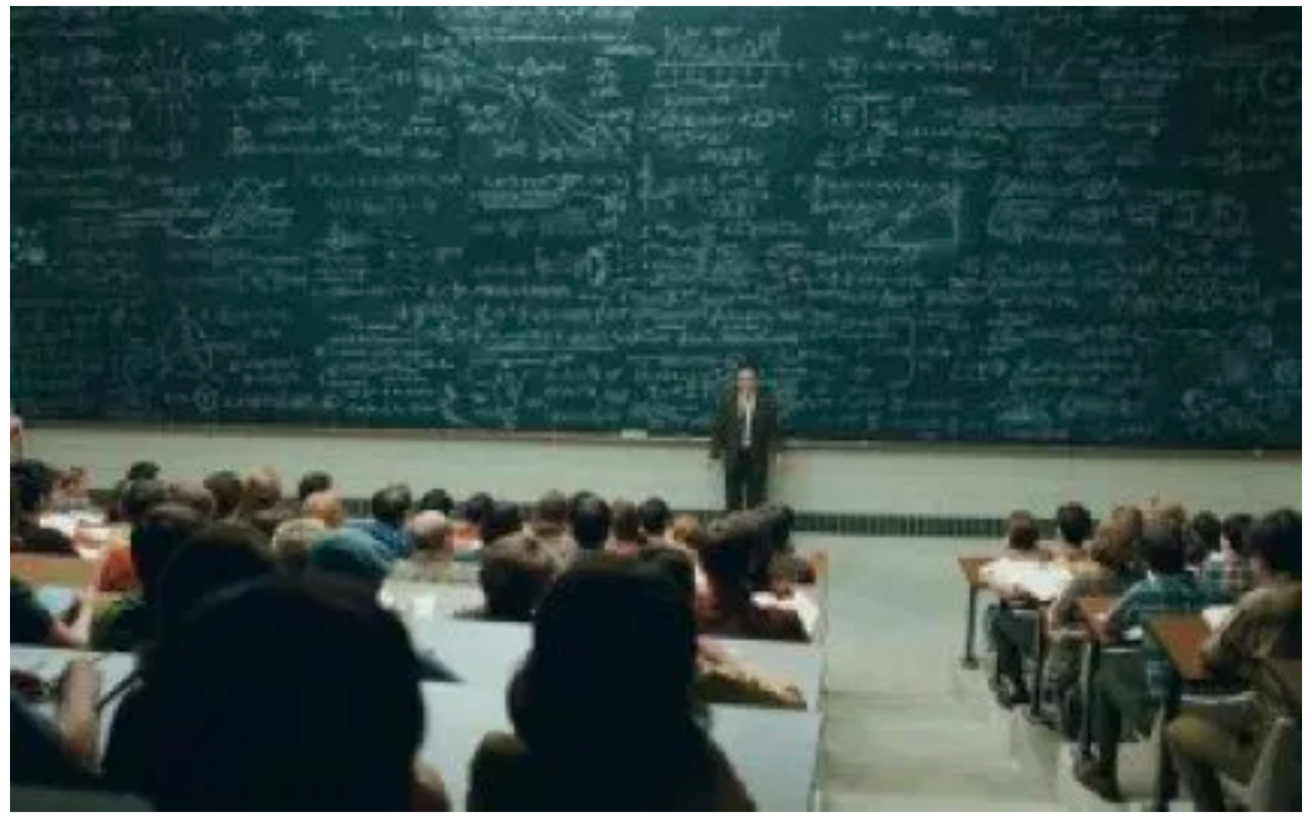
… for modularity and proof reuse

Functional Programming Ideas in…

Research

… for Modularity and Proof Reuse

Teaching

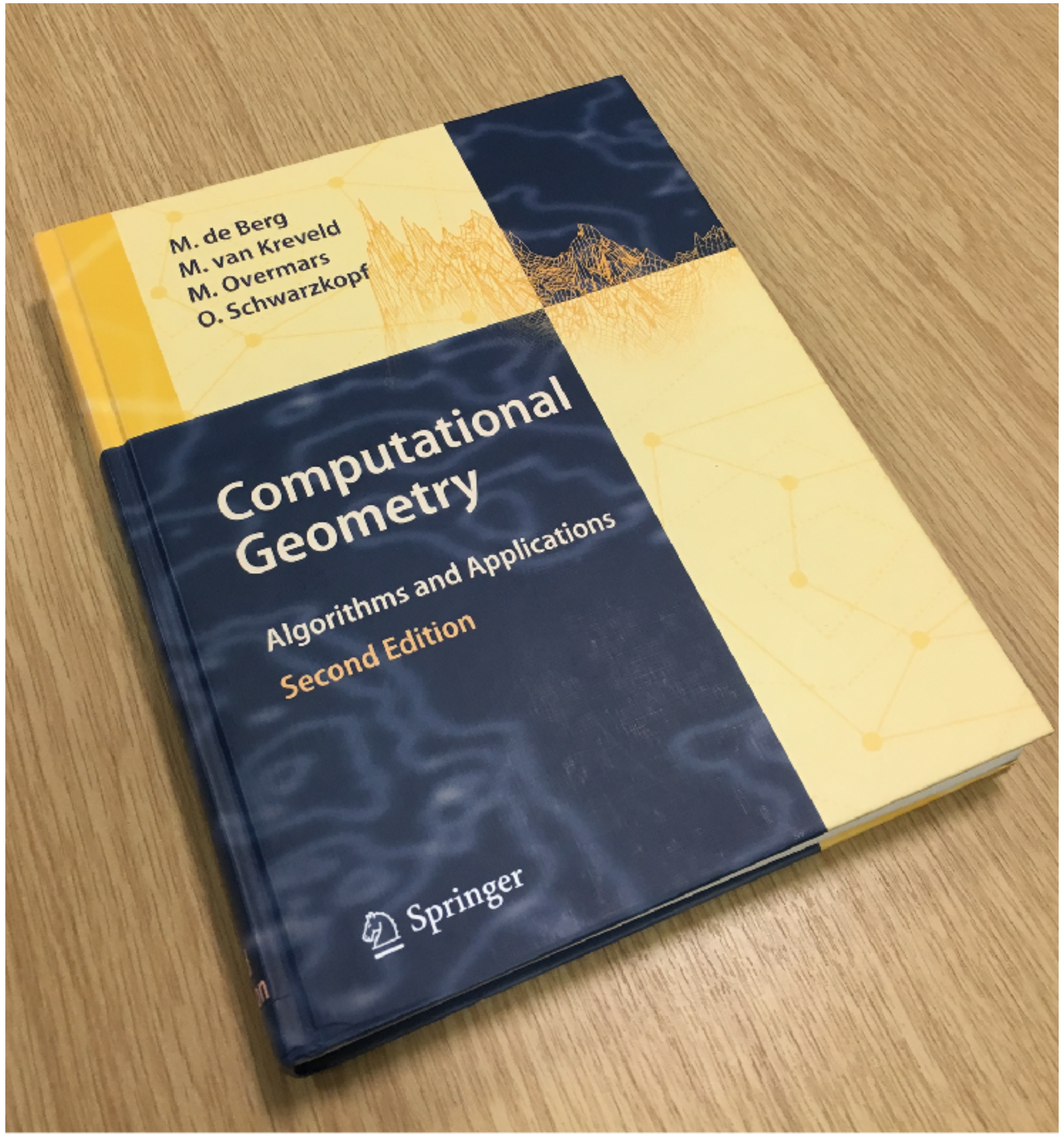


Software Engineering

Algorithmic Competitions at UCL (2016-2018)

- Targeting *2nd year* undergrads, team work
- One week-long
- Should involve *math* and *programming*
- *Challenging* for students, but *easy to assess*

Algorithmic Competitions at UCL (2016-2018)



1 Computational Geometry 1
Introduction

1.1 An Example: Convex Hulls
1.2 Degeneracies and Robustness
1.3 Application Domains
1.4 Notes and Comments
1.5 Exercises

2 Line Segment Intersection
Thematic Map Overlay

2.1 Line Segment Intersection
2.2 The Doubly-Connected Edge List
2.3 Computing the Overlay of Two Subdivisions
2.4 Boolean Operations
2.5 Notes and Comments
2.6 Exercises

3 Polygon Triangulation
Guarding an Art Gallery

3.1 Guarding and Triangulations
3.2 Partitioning a Polygon into Monotone Pieces
3.3 Triangulating a Monotone Polygon
3.4 Notes and Comments
3.5 Exercises

4 Linear Programming
Manufacturing with Molds

4.1 The Geometry of Casting
4.2 Half-Plane Intersection
4.3 Incremental Linear Programming
4.4 Randomized Linear Programming

5 Orthogonal Range Searching 95
Querying a Database

5.1 1-Dimensional Range Searching 96
5.2 Kd-Trees 99
5.3 Range Trees 105
5.4 Higher-Dimensional Range Trees 109
5.5 General Sets of Points 111
5.6* Fractional Cascading 112
5.7 Notes and Comments 115
5.8 Exercises 117

6 Point Location 121
Knowing Where You Are

6.1 Point Location and Trapezoidal Maps 122
6.2 A Randomized Incremental Algorithm 128
6.3 Dealing with Degenerate Cases 137
6.4* A Tail Estimate 140
6.5 Notes and Comments 143
6.6 Exercises 144

7 Voronoi Diagrams 147
The Post Office Problem

7.1 Definition and Basic Properties 148
7.2 Computing the Voronoi Diagram 151
7.3 Notes and Comments 160
7.4 Exercises 162

8 Arrangements and Duality 165
Supersampling in Ray Tracing

8.1 Computing the Discrepancy 167
8.2 Duality 169
8.3 Arrangements of Lines 172
8.4 Levels and Discrepancy 177
8.5 Notes and Comments 178
8.6 Exercises 180

9 Delaunay Triangulations 183
Height Interpolation

9.1 Triangulations of Planar Point Sets 185
9.2 The Delaunay Triangulation 188
9.3 Computing the Delaunay Triangulation 191
9.4 The Analysis 197
9.5* A Framework for Randomized Algorithms 200
9.6 Notes and Comments 206
9.7 Exercises 207

10 More Geometric Data Structures 211
Windowing

10.1 Interval Trees 212
10.2 Priority Search Trees 218
10.3 Segment Trees 223
10.4 Notes and Comments 229
10.5 Exercises 230

11 Convex Hulls 235
Mixing Things

11.1 The Complexity of Convex Hulls in 3-Space 236
11.2 Computing Convex Hulls in 3-Space 238
11.3* The Analysis 242
11.4* Convex Hulls and Half-Space Intersection 245
11.5* Voronoi Diagrams Revisited 247
11.6 Notes and Comments 248
11.7 Exercises 249

12 Binary Space Partitions 251
The Painter's Algorithm

12.1 The Definition of BSP Trees 253
12.2 BSP Trees and the Painter's Algorithm 255
12.3 Constructing a BSP Tree 256
12.4* The Size of BSP Trees in 3-Space 260
12.5 Notes and Comments 263
12.6 Exercises 264

13 Robot Motion Planning 267
Getting Where You Want to Be

13.1 Work Space and Configuration Space 268

The Competitions

- 2016: Art Gallery Competition
- 2017: Move-and-Tag Competition
- 2018: Room Furnishing

Art Gallery Competition

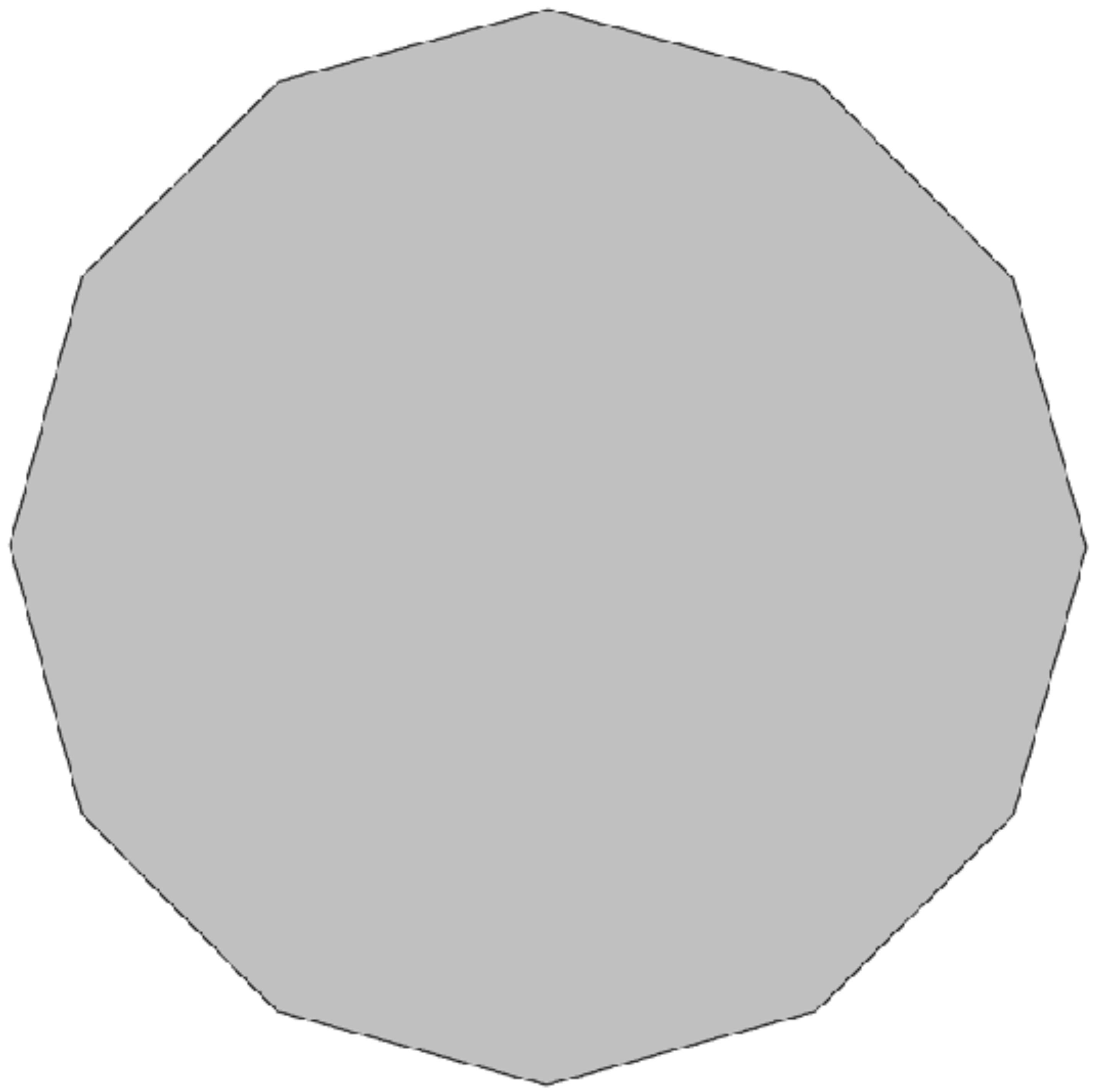
based on:

