Growing and Shrinking Polygons for Random Testing of Computational Geometry Algorithms

Experience Report

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

Abstract

This paper documents our experience of organising a medium-size programming contest for second year university students—an experiment we conducted as an attempt to introduce them to computational geometry. The main effort in organising the event was implementation of a solid infrastructure for testing and ranking solutions. For this, we employed functional programming techniques. The choice of the language and the paradigm made it possible for us to engineer, from scratch and in a very short period of time, a series of robust geometric primitives and algorithms, as well as implement a scalable framework for their randomised testing.

We describe the main insights, enabling efficient random testing of geometric algorithms, and report on our experience of using the testing framework to ensure the quality of our ranking artifacts.

1. Introduction

Imagine that we are put in charge of a museum that contains a large number of galleries, exhibiting precious pieces of art. Naturally, not all visitors of the museum are well-behaved, and some of them might try to vandalise the paintings and installations. In order to prevent this from happening, we will have to install security cameras in each gallery. While the cameras can observe all area around them as far as their line of sight is not interrupted by some obstacles (*e.g.*, walls), alas, they cannot move. They are also quite expensive, so we will not be able to buy too many of them, and instead we should choose their locations wisely. Therefore, the problem is as follows: *for each given gallery, find such locations, so security cameras installed in them would be able to survey the entire gallery's surface, while minimising their number*.

This setup describes the famous ***Art Gallery Problem*** (AGP), posed by Victor Klee in 1973 [7, 22]. Even though the problem's description is simple, as it is the case with many problems of computational geometry, AGP itself is proven to be NP-hard [17], therefore, no efficient way to find its optimal solution in a general case is known to date. However, many sufficiently good, although not optimal, algorithms have been proposed for solving AGP [22], with the upper boundary on the size of set of cameras $\lfloor \frac{n}{3} \rfloor$ (where n is a number of vertices in the gallery polygon) proved by Chvátal [2].

We drew inspiration from the variety of existing AGP algorithms, whose optimality depends on the properties of a polygon, to turn the Art Gallery Problem into an ICFP-style programming contest [8] for second year computer science students, introducing them to problems of computational geometry. The five days-long event, dubbed the *Art Gallery Competition*, during which the students were supposed to implement the best solution for the problem, has been organised as a part of the standard Software Engineering course offered by our department.

In order to make grading and ranking of the solutions in the competition of such scale feasible, we designed a framework for checking AGP solutions, implemented as a web-server, which ran during the time span of the event. While efficiency of solution checking was important (we wanted to provide automated feedback to students as fast as possible), what was far more important for us was *robustness* of our geometric machineries. In the competition, we fixed the set of problems (making it to be 30 large galleries of different shapes with floating-point coordinates), allowing the participants to submit *any* solution candidates, which we then checked for validity and optimality, ranking them accordingly. Therefore, we could not afford our checker to crash on arbitrary inputs.

There was no ready solution for our task, which could be easily integrated into a lightweight server-side application, so we had to develop our checking framework from scratch. For this quest, we chose a functional programming language with an expressive type system and a rich set of abstractions for server-side concurrent programming. Luckily, in such setting, we could also efficiently ensure the quality of our program artifacts, by applying QuickCheck-style random testing [3] to the implemented algorithms.

In this paper we mainly focus on the testing aspect of our implementation experience, outlining the key ideas behind the abstractions that we had to develop in order to employ randomised testing for checking and debugging geometric algorithms. We also report on our experience of using the functional approach for implementing from scratch a concurrent server-side application for automatic checking and grading solutions during the Art Gallery Competition.

2. Overview

Choice of programming language We chose Scala as a language for our implementation. The first reason for our choice was the rich library of collections and higher-order functions for data processing [21], provided by Scala, which we anticipated to come in handy when processing geometric data (and this proved to be a right expectation). The second reason for choosing Scala was its expressive type system with the support of implicit coercions and the ability to emulate type class-based polymorphism [4]. This feature of the language turned out to be essential for seamlessly switching between multiple representations of the same object (*e.g.*, of a point in cartesian or polar coordinates), augmenting existing data types with extension methods (*e.g.*, for checking ε-equivalence $\approx$ instead of equality for floating-point values). In combination with the support for monadic `do`-notation (expressed via Scala's `for`-comprehensions), it allowed us to implement a random testing framework, described in Section 3 and evaluated in Section 4.

Following the outlined reasons, we could also have picked Haskell. The additional motivation to use Scala was its smooth integration with various third-party JVM-based frameworks (most of which are implemented in Java), *e.g.*, for developing servlets or sending e-mails (more on that in Section 5), that were required in order to implement our solution-checking server. Finally, from

```
def triangulate(pol: Polygon): Set[Triangle] = {
  val vs = pol.getVertices
  val n  = vs.size
  if (n ≤ 2)  return Set.empty
  if (n == 3) return Set(mkTriangle(vs))
  val (i, j) = diagonalIndices(pol)
  val p1 = Polygon(vs(j) :: vs.slice(i, j))
  val p2 = Polygon(vs(i) :: vs.slice(j, n) ++ vs.slice(0, i))
  triangulate(p1) ++ triangulate(p2)
}

def diagonalIndices(pol: Polygon): (Int, Int) = {
  val vs = pol.getVertices
  val es = pol.getEdges
  val candidates = for {
    i    ← vs.indices.toStream
    j    ← i until vs.size
    cand = Segment(vs(i), vs(j))  // candidate diagonal
    c1   = !es.exists(e ⇒ e ≈ cand || e.flip ≈ cand)
    c2   = vs.forall(v ⇒ v ≈ cand.a || v ≈ cand.b ||
                    !cand.contains(v))
    c3   = es.forall(e ⇒ !intersect(cand, e))
    if c1 && c2 && c3
  } yield (i, j)
  candidates.head
}
```

