A Practice-Minded Approach to Computing Motorcycle Graphs*

Stefan Huber† Martin Held‡

Abstract

We study the computation of motorcycle graphs and give the first formal definition of the motorcycle graph as a set of constraints rather than as the result of some process. A constructive proof that the constraints can be fulfilled is cast into a simple algorithm for computing motorcycle graphs, with geometric hashing used for speeding up the algorithm. Extensive practical tests of the C++ implementation of our algorithm on over 22 000 data sets provide the surprising practical evidence that it processes n motorcycles in $O(n \log n)$ time on average. This observation is backed by a stochastic analysis which reveals that our hash-based algorithm can be expected to run in $O(n\sqrt{n} \log n)$ time, provided that the motorcycles are sufficiently uniformly distributed in the plane.

1 Introduction

1.1 Motivation

Consider n motorcycles in the plane, each starting at some point and driving with a constant speed along a straight path. As a motorcycle proceeds it leaves a trace behind it on the plane. Every motorcycle crashes when it reaches the trace left behind by another motorcycle: it stops moving, but its own trace remains. Roughly speaking, the motorcycle graph is the union of the resulting motorcycle traces.

Motorcycle graphs were introduced by Eppstein and Erickson [3], who also presented an algorithm which runs in $O(n^{17/11+\epsilon})$ time. Motorcycle graphs are known to be related to straight skeletons: Cheng and Vigneron [1] studied a straight-skeleton algorithm for simple polygons that computes the motorcycle graph in a preprocessing step in $O(n\sqrt{n} \log n)$ worst-case time. Motorcycle graphs are also related to other problems like visibility and art gallery problems, see [2, 3]. Recently, Eppstein et al. [4] introduced motorcycle graphs on quad literal meshes for extracting canonical partitions of those. However, despite of their practical usefulness no implementation of a sub-quadratic algorithm is known.[1]

1.2 Our contribution

We suggest an algorithm for computing motorcycle graphs that is both easy to implement and fast. We start with a formal definition of the motorcycle graph which does not define the motorcycle graph as the outcome of some process, and argue that our definition conforms to the one given by Eppstein and Erickson. Subsequently, we show how to apply geometric hashing in order to get a simple and easy-to-implement algorithm for computing motorcycle graphs.

Even though the motorcycles might move across large portions of the hash grid during the course of computation, our C++ implementation exhibits a surprisingly good performance in over 22 000 practical tests, involving tests with more than a million motorcycles. More precisely, in the vast majority of our tests we achieve an $O(n \log n)$ behavior. The practical speed of our algorithm prompted us to analyze the expected time complexity of our hash-based algorithm: investigating stochastic questions given by random rays on a rectangular grid allows us to predict an $O(n\sqrt{n} \log n)$ expected complexity of our algorithm for n random motorcycles.

Our implementation can handle a set of motorcycles and a set of rigid walls formed by straight-line segments. (If a motorcycle crashes into a wall then it stops, too.) Furthermore, our implementation can cope with simultaneous crashes of motorcycles at the same place, and new motorcycles can be launched at arbitrary points in the course of computation. Thus, our motorcycle-graph algorithm could be used to compute the straight skeleton by means of an approach similar to the one by Cheng and Vigneron [1].

1.3 Definitions

We regard a triple $m = (p, s, t^*) \in \mathbb{R}^2 \times \mathbb{R}^2 \times [0, \infty)$ as a motorcycle, where p denotes the start point, s denotes the speed vector, and t^* denotes the start time. We denote by $R_m(t) := \{p + t's \,:\, t^* \le t' \le t\}$ the supporting ray of m until time t and define and $R_m(\infty) := \{p + t's \,:\, t^* \le t'\}$. The time $T_m(q)$ when m reaches a point $q \in R_m(\infty)$ is given by $T_m(q) := \frac{(q-p)\cdot s}{||s||^2}$.

Let $m_i = (p_i, s_i, t_i^*)$, with $i \in \{1, \ldots, n\}$, be n motorcycles and assume that their start points $p_1, \ldots, p_n$

*Work supported by Austrian FWF Grant L367-N15.