Chvátal's *Art Gallery Problem* (1975)


GUARD
33

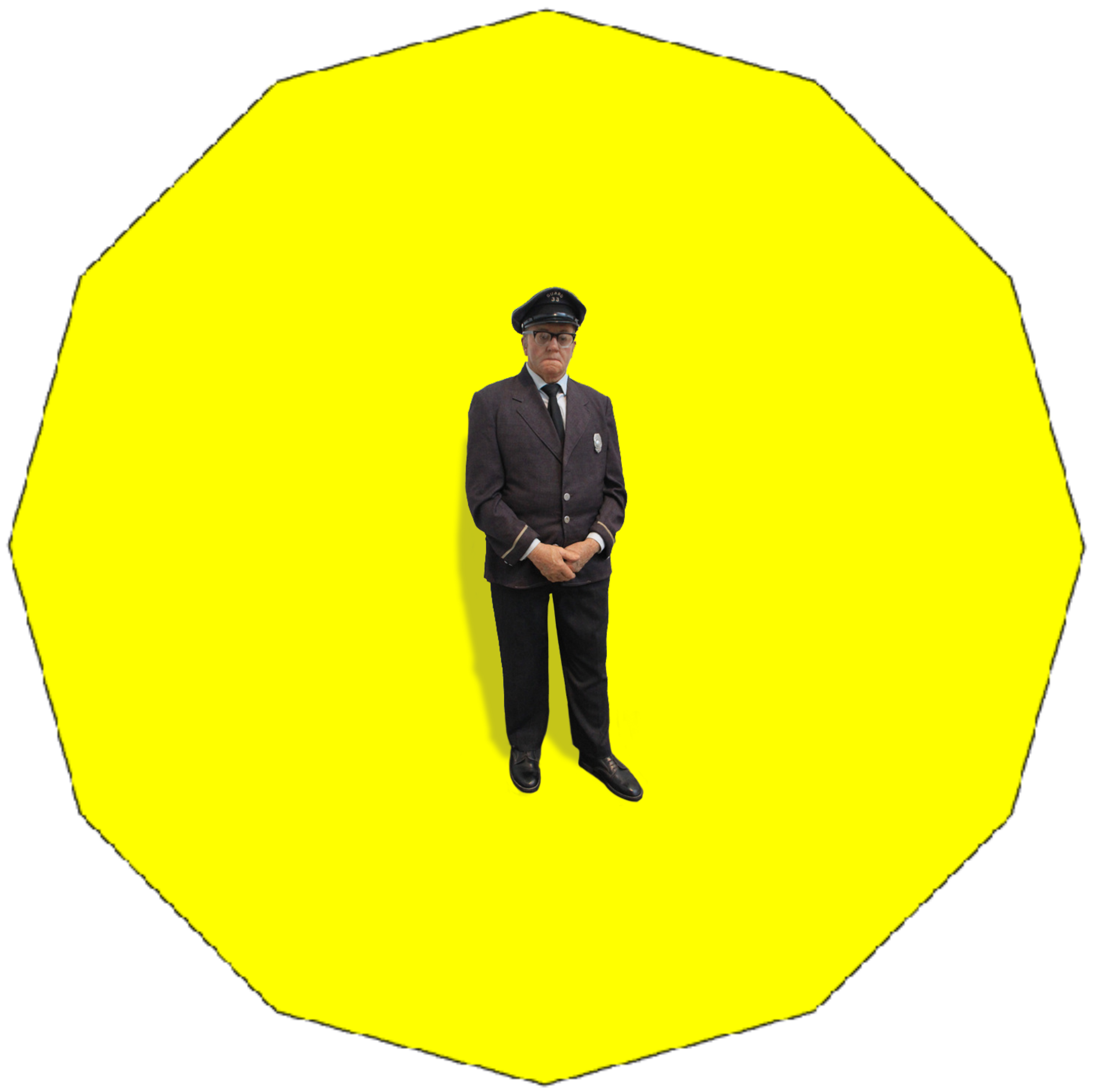
How many guards do we really need?

The answer depends on the shape of the gallery.

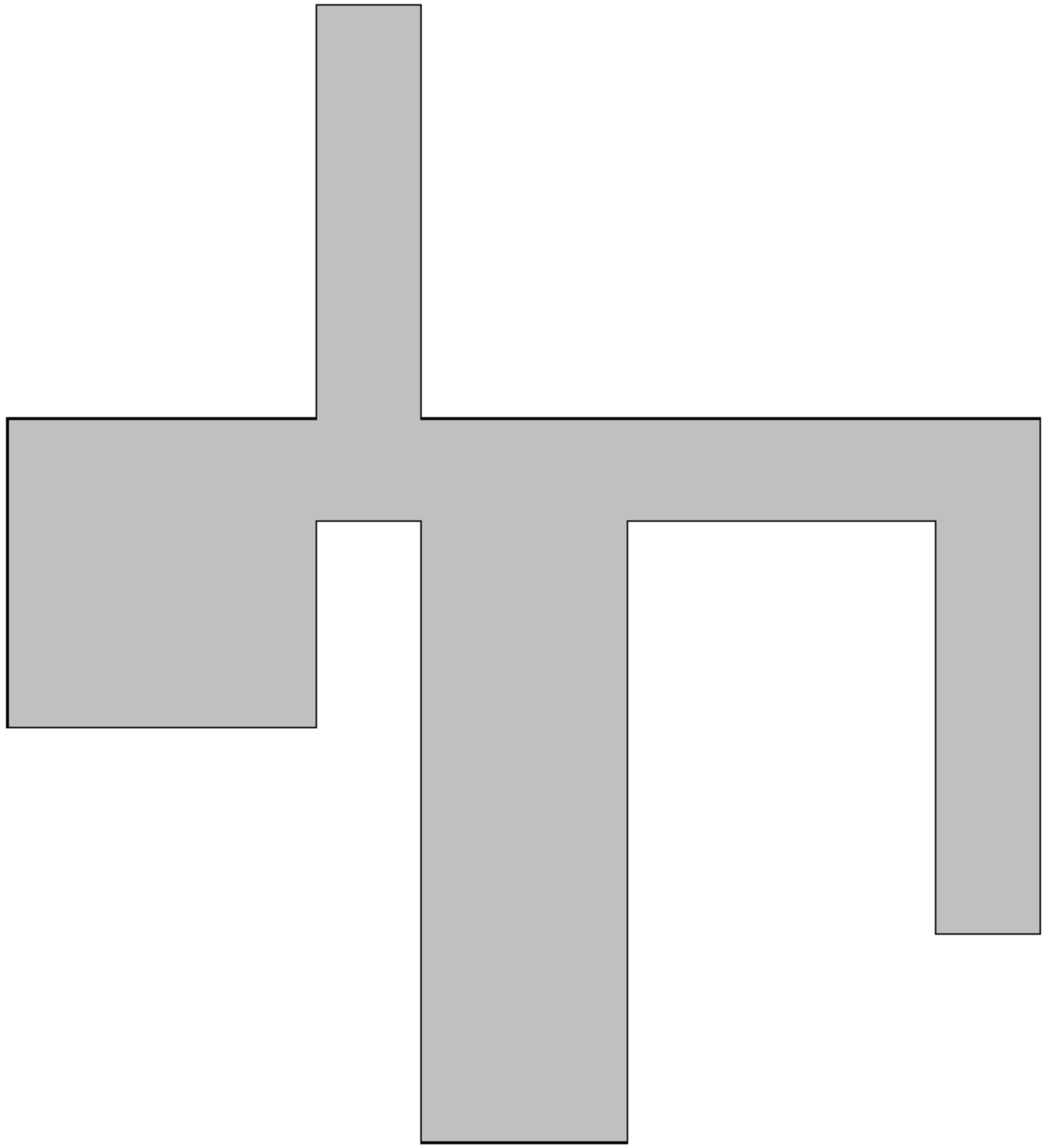


How many guards do we really need?

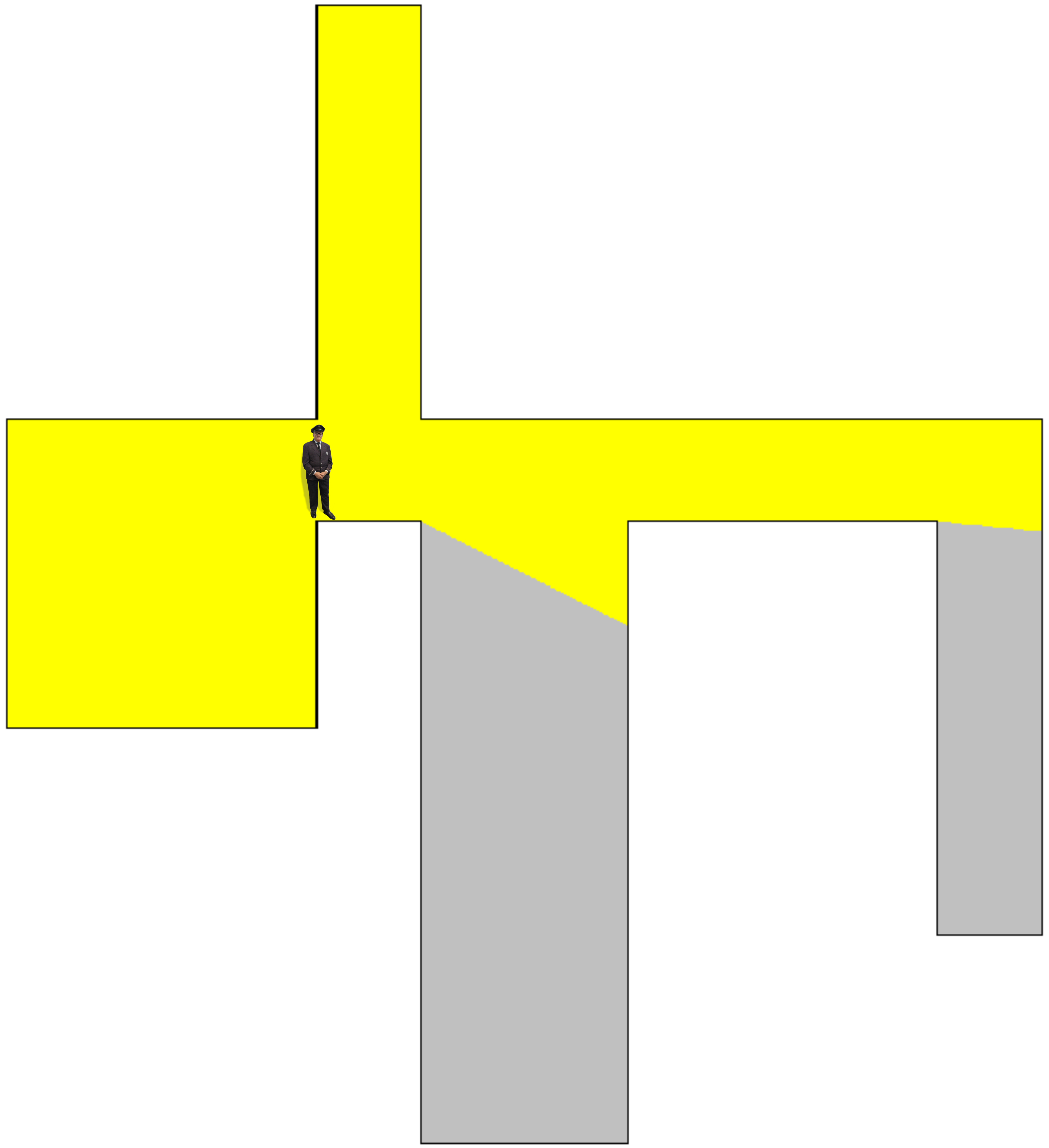
The answer depends on the shape of the gallery.



