# ConvexMerger: Algorithmic Optimisations & Challenges

An area maximisation game based on the idea of merging convex shapes

Roan Hofland
Eindhoven University of Technology
r.w.p.hofland@student.tue.nl

Emiliyan Greshkov
Eindhoven University of Technology
e.greshkov@student.tue.nl

14th of May, 2023

**Supervisor**
Irina Kostitsyna (i.kostitsyna@tue.nl)

# Contents



## Version History

| Version | Date | Comment |
|---|---|---|
| v1.0 | 2023-01-25 | Initial version |
| v1.1 | 2023-03-08 | Handle initial feedback & finish remaining sections |
| v1.2 | 2023-03-29 | Various minor improvements to most sections |
| v1.3 | 2023-05-14 | Various minor improvements based on received feedback |

# 1 Introduction

This work describes in detail the effort and results of a Geometric Algorithms Capita Selecta project. The goal of the project is to improve the processing of several key elements of ConvexMerger, an area maximisation game based on merging convex shapes. The focus is on improving the data structures and algorithms for efficient handling of several key operations or to come up with novel optimised solutions and to implement them. The key operations are point location, convex object merging, and segment intersection testing. Secondary goals include the addition of more and better animations and general code quality improvements.

In ConvexMerger, up to four players compete against each other in a playfield populated with non-overlapping convex objects. The goal of each player is to claim as large an area as possible while competing against their opponents. Players take turns sequentially and during their turn a player can do one of two actions:

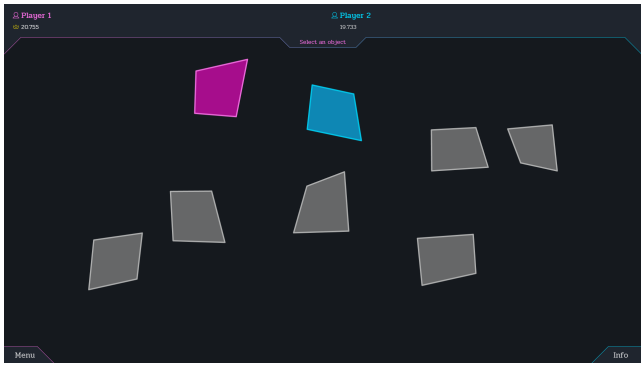
**1)** They can claim a new object which has not yet been claimed by any player.

**2)** They can merge one of their claimed objects with another object that is either unowned or is owned by them. When two objects are merged they are replaced by a new convex object that has the shape of the convex hull of the two merged objects. A merge is not allowed if there are any other objects on the boundary of the resulting convex object. However, objects fully contained within the result will be stolen from their current owner.

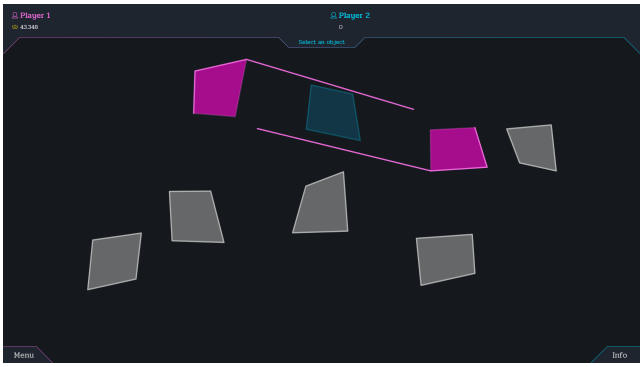
Figure 1a depicts a playfield where two players have each claimed a single object, and Figure 1b depicts a possible next move. In the followup Player 1 merges their claimed object with an unowned object, resulting in an object that absorbs Player 2's claimed object.

The objects on the initial playfield are convex and new objects created through merging are also convex. It is possible to play the game with up to four players, some or all of which can be Artificial Intelligence (AI) players, and it is possible to play online multiplayer. The game ends when the active player has no moves left, at which point the player with the largest owned area wins. The game is also available as open source software on GitHub [1].

ConvexMerger can be played entirely with the mouse. As mentioned, players can either claim an object or merge objects during their turn. Next we will give a more formal description of the merge of two objects. Let us refer to the selected object as the source of the merge $s$, to the other object as the target $t$, and to the convex hull that would result from the merge as $r$. The merge between $s$ and $t$ is only allowed if the boundary of $r$ does not intersect any other object's boundary. If the merge is allowed, then $r$ is created, and $s$, $t$ and any other objects on the interior of $r$ are absorbed into it, including objects that are claimed by other players. The game promotes strategic thinking for the selection of objects to claim and setting up advantageous merges is crucial to winning. For example, while a naive playstyle would be to always play the move that yields the largest increase in area for that turn, this strategy is unlikely to win in a serious game. In fact, one of the implemented AI players (Isla) follows this strategy. Instead, it is more beneficial to consider the long term gains. This can for example be done by merging two smaller objects into a longer object that can be merged with an object on the other side of the playfield next turn. Alternatively, extremely large merges like this can be used to try to absorb objects from an opposing player.




(a) A playfield where Player 1 and Player 2 have made a single object selection each.




(b) The same playfield right as Player 1 has chosen to merge, absorbing Player 2's claimed object.

Figure 1: Example of a merge on a generated playfield.

The game has several types of events that trigger throughout execution. One event is the human clicking on the playfield, which triggers a point location query. If the point is located to be inside an object, then that

[1] https://github.com/RoanH/ConvexMerger

object is to be selected or claimed if possible. Another event is the attempt of a merge. Here a few things happen. First, the segments that will be added to complete the merge are computed. These are the segments on the boundary of the result $r$ that are part of neither $s$ nor $t$, and we call these merge segments. Afterwards, the merge segments are tested for intersection with any other segments. If they intersect nothing, the merge is executed and the underlying data structures are updated, otherwise the merge is rejected.

At the start of this project, the result of Convex Object merges was computed using the Jarvis March algorithm, also known as the Gift Wrapping Algorithm [23]. The same algorithm was used for computing the helper lines visible while a player has selected an object that they want to merge from. For point location queries, we used the Vertical Trapezoidal Decomposition data structure [17], rebuilding it after each convex object merge. Segment intersection testing was done trivially, i.e. the query segment was compared against every segment of every object.

The main contributions of this project comprise optimisations to the computation of convex hull merges, point location queries, and segment intersection testing. A secondary contribution is the addition of visualisations for each of the improved algorithms. Convex hull merges were improved with a two-pass algorithm that computes both the segments needed to merge a pair of convex objects and the resulting hull of the merge. A more detailed explanation is available in Section 3.1 and the corresponding visualisation is described in Section 5.3. Point location query handling was improved by dynamically updating the previously static vertical trapezoidal decomposition instead of rebuilding it at every update, resulting in worst-case $O(\log n)$ query times and $O(n)$ update times. Further details on this can be seen in Section 3.2 and the corresponding visualisation is showcased in Section 5.1. Segment intersection testing is improved using partition trees to divide the plane and only test a subset of the segments stored only at some plane subdivisions. Two types of partition trees were implemented, and the use of conjugation trees is particularly interesting, as the static set of points throughout the game and the insertion of segments means that the structure does not need to be dynamic, resulting in $O(n^{0.695})$ query and $O(\log n)$ update times. We also looked into engineering a novel structure for intersection testing, and found several existing structures that would not work well for ConvexMerger. Further details on segment intersection testing can be found in Section 3.3 and the corresponding visualisations are discussed in Section 5.2.

In Section 2, we will present a survey on the literature that we have found relevant to the goals of the project. Then, in Section 3, we will go over each of the main algorithmic improvements and mention some auxiliary algorithms. For each algorithmic improvement, we will first quickly describe and analyse the starting point, then we will describe the idea of the improvement, perform runtime analysis, and discuss implementation details, design choices, and issues. Section 4 contains an evaluation on the performance and characteristics of the implemented data structures. Section 5 presents the various visualisations that showcase the structures on top of the playfield and the animations that can be triggered during gameplay.

# 2 Previous Work

**Convex Hull merging** The first topic we looked into was convex hull merging, more specifically the problem of determining the common convex hull of two convex objects. At the start of the project, the implemented algorithm for convex hull computation was the Jarvis March [12]. For a set of $n$ points it computes the convex hull in $O(nh)$ time, where $h$ is the number of points on the resulting convex hull. It does so by first finding the leftmost of the points, $p_0$, first (bottom one if multiple), then finding the point $p_1$ such that the line $p_0p_1$ has the smallest angle to the positive y axis. Both points are on the hull, and each next point $p_i$ on the hull can be selected as the point such that the line $p_{i-1}p_i$ has the smallest angle to $p_{i-2}p_{i-1}$. We find $h$ points this way, with each point requiring up to $n$ comparisons, leading to the described running time.

A faster alternative for finding the convex hull of a set of $n$ points is Graham's scan [11], which sorts all points around an interior point and adds them to the hull in order while removing the middle points of concavities during addition. The algorithm runs in $O(n \log n)$ time, dominated by the radial sort. Finding an interior point and the construction of the hull take linear time each. Several even faster algorithms exist, such as Chan's algorithm which uses a combination of Graham's scan and the Jarvis March for an $O(n \log h)$ runtime [4].

Another angle to attack the problem is based on the 'rotating callipers' approach, originally introduced by Shamos [18] as an advancing algorithm for finding the intersection of two convex polygons. Toussaint finds how to use the rotating calipers to compute the bridge lines between a pair of convex hulls that are required for building their common convex hulls [22]. The algorithm does a single pass of all $t$ points on the two hulls, resulting in $O(t)$ running time. The idea is that we start off with two vertical (or horizontal) lines passing through the leftmost (bottom-most) point of each hull. Then we rotate these in the same direction simultaneously along the boundaries of their respective hull. At exactly two angles, the lines will coincide and they will bridge the hulls such that all points of both hulls lie on or on one side of the coincident lines. At least one point of each hull will lie on the bridge line, and the segment between them is the bridge segment, or as we will refer to it later – the merge segment.

**Point Location using Vertical (Trapezoidal) decomposition** One of the first decompositions of polygons into trapezoids was done in 1981 by Lee within the context of computer graphics [13]. A couple years later Fournier and Montuno present an approach for the triangulation of simple polygons, possibly with simple polygon holes, via a trapezoidal decomposition constructed in in $O(n \log n)$ time [9]. The decomposition approach to triangulation was then extended for any set of segments that do not intersect outside of their endpoints by Seidel using a randomised incremental construction algorithm for a $O(n \log n)$ construction time in expectation [17]. While linear-time algorithms for triangulation exist, the trapezoidal decomposition is simpler. Trapezoidal decompositions can also be used for point location, with query times of $O(\log n)$ in expectation. These variants are static, however multiple works provide dynamic versions of trapezoidal decompositions for various settings [7, 5, 6, 10], including for the general case [2].

**Segment intersection testing** One of the founding works on segment intersection testing was published in 1976 by Shamos and Hoey, and it presents an $O(n \log n)$ algorithm for checking whether an intersection exists between any pair of $n$ segments [19]. Bentley and Ottman present an improvement that can count the number of intersections in $O(n \log n)$ and can find and report all $k$ intersections in $O(n \log n + k)$ time. These algorithms however operate on the whole set of segments and find all intersections between them, while we are only interested in the intersections with newly added segments. Furthermore, we only need an answer as to whether an intersection exists, creating the opportunity of further optimisations via early stops and ordering the search space.

We then looked in the direction of structures that support testing a query segment for intersection with any segment from a set of $n$ segments that do not intersect each other except at endpoints. One such structure is the visibility polygon [1, 20], which can be used to answer whether a query segment intersects any other segments by checking whether one of its end points can see the other. The construction of the visibility polygon is done for a single point and takes $O(n \log n)$ time in this setting. Therefore, it must be constructed for each query and has to be reconstructed after updates to the set of segments, leading to $O(n \log n)$ query and update times. Since the query segments for segment intersection in ConvexMerger are only between the existing segments' endpoints, an alternative approach would be to precompute the visibility complex of each point and take note of what points lie within, resulting in $O(n^2 \log n)$ construction and update time and faster queries.

Since we expect a lot of segment intersections, we surveyed dynamic structures that allow for fast insertion and deletion of segments. One class of such structures are dynamic partition trees [16]. These trees are built on top of partition trees [15], such as kd-trees [3] and conjugation trees [8]. Kd-trees alternate between vertical and horizontal bisectors to recursively partition the points in the plane in (near-)equal halves. Conjugation trees have one arbitrary starting bisector, and each next bisector is a conjugate of the previous, meaning that it bisects each of the halves split by the previous bisector. It is proven in [24] that a conjugate always exists for a bisector, and [14] provides an algorithm for computing a conjugate in linear time. Another interesting detail is that since the point set of ConvexMerger is static, there is no need to use the dynamic conjugation tree proposed in [16], and just the standard conjugation tree as in [8] can be used.

# 3 Algorithms

**Geometric Setting:** Before describing the algorithms it is useful to introduce the geometric setting of the game and the assumptions that can be derived from it. These assumptions can be used when designing new algorithms for the game, and the algorithms in this report rely on these assumptions. Note that some of these assumptions could be considered implementation detail, meaning that algorithms could be adapted to handle a different situation (e.g., for the 4th point, using clockwise winding order). Given that the game is based around non overlapping convex objects, some useful assumptions can be made about the points and lines algorithms need to work with. We will also specify some invariants we take care to maintain when merging objects. The following list summarises the key assumption of the setting, here line segments refers to the collection of all $m$ boundary line segments for all the convex objects, and points refers to the collection of all $n$ boundary points for all the convex objects. Note that $n$ and $m$ initially have the same value, since each object has the same number of boundary segments and boundary points. However, while the initial point set does not change during the game, the set of line segments does. Each merge executed during the game results in two new line segments being created between two preexisting points. While there are also segments that can be removed after a merge, this is not currently done by the majority of algorithms presented in this section. Therefore, it is worthwhile to make a distinction between $n$ and $m$. However, note that the maximum number of merges during a game is linear in the number of objects. This means that $m$ will always be bounded by $O(n)$.

**1)** None of the convex objects intersect. Note however that it is possible for remains of an object that was absorbed in a merge to be fully contained in another convex object.

**2)** All convex objects have a non-zero area.

**3)** The first point of the hull of points that defines a single convex object is the bottom left-most point.

**4)** All objects are convex and have a counter-clockwise winding order.

**5)** None of the line segments overlap or intersect, with the exception of the end points.

**6)** None of the points coincide.

**7)** On the boundary of a single convex object no three consecutive points are collinear. This is an invariant we take care to maintain when merging objects.

## 3.1 Convex Object Merging

Convex hull merging is the core idea behind ConvexMerger. When two objects $o_1$ and $o_2$ are merged into $r$, there are two line segments that need to be created that link $o_1$ and $o_2$, we call them merge segments. These are the only line segments on the hull of $r$ that are not on the boundary of either $o_1$ or $o_2$.

As mentioned in Section 1, the algorithm previously used for computing the convex hull of the merge of two convex objects is the Jarvis March [23]. It should be noted that this approach does not directly provide us with the merge segments. Instead, these were computed using a separate algorithm. The algorithm takes as input the two original hulls $o_1$ and $o_2$ and the new hull $r$. It assures that $o_1$ is the original hull with the leftmost point, by swapping $o_1$ and $o_2$ if that is not initially the case. It then iterates the points of $r$ and $o_1$ in parallel, both iterations starting from the bottom leftmost point on $r$ until the points no longer coincide. Then, the last iterated point on $r$, $p_1$, is part of $o_1$ and the current point $p_2$ is part of $o_2$, meaning $(p_1, p_2)$ is the first merge segment. From $p_2$ onward, the points of $r$ are iterated in parallel with the points on $o_2$, and again whenever they do not coincide, the last iterated point $p_3$ on $o_2$ and the current point $p_4$ on $o_1$ make up the second merge segment $(p_3, p_4)$. This algorithm runs in time linear to the number of points on all three input hulls.

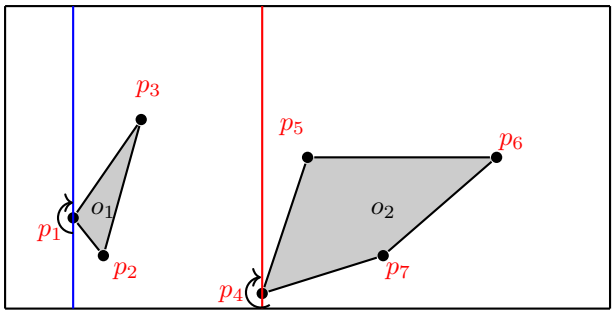
We improve on this with an approach loosely based on the algorithm for convex polygon intersection by Godfried Toussaint [21]. Our approach borrows the rotating caliper approach from Toussaint, which observes that the two line segments on the merged convex hull that have an endpoint on each of the initial convex object can be found by simultaneously rotating two lines $c_1$ and $c_2$, called calipers, along the peripheries of the hulls. At exactly two moments, $c_1$ and $c_2$ will coincide. The points around which the calipers rotate at these moments we call merge points, and the line segments between each pair of merge points – merge segments, or merge lines.

The calipers $c_1$ and $c_2$ start out vertically and oriented upwards, rotating clockwise around each object's leftmost point. Rotation happens simultaneously for the two calipers by the smallest angle such that the next point on either convex object is reached. For the convex object whose point was reached, we change the rotation point of the caliper to the farthest point in the object that the caliper intersects, skipping over collinear points. The rotation ends whenever the calipers have rotated around all points of the two objects and have reached the initial points. An example showing the process is given in Figure 2.