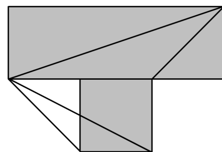
Figure 1. Naïve (and flawed) polygon triangulation.

all functional languages we knew, Scala was offering the best IDE support, facilitating debugging and major code refactorings.

Basic data types Our main data types are points, encoded via their cartesian or polar coordinates (with the implicit type-based conversion between the two views), and polygons on the plane. A polygon is represented by a list of vertices such that when "walking" along it the polygon's interior is "on the left". We didn't consider polygons with inner "holes" or self-intersections, and computed the polygon's properties, such as (non-)convexity, set of edges, or relation to a specific point via standard collection combinators.

Typical bugs in geometric algorithms It is difficult to get even seemingly simple geometric algorithms right from the first try. Consider, for example, the code in Figure 1 implementing an unoptimised triangulation algorithm, via the "ear clipping" method [18]. The function `triangulate` has $O(n^3)$ worst-time complexity (where n is the number of vertices) and implements the divide-and-conquer strategy to find a diagonal of the polygon `pol` (*i.e.*, a non-edge segment, which fully lies within it) when $n > 3$. It does so by calling `diagonalIndices`, which computes indices `i` and `j` of the diagonal's endpoints vertices. The indices are then used to split the polygon `pol`'s list of vertices to represent two smaller polygons, `p1` and `p2`, so `triangulate` proceeds to construct triangulations recursively, until the polygon `pol` is itself a triangle.

The function `diagonalIndices` iterates through indices of the polygon's vertices via **`for`**-comprehension (doing this lazily, thanks to the `toStream` conversion), looking for a suitable diagonal candidate `cand`. A candidate is considered suitable if the following three conditions hold: it is not an edge (`c1`), it only contains the two vertices of the polygon, which are its endpoints (`c2`), and it doesn't intersect internally any of the polygon's edges (`c3`). Unfortunately, these checks are not sufficient, which is demonstrated by the "triangulation" of the polygon on the right (it also has some additional spurious "diagonals"). What we forgot to check is that the candidate diagonal is also not *outside* of the polygon, which could be done by adding the simple condition `c4 = pol.contains(cand.middle)`.



3. Randomised Testing with Polygons

The bug in the flawed polygon triangulation was one of the first problems we caught in our geometric development. It has been detected via a unit test on a polygon similar to the one in the example above. It has soon become apparent that encoding polygons manually is not a good idea, as most of the "interesting" bugs can be discovered only on fairly large polygons (in terms of a number of vertices) with specific configurations of edges and angles.

To automate the process of detecting geometric bugs, we decided to use random property-based testing—an approach that has been implemented in the QuickCheck tool for Haskell [3], employed subsequently with great success in various areas [12–14, 19, 23, 24], and adopted in many other languages, including Scala [20], where it has become a part of major testing frameworks, such as ScalaTest.[1] But in order to employ QuickCheck-style random testing for debugging of polygon-manipulating procedures, we first need to supply two machineries: for *generating* and *shrinking* polygons. The former procedure is required for creating arbitrarily large inputs of various shapes, while the latter helps reducing inputs for failing tests. In this section we describe our approach for engineering scalable and customisable strategies for doing so.

3.1 Growing random polygons

If we are asked to generate an arbitrary polygon, the simplest solution will be to give a list of coordinates, describing a rectangle, for instance, `[(0, 0), (5, 0), (5, 2), (0, 2)]`. If we need something a bit more complicated, we can choose to "attach" another rectangle, let's say, with initial coordinates `[(0, 0), (3, 0), (3, 3), (0, 3)]`, to the segment `[(4, 2), (1, 2)]` of the edge `[(5, 2), (0, 2)]` of our "base" rectangle. This way, we will obtain a polygon with with the following encoding: `[(0, 0), (5, 0), (5, 2), (4, 2), (4, 5), (1, 5), (1, 2), (0, 2)]`.

We can then continue this process of *(i) picking* a suitable "primitive" polygon to attach, *(ii) locating* an edge of a base polygon and a segment on it (which might be the entire edge itself), where the attachment should be deployed, *(iii) attaching* the primitive by shifting, scaling and rotating it appropriately and *(iv)* checking that the newly deployed attachment didn't introduce self-intersection in the polygon. If the step *(iv)* fails, we repeat the steps *(i)-(iii)*.