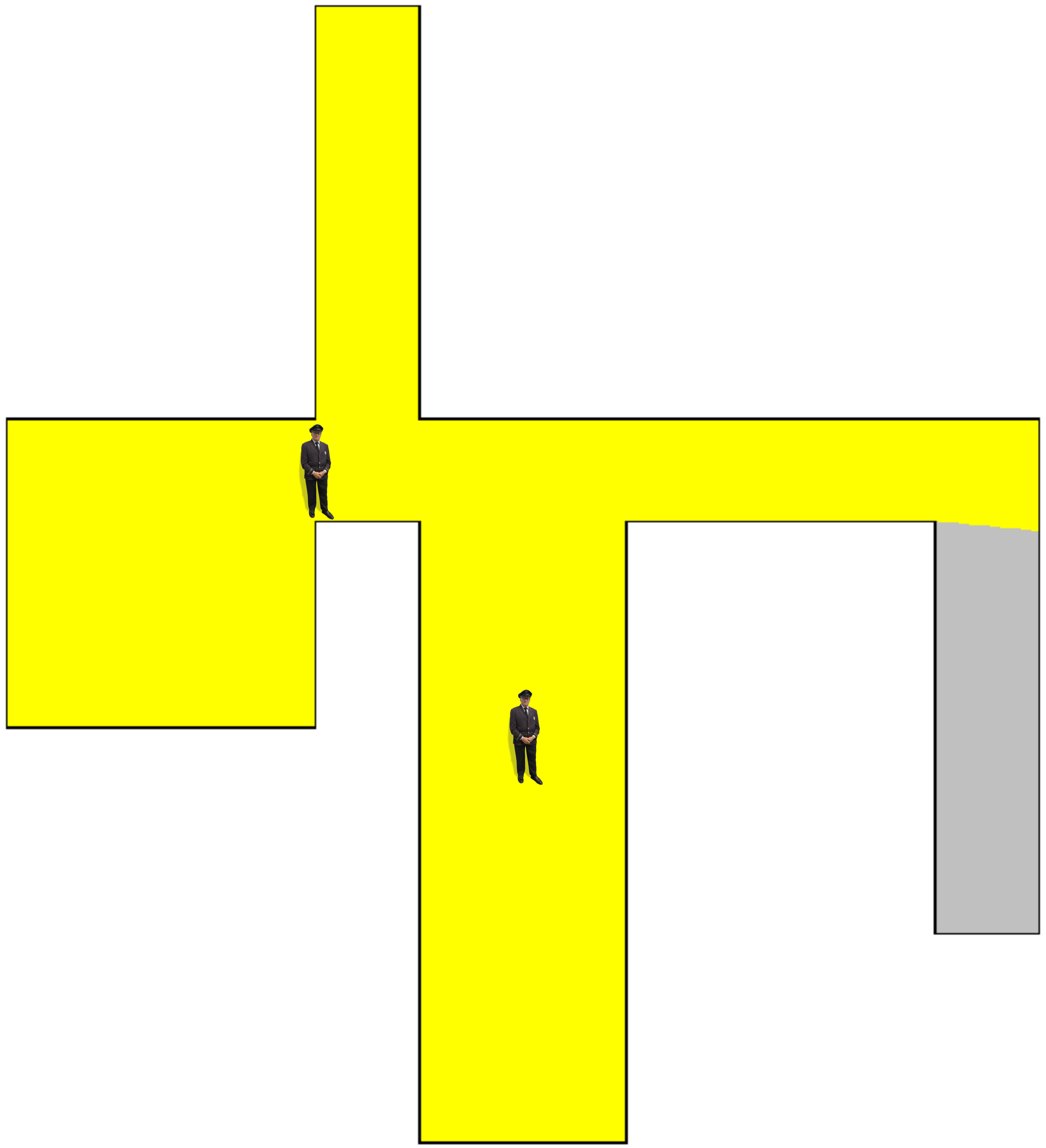
How many guards do we really need?



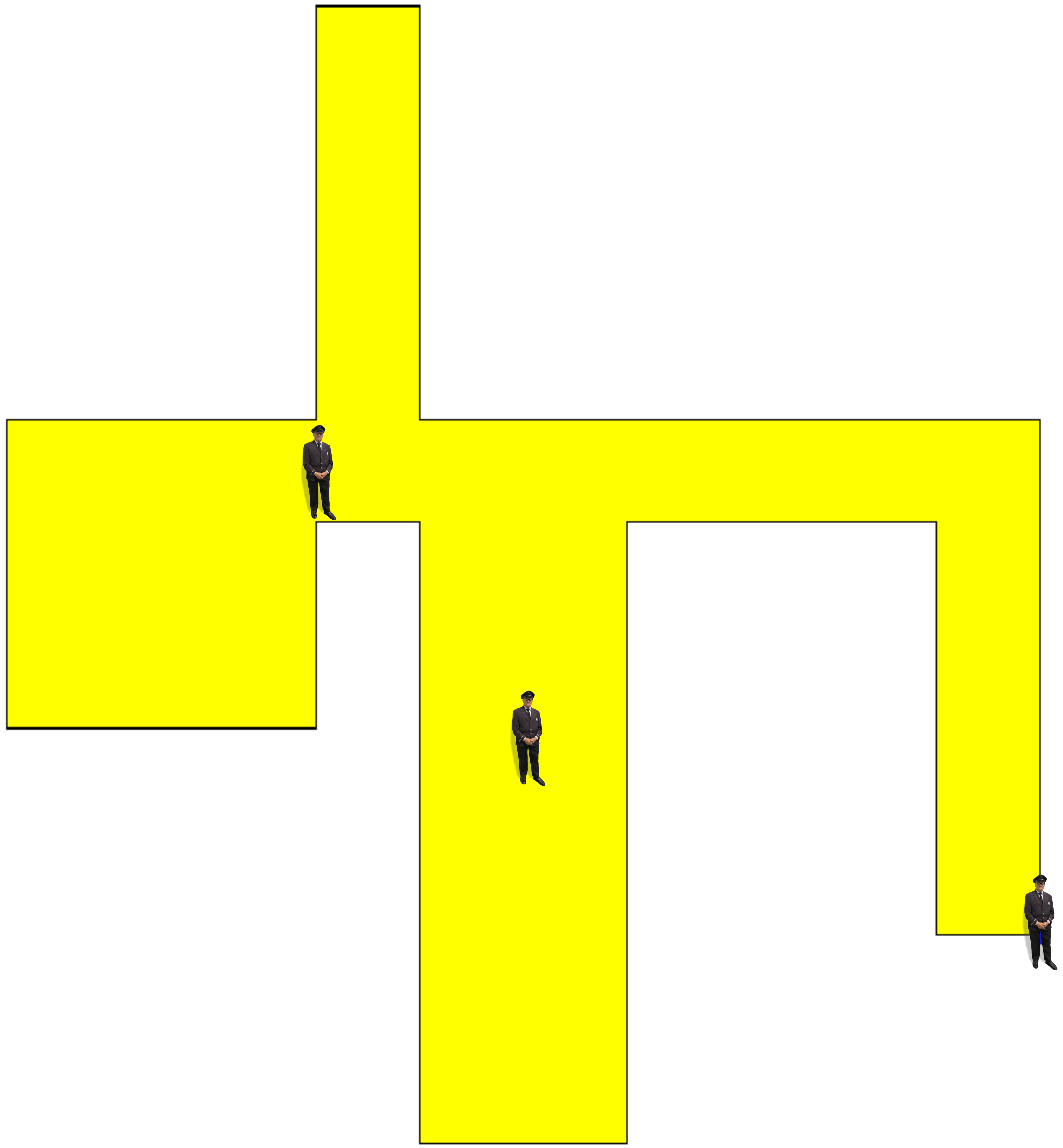
How many guards do we really need?



How many guards do we really need?



How many guards do we really need?

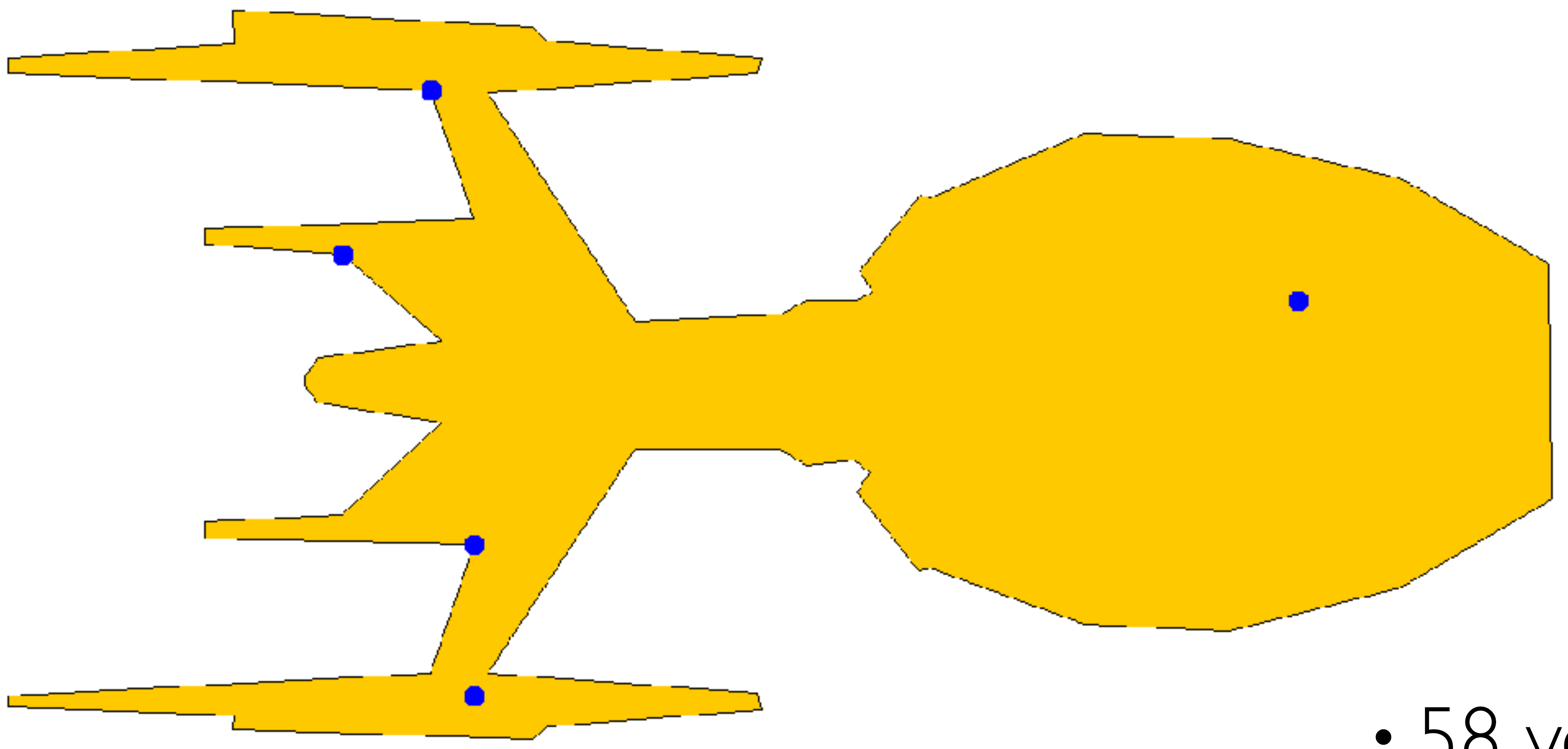


Art Gallery Problem

For a given gallery (polygon),
find the *minimal* set of guards' positions,
so together the guards can "see" the *whole* interior.

Project: *Art Gallery Competition*

Find the *best* solutions for a collection of *large* polygons.



- 58 vertices
- 5 guards

Making it Fun

- Problem generator;
  - Polygons with different *"features"* (convex, rectangular, *etc*.)
- Solution checker with online feedback
  - geometric machinery (triangulation, visibility, …)
  - web-server
- Make sure that it all works.

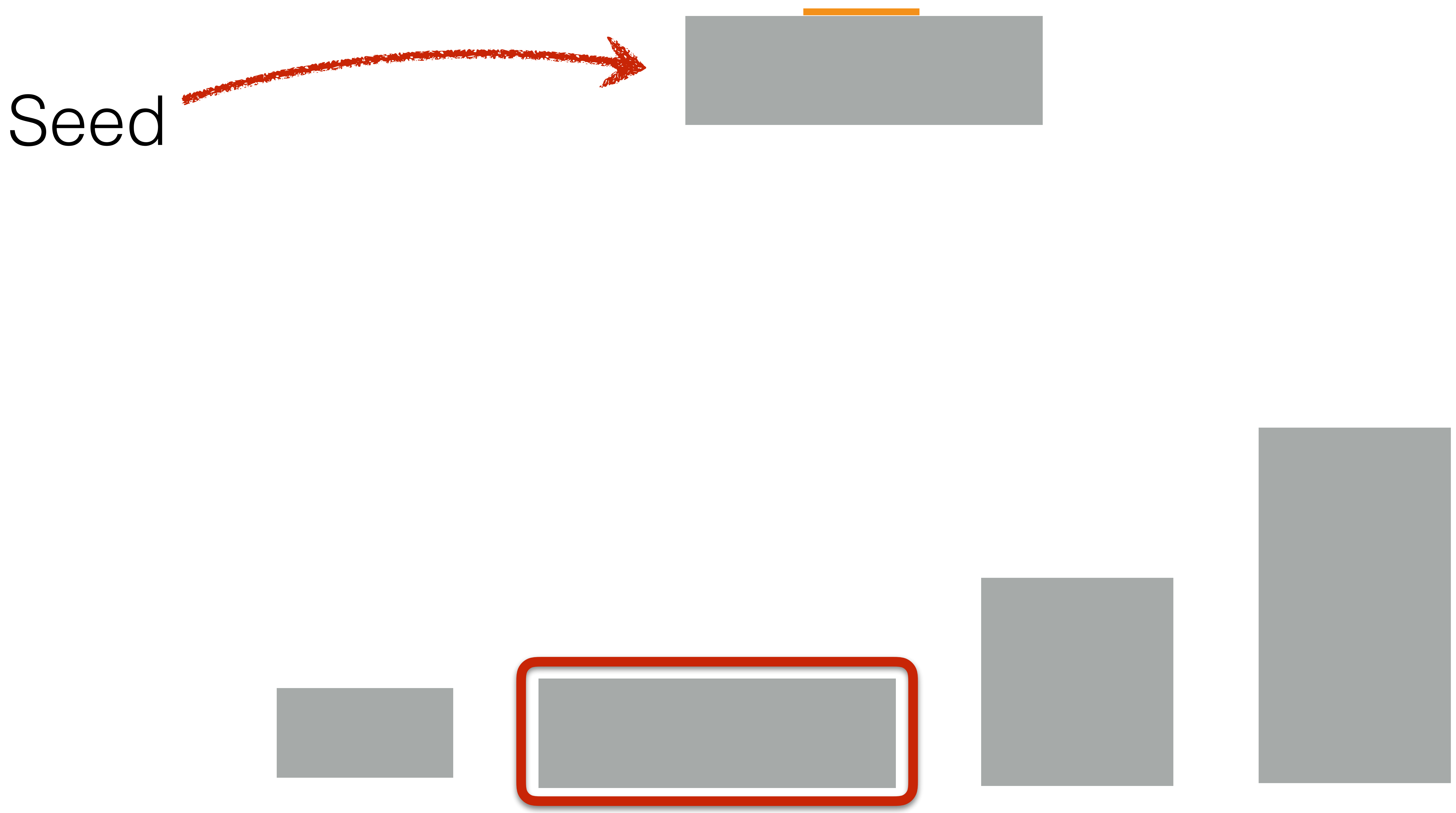
Making it Fun

- Problem generator;
  - Polygons with different *"features"* (convex, rectangular, *etc*.)
- Solution checker with online feedback
  - geometric machinery (triangulation, visibility, …)
  - web-server
- Make sure that it all works.