†Universität Salzburg, FB Computerwissenschaften, A-5020 Salzburg, Austria, shuber@cosy.sbg.ac.at

‡Universität Salzburg, FB Computerwissenschaften, A-5020 Salzburg, Austria, held@cosy.sbg.ac.at

[1] In personal communication with David Eppstein and Siu-Wing Cheng we learned that they also are not aware of implementations of their algorithms.

are pairwise distinct. We consider the following system of conditions for the times $t_1^\dagger, \ldots, t_n^\dagger \in [0, \infty]$:

$$\forall i \in \{1, \ldots, n\}\ \forall j \in \{1, \ldots, n\} \setminus \{i\} : \\ \forall p \in R_{m_i}(t_i^\dagger) \cap R_{m_j}(t_j^\dagger) : \\ T_{m_i}(p) \geq T_{m_j}(p) \Rightarrow t_i^\dagger \leq T_{m_i}(p) \quad (1)$$

Lemma 1 *There always exists a solution vector $(t_1^\dagger, \ldots, t_n^\dagger)$ that fulfills Conditions (1). The solution vector is unique if it is requested to be maximal according to lexicographical order, after rearranging all solutions $t_i^\dagger$ in sorted order.*

Proof. If the supporting rays $R_{m_1}(\infty), \ldots, R_{m_n}(\infty)$ do not cross, then we can set $t_1^\dagger = \cdots = t_n^\dagger := \infty$. So suppose that there are intersections among the supporting rays. We choose an intersection point $q \in \bigcup_{i,j \neq i} R_{m_i}(\infty) \cap R_{m_j}(\infty)$ such that $\max(T_{m_i}(q), T_{m_j}(q))$ is minimized over all intersections. W.l.o.g., we assume that $T_{m_i}(q) \geq T_{m_j}(q)$ and let $t := T_{m_i}(q)$. We can regard m_i as the first motorcycle which crashes; it crashes against m_j at point q in time t. Obviously, setting $t_1^\dagger = \cdots = t_n^\dagger := t$ allows us to fulfill Conditions (1). On the other hand, if we choose $t_i^\dagger > t$ then at least one of the conditions of (1) is violated. Hence, we set $t_i^\dagger := t$. Now we repeat our considerations for intersections among supporting rays of motorcycles in $\{m_1, \ldots, m_n\} \setminus \{m_i\}$, and interpret the result as the second crash of a motorcycle. We keep going until no intersections among the supporting rays of the remaining motorcycles exist. We set $t_k^\dagger := \infty$ for all remaining motorcycles m_k.

By construction, $t_1^\dagger, \ldots, t_n^\dagger$ fulfill Conditions (1). Also due to the construction, the solutions are obtained in sorted order and, thus, are maximal and unique. □

Definition 1 *We call $t_k^\dagger$ (of Conditions (1)) the crashing time of the k-th motorcycle m_k, and we denote by $S_k := R_{m_k}(t_k^\dagger)$ the trace of m_k.*

Definition 2 *We call $\bigcup_{k=1}^n S_k$ the motorcycle graph of the n motorcycles $m_1, \ldots, m_n$.*

Corollary 2 *Motorcycle traces do not intersect in their relative interiors.*

Algorithm 1 A simple algorithm for computing motorcycle graphs is obtained by converting the proof of Lem. 1 into an algorithm: we find the crashes of the motorcycles iteratively in chronological order.

We are not aware of an prior formal definition of a motorcycle graph. However, our definition fits to the considerations of prior work. This can be seen by figuring out that Algorithm 1 exactly computes what is described as "motorcycle graph" in prior publications.

In practice it may be requested to consider straight-line segments as rigid walls such that motorcycles crash when they run into a wall. We can easily extend our definition to satisfy this request: in Def. 2, we do not choose $t_k^\dagger \in [0, \infty]$, but $t_k^\dagger \in [0, t_k^\ddagger]$, where $t_i^\ddagger$ denotes the minimal time $T_{m_k}(q)$ for intersection points q of $R_{m_k}(\infty)$ with a wall. If no such intersection exists for m_k then we resort to the old definition by setting $t_k^\ddagger := \infty$.