This intuition summarises our method for growing polygons, which we call ***Pick-Locate-Attach*** (PLA). Even though we have presented it using rectangles, it can be instantiated with *base* and *primitive* polygons of any arbitrary shape. The only requirement for a polygon to be primitive is that it should have at least one *convex* edge, *i.e.*, an edge, which has the rest of the polygon in one half-plane with respect to it (for instance, some star-shaped polygons might not have convex edges). Our implementation ensures that it is always the case before making an attempt to attach. It also "normalises" a primitive polygon with respect to its arbitrary convex edge e, shifting and scaling it, so e would be a segment $[0, 1]$ on the X axis, and the whole primitive polygon is in the half-plane above it. This edge will then be used as a surface of attachment of the scaled/rotated primitive to the base edge's segment.

The PLA procedure, as described, might not terminate, or take a lot of time, due to possible failures of the check in step *(iv)*, therefore we have instrumented it with a "fuel" parameter, limiting the number of PLA "generations" and ensuring fast termination.

Figure 2 shows the base interface with partial implementation (defined as a Scala **`trait`**) for random polygon generators. Its first

[1] http://www.scalatest.org

```
trait RandomPolygonGenerator {
  val bases        : List[Polygon]
  val primitives   : List[(Int) ⇒ Polygon]
  val baseFreqs    : List[Int]
  val primFreqs    : List[Int]
  val locate       : Double ⇒ Option[(Double, Double)]
  val generations  : Int
  val scale        : Int

  // Random polygon generator
  implicit val arbitraryPolygon: Arbitrary[CompositePolygon] =
   Arbitrary( for {
    base   ← zipWithFreqs(bases, baseFreqs)
    iNum   ← Gen.choose(0, generations)
    primG  = zipWithFreqs(primitives, primFreqs)
    scaleG = Gen.choose(1, scale)
  } yield generatePolygon(base, primG, scaleG, iNum, locate) )

  // Relative frequency-based choice generator
  def zipWithFreqs[T](ps: List[T], freqs: List[Int]): Gen[T] =
    Gen.frequency(freqs.zip(ps.map(Gen.const(_))): _*)
}
```

Figure 2. Base Scala trait for random polygon generators.

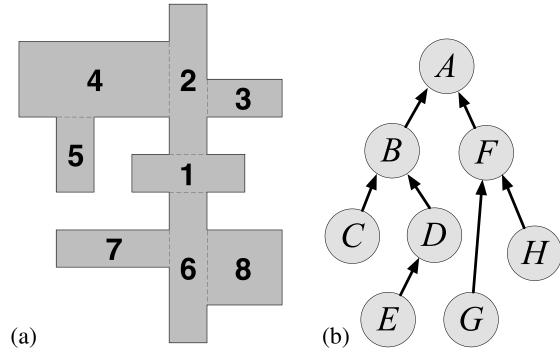


Figure 3. Composite polygon H (a), and its attachment tree (b) with $A = \mathbf{1}, B = (\mathbf{2}, -, A), \ldots, D = (\mathbf{4}, -, C), \ldots, G = (\mathbf{7}, -, F), H = (\mathbf{8}, -, G)$. The *attachment tree*-parents might differ from the cons-parent (*e.g.*, in the cases of D and H).

four abstract fields are used to provide, when instantiated, a set of base and primitive polygons, along with the relative frequencies, defining how often they should be picked. The parameter `locate` is a function, determining the strategy to choose endpoints of the attachment segment. Finally, the last two parameters, `generations` and `scale`, define the maximal number of times the PLA procedure should be iterated and the coefficient, used to "stretch" the primitive once attached (hence each of `primitives` takes `Int` as an input). What follows is the definition of the generator procedure `arbitraryPolygon`, defined using Scala's monadic **`for`**-notation, which draws random values for a base polygon and a number of generations, as well as creates a randomised generators `primG` and `scaleG` for primitives and scales, passing them to the `generatePolygon` function, implementing the PLA logic. The `CompositePolygon` type will be explained in Section 3.2.

While this interface could have been generalised even further to provide more flexibility in polygon generation, what is presented is already higher-order enough for the needs of our project. For instance, Figure 3(a) demonstrates a rectilinear polygon obtained via the PLA method with 7 generations, with rectangles as the base and primitives. The primitives are numbered as they were attached.

3.2 Trimming polygons

Once a geometry-specific property is violated, we would like to "shrink" the polygon, which served as a test case, to investigate the problem. However, "shrinking" here doesn't mean "scaling": it stands for reducing the polygon's size, *i.e.*, its number of vertices.

By simply removing vertices from the polygon's encoding, we risk to create self-intersections. What we should do instead is to "trim" the polygon, seeking a part of it that keeps the relevant shape, yet reproduces the bug. For this, we exploit the nature of the PLA method, generating a polygon as a *list* of attachements, which we record via the following Scala datatype `CompositePolygon` with only two constructors: `BasePolygon` and `Attached`.

```
sealed abstract class CompositePolygon { def pol: Polygon }
case class BasePolygon(pol: Polygon) extends CompositePolygon
case class Attached(base: CompositePolygon, e: Segment,
                    prim: Polygon)   extends CompositePolygon {
  def pol: Polygon = { /* render into actual polygon */ }
  lazy val parent: CompositePolygon = computeParent(this) }
```

The `BasePolygon` case merely stores the base polygon, while the *cons*-like `Attached` also records the base's edge `e`, which served for attachment and the primitive `prim` in its position *right before the attachment* (*i.e.*, shifted and scaled). We can now "render" the actual polygon by calling the `pol` method. Furthermore, for any composite polygon instance, we can render the whole series of its "pre-polygons" by compiling the prefixes of the list, obtained by unwinding its recursive structure, therefore getting meaningful "smaller" test cases to reproduce the bug.