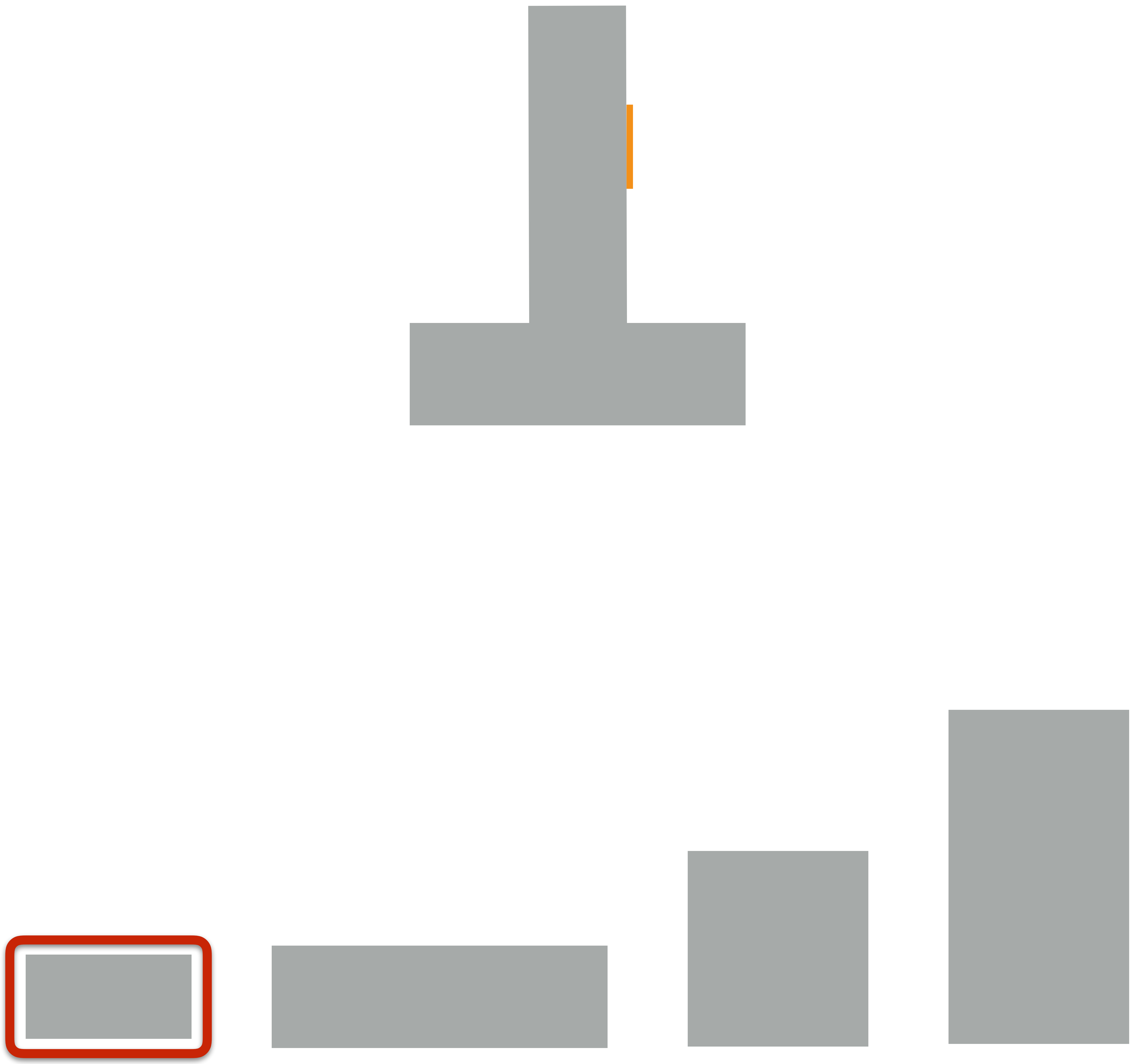
Growing polygons

Primitive polygons with specific “features”