2 Computing Motorcycle Graphs

2.1 Algorithm

We first discuss the algorithm by Cheng and Vigneron [1], as our algorithm can be interpreted as a practice-minded simplification of their algorithm. Since there are $O(n^2)$ many intersections of the supporting rays of the motorcycle traces, but only $O(n)$ of them realize a crash, Cheng and Vigneron try to reduce the complexity of interactions among the motorcycles. This is done by using so-called $1/\sqrt{n}$-cuttings, which can be interpreted as a very special kind of geometric hashing. A $1/\sqrt{n}$-cutting is a partition of $\mathbb{R}^2$ into a set of simplices; it has the powerful property that no more than $O(\sqrt{n})$ rays intersect a single simplex.

Basically, the algorithm of Cheng and Vigneron is a discrete simulation of the movement of the motorcycles on the cutting. The simulation consists of two types of events: crash events and switch events. The former one indicates a crash of a motorcycle; the later one indicates a switch of a motorcycle from one simplex of the cutting to a neighboring one. In the course of simulation, events are put into a priority-queue, and the algorithm iteratively fetches the earliest event and processes it. However, we are not aware of any implementation of Cheng and Vigneron's algorithm. In any case, implementing the algorithm for the $1/\sqrt{n}$-cutting does not seem to be easy. In our approach, we replace the $1/\sqrt{n}$-cutting of Cheng and Vigneron by a simple regular rectangular hash grid and drop their arrangements.

The input for our algorithm is a set of motorcycles $M = \{m_1, \ldots, m_n\}$, as defined in Sec. 1.3, and a set W of line segments representing the walls. We assume that all motorcycles start at distinct points that lie within a unit bounding box. We restrict our computation to a larger copy of the bounding box, which can be imposed easily by adding four walls representing the boundary. We maintain a priority queue[2] Q of pending crash and switch events and a list C of (balanced) binary search trees $C[m]$ for each motorcycle m. As already mentioned, we simulate the movement of the motorcycles, as Cheng and Vigneron do: we use a geometric hash H (consisting of uniform rect-

[2] We may use some sort of minimizing heap, where the priority is the occurrence time of an event.

angular cells) for tracking the motorcycle traces and a geometric hash G for the wall-segments. We use the same $h \times h$ uniform rectangular grid for both H and G, with $h \in \Theta(\sqrt{n})$.

Algorithm 2 The basic algorithm first fills G with all walls from W and invokes `insertMc(`m`)` for every motorcycle $m \in M$. Then, the main loop of the algorithm successively fetches the minimal element e from Q and invokes `handle(`e`)` on it, depending on the type of the event e. The procedures `insertMc()` and `handle()` are described in the sequel.

`insertMc(motorcycle` m`)`: We first add to C an empty binary search tree $C[m]$. Then we insert a switch event e for m to Q, with the time of e set to the start time of m.

`handle(switch event)`: We denote by t, m, c the occurrence time, the motorcycle affected, and the cell that is being entered. At first, we add the next switch event of m to Q, if m leaves c at some point in the future.

Then we get all walls from G that are in cells that intersect c and denote them by $walls$. If m crashes into elements of $walls$ then we determine the wall with minimal crashing time and add a respective crash event to Q.

Now we reconstruct the search tree of potential future crash events. First, we clear $C[m]$ and denote by mcs all other motorcycles currently associated with c. For every $m' \in mcs$, we check whether the supporting rays of m and m' intersect within c. If they intersect then we denote by q' the potential crash point. If m' reaches q' before m then we add a corresponding crash event of m into m' to $C[m]$. If, on the other hand, m reaches q' before m' does then we denote by e' a potential crash event of m' into m. If e' is not earlier than the earliest event in $C[m']$ then we insert e' into $C[m']$. Otherwise, we update Q: we remove all possibly remaining crash events of m' from Q and re-add them to $C[m']$, and also add e' to Q.