This is still not good enough, as this way we will only obtain a small number of "sub-polygons", all rooted in the base one. To make a significant improvement, let us notice that, in fact, the way composite pre-polygons are obtained makes it possible to arrange them not just in a *list* but in a *tree*. By storing an attachment edge `e` in `Attached`, we can track its origin back to a composite polygon instance, where it has appeared for the first time. This origin will be the "parent" of the current composite polygon. The intuition is that we can only attach the child's primitive if the parent has been constructed, providing the attachment edge. We can safely ignore "unrelated" parents in different subtrees. We call such a structure an ***attachment tree***, and example is given in Figure 3(b). By taking *any* partial traversal (*e.g.*, DFS or BFS) of a composite polygon's *reversed* attachment tree and rendering it as a series of primitive attachments, one gets a valid *sub-polygon* of the original one.

As the last improvement to our shrinking strategy, we can notice that one can traverse a reversed attachment tree starting from *any* internal node, as long as the edge, establishing the link between the chosen initial parent node and its child, belongs to the parent's primitive attachment `prim` and is not a result of *splitting* a previously existing edge. In this case, we can render a sub-polygon starting from the internal node's `prim`, instead of the base polygon.

To summarise, our final shrinking strategy works on the reversed attachment tree of a randomly generated composite polygon, lazily rendering all its traversals (including those from internal nodes) into "candidate" polygons for reproducing the failed test.

3.3 Testing using custom polygon generators

To make use of our testing framework, one should instantiate the interface from Figure 2 with appropriate fields. In our case, we have several instances for generating rectilinear, quasi-convex and particularly nasty polygons (see Figure 7). As an instance of the `locate` strategy, we often use the following one, which sticks to integer positions on edges, whose length is greater or equal than 3:

```
val locate = (length : Double) ⇒
  if (l < 3) None else {
    val start = randomIntBetween(1, length - 2)
    Some((start, randomIntBetween(start + 1, length - 1)))  }
```

Once a polygon generator is defined, it can be imported into the testing scope. The conversion from `CompositePolygon` to `Polygon` instance is made transparent thanks to Scala's mechanism of customisable *implicit conversions*. The following code

```
1: procedure VisPartition(pol, VPS)
2:     TS := triangulate(pol)
3:     for each visibility polygon vp ∈ VPS do
4:         for each edge e of vp do
5:             TI := triangles from TS, properly intersected by e
6:             TP := Δ-partitioning of all triangles in TI via e
7:             TS := TS \ TI ∪ TP
8:     return TS
```

Figure 4. Triangular partitioning via visibility polygons *VPS*.

snippet illustrates random testing of the triangulation property that centres of all triangles are within the triangulated polygon.

```
test("Centres of triangles are within the original polygon") {
  check((p: CompositePolygon) ⇒ {
    val triangles = Triangulation.triangulate(p)
    triangles.forall(t ⇒ p.contains(t.center))
  })}
```

We have also implemented a number of QuickCheck-style *collectors* to analyse distribution of random polygons in our test cases.

4. Case Study: AGP Solution Checker

How useful was our framework for testing with polygons after all?

One of the main components of the infrastructure we have developed for the competition is the checker for submitted solutions, implemented as a part of a server, running during the contest week. Specifically, we needed an algorithm to check whether a proposed solution for an Art Gallery Problem instance is indeed a solution, that is, the set of cameras can see the entire gallery. In order to assess the solutions precisely and provide the feedback in a timely fashion, we could not afford to use cheap-and-cheerful approaches, such as random sampling or ray casting, and had to employ a proper visibility checking algorithm. As we soon discovered, there was no ready-to-use algorithm for this problem implemented as a JVM-compatible library, so we had to implement it from scratch. After having done that, we employed random testing to make sure that our implementation is correct and sufficiently robust to serve as checker for the length of the competition.

4.1 Constructing visibility polygons for individual cameras

Our checker for the Art Gallery Problem solutions builds on a procedure for constructing a *visibility polygon* (VP) of a point within a simple polygon. For this role, we chose to implement the stack-based plane-sweeping algorithm by Joe and Simpson [15, 16], which runs in $O(n)$ time. Even though this algorithm is presented in the literature as one of the simplest and most efficient solutions for the problem [22], in our implementation we faced a number of subtleties, stemming from the simplifications in its presentation [15], identified via random testing (see Section 4.4).

4.2 Visibility checker for a set of cameras

Once we have implemented an algorithm for VP construction, the problem of identifying "non-complete" solutions for AGP seemed almost trivial: we would just need to take a union of VPs for all cameras in the solution and check whether it is the same as the gallery polygon itself. Unfortunately, computing the union (and, equivalently, the difference) of two simple polygons is a challenging task, as the result of such operation might itself be a *non-simple* polygon and, for instance, contain inner holes.