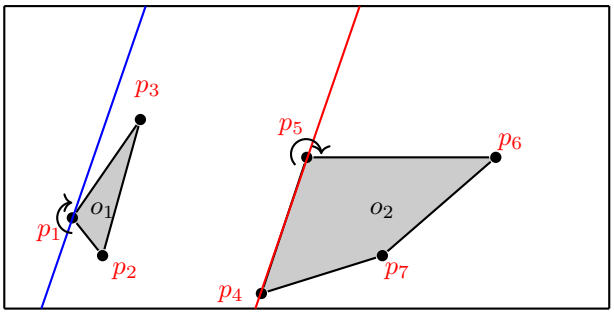
Let us name the convex object whose leftmost point is further left, or below in case of ties, $o_1$, and the other – $o_2$. Observe that initially $c_2$ does not lie counterclockwise to the orientation of $c_1$. Whenever this changes during rotation of the calipers, we observe that the points around which the calipers have been rotating are a pair of merge points. Since this happens exactly twice for a pair of convex objects, the two merge lines $m_1$ and $m_2$ are formed by four points $mp_1$, $mp_2$, $mp_3$ and $mp_2$. The first merge segment $m_1$ is directed from $o_1$ to $o_2$ with endpoints $mp_1$ and $mp_2$, and $m_2$ is oriented from $o_2$ back to $o_1$ with endpoints $mp_3$ and $mp_4$.

To obtain the convex hull of the merge, we start from the first and leftmost point of $o_1$, since it is guaranteed to be on the convex hull of the merge result. Then, we add all points of $o_1$ up to and including the source point of $m_1$, $mp_1$. For $o_2$, we add all points in sequence, starting from , $mp_2$, and ending with $mp_3$. Finally, we add $mp_4$ and the remaining points of $o_1$. In case of collinearity of points along the resulting convex hull, the middle collinear points are excluded from the result.

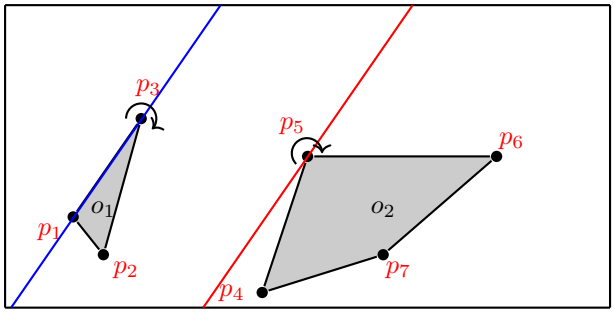
The runtime analysis for this algorithm is fairly simple. As can be seen in the algorithm description, we iterate the points for each of the input convex objects twice. Once for computing the merge lines, and a second time for obtaining the resulting convex object. The operations during each of these iterations take constant time, resulting in a total asymptotic runtime of $\Theta(t)$, where $t$ is the combined number of points on $o_1$ and $o_2$. Note that it is possible to compute both the merge lines and the result of the merge in a single pass, however we decided to keep the computations separate as we use the merge segments separately for other features.



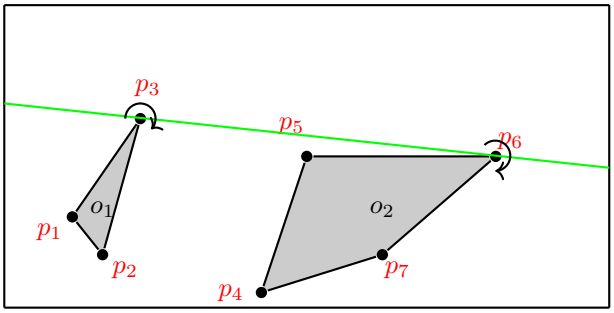

(a) Initial vertical state of calipers.



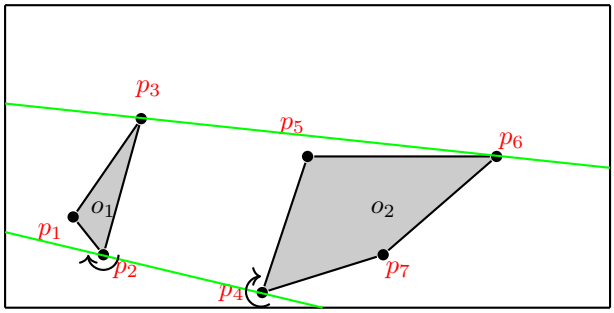

(b) Reaching $p_5$.



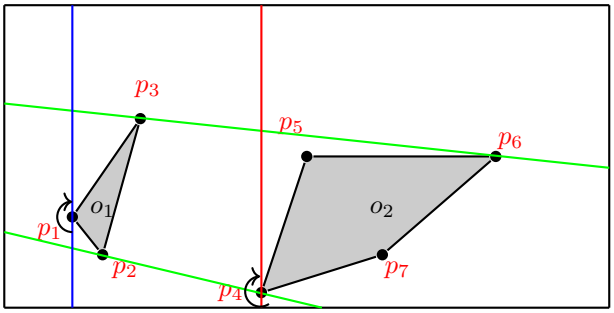

(c) Reaching $p_3$.




(d) Reaching $p_6$ and then the first caliper coincidence.




(e) Passing through $p_7$, $p_2$, reaching $p_4$ and the second caliper coincidence.



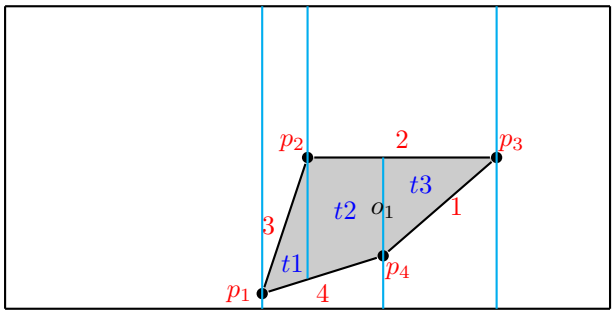

(f) Reaching $p_1$ again and finishing the rotation, reaching initial position.

Figure 2: Example depicting how the merge lines are found. The calipers start out vertically at the leftmost points and "roll" around a convex object each. Whenever the calipers coincide, a merge line is found.

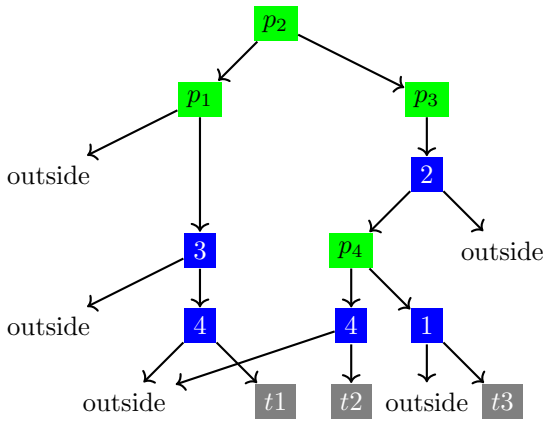
### 3.2 Dynamic Vertical Decomposition

Point location in ConvexMerger is used to determine whether the location that the user has clicked on is inside a convex object and if so – which object was clicked on. Initially, point location queries were done naively, where each query point $p$ was checked for containment in every convex object, resulting in a runtime equal to $O(m)$, since each segment is part of one object.

Before the start of this project, we implemented a Vertical Decomposition structure that partitions the playfield into trapezoids whose parallel segments are both vertical. Every query point $p$ is consecutively tested whether it lies left or right of points and below or above segments, in order to determine which trapezoid $p$ lies within. The control sequence of which points and segments are to be checked is stored within a search structure, which is a directed acyclic graph that has an expected height of $O(\log m)$ whenever the random incremental construction (RIC) approach is used to initialise the decomposition. Lastly, whenever the trapezoid that $p$ lies within is discovered, the convex object that the trapezoid lies within is considered. Each trapezoid is part of at most one convex object at a time, and as such in ConvexMerger we represent this link in the form of a mapping function from trapezoids to convex objects. Figure 3 shows an example vertical decomposition of a single object and Figure 4 shows how the decomposition visualisation looks in ConvexMerger on a playfield with some executed merges.




(a) Example of a vertical decomposition on a single convex object.




(b) The search structure for $o_1$ when the segments are inserted in the order $2, 3, 4, 1$. Note how both 4s point to the same "outside" trapezoid.

Figure 3: Example of a vertical decomposition for a single convex object. The outside trapezoids are not named here.

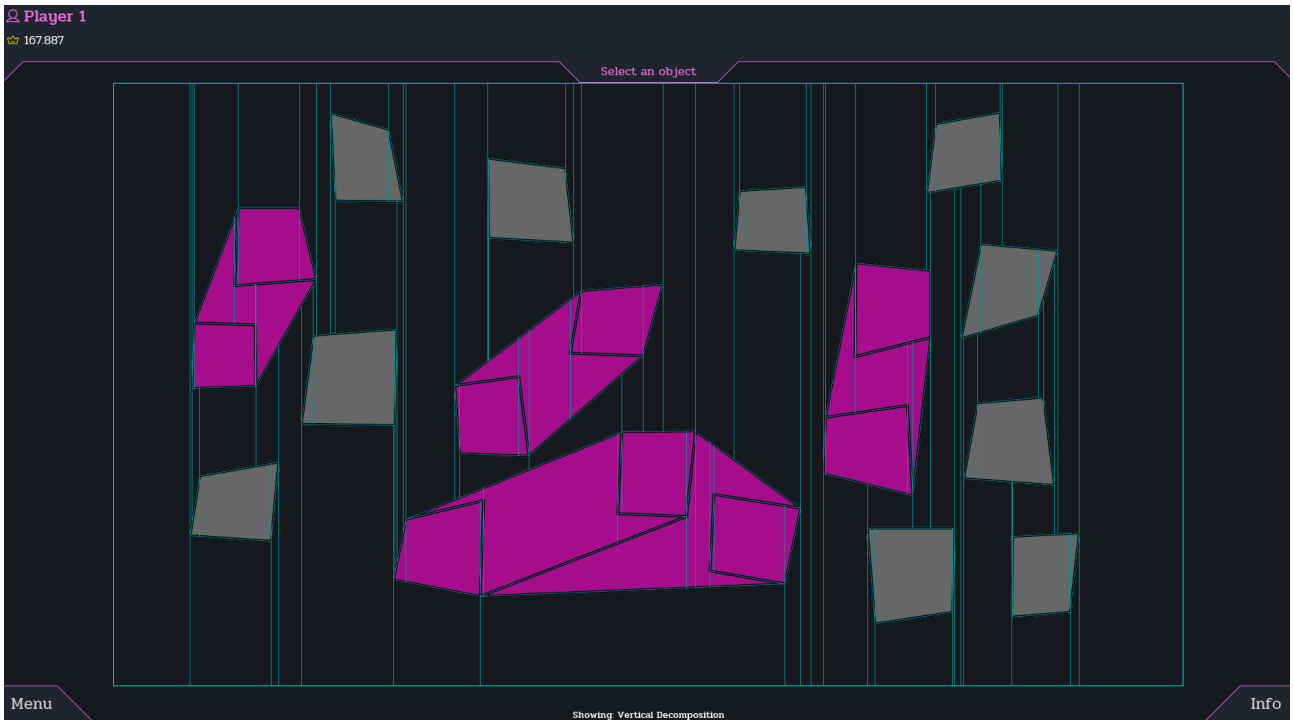



Figure 4: The visualised vertical decomposition of a playfield.

If the RIC approach for vertical decomposition is followed, the vertical decomposition construction time is expected to be $O(m \log m)$ [17]. Point location queries can be answered by going down the search structure in $O(\log m)$ expected time. However, at the start of the project, the decomposition was static, which meant that whenever a new segment was added to the decomposition, which is the case with merges, it would have to be rebuilt. Rebuilding after every merge means that updates to the decomposition have the same $O(m \log m)$ expected runtime as construction.

Here we will note that the RIC approach works under the assumption that there are no two points on the same $x$-coordinate, which bounds the number of neighbours of each trapezoid to at most four. The assumption can be lifted by defining a topological order on the points or by using the perturbation technique, that means to slightly shift all points along the $x$-axis based on $y$-coordinate until no two points are on the same $x$-coordinate. In ConvexMerger, the assumption is not used nor lifted, however occurrences where two points lie on the same $x$-coordinate, with different $y$-coordinates, are rare. Having more than two points on the same $x$-coordinate that are not separated by a segment is an even rarer occasion. Such cases are rare in practice, and no more than a few have been spotted per playfield, thus they carry little impact to the runtime of the decomposition

construction, queries, and updates.

The improvement to the vertical decomposition implemented in this project is to dynamize the structure. Instead of rebuilding the whole decomposition upon merging objects $s$ and $t$, only the merge segments $m_1$ and $m_2$ are inserted into it. This creates new trapezoids and requires extra bookkeeping for all affected trapezoids. After the new trapezoids are created, the trapezoids affected by a merge are all trapezoids within the resulting convex hull, that is all trapezoids within $s$ and $t$, and the area enclosed between $s$, $t$, $m_1$, and $m_2$. The extra bookkeeping is in the link between trapezoids and the convex object they are part of. Whenever a merge between two objects happens, the resulting convex object $r$ is considered a new object, so the affected trapezoids must be remapped towards $r$.
Another possible improvement that we did not implement as part of this project is to rebuild occasionally. This would be done to maintain the balance of the decomposition search structure. Using the RIC approach, the height of the search structure (a tree with internal nodes representing points and segments, and leaves representing trapezoids) is expected to be $O(\log m)$ after construction. After each segment addition, parts of the tree get deeper, which can disbalance the search structure and invalidate the $O(\log m)$ query time guarantee. To rebalance the decomposition, it can be rebuilt after some number of updates. This number can either be fixed or dynamic. For a static number of updates if the decomposition is rebuilt every $\log m$ merges, the $O(\log m)$ expected height is maintained. Alternatively, a criterion for disbalance can be introduced, e.g. average or maximum height difference between leaf nodes, and the decomposition can be rebuilt after a specific threshold for disbalance is reached.

**Analysis** The beginning point of the analysis on the cost of updates is the addition of the inserted segment $s_i$, which takes time linear in the total number of segments in the worst case, since there are $O(m)$ trapezoids total, each of which $s$ could intersect. Then, the updates to the search structure happen in constant time for each intersected trapezoid. Finally, during the update of the trapezoid-to-object mapping, no more than $O(m)$ trapezoids will be visited in the worst case, as each trapezoid is visited at most once.

The vertical decomposition is (re)built in $O(m \log m)$ time. Rebuilding every $x$ merges takes an amortised $O(\frac{m \log m}{x})$ time per merge. Therefore, when $x$ is chosen equal to $m$, the amortised rebuild time is $O(\log m)$. Seeing as merge segment insertion takes linear time, rebuilds can happen more often, and rebuilding once every $\log m$ updates would result in $O(m)$ amortised time per update. Middle points between these values exist, such as $x = \sqrt{m}$ resulting in amortised $O(\sqrt{n} \log n)$ update times. The value of $x$ should be chosen depending on the expected query-to-update ratio.

The analysis above shows that the improvements to the vertical decomposition in this project reduce the update time from $O(m \log m)$ to $O(m)$, and shows an approach to ensure the balance of the search tree and the $O(\log n)$ query time at the expense of updates with $O(m)$ amortised runtime.

## 3.3 Efficient Segment Intersection Testing

Segment intersection testing in ConvexMerger is used whenever a merge is attempted to check whether the merge lines between the source and target object intersect the segments of other objects. This is used both for human and AI players. In fact, each AI makes a multitude of such checks per turn before choosing their move.

Trivial segment intersection testing, by checking whether the query segment intersects each segment on the playfield, takes $O(m)$ time. To improve upon this, we implemented two segment partition trees, as presented by Overmans et al. [15]. One is based on kd-trees [3], and the other is based on conjugation trees [8]. It was not necessary to implement both, however we first implemented the kd-tree as a simpler proof of concept.

**Conjugation trees** Conjugation trees partition their set of points based on a bisector and its conjugate. A bisector is a line (or segment) that splits a set of points into two equal-sized sets, possibly containing some of the points on itself. A conjugate of a bisector is a line that intersects the bisector and is a bisector of both of the equal-sized split point sets. For any given bisector, a conjugate always exists[24].

The construction of the conjugation segment partition tree happens in two parts. First, for the root node $r$ we partition the complete point set into $p_l$ and $p_r$ of equal size using a vertical bisector $b$. Afterwards, for the child nodes we partition respectively $p_l$ and $p_r$ using the conjugate $c_b$ of the parent's bisector $b$. For each child node *chld*, the conjugation tree is recursively built with *chld* as its root node and $c_b$ as its bisector. Figure 5 depicts the bisectors of a possible conjugation tree on a set of 16 points.