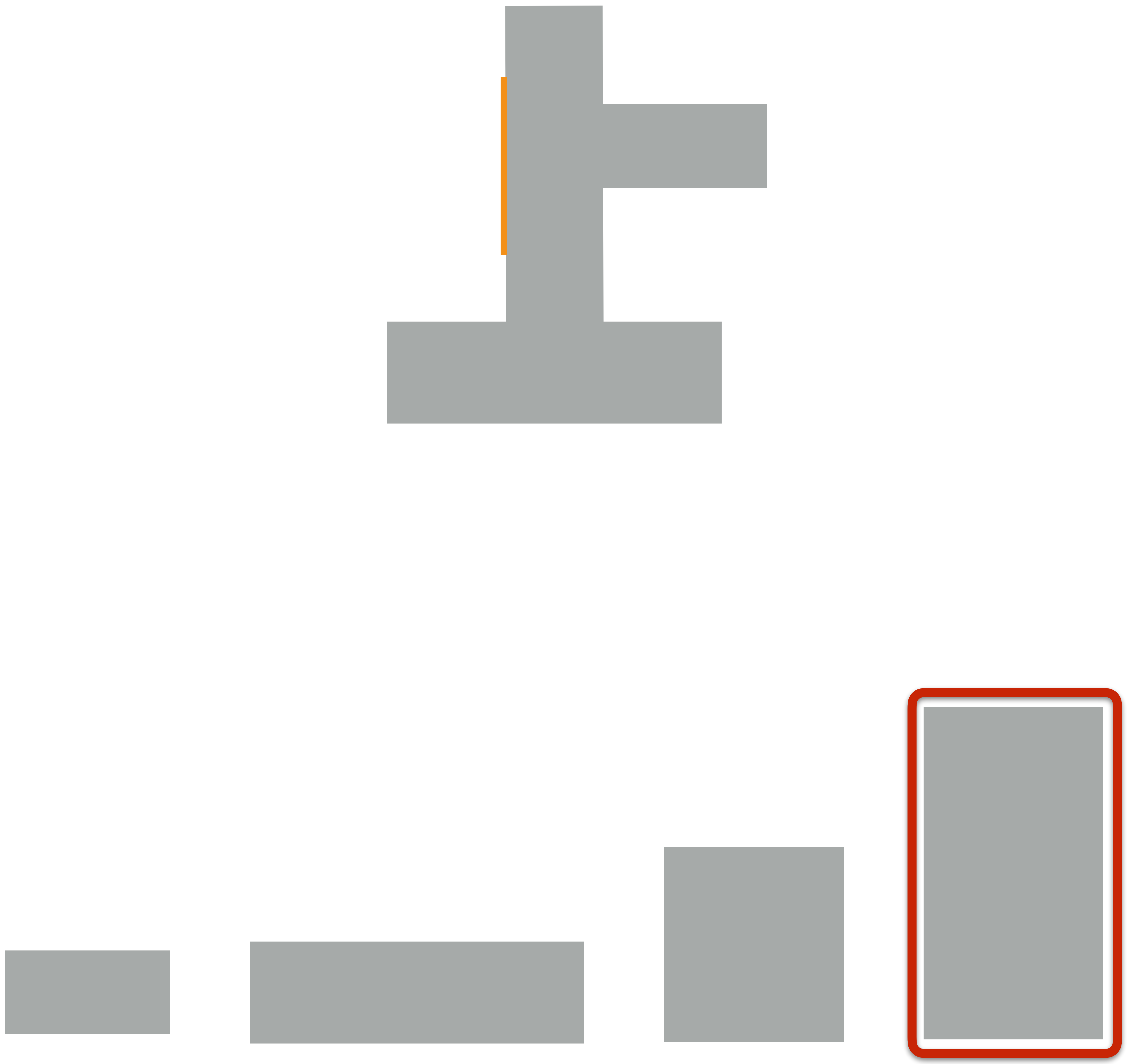
Growing polygons



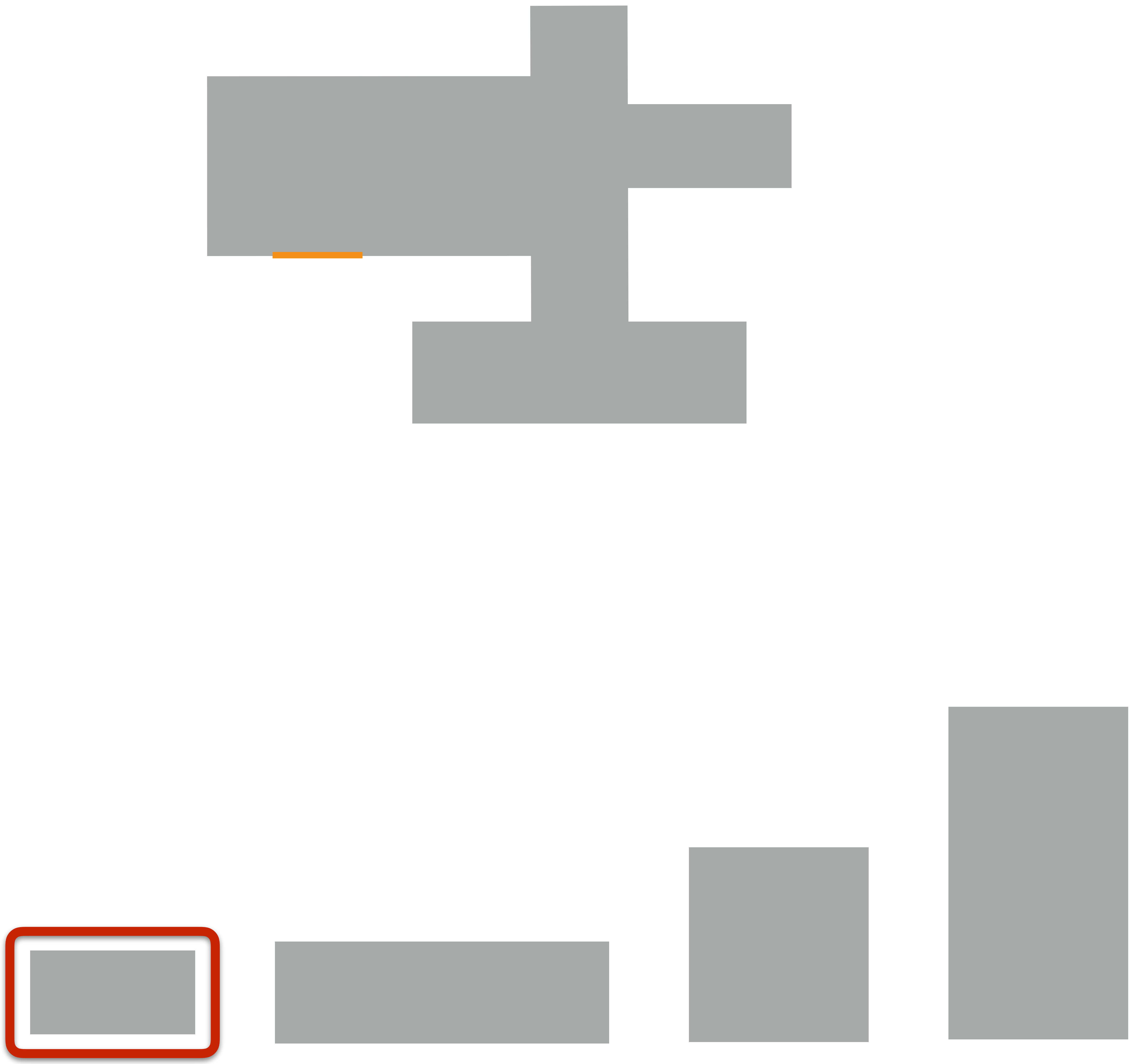
Growing polygons



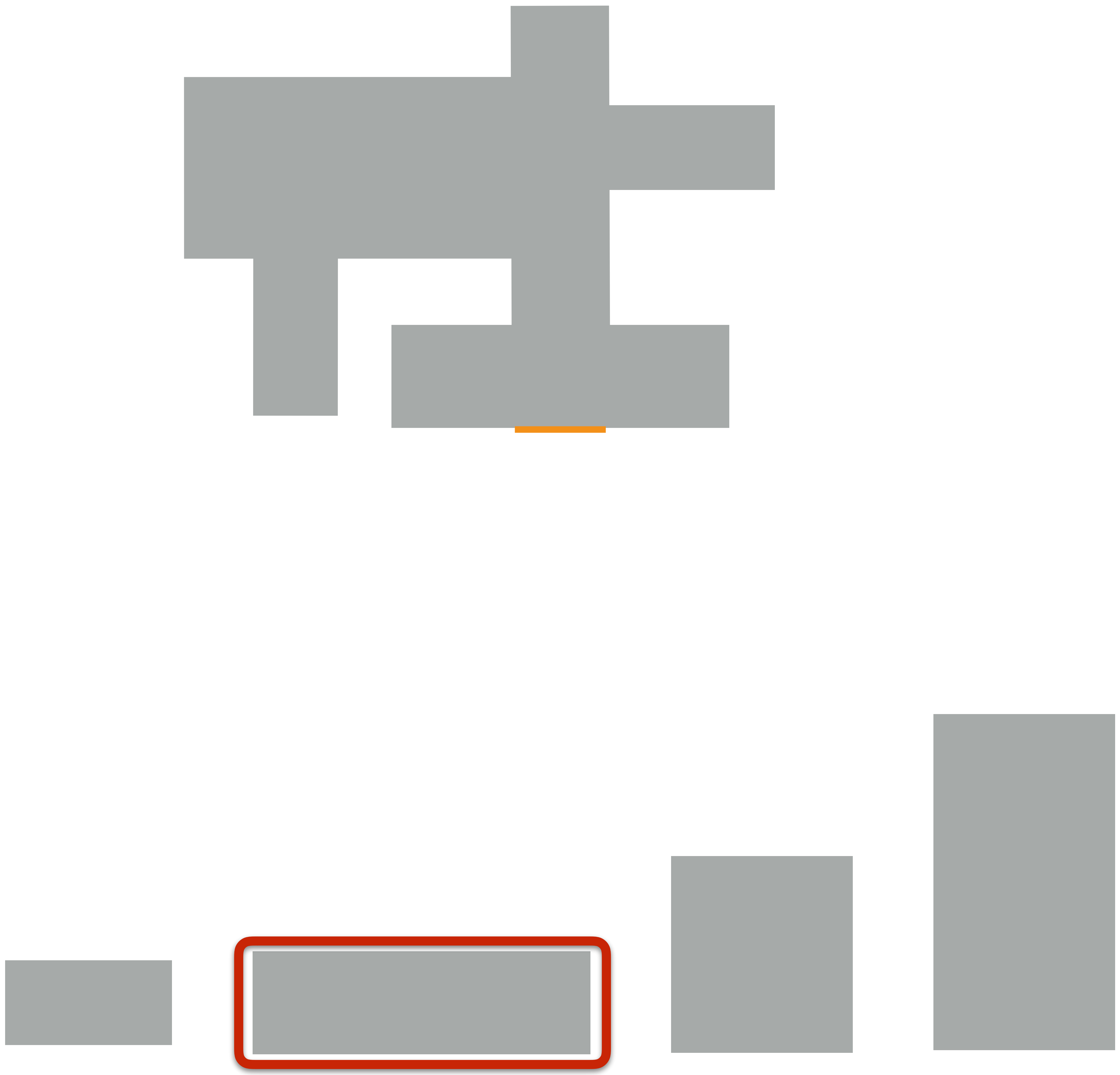
Growing polygons



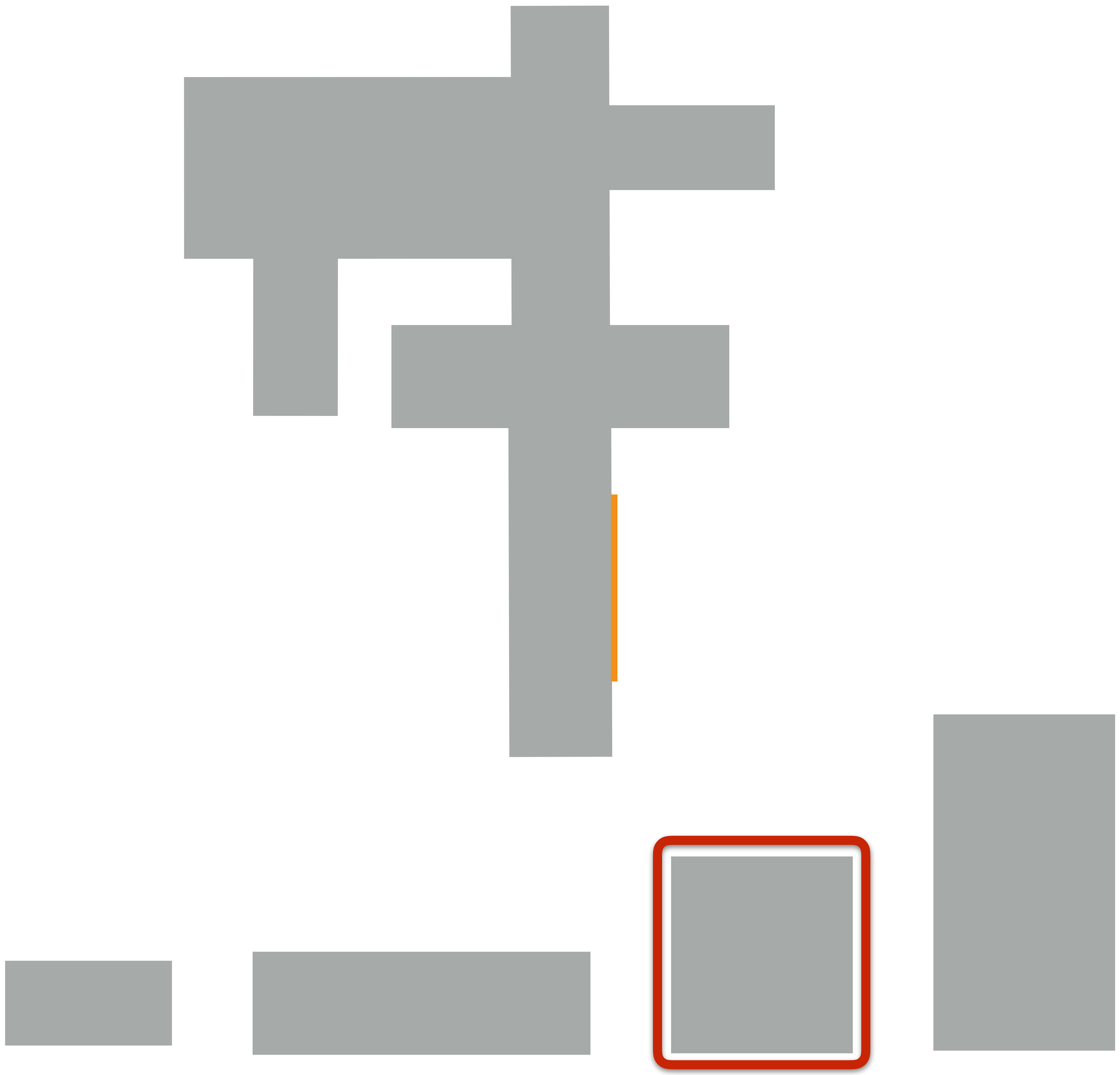
Growing polygons



Growing polygons



Growing polygons



Growing polygons

Can we *enumerate* “primitive” polygons and plug *arbitrary shapes generators*?

The Essence of Functional Languages

- Higher-order functions and closures
- Types and Type Inference
- Polymorphism
- Laziness
- Point-free style
- Combinator Libraries
- Algebraic Data Types
- Purely functional data structures
- Pattern Matching
- Folds
- Continuations and CPS
- Structural Recursion
- Type Classes
- Monads

The Essence of Functional Languages

- Higher-order functions and closures
- Types and Type Inference
- Polymorph
- Laziness
- Point-free style
- Combinator Libraries
- Algebraic Data Types

Enumeration and Extensibility

- Purely functional data structures
- Pattern Matching
- Folds
- Continuations and CPS
- Structural Recursion
- Type Classes
- Monads

QuickCheck Automatic Specification-Based Testing

Koen Claessen and John Hughes

QuickCheck is a tool for testing Haskell programs automatically. The programmer provides a *specification* of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

Resources

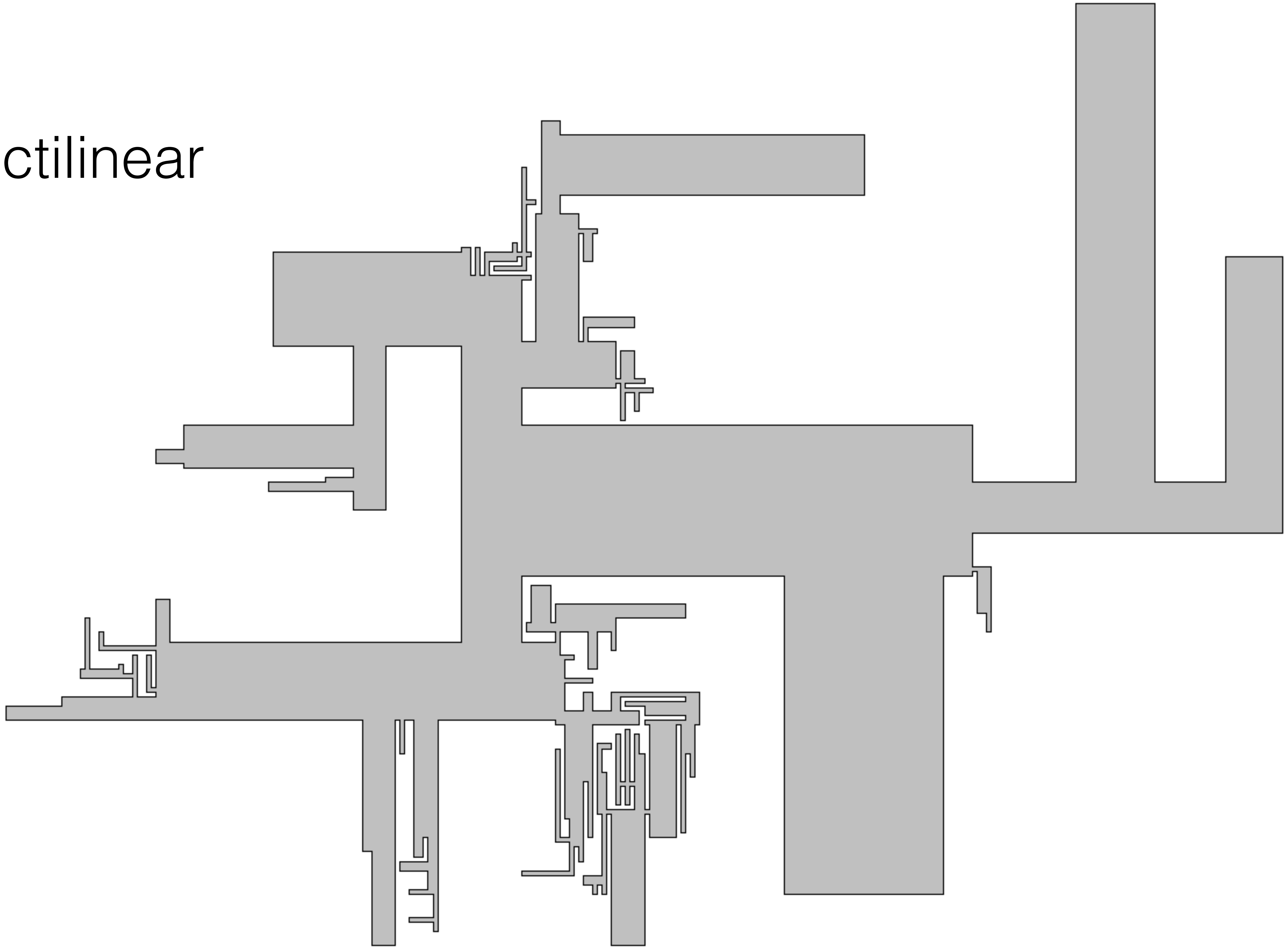
- Our paper from ICFP 2000.
- A new paper (presented at the Haskell Workshop 2002) on testing monadic programs, especially in the ST monad.

“Polygon Combinator”

```scala
trait PolygonGenerator extends GeneratorPrimitives {

  val seeds          : List[Polygon]
  val primitives     : List[(Int) => Polygon]
  val locate         : Double => Option[(Double, Double)]

  val seedFreqs      : List[Int]
  val primFreqs      : List[Int]

  val generations    : Int

  ...
}
```