Instead of following this path, we based our implementation of visibility checking and finding refutations (*i.e.*, points within a polygon that are not visible from any of the solution's cameras) on the idea of "progressive triangulation" by gradually adding constructed VPs and "refining" the initial triangulation *TS* of the polygon, via intersections of VPs' edges and "current" triangles. While

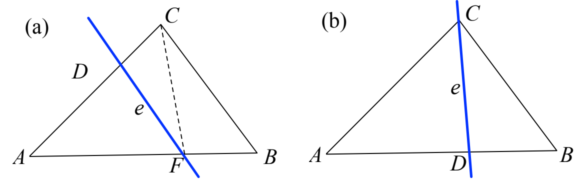


Figure 5. Δ-partitioning of the triangle *ABC* via the edge e (thick line), which intersects it, into (a) three or (b) two new tirangles.

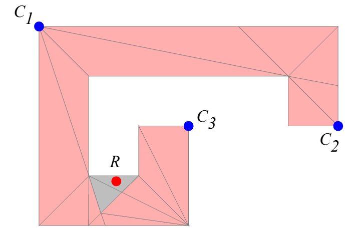
this idea is quite simple, we didn't encounter it in the literature on AGP and plane visibility [10], so we describe it here.

The procedure VisPartition, implementing the idea of fine-grained triangular partitioning, is presented in Figure 4. It takes as inputs the polygon *pol* and a list of visibility polygons *VPS*, constructed via the Joe-Simpson algorithm for the solution's cameras. All triangles from the previous partition *TS*, "affected" by an edge e of a polygon *vp* (and, hence, recorded in *TI*), are Δ-partitioned, as shown in Figure 5, into three or two new triangles each. Thus, the procedure of finding a refutation (if it exists) relies on the following theorem, establishing an invariant for VisPartition's main loop:

Theorem 4.1. *The triangles in the result partition, delivered by the procedure* VisPartition*, cover the whole polygon* `pol`*, and each of these triangles is either fully contained within some polygon vp ∈ VPS or is fully outside of any of them.*

Proof. By two-level induction: the top-level one is on the list of visibility polygons *VPS*, the inner one is on the list of the edges of a visibility polygon *vp* currently being processed. □

We can now iterate through the set of all obtained triangles, checking for each of them, whether its centre is not within any visibility polygon from *VPS*. If such triangle is found, its centre is the refutation, otherwise the polygon *pol* is fully covered. A result of the algorithm, with final triangulation, is illustrated on the right, with dots C_i indicating cameras, their VPs being pink, and the refutation R being a red dot in the bottom, in a gray triangle, invisible by the cameras.



4.3 Tested algorithms and properties

The foremost application of our framework for random testing with polygons was to simply check that none of the algorithms, critical for our goals (triangulation, visibility checking, *etc*), crashes on arbitrary large polygons and corresponding inputs. While this sounds like a trivial safety assertion, in the case of algorithms, such as the Joe-Simpson construction, we were surprised by the amount of possible subtle bugs that we initially missed and that immediately caused our implementation to crash on large non-trivial inputs.

Next, we employed randomised polygon generation for checking the following properties of our project's main algorithms:

1. **Triangulation of a polygon of size** n:
   (a) centre of each triangle lies within a polygon;
   (b) triangulation generates $n - 2$ (possibly degenerate) triangles;
2. **Joe-Simpson algorithm for visibility polygons** (VPs) [15]:
   (a) a vertex of a VP is also within the original polygon;
   (b) a middle point of a VP's edge is within the original polygon;
   (c) a random point *within* a VP is indeed visible from its origin;

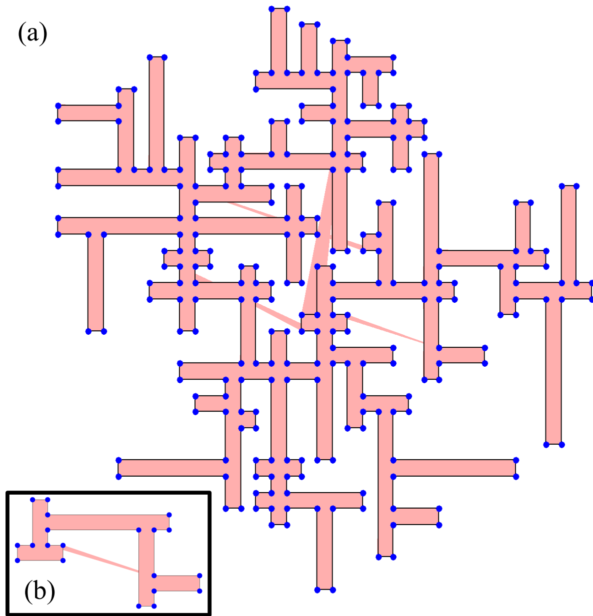


Figure 6. Randomly generated polygon (a), 260 vertices, which exhibits the bug in Joe-Simpson visibility algorithm, and its trimmed version (b), 20 vertices, reproducing the failed test case.

3. **Triangulation-based set visibility checker** (Section 4.2):
   (a) a refutation (if it exists) is within the original polygon;
   (b) a refutation for a set of cameras is not within any of their VPs;
4. **Fisk's basic algorithm for solving AGP** [9]:
   (a) delivers a solution of size within Chvátal's boundary $\lfloor \frac{n}{3} \rfloor$;
   (b) the visibility checker finds no refutations for its result.