**Kd-trees** The construction of the kd segment partition tree is done analogously to the conjugation segment partition tree, with a single difference. Instead of conjugates, the partition alternates between vertical and horizontal bisectors, which are lines that go through the median point(s) in the horizontal (resp. vertical) order of the point (sub)sets. For each visited node $v$, the points are sorted on the on the $x$-axis if $v$ is an even distance from the root, otherwise on the $y$-axis. Then, the median point $m_v$ is picked, and a line perpendicular to the

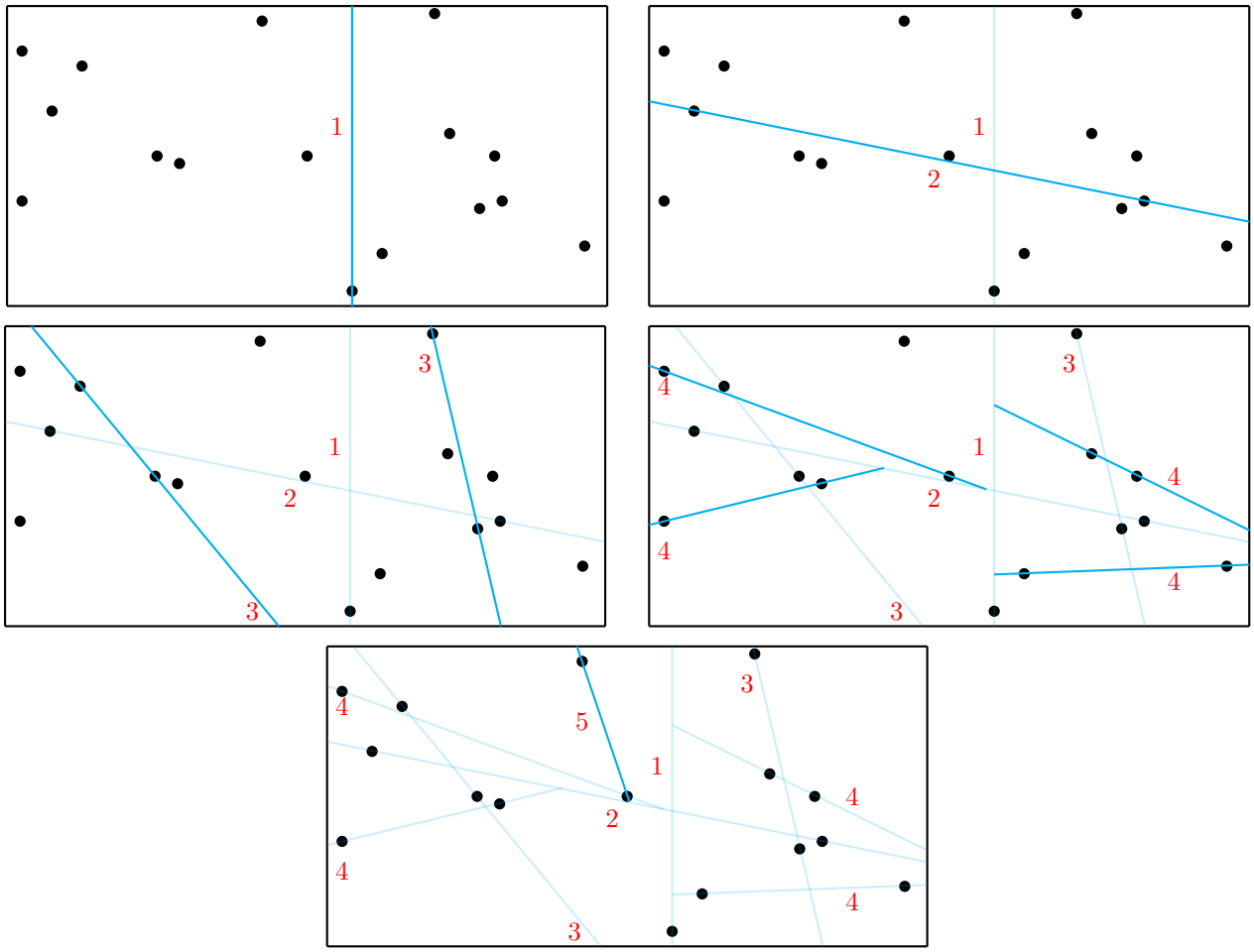



Figure 5: Level-by-level conjugation tree construction on 16 points. The bisectors are depicted in cyan, and the number next to each bisector represents the level of its node. The first bisector is vertical, and every next bisector is a conjugate of its predecessor.

sort axis is run through $m_v$, bisecting the point set. Afterwards the algorithm recursively executes on a pair of child vertices, containing one of the split subsets each. Figure 6 depicts the bisectors of a possible kd-tree on the same set of 16 points as in Figure 5.

The addition of segments to the two types of trees happens the same way. After the partition trees are constructed, we then insert the segments one by one into the constructed tree, beginning with a visit to the root node. For each visited node $v$, if the segment to be inserted $s$ fully intersects the area of $v$, meaning $s$ intersects $v$'s area but has no endpoints inside it, or if $v$ is a leaf node, we store the segment at that node. Otherwise, we check whether $s$ intersects the bisector of $v$, and if so, we visit both children of $v$. Otherwise only the child that is partially intersected by the segment is visited. Later updates to the playfield that result in the addition of segments also use this insertion approach.

Intersection testing queries are handled the same way for kd and conjugation trees, starting at the root node. For every visited node $v$, the query segment $s$ is checked for intersection against all segments stored in $v$. If $s$ fully intersects $v$'s area or if $s$ intersects the bisector of $v$ (clipped inside the area of $v$), both of $v$'s children are visited. Otherwise, only the child that is partially intersected by $s$ is visited. If at any point $v$ intersects a segment stored at a visited node, the search concludes.

**Analysis** The construction of *kd*-trees follows the standard algorithm [3]. Sorting the points takes $O(n \log n)$ time at each depth level (as there are $k = 2^i$ nodes of $\frac{n}{k}$ points each). After sorting, the remainder of the work at each node takes constant time, so the total construction time is $O(n \log n \cdot \log n) = O(n \log^2 n)$. Here we can note that the addition of the segments at the nodes takes $O(\log n)$ time per segment, for a total of $O(m \log n)$ time, however since in ConvexMerger $m = O(n)$, this does not impact the asymptotic bound. We can also note that with extra bookkeeping the points can be sorted only twice, once for each axis and filtered in linear time instead of $O(n \log n)$, which would reduce the running time by a factor of $O(\log n)$. This was omitted in ConvexMerger in order to simplify the complexity of the code.
Similarly, the construction of conjugation trees takes $O(n \log^2 n)$, as at the root level a vertical bisector is

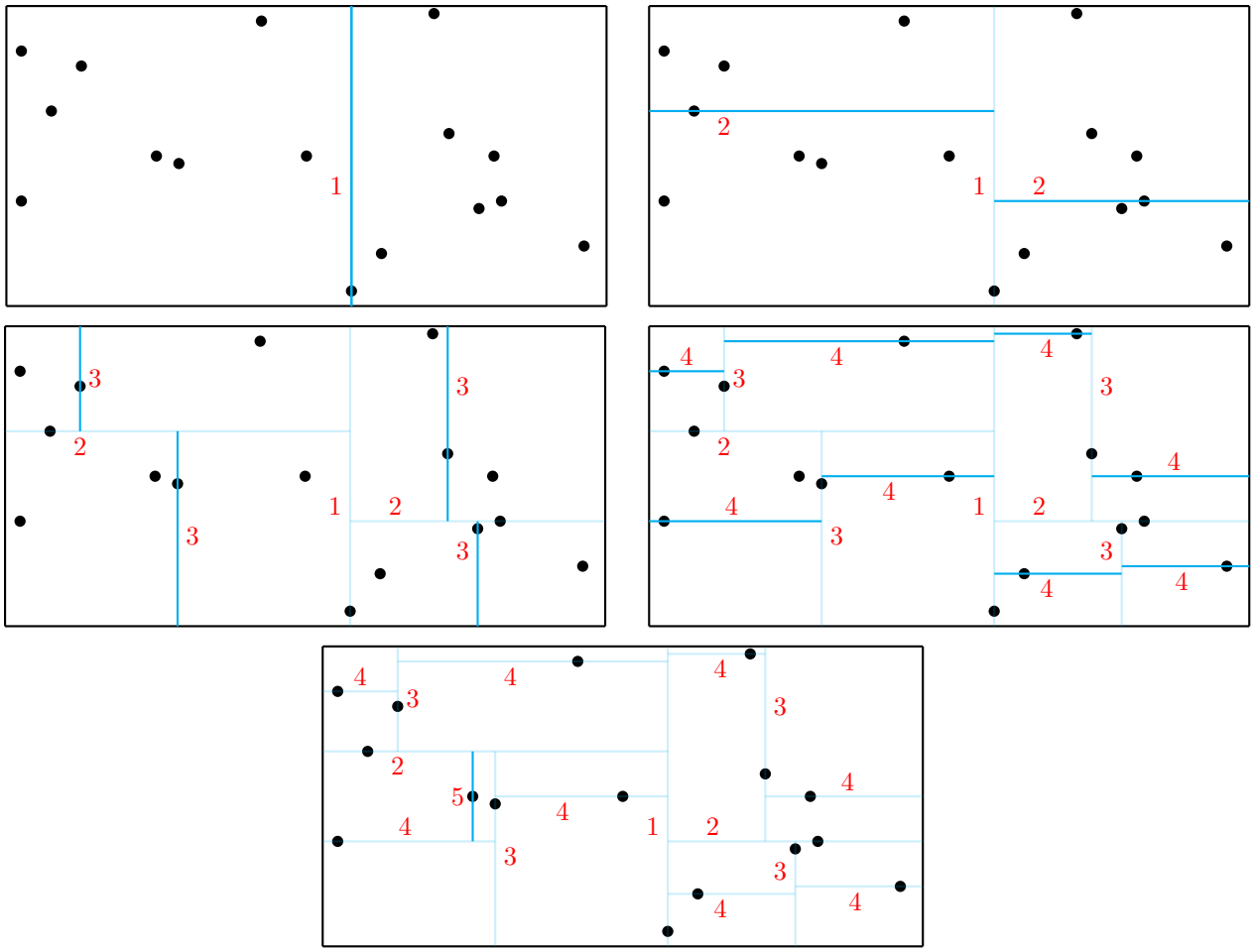


Figure 6: Level-by-level kd-tree construction on 16 points. The bisectors are depicted in cyan, and the number next to each bisector represents the level of its node. The first bisector is vertical, and every next level of bisectors alternates between horizontal and vertical.

computed in $O(n \log n)$ time analogously to the *kd*-tree, and at each internal node a conjugate on $n$ points is computed in $O(n \log n)$ operations (see Section 3.4.1). Similar to *kd*-trees, we take $O(n \log n)$ time at each depth, leading to the aforementioned $O(n \log^2 n)$ construction time.

For kd-trees, segment intersection queries on $m$ segments between $n$ points can take $O(n \cdot m) = O(n^2)$ time in the worst case. The case would be when a lot of segments are all linked to one point. A depiction of how this would look like for convex objects and their merges is depicted in Figure 7.

As can be seen on the top-right of the figure, when all objects are merged in a specific sequence, some node in the kd-tree will store one segment per object. In fact, more than just that node will store these segments. If the sequence of objects is generated at a particularly small angle, the merge segments would be stored in $O(n)$ cells. For a query segment that intersects all of these cells, such as a hypothetical merge segment of the next object following the example structure, this leads to to the aforementioned $O(n^2)$ runtime. Such a case is extremely unlikely in ConvexMerger, where on average each node only contains a small number of segments, as will be shown in Section 4.3. This puts a lower bound on the worst-case within ConvexMerger at $\Omega(n)$, however we were unable to prove an upper bound tighter than $O(n^2)$.

The runtime analysis of segment intersection queries for conjugation trees is based on Edelsbrunner and Welzl's discovery that the worst-case recurrence for queries on conjugation trees can be bounded using its similarity to the fibonacci sequence [8]. They found that in the worst case, for a set of points split by a bisector and its conjugate, any straight segment will intersect at most 3 out of the 4 divided areas of the plane. This is because from any point not on the lines, to enter another area would require intersecting one of the conjugates, and two straight lines can only intersect at most once, see Figure 8 as an illustrative example. The runtime $T(n)$ of a segment intersection query for some segment on a set of $n$ points is then the recurrence:

$$T(n) = T(\tfrac{1}{2}n) + T(\tfrac{1}{4}n) + x$$

Here $x$ is the time spent at each node. Edelsbrunner and Welzl prove that for $x = O(\log m)$, intersection testing

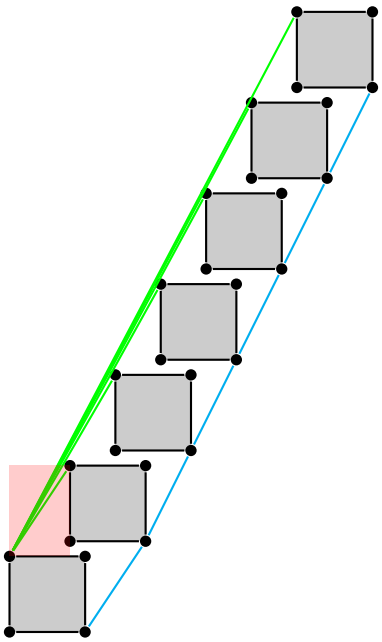


Figure 7: Sketch of worst-case scenario for kd-trees. If the bottom object is consecutively merged with all other objects from left to right, the kd-tree cell coloured in red just above the original bottom object will store a segment for every merge, meaning a segment for every object other than the bottom-most one.

using conjugation trees runs in $O(n^{\log_2 \phi}) = O(n^{0.695})$, where $\phi = \frac{1}{2}(1 + \sqrt{5})$ is the golden ratio. As can be seen in Section 4.3, in ConvexMerger the number of segments stored at a node averages at fewer than 2 for $m > 2000$. This is well within the $O(\log m)$ bound, so even when testing the query segment for intersection with all stored segments in visited nodes, the proof holds and the $O(n^{0.695})$ bound applies.

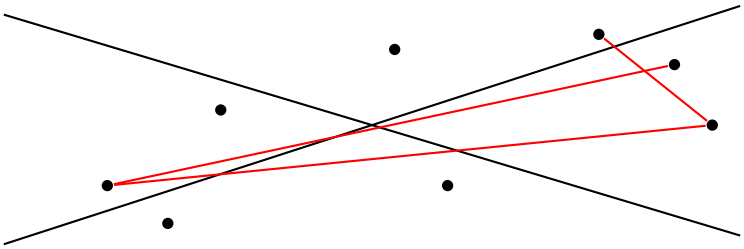


Figure 8: Example of two conjugate bisectors on a set of 8 points. Note how the red segments can not go through 4 of the areas divided by the bisectors.

## 3.4 Miscellaneous Algorithms

In this section we will describe some minor helper algorithms that are interesting to cover, but not necessarily very complicated.

### 3.4.1 Conjugate computation

For the implementation of conjugation trees, a key component is the computation of the bisectors and their conjugates that partition the tree. For a set of $n$ points, a bisector $b$ is a line that separates the point set into a left subset $p_l$ and right subset $p_r$ of equal size (or off by 1). A conjugate of the bisector is then a line $c_b$ that is a bisector of both $p_l$ and $p_r$ simultaneously.

Literature on the topic states that a conjugate always exists [24], and that an $O(n)$ algorithm is possible[24], however we found the algorithm complex to comprehend and implement. As such, we first implemented a naive approach and later came up with an alternate solution that compromises between speed and simplicity.

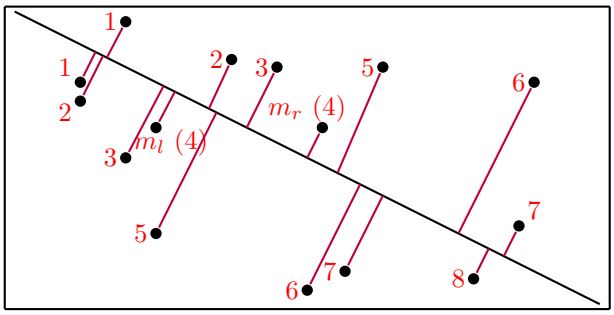
The naive solution to conjugate computation iterates over all pairs of points of the form $(x, y)$ where $x$ is a point in $p_l$ and $y$ is a point in $p_r$. It then considers the line passing through the pair of points as a candidate conjugate $c_{cand}$. For $c_{cand}$ the number of points on each side of it is computed, separately for $p_l$ and for $p_r$. If the number of points on each side differs by at most one for both subsets, $c_{cand}$ is returned as the answer.

The naive solution proved to be the bottleneck of the algorithm, taking up to tens of seconds to generate the conjugation tree on playfields with the highest population settings. To remedy this, we propose an alternative algorithm. The basis for the algorithm is the observation that in the general case, the computed conjugates pass through points that are either the median points of $p_l$ and $p_r$ when sorted on the projection of the points onto the bisector $b$, or that the points were very close to being median in their respective subset.

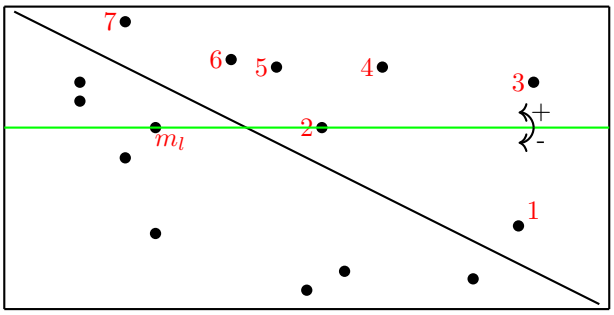
The algorithm starts by sorting $p_l$ and $p_r$ on the projection of each point along $b$. Then, the median points $m_l$ and $m_r$ of the subsets are selected and the line passing through them them is chosen as the first candidate conjugate $c_{cand}$. Afterwards, $p_r$ is sorted radially with relation to $m_l$, and $p_l$ is sorted radially with relation to $m_l$. If $m_l$ and $m_r$ are still the median points of respectively $p_l$ and $p_r$, then $c_{cand}$ is indeed a conjugate of $b$. Otherwise, select a new $c_{cand}$ passing through one of $m_l$ or $m_r$ and the current median of the opposite set, respectively the median of $p_r$ or $p_l$. The process continues until a conjugate is found.

One issue that was discovered was that in some cases of collinearity, the choice of candidates would loop. To tackle this, whenever $c_{cand}$ has already been chosen before, a new candidate is sought, starting with points that are adjacent in index to the current candidate points in the sorted subsets and searching for more distant points until a yet unchosen candidate conjugate is found.

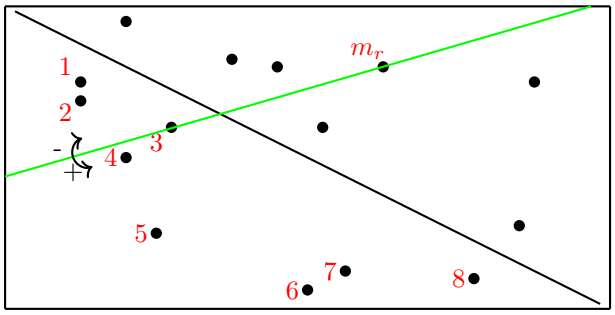
The naive implementation has an asymptotic runtime of $O(n^3)$. This comes from checking $|p_l| \cdot |p_r| = O(n^2)$ pairs of points as defining points of candidate conjugates, with each candidate taking $O(n)$ to iterate the subsets. The proposed algorithm first sorts $p_l$ and $p_r$ in $O(n \log n)$, and then takes $O(n \log n)$ time per candidate conjugate to sort the two subsets again. In the worst case, when all points are collinear, this would take $O(n^3 \log n)$ time total, as all pairs of points may be checked, depending on the initial order. A way to solve this case that we did not implement is to add a third dimension to all points and assign values to each point in that dimension that would define a topological order in case all points are collinear in 2D. This would reduce the worst-case to $O(n \log n)$ as only a constant number of candidates will be considered.