Finally, we add the minimal element of $C[m]$ to Q, if one exists, and associate m with the cell c of H.

`handle(crash event)`: Let t, m, c denote the occurrence time, the motorcycle affected, and the cell of H in which the crash event occurs. First, we mark m as being crashed, clear $C[m]$, and remove the possibly remaining switch event of m in Q. Note that the trace of m ends at the crash point.

Then we clean up the interactions with other motorcycles: let mcs be the other motorcycles that are associated with the cell c. For every $m' \in mcs$, we remove from Q the crash event of m' against m, if the crash event has become invalid since it occurs outside of the trace of m. Further, if $C[m']$ contains an invalid crash against m then it is also removed from $C[m']$.

Obviously, Alg. 2 determines the crash events in chronological order. Furthermore, we note that Q contains at no time t a crash event against a motorcycle m at place q, if m already crashed and never reached q. Thus, there are no "stale" crash events in Q.

Let k be the number of motorcycles in a hash cell. Then a crash event is handled in $O(k \log n)$ time[3]. Same holds for switch events. Since there are $O(n)$ crash events and $O(n\sqrt{n})$ switch events, and $k \in O(n)$, we get $O(nkh \log n) \subseteq O(n^2\sqrt{n} \log n)$ as the worst-case complexity. Obviously, the worst case would take place if $\Omega(n)$ motorcycles would cross $\Omega(h)$ hash-cells before crashing. For big data sets this basically means that most motorcycles move parallel to each other in a strip with the thickness of a few hash-cells and no other motorcycles cross the strip before them.

2.2 Expected run-time

As witnessed by our experiments, our implementation achieves an "almost linear" run-time on the vast majority of our data sets. This experimental result motivates a formal analysis of the expected run-time of our algorithm. Studying questions related to random rays within a hash grid allows us to substantiate claims for a good average-case complexity of our algorithm. We summarize our results in the following two theorems.

Theorem 3 *Let S be a unit square covered by a $h \times h$ grid. We distribute n random[4] rays in S. The expectation of the number of rays intersecting any cell of the grid is in $\Theta(\frac{n}{h})$.*

Theorem 4 *Consider n random motorcycles within the unit square S. The expected number of cells intersected by a motorcycle trace is in $\Theta(\sqrt[4]{n})$.*

Proofs related to our stochastic analysis are omitted due to lack of space. However, experimental evidence for the second theorem is provided by our tests: see the plot of the mean trace lengths in Fig. 1. As an immediate consequence, we get that the mean number of switch events per motorcycle is in $\Theta(\sqrt[4]{n})$. Vice versa, in a hash cell there are $\Theta(\sqrt[4]{n})$ motorcycles on average. A combination of these results yields an expected run-time of $O(n\sqrt{n} \log n)$, as claimed.

[3] We can remove an arbitrary element of Q in $O(\log n)$ time, if we maintain a pointer in Q.

[4] Start point and direction angle of each ray are distributed uniformly on $S \times [0, 2\pi)$.

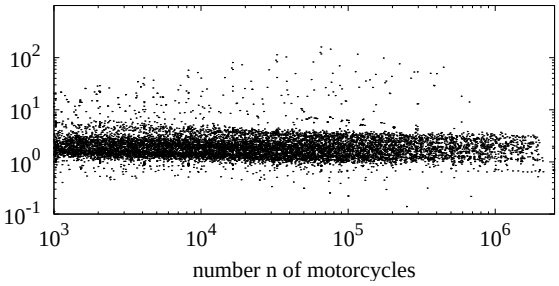

Figure 1: This plot shows for every data set the mean trace length of the motorcycles, multiplied by $\sqrt{n}$, i.e., by the square root of the number of motorcycles.