While this list might be incomplete, it was sufficient to reveal a number of problems, which all were fixed, so our final implementation was robust enough to fulfill its purpose (see Section 5).

4.4 Discovered bugs

The Joe-Simpson algorithm was by far the most sophisticated part of our geometric development. The vast majority of the problems discovered with our implementation originated from inaccurate treatment of results of floating-point computations. Not entirely surprisingly, not only equalities should have been replaced by ε-equivalences (which is a standard practice for floating points), but also some of the inequalities (including *strict* ones, appearing when comparing radial angles of two positions of a plane-sweeping ray) had to be treated not precisely, but modulo a selected ε.

Perhaps, more surprisingly, several bugs were discovered *in the algorithm* [15] *itself*, and below we report on three of them.

The first problem was a result of an erroneous notation, chosen in the paper [15] in order to unify two cases of a camera position in a polygon: *on the boundary* (i) and in *its proper interior* (ii). In the former case, a specific treatment should be given, in fact, only to the situation when the camera is located in the polygon's *vertex*. If it's on an *edge*, but not in a vertex, it should be treated in the same way as in the case (ii). Following the policy (i) in this case (as suggested by the authors) leads to errors for some polygons.

The next bug is almost trivial. The algorithm is formulated as iteration over vertices of the gallery polygon, but an important side condition was omitted in one of the cases, leading to an *index-out-of-range* error for specific configurations of angles between edges. The fix was easy: just add the necessary boundary check.

The last problem is the most subtle and occurred only in situations when several vertices of the polygon were aligned on a line of sight of a specific camera, with some of them visible and some others hidden. This has been discovered on large rectilinear polygons via the property test 2(b) from Section 4.3. An example is shown in Figure 6(a), where cameras are put in every vertex of the polygon. Thanks to the shrinking strategy, the testcase has been minimised down to a polygon of size 20 (Figure 6(b)), used to locate the problem in the algorithm. In this particular case, the problem is caused by the three vertices, with the ray origin being the rightmost bottommost one. As a fix, we added the corresponding distance check to the plane-sweeping algorithm [15], so it would exclude aligned hidden vertices from consideration.

5. The Art Gallery Competition

While debugging and optimising individual geometric algorithms was fun, the ultimate goal of our project was to implement a working server to run the geometric programming contest. Having very limited time to develop the infrastructure, we made use of existing Scala–powered frameworks for the server-side programming.

5.1 Implementing the server

The server for checking solutions was implemented using *Spray*,[2] a Scala-based open-source toolkit for building REST/HTTP-based applications using servlets. *Spray* comes with a lightweight embedded web-server and builds on top of Akka [27], a Scala-powered framework for concurrent applications, facilitating distributed request processing using Scala actors [11].

Since the participants of the competition were working in teams of four, we distributed the workload of checking submitted solutions by allocating a separate Akka actor for each team. We did not use any specific database backend for persistent storage, resorting instead to generating per-submission log files (backed up to the cloud), managed atomically and storing all data about submitted/accepted solutions as well as submission times. The front-end webpages with scoreboards were rendered dynamically from the stored submission data via Scala's native support for XML.

5.2 Overall implementation effort

The server implementation for the competition is about 1500 LOC. The implementation of geometric primitives and procedures is 1450 LOC, and the random polygon testing framework is about 350 LOC. The work on the implementation was carried out in less than *one man-month*, starting from sketching the initial idea and including the planning of the competition, multiple discussions, playing with the random testing framework, engineering the problem sets and testing the server. Overall, we consider it to be a very modest implementation effort for the problem, given that we had to implement the core logic for solution checking from scratch.

5.3 Running the competition

In order to generate a set of 30 challenging galleries for the competition, we employed the random polygon generating framework, described in Section 3. The sizes of problems ranged from a couple of dozen vertices to about 500. For instance, Figure 7 features a large polygon, which was generated using rectangles, triangles, convex decagons, and "Chvátal's combs" as primitives.

94 second year CS students took part in the competition, working in groups, making it 24 separate teams. None of the participants had systematic exposure to geometric algorithms before, as they are

[2] http://spray.io

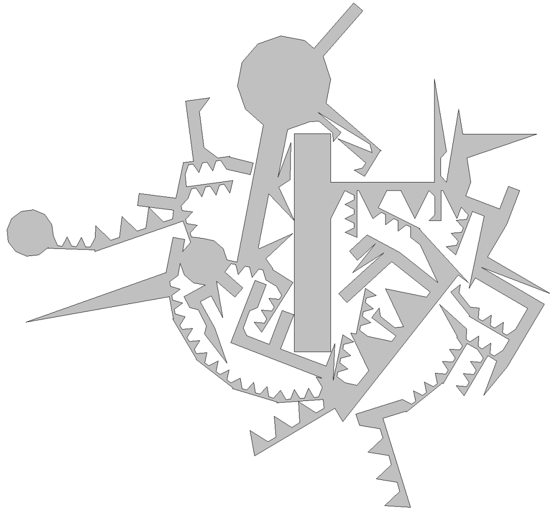


Figure 7. One of the polygons from the competition (469 vertices), obtained by the PLA method using six primitives, including convex polygons, rectangles, decagons and "Chvátal's combs".

not a part of the first or second-year curriculum. Each team has been given unique credentials for authentication and submitting their solutions. Teams were allowed to submit their batch solutions (encoded as ASCII text files) multiple times, improving their results. Overall, 2360 files with solutions were submitted and processed.