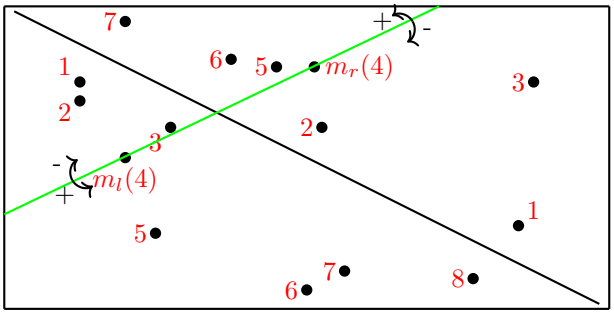

(a) Initial state, including the bisector and the perpendicular segment of each point to the bisector. The segments on each side are numbered in order from top-left to bottom-right.




(b) First candidate conjugate, passing through the median point on each side. The points to the top-right are sorted w.r.t. the angle that the segment from $m_l$ to the point makes with the green line.




(c) Last candidate conjugate, passing through the old $m_l$ the median point on the top-right. The points on the bottom-left are sorted on their angle to $m_r$.




(d) Last candidate conjugate, passing through the old $m_l$ the median point on the top-right. The points on each side are sorted on their angle to $m_l$ or $m_r$. Since both points the line passes through are medians, the candidate is indeed a conjugate.

Figure 9: Conjugate computation on a set of 15 points and an existing bisector.

In our observation of the proposed algorithm's implementation in ConvexMerger we found that typically at most three candidates are considered before the algorithm converges, which confirms the average $O(n \log n)$ runtime. While the proposed algorithm does not match the best achieved theoretical running time bound of $O(n)$, it is easy to implement and it is intuitive. An example showing the computation of a conjugate with this

algorithm is shown in Figure 9.

### 3.4.2 Line extension and clipping

In order to visualise how the playfield space is divided by conjugation trees, there is a need to resize the partitioning segments to the bounds of the cell that they split. These operations are not necessary for *kd*-trees, as all partitioning line segments are orthogonal to the axes and each cell is a rectangle.

Before delving into line extension and clipping, it is important to note that the underlying data representation mentioned in Section 3.4.1 is not actually a line, but a line segment between points, as that is sufficient for finding out through which points the conjugate passes.

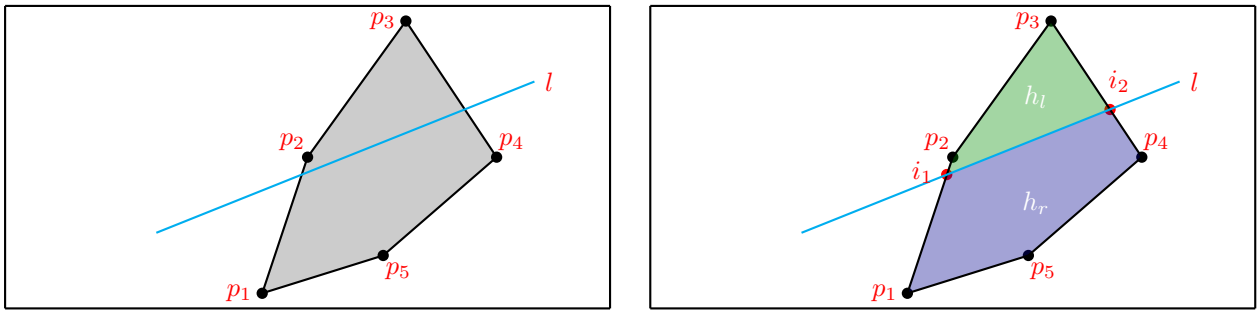
**Extension to the ends of the playfield** In order to obtain the final conjugate line within the bounds of the playfield from the chosen candidate segment, first the slope $s$ and the intercept $b$ at the zero $x$-value are computed from the segment. Then, the $x$-coordinate of the two points $p_1$ is computed as $x_1 = \min(playfield\_width, \max(0, \frac{-b}{s}))$ and $x_2 = \min(playfield\_width, \max(0, \frac{playfield\_width - b}{s}))$. The minimum and maximum operations are included to ensure that the $x$-coordinates are within the bounds of the playfield. The $y$-coordinates are then computed as $y_1 = b + s \cdot x_1$ and $y_2 = b + s \cdot x_2$. The line segment $((x_1, y_1), (x_2, y_2))$ is then guaranteed to end at the boundaries of the playfield due to the way that the $x$-coordinates were computed.

**Clipping to the area of a node** A conjugation tree node's area is bound by the playfield bounding box and by the bisectors of the node and a (non-strict, possibly empty) subset of its ancestors. For the node and each of its ancestors in order, the line to be clipped $l$ is tested for intersection with each of the bisectors $b_i$. If $l$ intersects with a $b_i$ at point $p$, $l$ is limited to $p$ in the direction that heads past the bound of the node's area. When all ancestors are visited, the line will be clipped down to the node's area.

### 3.4.3 Hull splitting

The conjugation trees presented in Section 3.3 partition the playfield on each side of multiple segments. One way to look at the playfield is as the convex hull of four points that form a rectangle. Each bisector of the partition tree splits the hull into smaller hulls that cover the same area. Hull splitting takes a hull $h_o$ and a splitting line $l$, and splits $h_o$ into two hulls $h_l$ and $h_r$ that cover the exact same area as $h_o$ such that the points of $h_l$ are on one side of $l$ or on $l$ and the points of $h_r$ are on the other side of $h_r$ or on $l$. Hull splitting is used to efficiently maintain the areas that the playfield is split into by the conjugation trees. The resulting hulls are used in the visualisations in Section 5.2 and for the segment intersection testing in-game animations.

The hull splitting algorithm first iterates through the points of the original hull $h_o$ until a point $p_i$ is found that is on a different side of the splitting line than the previous visited point $p_{i-1}$. This indicates that there is an intersection at a point $i_1$ between $l$ and the segment $(p_{i-1}, p_i)$. Since it is on both hulls, $i_1$ is added to both $h_l$ and $h_r$. Then, iteration continues until again a point $p_j$ is found on a different side of $l$ compared to $p_{j-1}$. This indicates that $l$ intersects $(p_{j-1}, p_j)$ at $i_2$. All points from $p_i$ up to but excluding $p_j$ are added to $h_r$ in order of iteration. Then, $i_2$ is added to both $h_l$ and $h_r$. Finally, the remaining points, from $p_j$ up to and including the final point on the hull are added to $h_l$, followed by all points from the first point on the hull up to but excluding $p_i$. In the edge case that the splitting line intersects the hull at a single point (or not at all), the algorithm immediately outputs a zero-area shape and $h_o$. A picture of how a hull is split in two by a line can be seen in Figure 10.




(a) The original hull $h_o$ and the splitting line $l$

(b) $h_o$ split into $h_l$ and $h_r$.

Figure 10: Hull splitting example on a hull with 5 points.

As can be seen from the description, the algorithm features at most two passes of all $n$ points on the hull of $h_o$. Therefore, the running time of this algorithm is $\Theta(n)$.

### 3.4.4 Helper line computation

Whenever a player clicks on an object they own, two helper lines project from the clicked object towards the cursor and move along with it. The helper lines are useful for players to more easily estimate whether a merge is possible by moving the cursor around. A picture of how this looks in-game can be seen in Figure 11.

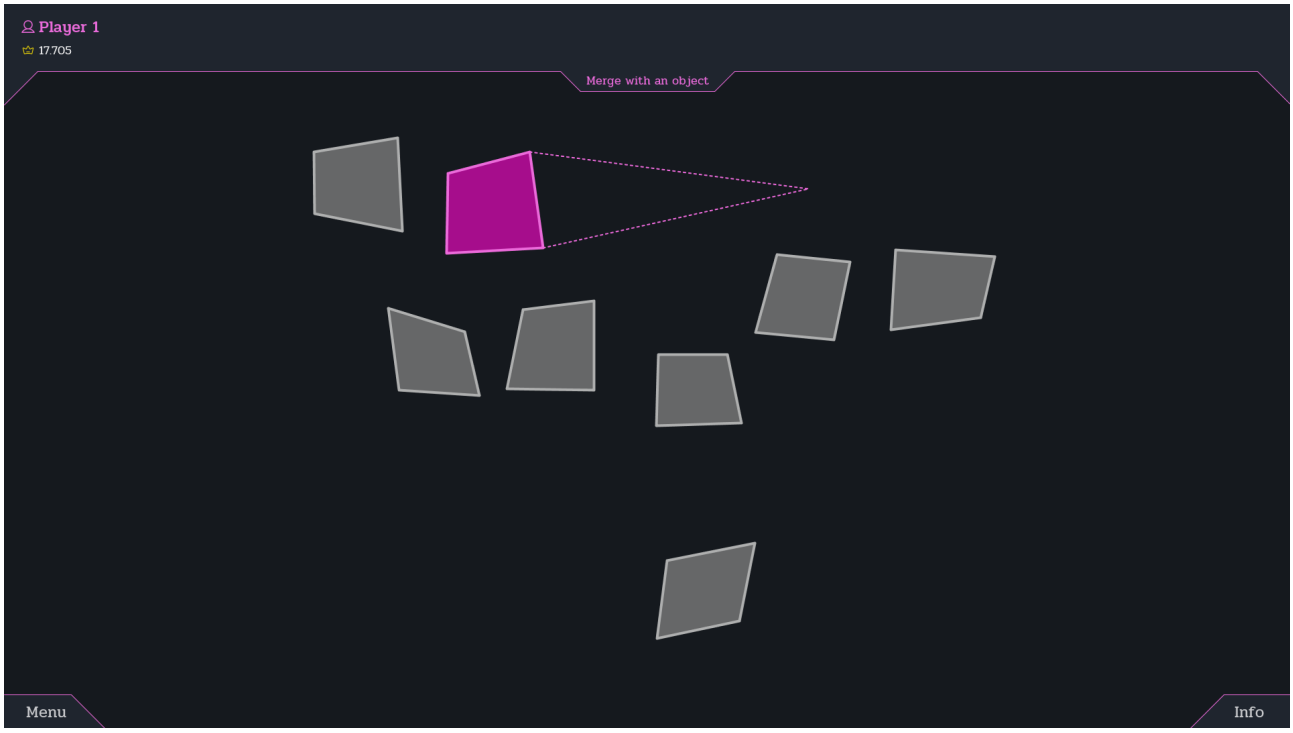



Figure 11: In game view of the helper lines.

The lines are computed by simulating a merge between the selected object $obj$ and the point $p$ at which the cursor is located, using a modified version of the caliper approach described in Section 3.1. The helper lines are exactly the merge lines between $obj$ and $p$. The rotation of a caliper $c$ is simulated, starting from a vertical position at the leftmost point of $obj$ and rotating around $obj$ by iterating its points in order. We keep track whether the orientation of $p$ changes with respect to $c$ as $c$ rotates. Again, as with the convex object merging approach, the two merge lines are found exactly whenever the orientation changes (see Figure 2). Since the goal is to find the merge lines, this algorithm can stop as soon as the second merge line is found instead of completing the rotation.

# 4 Evaluation

In this section we will evaluate the performance and characteristics of the data structures we have implemented. The goal of this section is twofold, the first goal is to explore interesting questions that arose during development, and the second goal is to justify some assumptions that were made in the rest of this report.

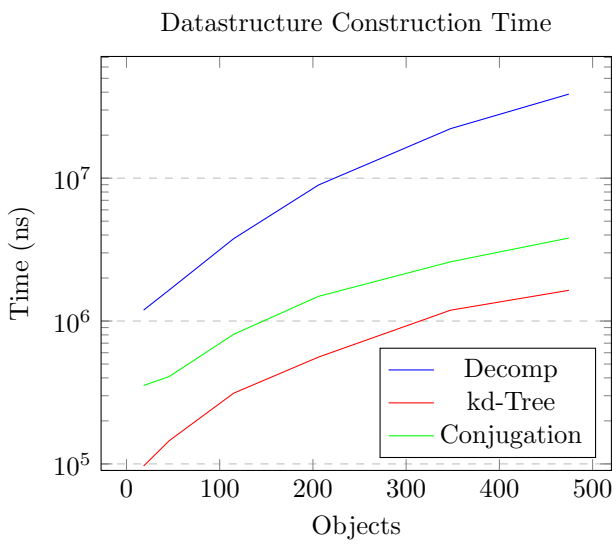
## 4.1 Construction Characteristics of Partition Trees

Although the primary goal of this section is to explore if there are any notable differences between kd-trees and conjugation trees, the vertical decomposition also fits the definition of a partition tree. As such, it is theoretically possible to make a segment partition tree that is based on a vertical decomposition. Therefore, the vertical decomposition will also be included in the analyses in this section. For the evaluations in this section we will make use of six datasets, each containing 100 playfields representative of possible games. For each of these datasets the playfield density and spacing are set to medium while the object size is varied from small to large. We used seeds with custom object sizes to generate the playfields between the in game medium and large setting. Note that the size of objects is inversely correlated with the number of objects that can be placed on a playfield. Some statistics for the average sizes of these datasets are shown in Table 1 and the datasets themselves are available in Appendices A.1, A.2, A.3, A.4, A.5 and A.6.

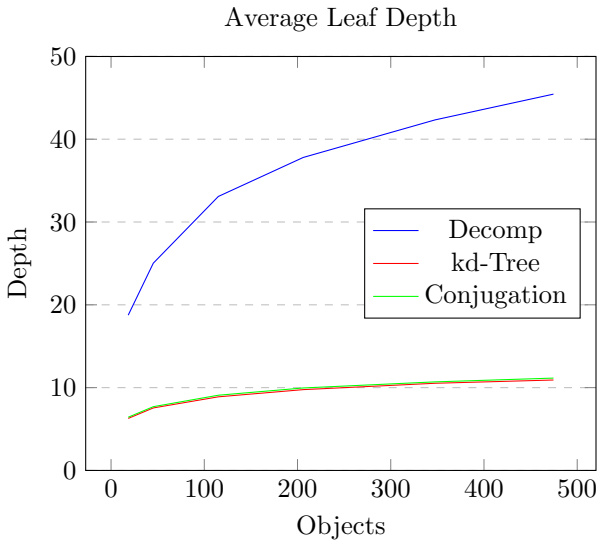
| Dataset | Objects | Points | Segments |
|---|---|---|---|
| Large | 18.41 | 73.64 | 92.05 |
| Medium | 45.34 | 176.41 | 221.75 |
| 20-40 | 115.07 | 460.28 | 575.35 |
| 15-30 | 206.11 | 824.44 | 1030.55 |
| 10-25 | 347.59 | 1390.36 | 1737.95 |
| Small | 474.53 | 1898.12 | 2372.65 |

Table 1: Average sizes of the playfields in the datasets

Figure 12a shows the data structure construction time increasing when the number of objects increases. Judging from the plots all data structures have similar runtime scaling, which for all should be $O(n \log n)$ ($n$ being either points or segments), the plots appear to be in line with this runtime. The average depth of a leaf node for each of the structures is shown in 12b. Here we see that kd-trees and conjugation trees have an average depth that corresponds almost exactly with the optimal height of these structures, which is $\log(n)$. Notably, conjugation trees are slightly deeper, presumably due to the fact that the point set does not always need to be split in two parts of exactly equal size. However, the vertical decomposition seems to be deeper than expected, this issue is explored further in Section 4.4.




(a) Construction time by object count.




(b) Average leaf node depth by object count.

Figure 12: Datastructure construction scaling statistics.

## 4.2 Average depth of segments in Segment Partition Trees

One core feature of segment partition trees is that segments can be stored at higher levels in the tree when they fully intersect a cell of the induced partitioning. This feature means that a query potentially has no need to travel all the way to a leaf node to determine if an intersection exists. While this feature is not very useful with regard to human players, who presumably make mostly valid moves without intersections and thus always cause the search to hit leaf nodes. For AI players this is different, as they require the segment partition tree to determine if a move is legal. As such it is interesting to investigate where segments are stored in our segment partition tree implementation. Moreover, it will be interesting to see if there are any major differences between kd-tree and conjugation tree based segment partition trees. For this investigation we make use of 100 games and record the average number of segments stored at each level of the segment partition tree before and after the games. The games will be played by two greedy AIs (Isla) and the game seeds are available in Appendix A.7.