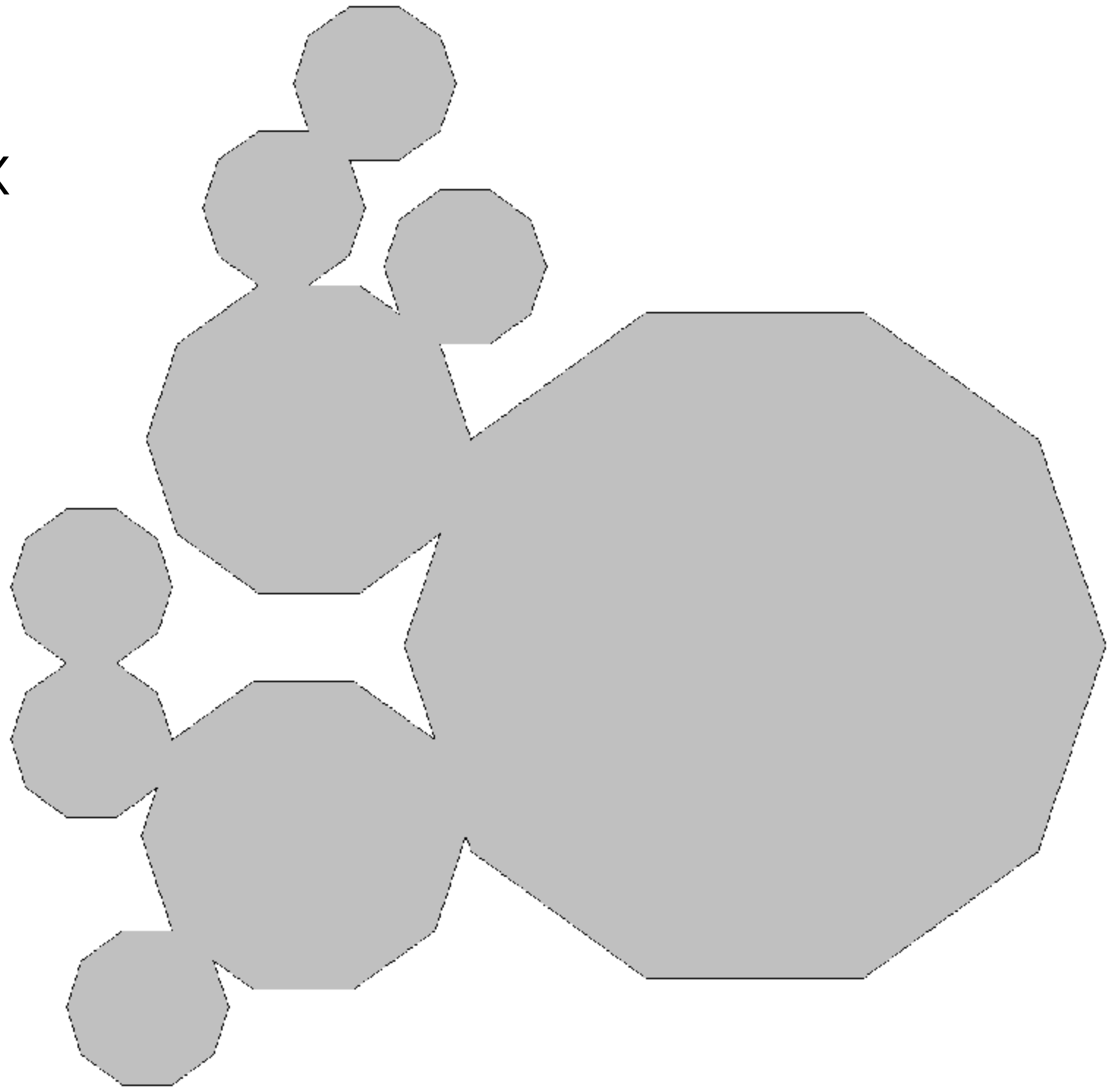
Generating random polygons

Rectilinear



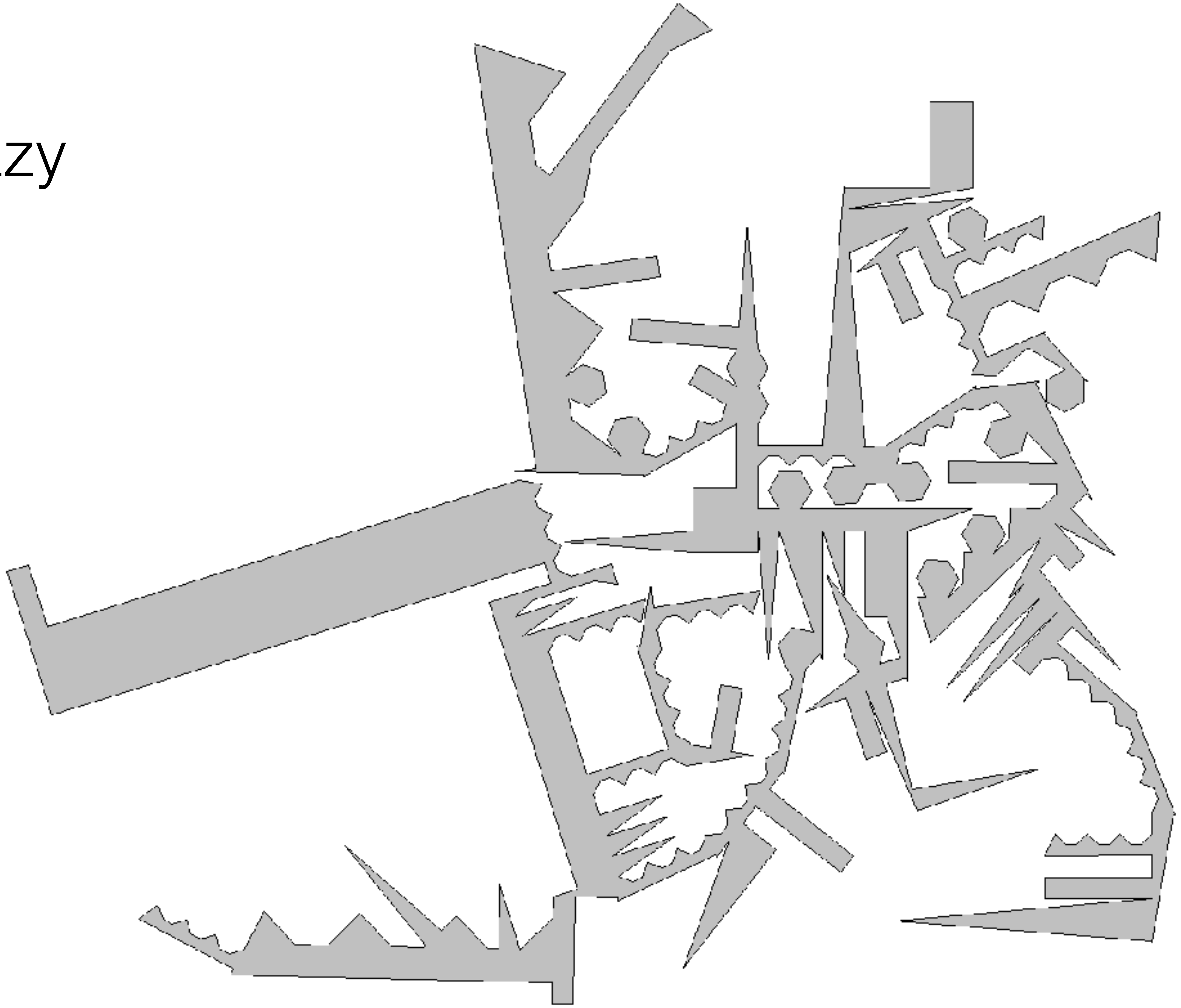
Generating random polygons

Quasi-convex



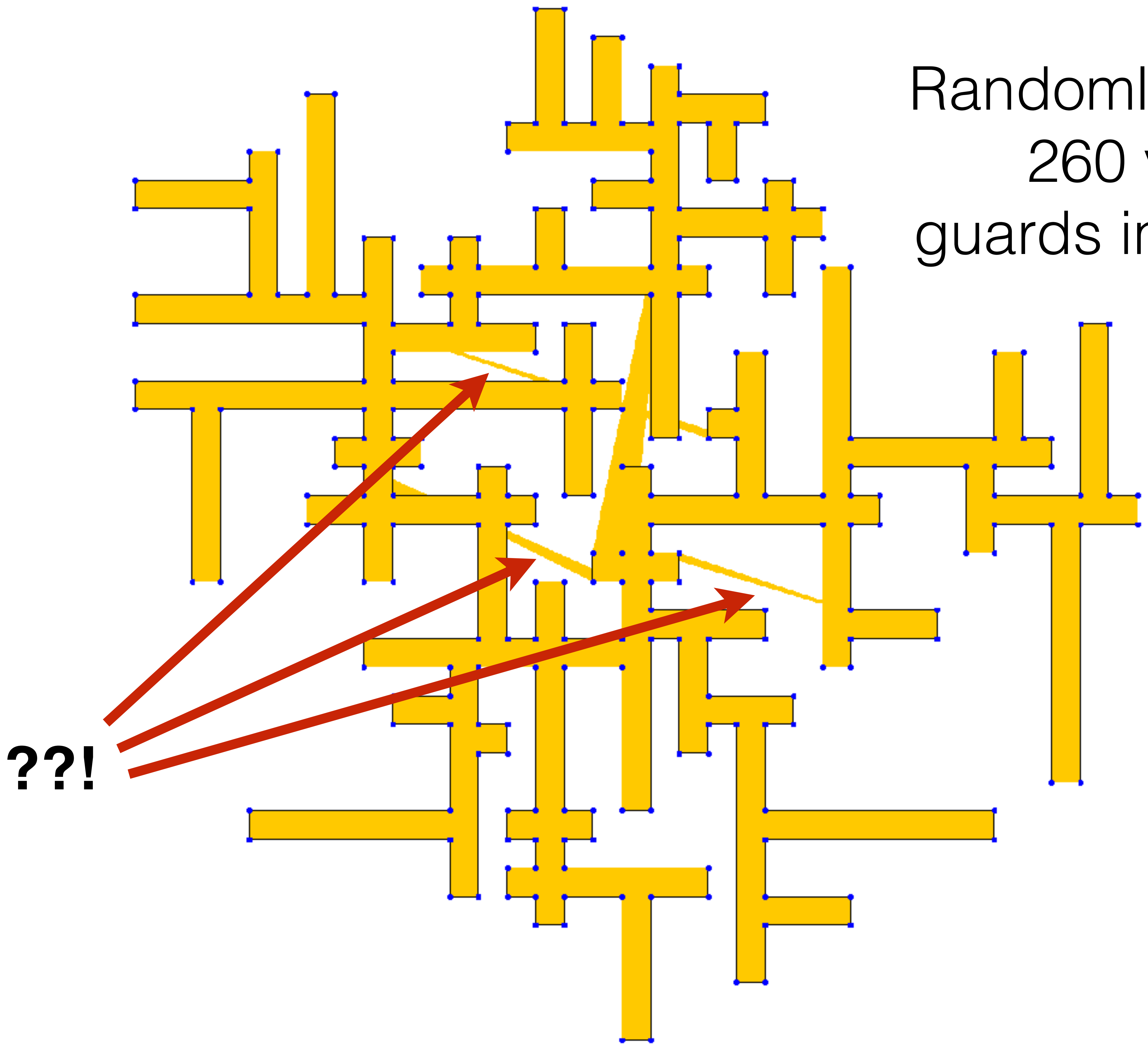
Generating random polygons

Crazy



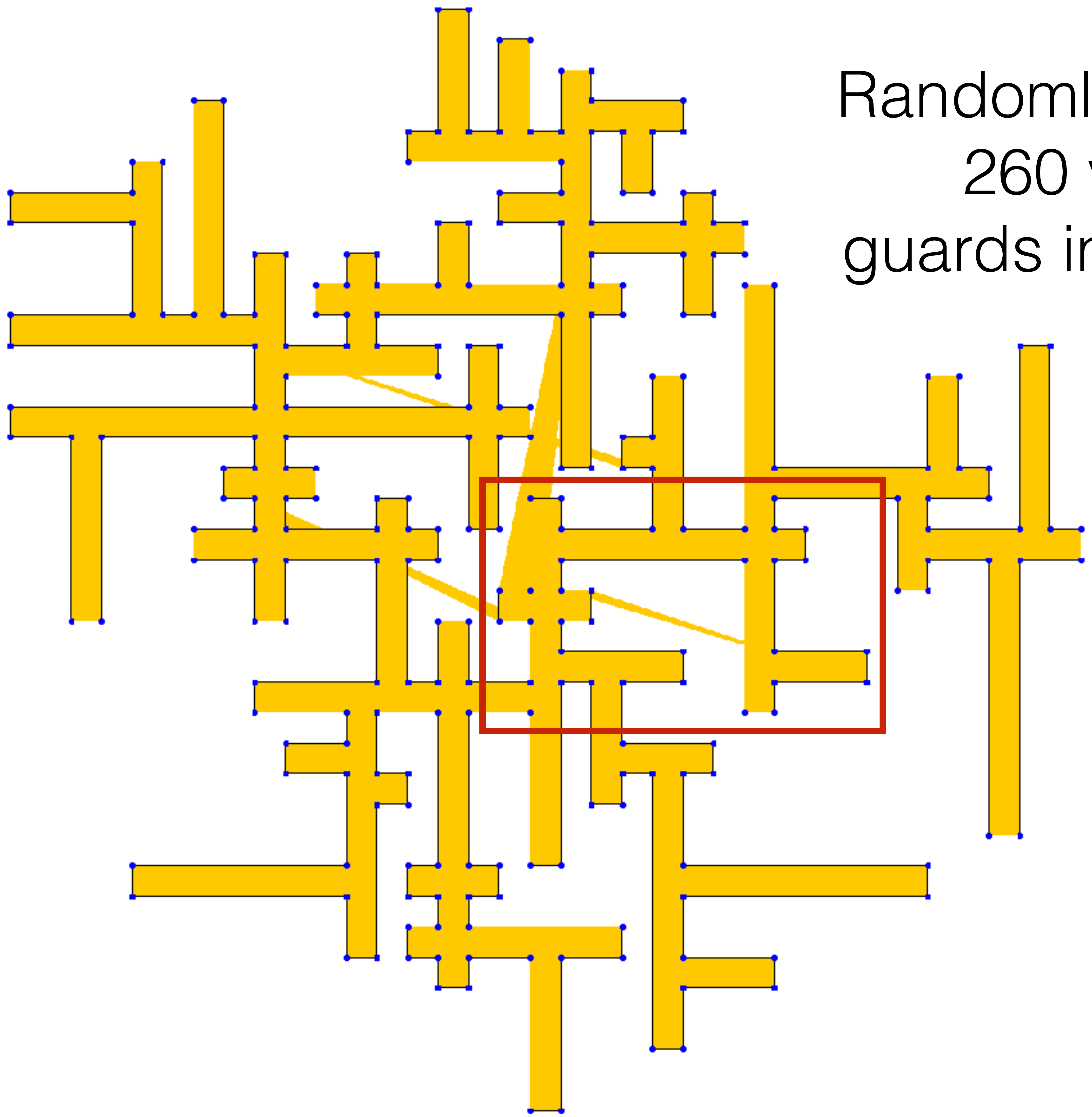
Can we *Quick-Check* geometric algorithms?