We ran our server on an Intel Xeon 2.67 GHz Linux machine with 4GB RAM. It was rejecting solutions for individual galleries (*i.e.*, sets of camera coordinates), which were larger than Chvátal's boundary $\lfloor \frac{n}{3} \rfloor$. Checking a complete batch of solutions of this size for all 30 galleries from the problem set was taking 1–3 minutes, depending on the current load, which was up to 10 simultaneous submissions. The server was running for the entire duration of the competition without crashing or noticeable slowdowns.

Since we didn't require to submit the code, the efficiency of the implementations was not of the teams' concern. As implementation languages, the students employed (in descending order of popularity) Python, Java, JavaScript, C, C#, C++ and MATLAB. While several teams tried to visualise the problems and find the optimal solutions "by hand" using the human intuition and visual inspections, the majority of the participants started from implementing the textbook Fisk's algorithm [9], delivering a solution within Chvátal's boundary. Many teams then implemented a conservative or imprecise visibility checker, and used "greedy" algorithms in order to optimise their solution by throwing away some of the cameras and checking whether the remaining ones still cover the whole gallery. Some other teams tried to use the server for the same purpose, as an oracle, "querying" it manually and receiving the results by e-mail, although, without much success. We were pleasantly surprised by one team's discovery of the recent result (which we were not aware of), reducing AGP to the Set Cover Problem (SCP) [26], which they solved using Integer Linear Programming techniques. As the result, the team obtained optimal solutions for almost all of the problems from our set, beating other top-ranked teams, whose best solutions were, as we suspect, hand-crafted.

6. Discussion and Lessons Learned

We now summarise the lessons we learned from the project.

Our background and development structure When this project started, all of us had experience with functional programming in Haskell, OCaml and Scala, but none of us had expertise in computational geometry beyond the contents of the standard textbook [7]. Neither did we have a lot of experience with randomised testing, Spray or Akka. Deciding to use Scala and the infrastructure it provides was, thus, a venture, which turned out to be the right choice for our purposes, as our priorities were more towards the speed of development and robustness, than efficiency of the final artifact.

The idea to use QuickCheck-style random testing was not pursued from the beginning, but only occurred to us, once the sizes of unit tests we had to write to exercise hypotheses about potential problems, went beyond 30 vertices in a polygon. The necessity to implement a shrinking strategy came even later from the fact that debugging an algorithm on a 200-vertices polygon is unpleasant.

Using alternative implementations An alternative solution was, indeed, to use a well-established state-of-the-art library of computational geometry algorithms, *i.e.*, CGAL.[3] However, doing so would require us to give up the opportunity of using the functional programming benefits outlined in Section 2. It would also force us to stick with C++ as an implementation language, or, alternatively, sacrifice the uniformity of the core logic/server implementation.

At the very late stages of our development, right before the beginning of the competition, we have asked a colleague, who had previous experience with CGAL, to test our server by engineering solution candidates and uploading them for checking. The colleague immediately implemented the approach with random camera generation within a polygon, followed by subsequent construction of its VP (using the Joe-Simpson algorithm, which was recently implemented in CGAL [1]) and taking the union of VPs for all generated cameras. To our surprise, CGAL didn't sustain this stress-testing, and its VP construction implementation crashed on several inputs from our set (one of them being of size 338). We are currently planning to submit the bug report to the authors of CGAL.

We have also investigated existing alternative approaches to generate random orthogonal polygons [25], but could not adopt them, as they (a) didn't generalise easily to arbitrary shapes and (b) didn't provide a natural way of implementing a shrinking strategy.

What can you learn from our experience? Lifting the hard requirement for the best possible efficiency of our implementation was, indeed, a facilitating factor, allowing us to choose the language, which is declarative enough to make the project feasible in the time period we could afford. For what it's worth, from our experience, using Scala and functional programming in an educational medium-size project involving computational geometry is not a bad idea, and overall we had a very pleasant time developing and playing with it (and so did the participants of the competition).

The classical algorithms of computational geometry are typically formulated in a very imperative fashion. For instance, the Joe-Simpson algorithm [15] is presented in FORTRAN-like pseudocode with several global mutable variables and a single global driver loop. Translating this to Scala and idiomatic functional style was, nevertheless, fairly straightforward, essentially requiring us to *refunctionalize* [6] and *direct-style transform* [5] the first-order imperative code. At this point of the development, it was, indeed, invaluable to have a good testing/debugging framework to immediately check the properties of our implementation.

For the purposes of debugging, having a good IDE really helps. For Scala programming we used IntelliJ IDEA with the Scala plugin,[4] which provided great support for large code refactorings (*e.g.*, consistent renaming and code relocation) and was very helpful for debugging failed test cases, allowing us to inspect several call frames simultaneously and re-run parts of the code at breakpoints.