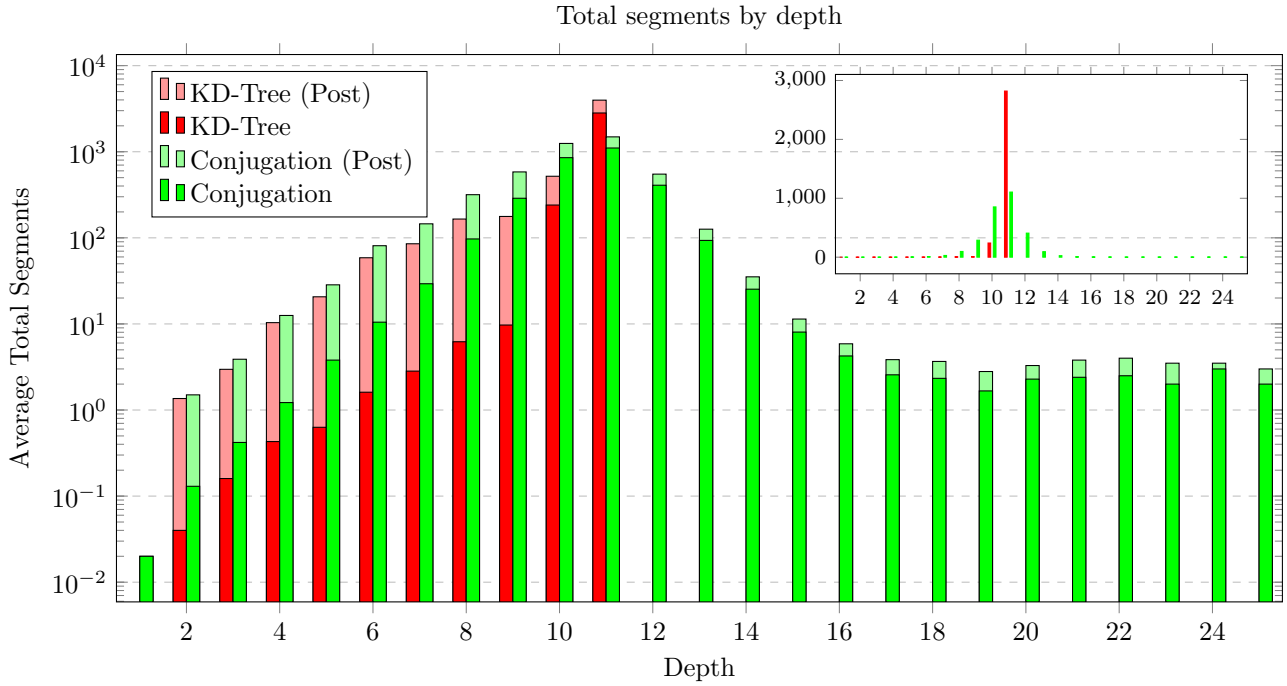



Figure 13: Distribution of stored segments across the layers of KD-tree and conjugation tree based segment partition trees before and after a game, based on the dataset in Appendix A.7.

Figure 13 shows the average number of segments stored at each level before and after the mentioned games, post game averages are marked with 'Post'. Due to the large variation in where segments are stored a log scale is used for the main figure. However, to provide a better view of the distribution a miniature plot without log scaling is provided in the top right corner. From this figure it becomes clear that kd-tree based segment partition trees store nearly all segments at leaf nodes. Conjugation tree based segment partition trees on the other hand appear to store segments in a normal distribution around the average leaf depth. We currently do not have a good explanation for this difference, so this might be interesting to further investigate in the future. Finally, it is worth noting that during a game more segments get added to higher levels of the tree. This is an expected result as good merges are as large as possible and thus naturally have longer merge lines. These long merge lines then have a higher chance of fully interesting a high cell in the induced partitioning, allowing it to be stored higher.

## 4.3 Segments per node in Segment Partition Trees

One core assumption on which we based our runtime analysis for segment partition trees in Section 3.3, is that a relatively small number of segments will be stored at any given node in the segment partition tree. To validate this assumption we have computed the average number of segments stored at each depth level of the segment partition tree across 100 games. In order to get an idea of how this number changes over the course of a game we let each game be played to completion by two greedy AIs (Isla) and the game seeds are available in Appendix A.7. The results of this experiment can be seen in Figure 14, the post-game results are marked with 'Post'.

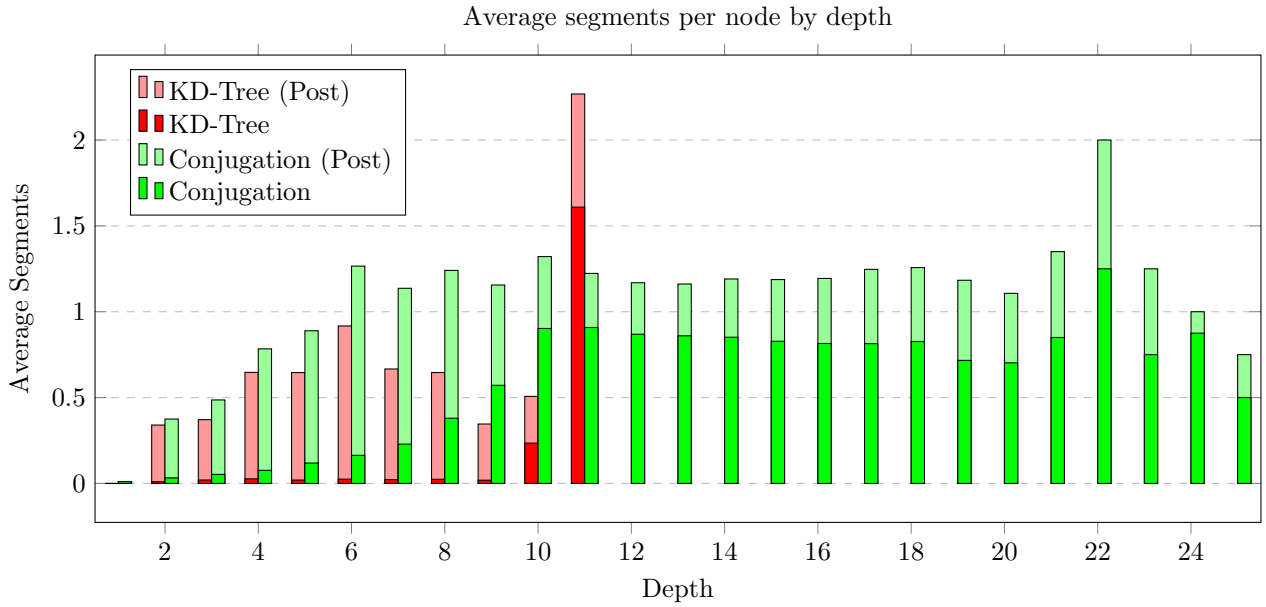



Figure 14: The average number of segments stored at each layer of KD-tree and conjugation tree based segment partition trees before and after a game, based on the dataset in Appendix A.7.

It is clear from these results that our assumption is valid, with the average number of segments per node rarely reaching 2. Furthermore, there are no major outliers. On average, before the game the maximum number of segments stored in a single cell was 5.26 for kd-tree based segment partition trees and 3.7 for conjugation tree based segment partition trees. After the games were played this changed to 8.55 for kd-tree based segment partition trees and 6.59 for conjugation tree based segment partition trees. Although these numbers are larger than the averages, they are not indicative of any major outlier issues and also do not invalidate our assumption for the runtime analysis. Notable is that the for both kd-tree and conjugation tree based segment partition trees a comparatively large number of segments is added to the higher levels of the tree during a game. This can be explained by the fact that primarily long line segments get added during a game. Since the goal of the game involves finding the largest merges possible, the added merge lines are also generally longer than the segments that make up the initial set of convex objects. As a result these segments are more likely to fully intersect the larger higher level cells of the segment partition tree, which is required for them to get stored at these levels. Finally, as already noted during the analysis of where segments get stored, using kd-trees results in a relatively high number of segments being stored at the leaf nodes. Consequently, the average number of segments stored at the leaf nodes for a kd-tree based segment partition tree is also relatively high.

## 4.4 Trapezoid depth in the Vertical Decomposition

As mentioned in Section 3.2 our implementation of the vertical decomposition does not re-balance itself during the game. As such, it is interesting to investigate if it would be worthwhile to implement techniques to keep the search structure more balanced (e.g., rebuilding when the structure becomes too unbalanced). For our investigation we observed 100 games and measured the average depth of all trapezoids before and after each game. The games were played by two greedy AIs (Isla) and the game seeds are available in Appendix A.8. Since the search structure for the vertical decomposition is a DAG there could be more than one path from the root to a trapezoid, we always use the longest path to compute the depth. Table 2 shows the average total number of trapezoids and the average depth of an individual trapezoid before and after the games. Figure 15 shows how the trapezoids are distributed depth wise through the vertical decomposition before and after the games.

| State | Total Trapezoids | Average Depth |
|---|---|---|
| Pre-game | 3607.04 | 44.61 |
| Post-game | 4152.20 | 48.08 |

Table 2: Average vertical decomposition statistics before and after the games.

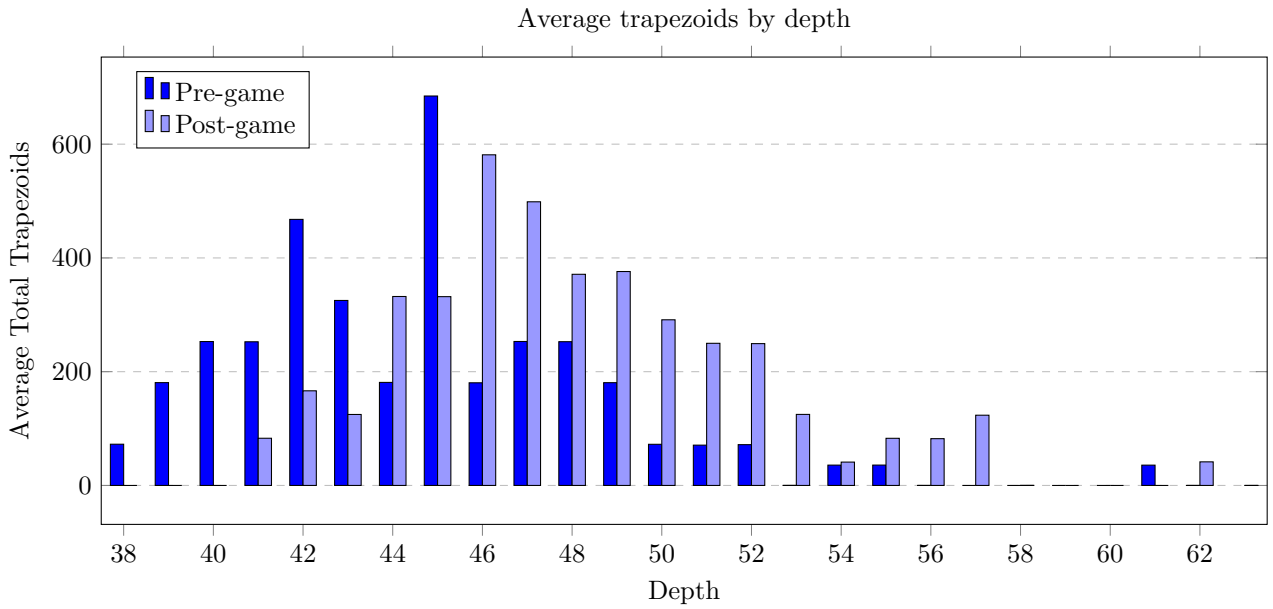



Figure 15: Maximum depth of vertical decomposition trapezoids before and after a game, data is averaged across the 100 games in Appendix A.8.

From these results we can surmise that the average depth of trapezoids in the search structure increases by approximately 3.47. However, considering that most trapezoids were already stored at a depth of 38 or deeper, this is not a very significant increase. Moreover, the merges that take place during a game naturally make the number of trapezoids grow, meaning at least some increase in search structure size is expected. Thus, we do not believe that there is much value in re-balancing the vertical decomposition during a game. However, the vertical decomposition does seem to be deeper than one would expect. Given that $\log(4200) \approx 12$, the search structure's depth of $O(\log(m))$ has a constant factor significantly above 3. This is partially attributed to the depth increasing by at most 3 per insertion, as is the case when an inserted segment lies entirely within a trapezoid. Another commonly seen case is when one segment endpoint is in one trapezoid and the other is in another trapezoid, which adds the newly created trapezoids at a depth of 2 deeper than the previously existing ones. One of these cases occurs at almost every segment addition, which would explain the constant factor being between 2 and 3. The constant factor being above that can be attributed to a poor ordering of the segment additions during initialisation that in turn leads to an imbalanced search structure. Consequently, an improved initialisation approach, such as true random initialisation[2], might be more beneficial to look into.

# 5 Visualisations

In this section we will discuss some of the data structure visualisations and animations that were implemented in the game. Generally, these visualisations are implemented as overlays on top of the normal playfield that can be toggled on or off with a keybind. For users an overview of all these keybinds is available on the 'Info' tab, which can be accessed from either the main menu or the bottom right corner during a game. It is also possible to enable multiple overlays at the same time, with the names of active overlays being displayed along the bottom edge of the playfield. However, since we do not precompute or cache a lot of geometry for these data structures game rendering performance can suffer substantially, especially when enabling a lot of visualisations or when the playfield has a lot of objects. Moreover, some combinations of overlays are too chaotic to read properly. Next follows a quick overview of all implemented visualisations and their keybinds.

**1)** ctrl + C Shows the centroids of the convex objects. This visualisation is not particularly interesting, so we will not dedicate a section to it. However, centroids play an important role in playfield generation and the object claim animation.

**2)** ctrl + D Shows the vertical decomposition, further discussed in Section 5.1.

**3)** ctrl + S Shows the conjugation tree based segment partition tree, further discussed in Section 5.2.

**4)** ctrl + K Shows the kd-tree based segment partition tree, further discussed in Section 5.2.

**5)** ctrl + M Enables the merge calliper animation for merges, further discussed in Section 5.3.

[2]Note that we currently add *objects* in a random order, which are essentially sets of *segments* closely located together.

## 5.1 Vertical Decomposition

The visualisation of the vertical decomposition can be toggled on or off by pressing ctrl + D. When shown the bounding box and the vertical lines that make up the vertical decomposition are rendered in cyan and all segments that were ever added to the decomposition are drawn in black. Incidentally, because segments are never deleted from the vertical decomposition, it is possible to see how large objects were created via merges from the original playfield objects. Adding segments to the vertical decomposition is done with a slight delay when the visualisation is active to make it slightly easier to see the effect of an individual insertion. However, this feature is primarily used to animate the vertical decomposition being built from scratch. On the new game menu it is possible to toggle the vertical decomposition animation using ctrl + D. When a game is started with the visualisation active the user can then see the vertical decomposition being built segment by segment in real time. An example of a vertical decomposition visualisation is shown in Figure 16.

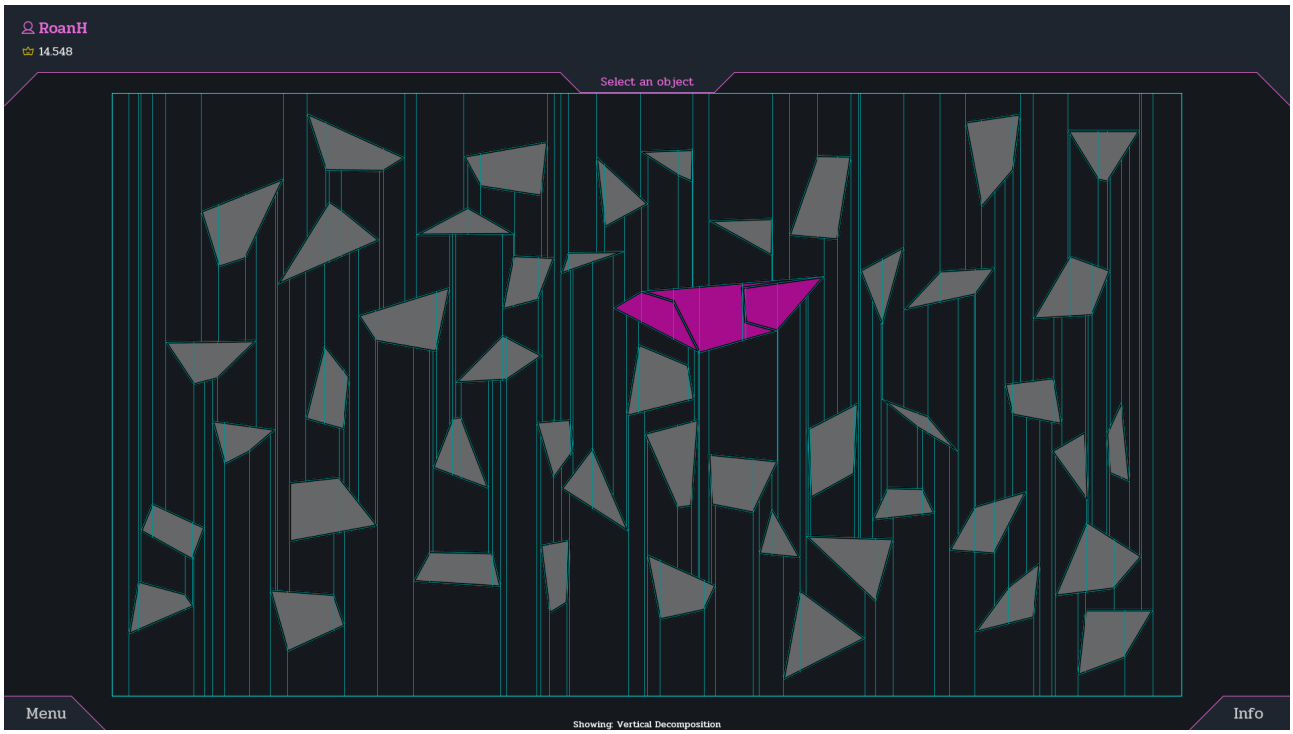



Figure 16: In game visualisation of the vertical decomposition.

## 5.2 Segment Partition Tree

Since the game includes two implementations of a segment partition tree there are also two visualisations. The visualisation of a kd-tree based segment partition tree can be shown by pressing ctrl + K and the visualisation of a conjugation tree based segment partition tree can be shown by pressing ctrl + S. Functionally, both visualisations work the same. For the underlying base partition tree the points from the point set that the partition tree was built from are shown in blue, the structure of the partition tree itself is shown in cyan with the dividing lines of deeper tree cells using a darker shade of cyan. The segments contained in the segment partition tree are drawn in black. Similar to the vertical decomposition visualisation, this means it is possible to see how larger convex objects were constructed. A segment partition tree visualisation based on a kd-tree is shown in Figure 17 and a segment partition tree based on conjugation trees is shown is show in Figure 18.