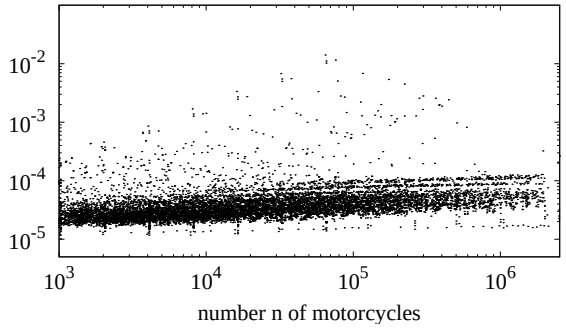

Figure 2: This plot shows the actual run time in seconds on about 22 000 data sets. Depicted are the run-times divided by the number n of motorcycles.

3 Experimental Results

Our code is called MOCA[5]. It was implemented in C++, based on standard IEEE 754 double-precision floating-point arithmetic. To the best of our knowledge, this is the first implementation of a sub-quadratic motorcycle-graph algorithm. For this reason, we do not compare our code with other implementations, but content ourselves with a discussion of the performance of MOCA. The tests presented were run on a 32-bit Debian Linux machine, with a 2.66 GHz Core Duo Intel processor, using 4 GB of RAM. For time measurement, we used the C function `getrusage()` and sum up user and system time.

For our tests we obtained motorcycles from straight-line polygonal chains[6]: We generate motorcycles by considering three consecutive vertices v', v'' and v''' in a chain. A motorcycle starts at time 0 and place v'', in direction $\frac{v''-v'}{||v''-v'||} + \frac{v''-v'''}{||v''-v'''||}$, and with speed $\frac{1}{\sin \alpha/2}$, where α is the angle between the vertices v', v'' and v'''. This corresponds to the set-up used by Cheng and Vigneron [1]. We ran MOCA on more than 22 000 polygonal data sets, consisting of synthetic and real-world data. Our real-world data sets — obtained from companies, colleagues, and the web — include polygonal cross-sections of human organs, GIS maps of roads and river networks, polygonal outlines of fonts, and boundaries of work-pieces for NC machining or stereo-lithography, and the like. The synthetic data also contains contrived data, like extremely smooth polygons or highly irregularly distributed vertices.

Figure 2 illustrates the actual run-time on every single data set in a double-logarithmic run-time plot, with the time given in seconds on the y-axis. For a better illustration, the run-times are divided by the number n of motorcycles processed. (To avoid unreliable timings and other idiosyncrasies of small data sets, we only plot results for test runs with at least 1000 motorcycles.) A least-squares fit reveals that MOCA processes n motorcycles in $5.05489 \cdot 10^{-6} n \log n$ milliseconds on average on our computer. In our experiments, the polygonal chains were inserted as walls. Anyhow, additional tests demonstrated that inserting or disregarding the polygonal chains has hardly any impact on the run-time.

[5] MOtorcycle CrAsher.

[6] The polygonal chains may be open or closed and need not be simple. Several chains per test can be handled.

4 Conclusion

We introduce an easy-to-implement algorithm for computing motorcycle graphs and obtain the surprising result that the combination with geometric hashing makes it very competitive in practice. Extensive practical tests of our C++ on over 22 000 data sets clearly demonstrate the reliability and speed of the C++ implementation of our algorithm, based on standard IEEE 754 floating-point arithmetic: Our implementation processes n motorcycles in $5.05489 \cdot 10^{-6} n \log n$ ms on average on our computer. This experimental result is backed by a stochastic analysis, which leads to $O(n\sqrt{n} \log n)$ as the expected run-time.

References

[1] S.-W. Cheng and A. Vigneron. Motorcycle Graphs and Straight Skeletons. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 156–165, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[2] J. Czyzowicz, I. Rival, and J. Urrutia. Galleries, Light Matchings and Visibility Graphs. In *WADS '89: Proc. of the Workshop on Algorithms and Data Structures*, pages 316–324, London, UK, 1989. Springer-Verlag.

[3] D. Eppstein and J. Erickson. Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. *Discrete Comput. Geom.*, 22(4):569–592, Dec 1999.

[4] D. Eppstein, M. T. Goodrich, E. Kim, and R. Tamstorf. Motorcycle Graphs: Canonical Quad Mesh Partitioning. *Computer Graphics Forum*, 27(5):1477–1486, Sep 2008.