[3] http://www.cgal.org

[4] http://blog.jetbrains.com/scala

What didn't work so well Scala is notorious for its long compilation times, and it was the price we agreed to pay for using its expressive type system. When implementing QuickCheck-style generators (*cf.* Figure 2), we had to "delay" fetching of some random values from sub-generators (*e.g.*, `primG`) until later computation stages (instead of querying them immediately in the top-level **`for`**-comprehension), and then query them directly within the procedure implementing PLA. There might be a more elegant way to implement this logic in ScalaCheck that we are not aware of.

7. Conclusion

We reported on our experience of taking functional programming to the field where it is rarely used—computational geometry. We have demonstrated that random property-based testing—a standard tool of a working functional programmer—is applicable and highly beneficial for checking and debugging realistic geometric algorithms. Our experience shows that when the code robustness and speed of development are of bigger concern than fine-tuned performance, the functional programming approach provides a reasonable way to tackle the complexities of programming geometric applications.

Acknowledgments

I would like to thank James Brotherston, Lewis Griffin, Robin Hirsch, Kareem Khazem, Gilles Rainer, Reuben Rowe and other members of UCL PPLV group for the fruitful discussions and for the feedback on the design of the Art Gallery Competition. Joshua Moerman went beyond the call of duty when testing the solution checker, discovering the bug in the CGAL library. I benefited from discussions with Leonid Shalupov, who hinted the "discretisation" approach that eventually inspired the checking algorithm from Section 4.2. I am grateful to Jan Midtgaard for his deep comments on the paper and general insights on property-based testing. Finally, I wish to thank Kira Vyatkina for introducing me, as a student, to the exciting field of computational geometry ten years ago.

References

[1] F. Bungiu, M. Hemmer, J. Hershberger, K. Huang, and A. Kröller. Efficient computation of visibility polygons. *CoRR*, abs/1403.3905, 2014.

[2] V. Chvátal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B*, 18:39–41, 1975.

[3] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279. ACM, 2000.

[4] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360. ACM, 2010.

[5] O. Danvy. Back to direct style. *Sci. Comput. Program.*, 22(3):183–195, 1994.

[6] O. Danvy and K. Millikin. Refunctionalization at work. *Sci. Comput. Program.*, 74(8):534–549, 2009.

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 3rd edition, 2008.

[8] E. Dolstra, J. Hage, B. Heeren, S. Holdermans, J. Jeuring, A. Löh, C. Löh, A. Middelkoop, A. Rodriguez, and J. van Schie. Report on the Tenth ICFP Programming Contest. In *ICFP*, pages 397–408. ACM Press, 2008.

[9] S. Fisk. A short proof of Chvátal's watchman theorem. *J. Comb. Theory, Ser. B*, 24(3):374, 1978.

[10] S. Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, 2007.

[11] P. Haller and M. Odersky. Actors that unify threads and events. In *COORDINATION*, volume 4467 of *LNCS*, pages 171–190. Springer, 2007.

[12] S. Holdermans. Random testing of purely functional abstract datatypes: guidelines for dealing with operation invariance. In *PPDP*, pages 275–284. ACM, 2013.

[13] C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *ICFP*, pages 455–468. ACM, 2013.

[14] J. Hughes. QuickCheck Testing for Fun and Profit. In *PADL*, volume 4354 of *LNCS*, pages 1–32. Springer, 2007.

[15] B. Joe and R. B. Simpson. Visibility of a simple polygon from a point. Technical Report CS-85-38, Dept. of Math and Computer Science, Drexel University, 1985.

[16] B. Joe and R. B. Simpson. Corrections to Lee's Visibility Polygon Algorithm. *BIT Numerical Mathematics*, 27(4):458–473, 1987.

[17] D. Lee and A. Lin. Computational complexity of art gallery problems. *IEEE Transactions on Information Theory*, 32(2):276–282, 1986.

[18] G. H. Meisters. Polygons Have Ears. *The American Mathematical Monthly*, 82(6):648–651, 1975.

[19] J. Midtgaard and A. Møller. QuickChecking Static Analysis Properties. In *ICST*, pages 1–10. IEEE, 2015.

[20] R. Nilsson. *ScalaCheck: The Definitive Guide – Property-based testing on the Java Platform*. Artima Press, 2014.

[21] M. Odersky and A. Moors. Fighting bit Rot with Types (Experience Report: Scala Collections). In *FSTTCS*, volume 4 of *LIPIcs*, pages 427–451. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009.

[22] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[23] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *AST*, pages 91–97. ACM, 2011.

[24] V. St-Amour and N. Toronto. Experience report: applying random testing to a base type environment. In *ICFP*, pages 351–356. ACM, 2013.

[25] A. P. Tomás and A. L. Bajuelos. Quadratic-time linear-space algorithms for generating orthogonal polygons with a given number of vertices. In *ICCSA (3)*, volume 3045 of *LNCS*, pages 117–126. Springer, 2004.

[26] D. C. Tozoni, P. J. de Rezende, and C. C. de Souza. The quest for optimal solutions for the Art Gallery Problem: A practical iterative algorithm. In *SEA*, volume 7933 of *LNCS*, pages 320–336. Springer, 2013.

[27] D. Wyatt. *Akka Concurrency: Building Reliable Software in a Multicore World*. Artima Press, 2013.