When a segment partition tree visualisation is active it will animate queries that are executed. However, due to technical challenges and usability concerns only queries that result in a segment being added to the tree are animated. Consequently, this means that no visualised queries ever find an intersection. However, this also means that the animation always continues all the way to the leaf nodes of the tree, clearly showing which cells are checked. The animation itself works as follows. The query segment is shown in blue at all times. The animation then shows in sequence which cells are searched at each level of the tree from root to leaves, with a slight delay before continuing to the next tree level. At each level all the cells being checked are shaded red, any segments checked for intersection in these cells also light up in red. It is worth noting that the animation can be cancelled at any time by simply disabling the visualisation overlay.

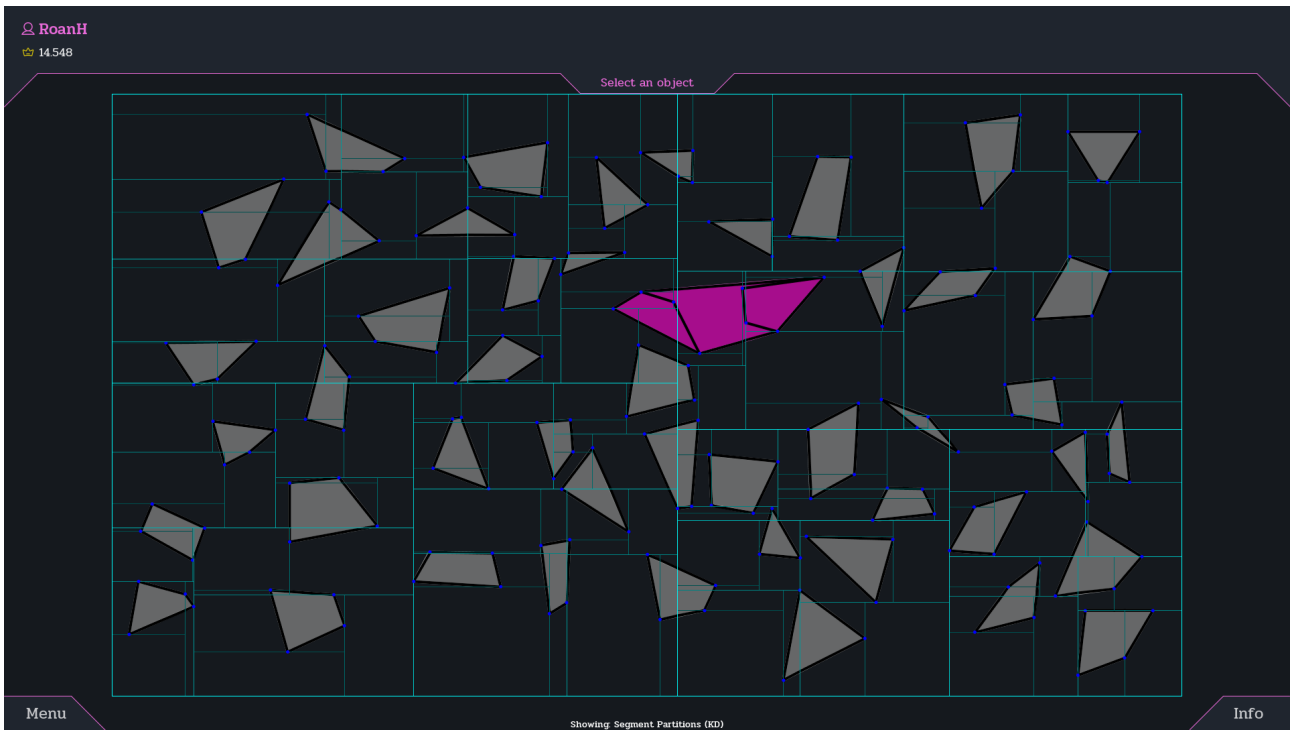



Figure 17: In game visualisation of a kd-tree based segment partition tree.

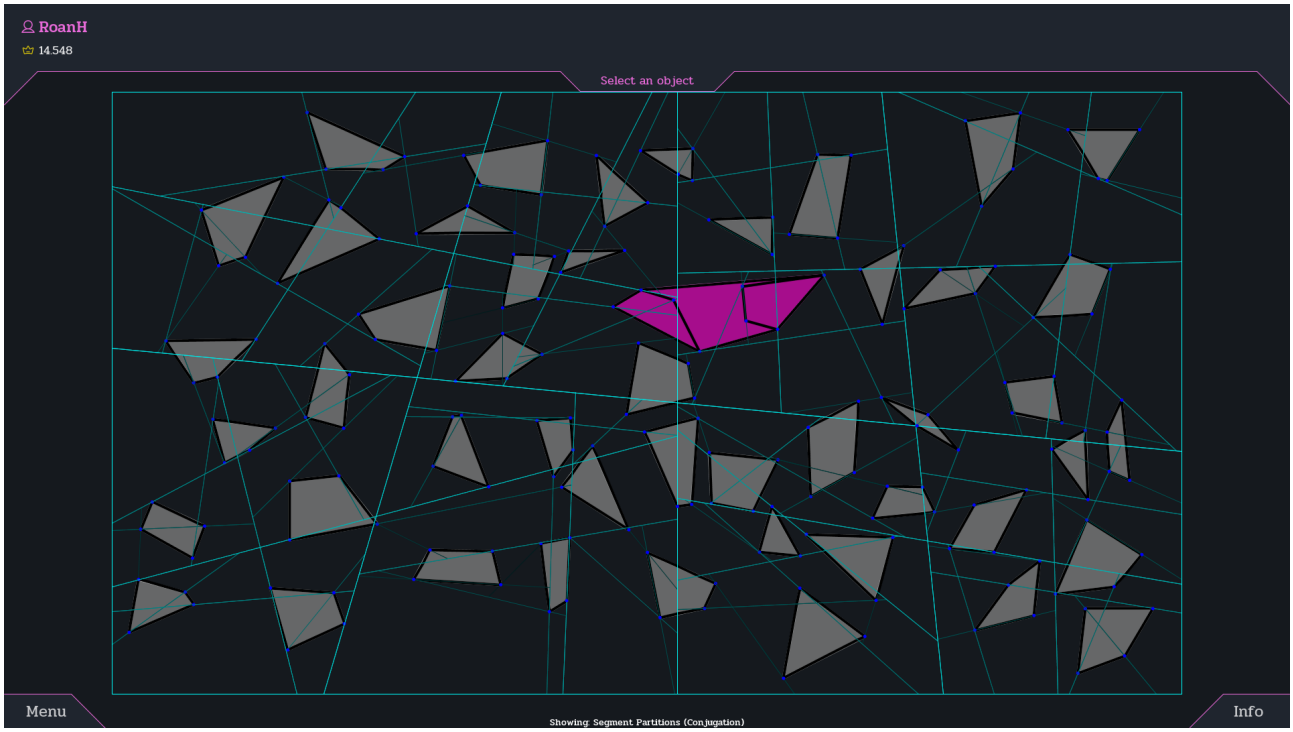
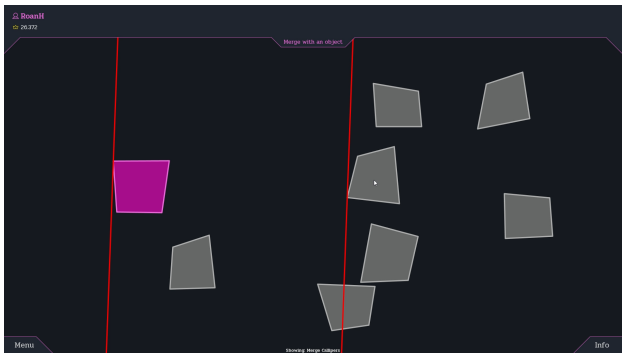


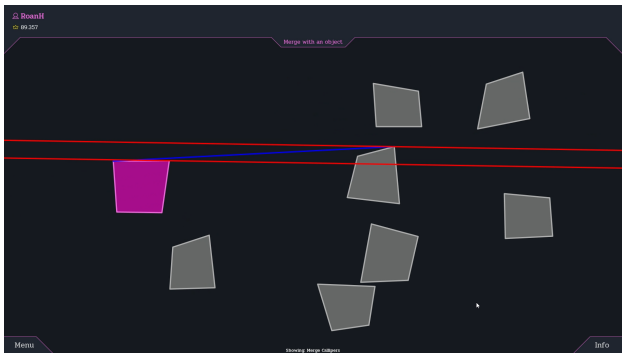

Figure 18: In game visualisation of a conjugation tree based segment partition tree.

### 5.3 Merge Callipers

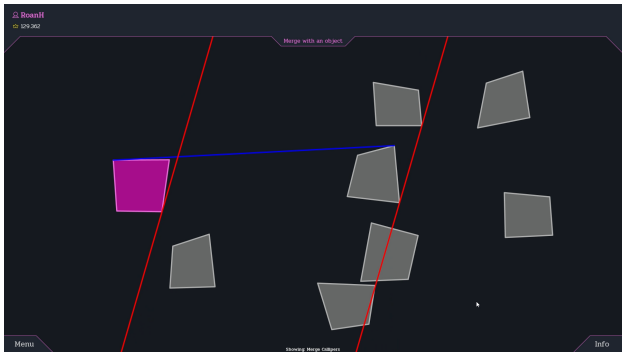
The visualisation to animate merge callipers can be toggled using ctrl + M. When active, this visualisation shows the merge callipers rotating around the objects involved in a merge in red. Any found merge lines are shown in blue as soon as they are encountered. Four figures showing successive snapshots of the animation can be seen in Figure 19.



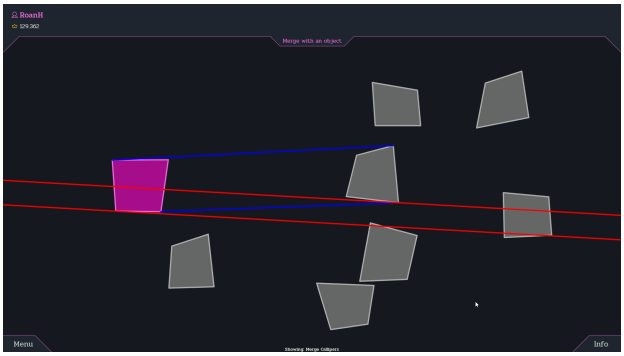
(a) Start of the animation.



(b) Just after finding the first merge line.



(c) Midway point of the animation.



(d) Just after finding the second merge line.

Figure 19: In game animation of the callipers used to find merge lines.

## 6 Concluding Remarks

In this work we have presented several algorithmic improvements to various aspects of ConvexMerger and their visualisations. The improvements concern vital elements of the game, namely point location, used for user clicks, convex object merging, which is the core game mechanic, and segment intersection testing, which is key in testing whether a convex object merge is possible. For each game element, we mentioned the base approach used at the start of this project, explained what the new algorithms are and how they improve upon the basis. The most notable improvements are to convex object merging, with the caliper approach running in time linear to the number of points on the two merged objects, and segment intersection testing, with segment intersection query times sub-linear to the number of segments.

Whilst the improvements to the running times of the various query types are significant, there is still room for improvement. Namely, point location, handled by the dynamic trapezoidal decomposition, can be further improved by inserting the segments in a truly randomised fashion. Another point of improvement would be the computation of conjugates. A more efficient approach is already known [14], yet it was not incorporated in favour of a custom approach that was simpler to implement.

## References

[1] Tetsuo Asano. “An efficient algorithm for finding the visibility polygon for a polygonal region with holes”. In: *IEICE TRANSACTIONS (1976-1990)* 68.9 (1985), pp. 557–559.

[2] N Baumgarten, Hermann Jung, and Kurt Mehlhorn. “Dynamic point location in general subdivisions”. In: *Journal of Algorithms* 17.3 (1994), pp. 342–380.

[3] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[4] Timothy M Chan. "Optimal output-sensitive convex hull algorithms in two and three dimensions". In: *Discrete & Computational Geometry* 16.4 (1996), pp. 361–368.

[5] Timothy M. Chan and Yakov Nekrich. "Towards an Optimal Method for Dynamic Planar Point Location". In: *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. 2015, pp. 390–409. DOI: 10.1109/FOCS.2015.31.

[6] Yi-Jen Chiang, Franco P. Preparata, and Roberto Tamassia. "A Unified Approach to Dynamic Point Location, Ray shooting, and Shortest Paths in Planar Maps". In: *SIAM Journal on Computing* 25.1 (1996), pp. 207–233. DOI: 10.1137/S0097539792224516. URL: https://doi.org/10.1137/S0097539792224516.

[7] Herbert Edelsbrunner, Leonidas J Guibas, and Jorge Stolfi. "Optimal point location in a monotone subdivision". In: *SIAM Journal on Computing* 15.2 (1986), pp. 317–340.

[8] Herbert Edelsbrunner and Emo Welzl. "Halfplanar range search in linear space and O (n0. 695) query time". In: *Information processing letters* 23.5 (1986), pp. 289–293.

[9] Alain Fournier and Delfin Y Montuno. "Triangulating simple polygons and equivalent problems". In: *ACM Transactions on Graphics (TOG)* 3.2 (1984), pp. 153–174.

[10] Michael T Goodrich and Roberto Tamassia. "Dynamic trees and dynamic point location". In: *SIAM Journal on Computing* 28.2 (1998), pp. 612–636.

[11] Ronald L. Graham. "An efficient algorithm for determining the convex hull of a finite planar set". In: *Info. Proc. Lett.* 1 (1972), pp. 132–133.

[12] Ray A Jarvis. "On the identification of the convex hull of a finite set of points in the plane". In: *Information processing letters* 2.1 (1973), pp. 18–21.

[13] DT Lee. "Shading of regions on vector display devises". In: *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*. 1981, pp. 37–44.

[14] Nimrod Megiddo. "Partitioning with two lines in the plane". In: *Journal of Algorithms* 6.3 (1985), pp. 430–433.

[15] Mark H Overmars, Haijo Schipper, and Micha Sharir. "Storing line segments in partition trees". In: *BIT Numerical Mathematics* 30.3 (1990), pp. 385–403.

[16] Haijo Schipper and Mark H Overmars. "Dynamic partition trees". In: *SWAT 90: 2nd Scandinavian Workshop on Algorithm Theory Bergen, Sweden, July 11–14, 1990 Proceedings*. Springer. 2005, pp. 404–417.

[17] Raimund Seidel. "A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons". In: *Computational Geometry* 1.1 (1991), pp. 51–64.

[18] Michael Ian Shamos. *Computational geometry.* Yale University, 1978.

[19] Michael Ian Shamos and Dan Hoey. "Geometric intersection problems". In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE. 1976, pp. 208–215.

[20] Subhash Suri and Joseph O'Rourke. "Worst-case optimal algorithms for constructing visibility polygons with holes". In: *Proceedings of the second annual symposium on Computational geometry*. 1986, pp. 14–23.

[21] Godfried T Toussaint. "A simple linear algorithm for intersecting convex polygons". In: *The visual computer* 1.2 (1985), pp. 118–123.

[22] Godfried T Toussaint. "Solving geometric problems with the rotating calipers". In: *Proc. IEEE Melecon.* Vol. 83. 83. 1983, A10.

[23] Wikipedia contributors. *Gift wrapping algorithm — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Gift_wrapping_algorithm&oldid=1008251630. [Online; accessed 22-January-2022]. 2021.

[24] Dan E. Willard. "Polygon Retrieval". In: *SIAM Journal on Computing* 11.1 (1982), pp. 149–165. DOI: 10.1137/0211012. eprint: https://doi.org/10.1137/0211012. URL: https://doi.org/10.1137/0211012.

# A Datasets

This appendix lists the game seeds that were used to generate the datasets used for the evaluation in Section 4.