Bug in textbook visibility algorithm



Randomly generated,
260 vertices,
guards in every node

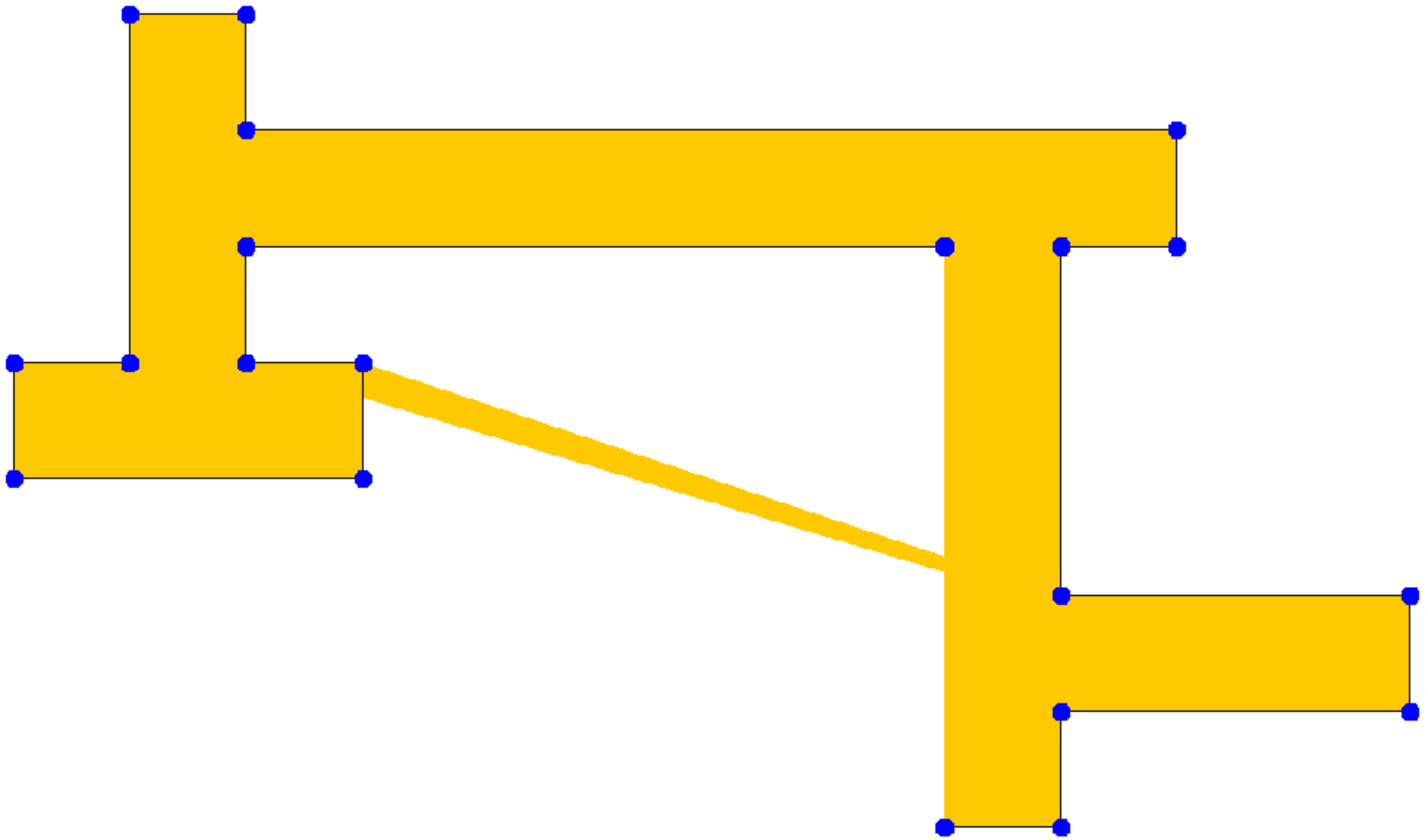
Bug in textbook visibility algorithm



Randomly generated,
260 vertices,
guards in every node

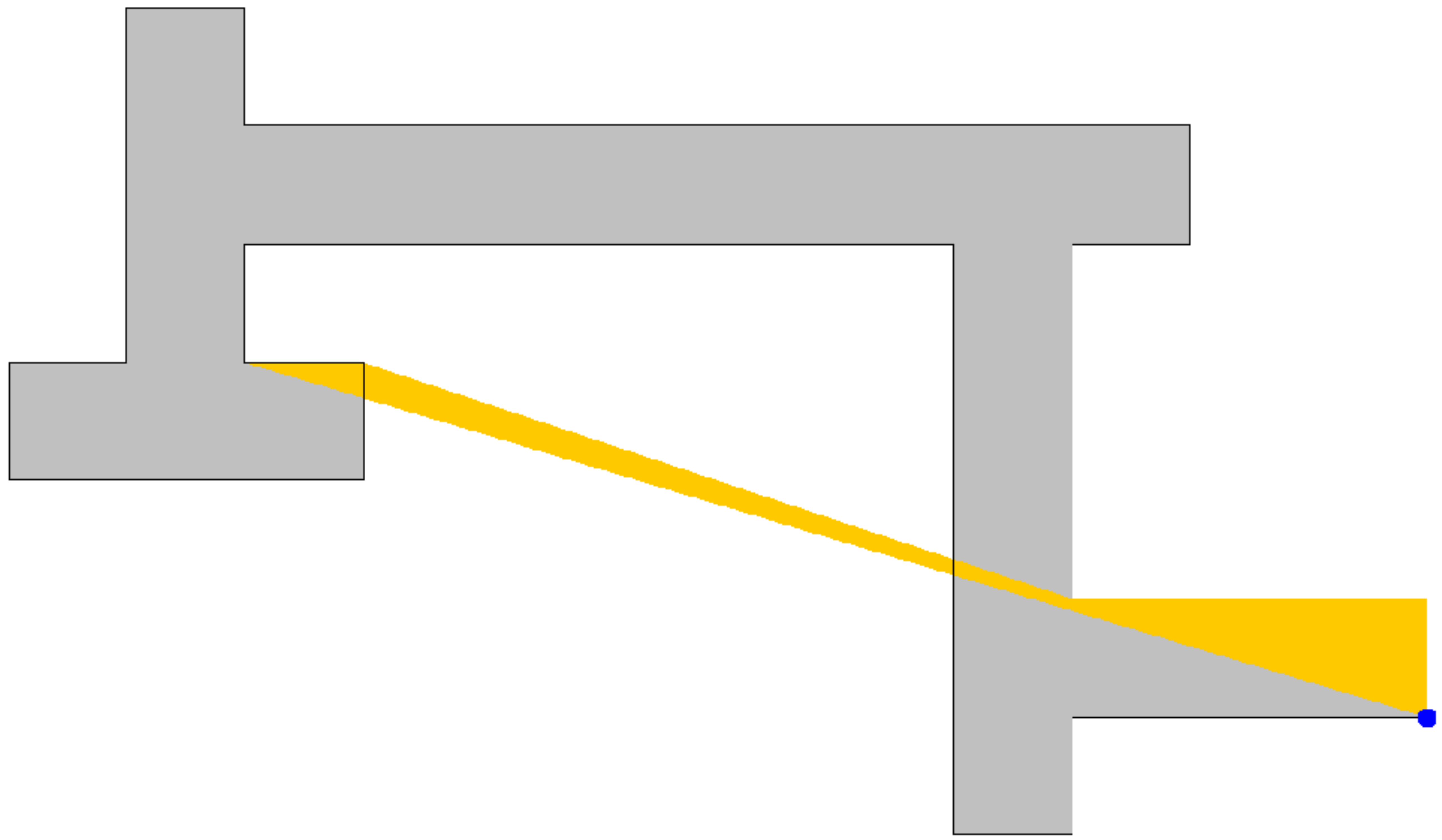
Bug in textbook visibility algorithm

After shrinking:
20 vertices



Bug in textbook visibility algorithm

Removed irrelevant light sources



Bug in textbook visibility algorithm

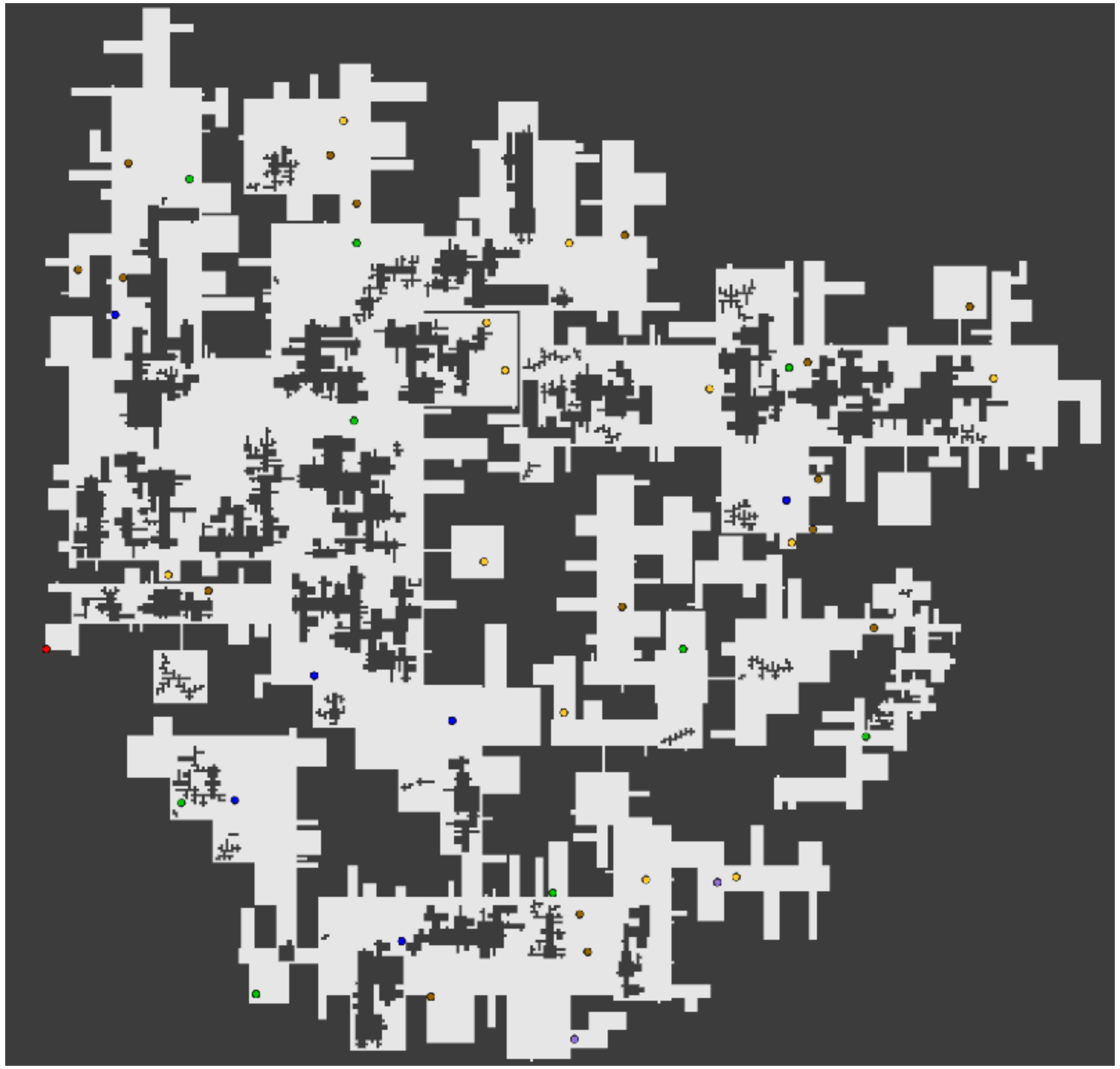
Removed irrelevant light sources

Experience Report: Growing and Shrinking Polygons for Random Testing of Computational Geometry Algorithms