## A.1 Dataset (Large)

Seed playfield generation options: *object size = large, density = medium, spacing = medium.*

4C1KNW03AT11XIFYDMVQ 4C1KNW00FIRADDVBBSGS 4C1KNW00GL0Z4SQDZ4TR 4C1KNW0021SO8K23G3GK
4C1KNW012HQ0S5WDOJ66 4C1KNW002U3HDVC9951K 4C1KNW00T8QL1XWM74XW 4C1KNW03UJ0CWO2SMESN
4C1KNW00DPCDV3CBS6Y9 4C1KNW00OI6JN93MT8MD 4C1KNW02G5RFG0WT677M 4C1KNW02EGC7XQS0HHCG
4C1KNW00LX36O9QY8XMM 4C1KNW009NX75CY4UT59 4C1KNW00ZCGA93PFIF8A 4C1KNW0382CMVEOU8N08
4C1KNW02JOO4MG6SQ0G5 4C1KNW03M87HTF734OCV 4C1KNW01T46JD3UQT7HJ 4C1KNW02FCTWGPRUD4YH
4C1KNW00YVWD36ZONYR7 4C1KNW00J1P9448U4O8W 4C1KNW023PL6FC7KB2AD 4C1KNW03KZ8PECLPF0YG
4C1KNW02R9XQKQ78O57D 4C1KNW00K8KY014RQPYZ 4C1KNW0214CPPU4IA0CJ 4C1KNW03GQ4HW801OIPD
4C1KNW0071SXVVV0LFR4 4C1KNW005BY5YYUO21D6 4C1KNW0087FVHQ7A76S8 4C1KNW00OFG9R2ID5XDO
4C1KNW00VKLBRRPHNYHK 4C1KNW0283PLATHZWNUM 4C1KNW00E2NKKEB1RQSC 4C1KNW00DUSDUZ4ZF2RE
4C1KNW03DP00I0RCHBBI 4C1KNW009X7TLZK0724D 4C1KNW0202OVVR0LX9BN 4C1KNW03TW8RRFW8IY5E
4C1KNW007Y0ZO30UTGE6 4C1KNW015DPR8IE8P7D8 4C1KNW001L7EF3EPMDI6 4C1KNW00FHKHX2N5BJIU
4C1KNW02XRJBR7RX0ZSQ 4C1KNW03DZAKWC9YX3PD 4C1KNW00SNYM8ZB6357R 4C1KNW01P2NJ4M145UM2
4C1KNW02ZUW85CYERN4V 4C1KNW00ZYALE8FZIFNY 4C1KNW00K2HLV56F8K9V 4C1KNW004BB7VRYF5EIL
4C1KNW02FS9AYHAFE9RG 4C1KNW03SGW04XTIP34E 4C1KNW00NBMLOPXUQWQD 4C1KNW021E1YAMM6S1RT
4C1KNW033FUSPKNR0C1N 4C1KNW029W79ZFCYDY7A 4C1KNW004ESJCRJIA2BK 4C1KNW02REY66IXXTA7S
4C1KNW009WAE5GBCYGBW 4C1KNW02ORGXUKOUG4BI 4C1KNW022JXR2IXOICCJ 4C1KNW011YXAR8XPL6IK
4C1KNW00C221IZH6L27V 4C1KNW00ASEJGDKKSFUP 4C1KNW01M6MOTDD44TEU 4C1KNW00GHLJZU34ZK7D
4C1KNW03IJR3MC903FZL 4C1KNW001J8ANI2C6ZMW 4C1KNW02532FTCH6S3OB 4C1KNW0358YDGE2FN0M0
4C1KNW03MO38RGOIFTWV 4C1KNW01WB6O5M8ZBBYS 4C1KNW032Y2D89VAR0D5 4C1KNW02LOAM32ENAZSA
4C1KNW0397B403EKS5CQ 4C1KNW01GPS7OMFJE6KP 4C1KNW01ZZ8GWIB7IEMM 4C1KNW00WV08MIUH2UU9
4C1KNW02G4131PLCK2OH 4C1KNW034MQOUE9ZKR0B 4C1KNW0058UUPVGELYBK 4C1KNW03A5448J1ZQ1DR
4C1KNW03FTP8CPPPUH1G 4C1KNW00453BFWPFJW1K 4C1KNW02OLSGJS8MJOL9 4C1KNW03OJ97QMLP0FSA
4C1KNW014MJUAEEKBVZ0 4C1KNW00JX3G54P0FMNY 4C1KNW03GBBEO500V8UY 4C1KNW03IMIK38DSSY4W
4C1KNW00UTUOHJLCMGIW 4C1KNW0144Y60SQQYHO4 4C1KNW02RR0T3ELYY7LU 4C1KNW00CSS76L6ZXOVF
4C1KNW03MRBO4RF75MGN 4C1KNW00K9ETLUZCAW9Z 4C1KNW00J72CW231XURO 4C1KNW02BFR8OGQG0ND8

## A.2 Dataset (Medium)

Seed playfield generation options: *object size = medium, density = medium, spacing = medium.*

3Y64YR436S1HS4T4OGBQ 3Y64YR41AOBPFQJNGRBE 3Y64YR436JZS7M0RPXQ3 3Y64YR42JKTG22KJOEOH
3Y64YR4033GBPK0VQ3B5 3Y64YR4154COXBXL9ML5 3Y64YR40KNMO356FXN7C 3Y64YR41MU9JW85HRCVF
3Y64YR4015OOZI2601M3 3Y64YR4150Y93BYFEGVR 3Y64YR40UFR9L8Q0OXRD 3Y64YR43UIXDT7MFQCC7
3Y64YR43P7WTP3TVJ0A9 3Y64YR40PIH85W1EMH80 3Y64YR432NA383EDG0BF 3Y64YR43Q2LUXBO7AFDB
3Y64YR42V0SVKGPH33BU 3Y64YR40UOLWZ4BKXYOE 3Y64YR40F4RTNARAH2RT 3Y64YR40UUCFQWTXZU0W
3Y64YR412Z1UMRBUBMGF 3Y64YR42Q7ZS723M8DRI 3Y64YR40DWPYVZNUZQK2 3Y64YR42E50U72FHBHV0
3Y64YR40SMUUY1JTDDSY 3Y64YR40X5AFA5KPV055 3Y64YR41TEMAU9HU46DS 3Y64YR43FNDSKUPYW71H
3Y64YR43EIFOIVIPMV8V 3Y64YR4203G1KRRAB2C3 3Y64YR430OD7NOOLEHVI 3Y64YR43OB5I17S42PEL
3Y64YR42V9BBEXAQWLXF 3Y64YR40U9Q7ASZ1VHTM 3Y64YR42J2THP48C2VY0 3Y64YR40KAID3MX3S3RY
3Y64YR415PQMEVZYDF8L 3Y64YR401RUCPT6C22ZJ 3Y64YR4029K6CAUKFCFS 3Y64YR4367WOU3ZLCV4L
3Y64YR42KQ2HWQUPET6H 3Y64YR427PW84WLHO8NN 3Y64YR431C4IP53KTYGL 3Y64YR40UMYVI188GZD8
3Y64YR40H642ERPWKK3T 3Y64YR407SY73OEU481Y 3Y64YR42UEKYMY4CY61J 3Y64YR42M9ZMN124WGK3
3Y64YR42YFF0NP7UMGUK 3Y64YR41B4MA5GUGOD1E 3Y64YR41NQ6CL3PER32B 3Y64YR42FSUEUK0YKYZF
3Y64YR40EBEEXZK4DK9O 3Y64YR40MGC60QEO12NY 3Y64YR42MLSY7ZONZRDX 3Y64YR43D3N9MBNDRABJ
3Y64YR42PPTGRGA3SR58 3Y64YR41XYGJPL5BWCGX 3Y64YR43VWO2KZ0V722C 3Y64YR43VAG9QIGX6YWG
3Y64YR41LKSGAAOSCSXW 3Y64YR40ABWF4B5OELG8 3Y64YR43TMU4MSPMK4EN 3Y64YR43494BVUAZ1KDL
3Y64YR43UTLP6U6MUYZB 3Y64YR43IZ14PSTLZJYL 3Y64YR4357QHH2CVH1GA 3Y64YR42SQJJPI73GI6V
3Y64YR42NDPB2PA9PGBS 3Y64YR40Z98LWAR3KNR7 3Y64YR42T7L5KH95Q588 3Y64YR43TLZKC4OGM2QO
3Y64YR41S9ILOUQR5CEW 3Y64YR40ZJHO9WHC11WL 3Y64YR41PL0890KU8YAK 3Y64YR40WJVCEDSV4ZDB
3Y64YR42Q158PO2901OC 3Y64YR40KEOL6UDUKQIU 3Y64YR402Q6NW49TFVSF 3Y64YR43HSUSXQ6303RU
3Y64YR42ZZC5KTXO2RCG 3Y64YR41N43YGRJ8ML3M 3Y64YR4148DMR643OIRH 3Y64YR43NE3Z9RYVMSFC
3Y64YR42DMMOSZ92FD3G 3Y64YR42R4IEC8PO2DJV 3Y64YR416FT67L66CMIJ 3Y64YR42XP4WRU4CLT9Y
3Y64YR405WDFH440EIMY 3Y64YR43G9J5DU45OVKS 3Y64YR42HB2XKMFNKHZA 3Y64YR41QIQDCLHZWKZK
3Y64YR40A7SUZHOZ73KF 3Y64YR42R4D9AKVZE7HV 3Y64YR42R8FP7DJM8X2E 3Y64YR43KPHDN62GJ4BN
3Y64YR4388VE4O9W4XEO 3Y64YR41L2UO0965ADLJ 3Y64YR41RWBWINMBXLKE 3Y64YR41STOR557D65F7

## A.3 Dataset (20-40)

Seed playfield generation options: *object size = 20 to 40, density = medium, spacing = medium.*

43NKKC02ZX3VI4Q84S6O 43NKKC01SEOYXG9YNKBT 43NKKC0311MT04T6YCE1 43NKKC02IZYN2TUF7QKO
43NKKC00BUENP8UIAXS3 43NKKC00W7ARSSPH4N8V 43NKKC007OJOMZPWGET9 43NKKC037YDDUVNWN3GO
43NKKC00Y9IWD4BI4SKA 43NKKC02Q7CG8CCLBR24 43NKKC03MTZ0QRP473DG 43NKKC00VSU3YTV5THEX
43NKKC032OGIS8QK2MO1 43NKKC00MNU4OPDAGWNB 43NKKC017JW7Y7FKMGCA 43NKKC032PIUIBWBA7MS
43NKKC01F6R8KV2J7PSM 43NKKC01J719ELR5XROO 43NKKC031MHN4QWLA84F 43NKKC01MMZWVWBLQMKO
43NKKC00QLFMU9CB4B1U 43NKKC02LS3OPL75D24D 43NKKC02WLK4KT6F2R4S 43NKKC03OZNFDESCCISU
43NKKC02D4ZGQI7I1T2M 43NKKC00AD4AYXCYIZUP 43NKKC02J8L1KI7U72SY 43NKKC001CNUY6N9IFQR
43NKKC02QF3GWAI4SWY2 43NKKC01TQ4Z20SXBLGE 43NKKC02P4P0BBG2HHFD 43NKKC03QAHPAIG5XESR
43NKKC022DFPZV1RY5GE 43NKKC03W4PH1F2F8H0X 43NKKC03AZFK3150ZAJZ 43NKKC00RWNTH7XXFX7U
43NKKC02WI7HJI1A0M17 43NKKC00B28RFN57K7FJ 43NKKC019ZA0FNOH44KF 43NKKC00UAC17MMNUCBW
43NKKC02J4P8RL3N7TBE 43NKKC03K0X0ASS3QHLJ 43NKKC01GS9N5UMI7ZJV 43NKKC01PHX3G7UVPAVM
43NKKC007V79M23A5W5F 43NKKC0137UF12JMZDZF 43NKKC01DQZMLU8ITG1Y 43NKKC019S772V1KXJP9
43NKKC00E9JS5VM0II5D 43NKKC01WQGOBUNK1Y5Q 43NKKC02ZLJNOQQB5VN4 43NKKC02ODORNSHWKFZ3
43NKKC0282YG4QLE9O7V 43NKKC02FDHMNDC3COP1 43NKKC01EYKF4LWUTUC1 43NKKC00SK3PTKFKN1L1
43NKKC0047F79I0R6BBP 43NKKC01ZCVSG88HD6VB 43NKKC014EL2VU02GNBW 43NKKC02817QRYJTJT0D
43NKKC01309OBJMM6MK0 43NKKC000D2JBKG6VLG4 43NKKC03OHBDNFOW447T 43NKKC02MTAVC0L39AAZ
43NKKC03QEGRXVTZNY9Z 43NKKC0247RWZFXEQ9R5 43NKKC01N37GZFB8NKAX 43NKKC02JYG0S6A0ZX6V
43NKKC00IIM96SQ5KL7U 43NKKC0272QDD73VVJBV 43NKKC0318S7FTLJWXD3 43NKKC01PD8DEIKYG9PG
43NKKC01LUP5DFE8QF1I 43NKKC03IMWPL7C9PVGS 43NKKC01PB4U406YHPBU 43NKKC01ONUD3FBMOE99
43NKKC00I4IBPIC6QSZK 43NKKC01TKR3NSEJ5M2H 43NKKC03HPVRVBMES7H5 43NKKC01SRPH5XA67XMJ
43NKKC01J4D6EEFRL8S3 43NKKC009TYV0Y1S0NXG 43NKKC0223Y8AVL3GUOS 43NKKC00DG9G9WAX9OIV
43NKKC00ALXDAAVGP07P 43NKKC020MRZMEWIK2L7 43NKKC01N1DGDK72939R 43NKKC01QFVZIAJO3XL3
43NKKC03A3PXXML178U7 43NKKC02569DXJXB3R00 43NKKC012P0ITG2Y7EPE 43NKKC01G1M6KNSTSRC3
43NKKC010UBR0VT7EFJ8 43NKKC03Q3K9TQGYZXFN 43NKKC02IPLNGMOLSQXT 43NKKC0156ZXZ5GYZHQV
43NKKC03AZQHY8P9YQ5B 43NKKC02KX0O1V1SK920 43NKKC01RYV7KMHLUPOT 43NKKC021J8J3R1VMSG6

## A.4 Dataset (15-30)

Seed playfield generation options: *object size = 15 to 30, density = medium, spacing = medium.*

4298JQO2HLK2K0U4I25D 4298JQO3S5QWR3NOIOSM 4298JQO3ES33FWGVGSQ2 4298JQOO2NV3DMBC972M
4298JQOO046XGE5S5TJ7I 4298JQO23RHKOI6XNIMO 4298JQO1DRMIW8TU0THK 4298JQO0Y57WRQ8U3DO6
4298JQO2X9ND9DYBN7IJ 4298JQOO1KPEE3JO5JKI 4298JQO1MZ3OUAE9T6WQ 4298JQO1N8OYY3C1384B
4298JQO38PJ17ACH474X 4298JQOOQBYQMMFVGJGA 4298JQO2L1H909YT06P3 4298JQO13GVP17G32XUC
4298JQO2IYPL8NRF3RP8 4298JQO2FM79HX6P2QKI 4298JQOOUQBNWXV6F719 4298JQO2A9F14NUBAYUC
4298JQOOUNOEUF8229OP 4298JQOOG7KLJM2YDD1I 4298JQO2YLXEA5KD9QVA 4298JQO2PE176NRL55NU
4298JQO3KZZFF0FCRD2O 4298JQO3DKM5ZPAHHM1U 4298JQO2A00Q2JLRRL6F 4298JQO2PCU3RL0NM6CT
4298JQO3SGYW4JA05TD7 4298JQO1H4OM0RS0N50T 4298JQO1GY1QQU2B2GBJ 4298JQO1942GKBUS2OR9
4298JQOOOVWO2S7VYZ16 4298JQO36PX6WEW711SI 4298JQO3F8G6QOI5P3JG 4298JQO2IEL3EKYV6W9Q
4298JQO235SKA46BTD9E 4298JQOOZ5GW1UO6EN1M 4298JQO2WF6XDT3R25AS 4298JQO3EG06BE5V0BA2
4298JQO26594FIPNE94D 4298JQO1IL2OTRO3U8SS 4298JQO3JPY2C1IKX5SM 4298JQO1XHQHIT1Q0XFF
4298JQO1ILTQW4UUZK8B 4298JQO32O48ZWVGGJC2 4298JQO3PJQ491LUHRQD 4298JQO315HH1NRCGGTE
4298JQO2XZA62RJPOKK6 4298JQO151RLY4Q4TCH2 4298JQO1EXZJY4C0RVIC 4298JQOO1EN8T21OD9L3
4298JQO2ODGO83S502LO 4298JQO28O2GE299EJ5F 4298JQOOE3QOKSM1XZWC 4298JQOODYLMJGY6XMZ9
4298JQO35FBTZ2SVLQ5M 4298JQO1V73YZJ1DU8SF 4298JQOOYKOBWSSV1SCL 4298JQO3B3GX158AELAD
4298JQOOH9Y4USWB087S 4298JQO2NKTU54T75XJA 4298JQOOO7HQOBU1XZUT 4298JQO10I64ZPX4FAZO
4298JQO1Q5LK6K7VCW41 4298JQO16EAD0MQPE9W2 4298JQO2DRCKOQR4CE1N 4298JQO25GZF61RR9CGJ
4298JQO2NP2JKRUOGZC3 4298JQO37RW55CAIR1X9 4298JQO1CP5W5INKKCML 4298JQO2JOQ6XTM60DRS
4298JQO321PZ92VXQU76 4298JQO1REITXCFF9KAB 4298JQO10PZCMDX4G79K 4298JQO3RQHVKJQHE58A
4298JQO2BD99CHJDM1NJ 4298JQO1TMTZDODP9FV9 4298JQO3CJZWFECNCA15 4298JQO21726M81HH94K
4298JQOOW0T3UWLOID6S 4298JQOO2GO07BNUGY5G 4298JQO1ZDXE4SO1V4W1 4298JQO19SA8G5WP1G6F
4298JQO3NG5U9J3O1CT0 4298JQO2FU6AW4HWD3CD 4298JQO3JUD9P5R7YNS5 4298JQO2WGBKE8OKSZ4C
4298JQO2FQE3FI6W8M5K 4298JQOOBQ5ZMMHFW0JC 4298JQO1NKPQ0Z24JOON 4298JQO225VD8O70YVC6
4298JQO10V30OMLTLBUP 4298JQO1EWQSOOH3Y41U 4298JQO2D58VMBDXB9T5 4298JQOO2Q6QRC5YFZGT
4298JQO1LS9OOHJQUI82 4298JQO1A892KFTS4CL6 4298JQO2NFGD07EQAJ9H 4298JQO2HJYN6CN546P9

## A.5 Dataset (10-25)

Seed playfield generation options: *object size = 10 to 25, density = medium, spacing = medium.*

40V3JZK2RROA7CFI9PKF 40V3JZK1KMPL1ZSIPSAI 40V3JZK1J0QHGV7Z385X 40V3JZK13HFLZ9DS7HAJ
40V3JZK0XYDI2Y4R3D4F 40V3JZK2QM85BB47C6AQ 40V3JZK3SLOSPKLDDOG3 40V3JZK1NZHEVVOO9FQ0

40V3JZK1LWJGL937G88T 40V3JZK0OQSYIW3O7WT6 40V3JZKO67JJHUK9NGHG 40V3JZK2HAZ3800MLC1C
40V3JZKOPHXD8DHM2LNZ 40V3JZK2CVPARAH31ON4 40V3JZK17VNRDU3ABKFM 40V3JZKO2N3TPQLCYRJW
40V3JZK1ELCIMP6I5BGA 40V3JZK3B4STEURV27SH 40V3JZKOPWV23ET73QQ3 40V3JZK3O5TP4HWJI9HO
40V3JZK2POLXXW74YHR9 40V3JZK3MY6OA4O8NKLR 40V3JZK3O8FL87SX4OCW 40V3JZK1PMWG9PI4PKML
40V3JZK3OZDOO4AHVIOB 40V3JZK18ZTI511B8RSV 40V3JZK1D1TN38T1PA5O 40V3JZK14HQX89RBEI7P
40V3JZK3S1L2RGE7YXOM 40V3JZK1AEWOJTIY9ALV 40V3JZK1FAYSUQ1JYAQS 40V3JZK28EVD8N1MAPO7
40V3JZK3KO75RTW6O4PK 40V3JZK1AC7MJCW31XRL 40V3JZKOHRPA37ECEVX8 40V3JZK1RCJG4P4PUX7X
40V3JZK3CR1141Y2804X 40V3JZK1M9JHKVCTD1LX 40V3JZKOTZNLOQBLK6IZ 40V3JZKOGM71YRDH9D23
40V3JZK37G4HEOYI3YMR 40V3JZK129PZXFQFZ8AC 40V3JZK3CG3NONKH8RPN 40V3JZKOBDBAOKB78EEY
40V3JZK2U1CYVOZ6LQJS 40V3JZK3PZ7E36KJJ6I8 40V3JZK2K9VS2SFO8NIB 40V3JZK1AS4JEJBMSYZJ
40V3JZK2ZOEO6S19TDIL 40V3JZKOU9RDRHVNARXL 40V3JZKOXGFFND48OB1N 40V3JZK3P7MNPOTCIA9H
40V3JZK1PMB3UOL7LD1D 40V3JZK1WAK79Y414I9U 40V3JZK1AL9MX8V1YB99 40V3JZK2LR7GWOFPIWHO
40V3JZK3IFYSM25BHVVJ 40V3JZKOQ4G7YHKFTXDM 40V3JZKO4Z2ZWQ5MR7V9 40V3JZK1X5PN1TXR3T19
40V3JZK22YRB67528JL2 40V3JZK1W8XK2L9Q5ECF 40V3JZKOT48H8W6H4485 40V3JZK10V2RD9KYAHIO
40V3JZKOWSAEW6TORNGA 40V3JZKOMM21AMCO1KK8 40V3JZK3OPXSOJ933GTT 40V3JZK3OZKQ2NYDE7F4
40V3JZKOUYXNFZTOJ8GL 40V3JZK1KXJK59ZB1GEN 40V3JZK19VPIPNK7EKT5 40V3JZKO63I5PXCCOU3M
40V3JZK2S789G279GAI8 40V3JZK2NI4KXBH1GWT6 40V3JZK3CV1H7ZDKN89I 40V3JZK1VIM9EQCZWETQ
40V3JZKOWSY7ZXRTXHNN 40V3JZK1OQOW4XDH3SLX 40V3JZKOXJAOSR9EV7F4 40V3JZKOSNXP2HUS7GXI
40V3JZK2OQTL6CT14YW3 40V3JZKOGLHNT3ZM8UCS 40V3JZK1IIXYDO4L1OKY 40V3JZKOIYRPODXWJO6V
40V3JZK34CPF8IGQKCMO 40V3JZK1WTW74QUTH6LH 40V3JZK2KEM7RDNIRMRF 40V3JZK27A3LCMMVVE9A
40V3JZK23U7XFGF76PMH 40V3JZKOKZWHT8YBM7TW 40V3JZK2YDH2IDUXFOPI 40V3JZK3VE8B1PDLHGZR
40V3JZK1EL18S4WYXFWK 40V3JZK1G436OL9Q2COG 40V3JZK1NFRR9M5TK3LN 40V3JZK1P2Z9T5BCY3U3
40V3JZK2IIUA1YLMBJDW 40V3JZK3VAGDEK9NOPY9 40V3JZKO3I9PIA51K108 40V3JZK31BNH87DMEORY

## A.6 Dataset (Small)

Seed playfield generation options: *object size = small, density = medium, spacing = medium.*

40UWJ5C21PC9TZQ8QC43 40UWJ5CO371FJOSFVV9W 40UWJ5C2JKXW54XOD4ZU 40UWJ5C3UBPPE681KXJO
40UWJ5C29D63NVT8202K 40UWJ5C26NC80E1DWQPT 40UWJ5C1R9VG109WJPCI 40UWJ5C2TCFVKNBSO3J2
40UWJ5C248YMOYAXVTFU 40UWJ5COX681KMR6D4RJ 40UWJ5C1GYZ5LH6SYMUY 40UWJ5COTTL3KNMSQ8Z9
40UWJ5C2HGK76UZEF9OI 40UWJ5C2FUZIGPYSDW36 40UWJ5C38O5YPTYNHAF5 40UWJ5C3QAHYST1GRFCO
40UWJ5C3HV9XQBP1MNS3 40UWJ5C16FZD4YEHXO84 40UWJ5COPGIT81KQ3Z1S 40UWJ5C2L4V3G5XM7HX8
40UWJ5C1YCTTVR3P9XRF 40UWJ5C3CW77PSV5Z2J7 40UWJ5C1U2A5FP8VAMBM 40UWJ5CO29UI7MBGEE9C
40UWJ5C23SOOE2YDPIVX 40UWJ5C1AA25CK2P2REJ 40UWJ5COMOULNKN1U3VV 40UWJ5C2BXFEWM2LTLRL
40UWJ5CO73KEH9EFTXAL 40UWJ5C3OM5HQC1TSDHC 40UWJ5C1S2J7OKTTN14L 40UWJ5C2UG6SEYX4NUF3
40UWJ5C1FZQBAVNSKURY 40UWJ5COTIIFDXKC8WOF 40UWJ5C128IWADCH1FZS 40UWJ5C339FZRUBTGTZI
40UWJ5C3HGQL5YRIOPKZ 40UWJ5C3E9PRTICOWMYM 40UWJ5C31NXH5FAPWKTX 40UWJ5C14CDT3YHCARNZ
40UWJ5C2K7NGSK35OUJS 40UWJ5C3ADV55MNFV5RQ 40UWJ5COZXEVM9URUDK5 40UWJ5C2TZ6W9EA1ADTF
40UWJ5C1HWDOMNYI9W3F 40UWJ5COO6OW6US7LV7J 40UWJ5C19C7GCA3FRYXF 40UWJ5CO199ROFPRFCI4
40UWJ5CO34D3WNZ8WKQC 40UWJ5C2DMFEL5KOZT7D 40UWJ5C2HAB9J7DOFZM8 40UWJ5C1GHU37UHZB383
40UWJ5C1J958YZPPQLAR 40UWJ5C176HHBYMLUB7T 40UWJ5C2L64IIPPKZKPP 40UWJ5C1ZAZ4JK426HZE
40UWJ5C2HXEYD164YU8S 40UWJ5C1ASCTSEUBZFHI 40UWJ5C1FUD6ID1WT1E8 40UWJ5COF416CIOT33N7
40UWJ5COWIRLAKR1QDHY 40UWJ5C2C5XYRI6HK6OY 40UWJ5CO6E8H31SSQ4VZ 40UWJ5C34KY5OHNZO9R7
40UWJ5C1UAY4NLV94DGR 40UWJ5C1FJ39Y7HIGOOP 40UWJ5C1S7I3GV85W704 40UWJ5C3VXAPJZR5OUG2
40UWJ5COO2N31OW5VCUL 40UWJ5C2F7NOQPCBAYPB 40UWJ5CO2OLPRQ3765HW 40UWJ5C29SJO8LP2XUL8
40UWJ5C2GZ4E6PXGLP25 40UWJ5COAZBZ8Z4HH6AB 40UWJ5C3QEMAA8RWR6XG 40UWJ5C14V2OTMR38HPM
40UWJ5COK1HEWPONXUJB 40UWJ5COP55NQNY3DAIY 40UWJ5C1QEDUVD18WMQ2 40UWJ5CO52RCYUZCOQOI
40UWJ5C3H6NNPLID58EN 40UWJ5C2C11K7FO5FO3I 40UWJ5C21YQQR2XS4G52 40UWJ5C3TCN23H1DJR1J
40UWJ5COV4GMBC91VJ8P 40UWJ5C2I46CY7QSKM2C 40UWJ5C1LMWRFNFG6L92 40UWJ5C205Z4R4Z10ZWV
40UWJ5C2T8OBBZI6KWOQ 40UWJ5C22M3OGF8ORY60 40UWJ5C3NOM41PCS57QT 40UWJ5C2EARDT7MCGHOM
40UWJ5C1967UJ2G97DZ8 40UWJ5C3R5SW1VCOJILX 40UWJ5C1E3PYKH64TB4F 40UWJ5C1B4ESRN8UTUUE
40UWJ5C3SJOIGKTVLQV5 40UWJ5C3C5B5XVNSM2ET 40UWJ5COXEJH2PB4ZRKD 40UWJ5C1PP1EFMGOIERA

## A.7 Dataset (Segmentation)

Seed playfield generation options: *object size = small, density = medium, spacing = medium.*

40UWJ48276AXVDYARA35 40UWJ482QU9KFBAHAMHK 40UWJ482U7JZRNYOM878 40UWJ481BV180TRUU4BE
40UWJ482ON8VEN8PGMTC 40UWJ483ROR92CBXVHY8 40UWJ4832XKIJ34Z44EF 40UWJ482FDYKZH11I2P9
40UWJ482KHE5IY5OMAJB 40UWJ481MRKG3AGZ42RV 40UWJ4832YOUMZNPQFU6 40UWJ483BI1A71HUTZKQ
40UWJ4827UVQCXANBB6X 40UWJ481B3SGHQQOLBBJ 40UWJ482JKI19NM92W8P 40UWJ482J6IKTHAE3KSR

```
40UWJ4801XL4QCUMS9P4 40UWJ4826T4IFCZV107I 40UWJ481WF2S7H7NZLSA 40UWJ483PVKG96OQN5TS
40UWJ482QSZ7F1A2J6K1 40UWJ480ZSSJZIOFI4DO 40UWJ482XLPXDOI1GBXU 40UWJ4810WZSYUPS8LX9
40UWJ480K9ZOFPTYKI6M 40UWJ483BAF8QOTGI41Y 40UWJ480DAH1M471128N 40UWJ482T2R806LYVX5Z
40UWJ483P5AHY9BFV94G 40UWJ4809WHTLP1Q8RA5 40UWJ480443ZUFV9BLP5 40UWJ482KI2FI1PKNHGL
40UWJ482VFMENEPOV1AF 40UWJ4829KJXOWP7W2DI 40UWJ48284F81XURJ5AH 40UWJ481K8FKDEKTK8N8
40UWJ48043R97BXO2YLT 40UWJ48053APQ846E82Y 40UWJ4820F98L9VB423O 40UWJ4825CBYSWKJJWNO
40UWJ4815LYPX7SBOO2F 40UWJ4808U6PW1I18OYB 40UWJ481QCRKB4DL57WA 40UWJ481QI676E637UJJ
40UWJ4802U6ON4OXM9NC 40UWJ480IWTD41YZ4RNO 40UWJ480YK21POROQX3H 40UWJ483NIPQG2WCFK1J
40UWJ481L1WQYAAQDMTU 40UWJ480FJPG27LACVDE 40UWJ482TEEFAOPBR4KS 40UWJ4802UJ5WECDLAW5
40UWJ483GWI7POJBK3TP 40UWJ480MJ76HROO2QVL 40UWJ482ZSA22LHWODGY 40UWJ4826B7ROWV3UTN3
40UWJ482VCOQ65KCHO5H 40UWJ481UYPCDWYEXOP7 40UWJ483M7CKMIVJNCL5 40UWJ482UNIWDOF932HR
40UWJ481IFEXF5EUW17P 40UWJ481R6O7UKVF8HG9 40UWJ480EOD3TBPKG5UO 40UWJ483LB8BRZFQZSED
40UWJ481HTZY4SFY6CTO 40UWJ481D7ABEMTOW88K 40UWJ481JDGCM3S2ZMZ3 40UWJ4813FM90G2VCJGZ
40UWJ482MR3JEIOMJPZF 40UWJ481XZKDO8QUJ9WC 40UWJ483DDS5LY1KO8QK 40UWJ483RQFOOT2TMNTM
40UWJ482MOI56LYQCOS6 40UWJ483I3TN3QU7XLUC 40UWJ483CKEP9JY999T2 40UWJ482MU2IJWTYCHP5
40UWJ481QXOLV3TLXPT8 40UWJ48041TQITIYL1IU 40UWJ4827BPZT9YO1FPX 40UWJ482HUHXN7SQOT8D
40UWJ482URO1DGUGD2RL 40UWJ483DP9JQ4QQR1I7 40UWJ4828IBWLE5I6MFL 40UWJ480KNISDMMC91A4
40UWJ4828R1302PMDOAF 40UWJ48228COBV8NH1ZC 40UWJ480LP47B5CJP2QI 40UWJ482QAGXFRH23X92
40UWJ482JP8LLSYVSANP 40UWJ483UOROF1XNVGKH 40UWJ482QY5FRH21UYUG 40UWJ480FREBOOMHA3KO
40UWJ480OEZMO6W6WGUP 40UWJ483BPKATWBRTST1 40UWJ482OT6Z3WHRQX1N 40UWJ48270GPBWMRSU2Y
40UWJ483B2QS8ZQUWFZJ 40UWJ482MV7BGXJ27DH4 40UWJ481Y1O8EV1HICHC 40UWJ482E5ZORK4NR2T6
```

## A.8 Dataset (Decomposition)

Seed playfield generation options: *object size = small, density = medium, spacing = medium.*

```
40UWJ483OI977M388P9U 40UWJ481OB3BDOHJI52V 40UWJ482FIG8B7JLJ8SY 40UWJ48173HJDNJ2WPZM
40UWJ481XTFMDVOOXGYW 40UWJ482V5FDJPRILB6S 40UWJ482VX12AM8C8ZHT 40UWJ481O3YLFZNTZDD4
40UWJ483ICQULE1ZK4U7 40UWJ481SAYOWMAF8V2P 40UWJ4804O30AMH4VVLC 40UWJ4813ZKDEI77RZEC
40UWJ480XIRRLXLN52BD 40UWJ4819JNMC1HP3IIO 40UWJ4822SWLS8OTC6T6 40UWJ480GAUQLEU51NE5
40UWJ483A2GYGVL5H3PY 40UWJ482E1O3IZS85CG2 40UWJ480E1TJIQOQFTZB 40UWJ483JX58FFBB1OOZ
40UWJ48123JK3OOFISAE 40UWJ480OPNEHUB8HHS3 40UWJ4813CFVVQY6XNO3 40UWJ482EBPMQ5U499HI
40UWJ481YMOAM9O2JXAH 40UWJ481DKHLGF98PIRO 40UWJ481MXO1W1YUOZ61 40UWJ481A4VEUHVS81RI
40UWJ482ROA8B8JQOQTL 40UWJ483JOGY9ICMYAZN 40UWJ482RYX228AE3MWB 40UWJ481ROCB7RI2E1HB
40UWJ481MY16FN46WUN5 40UWJ48010NOMVOEMSO9 40UWJ4835UHQ5OA8CG4V 40UWJ482M60ILI12TUSZ
40UWJ4824PDOEM5XN978 40UWJ480XYQDAHT119XE 40UWJ480NQEXWMTNWJ9X 40UWJ483481OC1AERX5M
40UWJ48395ZP3OSRUAPI 40UWJ481EIWEFM13IAO3 40UWJ483Q8ODZ9KVGAG6 40UWJ482SM21SBM2TI2O
40UWJ4801R4J6ZQYD7MI 40UWJ482HJLGNTI1GTA6 40UWJ483H7JQ6FKO9ZFU 40UWJ481BKUEV1VMIDDT
40UWJ480AE9I6ZJEJW0A 40UWJ4812OQRWYATUN6Q 40UWJ480KF306VGE4GRK 40UWJ482D3XPEK1IOLEX
40UWJ48384WHG92MNDJY 40UWJ480R5E9U2E9DEPT 40UWJ480YW2SKTC1LUHK 40UWJ480L4SCJOCKG9NX
40UWJ482UY42PCG3HXPF 40UWJ483I3ZFO8Y34XZ6 40UWJ480F35OV27GCRPD 40UWJ480PA1N6VVN7P1B
40UWJ481DNTRIGPZI3ZW 40UWJ4832MCT8MLXOAUO 40UWJ483IA3G45NYFIO1 40UWJ4837FX5D2KAP9AO
40UWJ480OWQ7KRQ5OFL6 40UWJ480MEG83K62Q1CA 40UWJ483T9HN8PVB5BKH 40UWJ480WDDORVO7LY8Y
40UWJ480KRPXWH3Q7Y35 40UWJ480J35O3WRP2T3M 40UWJ481NXR7Q41V6GIC 40UWJ483O19ZZ64NZP6Z
40UWJ483O5IZF8XGAHOM 40UWJ480OKIVOZQETUE5 40UWJ4839NDHBK5C9TIT 40UWJ482NOCGWP2LT9JG
40UWJ4834D6USN7P7PX3 40UWJ480JNBWQYKBY1HX 40UWJ482MQ1VT52WX368 40UWJ4803141EXTYH2DV
40UWJ483HOS69EN4JLP8 40UWJ482AIDI7VNXOZOE 40UWJ483J8YYLUHZUGDR 40UWJ481NEG3BRGIBLVL
40UWJ483ILJRX15QQTAL 40UWJ480UJM5DIELEVD3 40UWJ483T2D411EVXNYB 40UWJ4831JEQKGPJNUN7
40UWJ481959O90J34HTR 40UWJ48223ELKVBT3G7N 40UWJ4808H3I9XRNVJX8 40UWJ48259ZBOGU7DEBG
40UWJ4839CSQFXD8P4C9 40UWJ481Y5C5867UEUOQ 40UWJ481JMREPP99ODKT 40UWJ481BTAEUBJ3HGE1
40UWJ482RWWFBQG7MUFW 40UWJ481CI4OWMNRNU6I 40UWJ4836LHSY2LTVP3J 40UWJ480SZ3PWDIVIUTM
```