Ilya Sergey
University College London, UK
i.sergey@ucl.ac.uk

ICFP 2016

Beyond Classroom: ICFP Programming Contest 2019



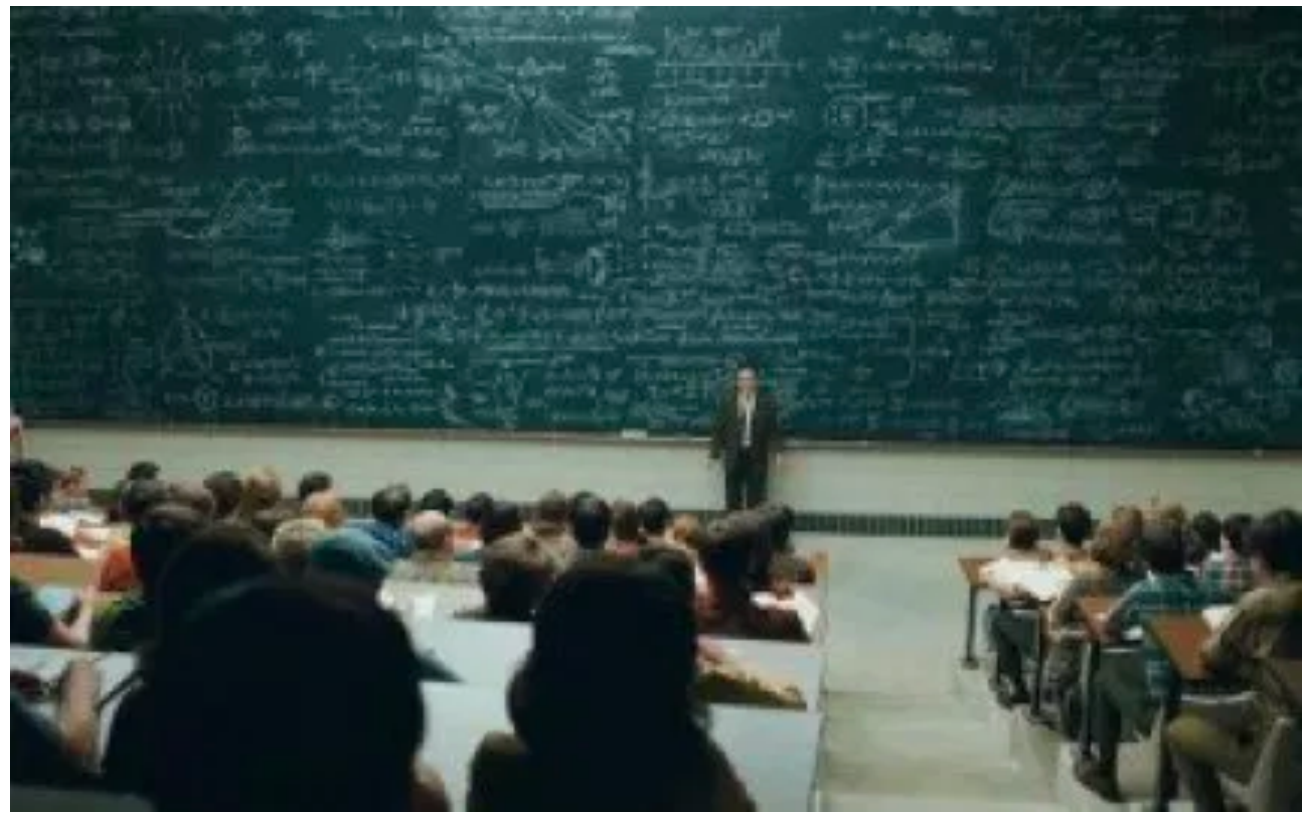
Contest Report
on Tuesday, 17:45

Functional Programming Ideas in…

Research

… for modularity and proof reuse

Teaching



… for creating fun assignments

Software Engineering

Functional Programming Ideas in…

Research

… for modularity and proof reuse

Teaching

… for creating fun assignments

Software Engineering

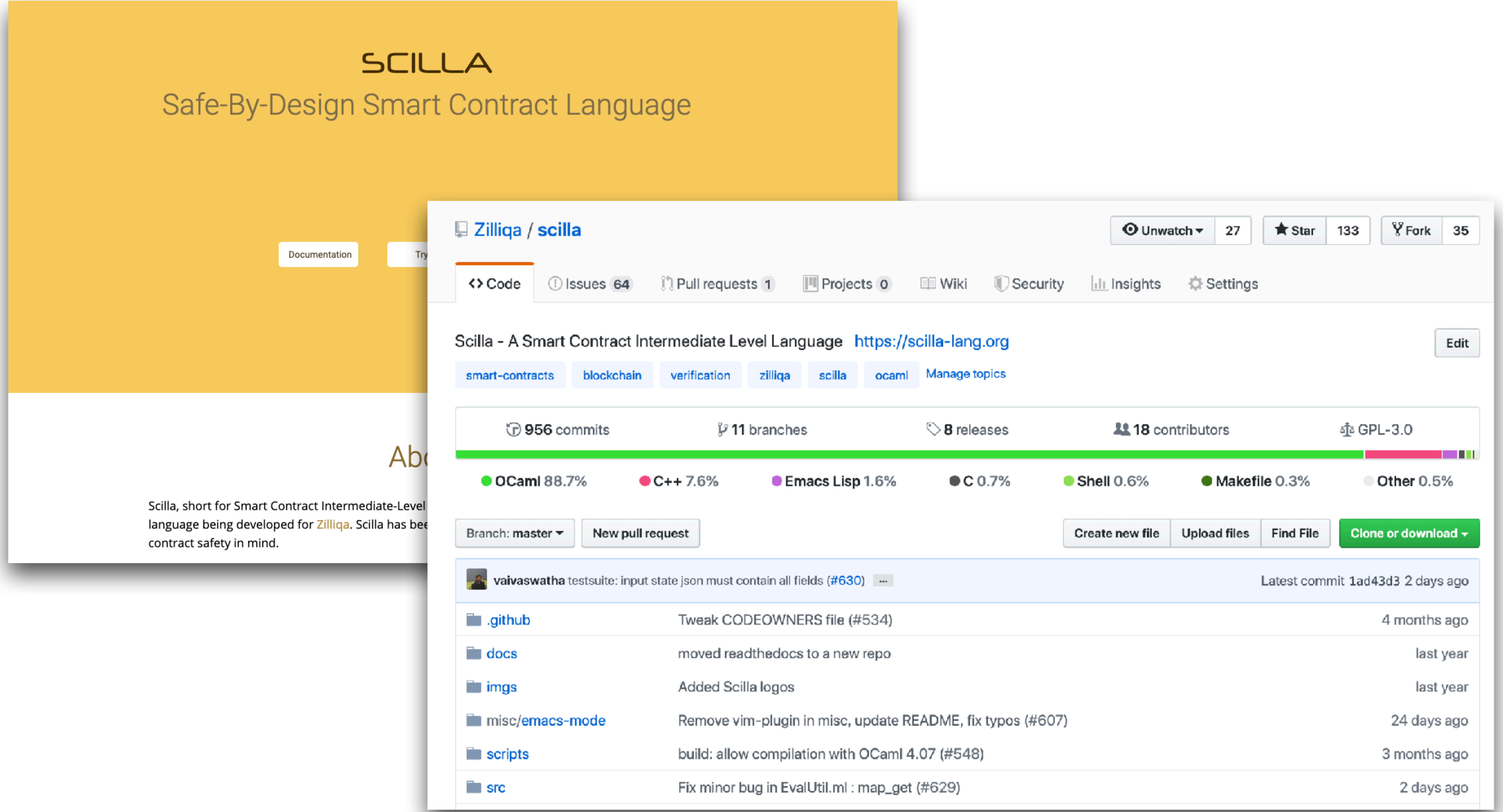
SCILLA

Safe-By-Design Smart Contract Language

Documentation   Try it!   GitHub

About

Scilla, short for Smart Contract Intermediate-Level Language, is an intermediate-level smart contract language being developed for Zilliqa. Scilla has been designed as a principled language with smart contract safety in mind.


SCILLA
Safe-By-Design Smart Contract Language
Documentation
Try
Abo
Scilla, short for Smart Contract Intermediate-Level
language being developed for Zilliqa. Scilla has bee
contract safety in mind.
Zilliqa / scilla
Unwatch 27
Star 133
Fork 35
Code
Issues 64
Pull requests 1
Projects 0
Wiki
Security
Insights
Settings
Scilla - A Smart Contract Intermediate Level Language https://scilla-lang.org
Edit
smart-contracts
blockchain
verification
zilliqa
scilla
ocaml
Manage topics
956 commits
11 branches
8 releases
18 contributors
GPL-3.0
OCaml 88.7%
C++ 7.6%
Emacs Lisp 1.6%
C 0.7%
Shell 0.6%
Makefile 0.3%
Other 0.5%
Branch: master
New pull request
Create new file
Upload files
Find File
Clone or download
vaivaswatha testsuite: input state json must contain all fields (#630)
Latest commit 1ad43d3 2 days ago
.github
Tweak CODEOWNERS file (#534)
4 months ago
docs
moved readthedocs to a new repo
last year
imgs
Added Scilla logos
last year
misc/emacs-mode
Remove vim-plugin in misc, update README, fix typos (#607)
24 days ago
scripts
build: allow compilation with OCaml 4.07 (#548)
3 months ago
src
Fix minor bug in EvalUtil.ml : map_get (#629)
2 days ago

Smart Contracts

- *Stateful mutable* objects replicated via a *consensus protocol*
- Use *valuable resource (gas)* to prevent “expensive” computations
- Yet, should be able to *handle arbitrarily large* data
- Can fail at any moment and roll-back (*transactional behaviour*)

Smart Contracts

- *Stateful mutable* objects replicated via a consensus protocol
- Use *valuable resource (gas)* to prevent “expensive” computations
- Yet, should be able to *handle arbitrarily large* data
- Can fail at any moment and roll-back (*transactional behaviour*)

Can we have *an interpreter* supporting all of these, while keeping the “core” semantics ***simple*** and ***easy to maintain***?

The Essence of Functional Languages

- Higher-order functions and closures
- Types and Type Inference
- Polymorphism
- Laziness
- Point-free style
- Combinator Libraries
- Algebraic Data Types
- Purely functional data structures
- Pattern Matching
- Folds
- Structural Recursion
- Continuations and CPS
- Type Classes
- Monads

The Essence of Functional Languages

Representing Monads*

Andrzej Filinski

School o…
Carnegi…
Pittsbu…
andrz…

…s and closures

…nce

- Laz…
- Poi…
- Co…
- Al…

Monad Transformers and Modular Interpreters*

Sheng Liang   Paul Hudak   Mark Jones†

Yale University
Department of Computer Science
New Haven, CT 06520-8285
{liang,hudak,jones-mark}@cs.yale.edu

We show how a ***set of building blocks*** can be used to construct programming language interpreters, and present implementations of such building blocks capable of supporting many ***commonly known features, including simple expressions, three different function call mechanisms […], references and assignment, nondeterminism, first-class continuations, and program tracing***.

- Purely functional data structures

Expressing any Effects
&
Modular Interpreters

- Structural Recursion
- Continuations and CPS
- Type Classes
- Monads

```ocaml
(*******************************************************)
(* A monadic big-step evaluator for Scilla expressions *)
(*******************************************************)

(* [Evaluation in CPS]

   The following evaluator is implemented in a monadic style, with the
   monad, at the moment to be CPS, with the specialised return result
   type as described in [Specialising the Return Type of Closures].
 *)

let rec exp_eval erep env  =
  let (e, loc) = erep in
  match e with
  | Literal l ->
      pure (l, env)
  | Var i ->
      let%bind v = Env.lookup env i in
      pure @@ (v, env)
  | Let (i, _, lhs, rhs) ->
      let%bind (lval, _) = exp_eval_wrapper lhs env in
      let env' = Env.bind env (get_id i) lval in
      exp_eval_wrapper rhs env'
  | Message bs ->
```

- About 200 LOC of OCaml
- *Hasn't been affected* by multiple modifications in the back-end protocol
- Changes in gas accounting *have not affected* the core interpreter
- Lots of performance bottlenecks fixed *without ever touching* the evaluator

Powered by Monads

Functional Programming Ideas in…

Research

… for modularity and proof reuse

Teaching

… for creating fun assignments

Software Engineering

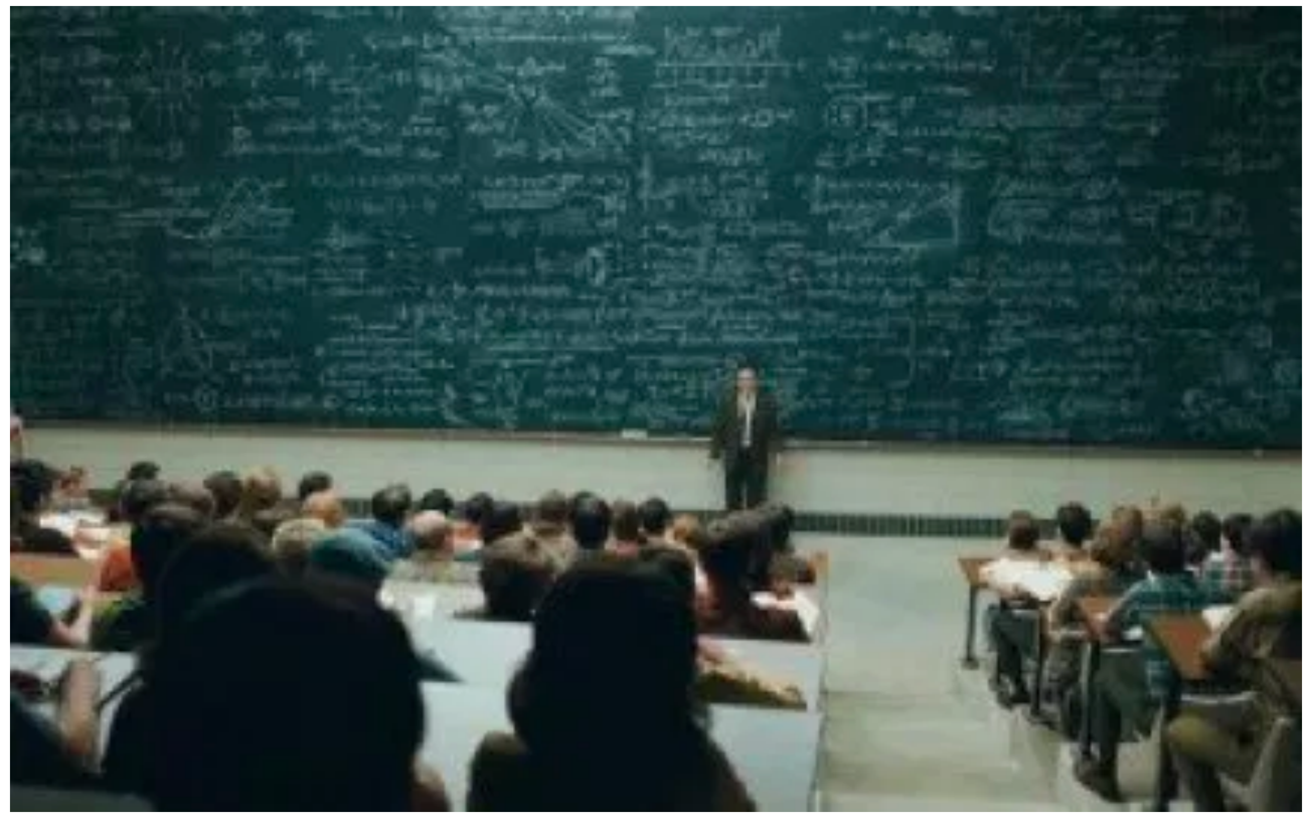
… for robust and maintainable artefacts

Functional Programming Ideas in…

Research

… for modularity and proof reuse

Teaching



… for creating fun assignments

Software Engineering

… for robust and maintainable artefacts

To Take Away

- FP insights *spread far beyond programming* in OCaml, Haskell, Racket, *etc.*
- FP keeps evolving: *new powerful ideas* are constantly emerging: *effect handlers, staging, automatic differentiation, security type systems…*
- Those ideas can be *your tools*, too!

Thanks!