

Applying graphics hardware to achieve extremely fast geometric pattern matching in two and three dimensional transformation space ^{*†}

Dror Aiger[‡]Klara Kedem[§]

Abstract

We present a GPU-based approach to geometric pattern matching. We reduce this problem to finding the depth (maximally covered point) of an arrangement of polytopes in transformation space and describe hardware assisted (GPU) algorithms, which exploit the available set of graphics operations to perform a fast rasterized depth computation.

1 Introduction

A central problem in pattern recognition, computer vision, and robotics, is the question of whether two point sets P and Q resemble each other. One approach to this problem, first used by Huttenlocher et al. [8, 7] is based on the minimum Hausdorff distance between point sets in the plane under translation.

The Hausdorff distance between two point sets P and Q is defined as $H(P, Q) = \max(h(P, Q), h(Q, P))$ where $h(P, Q) = \max_{p \in P} \min_{q \in Q} d(p, q)$ and $d(\cdot, \cdot)$ is a standard metric on points.

In this paper we consider the following geometric pattern matching problem: Given two point sets P and Q in the plane, and some $\delta > 0$, find a 3-parameters transformation $T \in G$ that brings the largest subset S of P to distance $h(T(S), Q) \leq \delta$, where $h(\cdot, \cdot)$ is the directional Hausdorff distance with L_∞ as the underlying metric. We describe two alternatives, in one G is the set of translation + scale transformations, and in the other the set of rigid transformations. We reduce this problem to finding the depth (maximally covered point) of an arrangement of polytopes in transformation space. This reduction is quite common (see, e.g. [4, 3]).

In [2] we gave a randomized algorithm for approximating the *Pattern Matching* problem for point sets under similarity transformation. We refer the reader to that paper for description of related problems and

previous work. The geometric pattern matching problem we solve in this paper is also termed the *Largest Common Pointset (LCP)* problem.

Based on the reduction to depth in a polytopes arrangement, we show that the problem can be solved very fast using a modern standard graphics hardware. Though translation + scale is a linear transformation rigid transformation (rotation and translation) is not. We will discuss approximating the regions in transformation space by a union of convex polytopes with linear boundaries. We use the so called *depth peeling* algorithm [5] implemented on the GPU to get the k -levels one after another while maintaining some structures. This enables us to get the deepest point in the arrangement (with a bounded error which depends on the rasterization) in $O(L)$ passes over the data where L is the number of levels in the arrangement.

2 The GPU as a stream computer

Recently, many GPU-based algorithms for geometry, image processing and other problems have been considered by researchers (see, e.g. [9, 6]). In particular, the GPU as a stream computer for geometric optimization was considered by [1]. In many applications, performing a computation on the graphics card is far faster than performing it on the CPU. The GPU provides spatial parallelism where each pixel on the screen can be viewed as a stream processor, enabling an application to be computed in highly parallel mode.

3 The Largest Common Point set problem (LCP) and its reduction to depth in an arrangement

We reduce the LCP problem to depth in arrangement as follows: All the transformations that bring a point $p \in P$ up to L_∞ distance δ from $q \in Q$ correspond to a region in transformation space [2]. This region is an intersection of four constraints defined by the side of the square of size 2δ around q (see Figure 1). For linear transformations, one such region forms a convex polytope in transformation space where each polytope is the Minkowski sum of a 3D line by a planar square of side size 2δ . The 3D line in transformation space corresponds to all transformations that bring $p \in P$ exactly to $q \in Q$ and its Minkowski sum

^{*}This work was partly supported by the MAGNET program of the Israel Ministry of Industry and Trade (IMG4 consortium)

[†]The authors thank the Frankel Foundation for supporting presentation of this paper in EWCG'07 in Graz, Austria

[‡]Department of Computer Science, Ben Gurion University, Be'er Sheva, Israel and Orbotech LTD aiger@cs.bgu.ac.il, dror-ai@orbotech.com

[§]Department of Computer Science, Ben Gurion University and Cornell University, klara@cs.bgu.ac.il

corresponds to all transformations that bring a point $p \in P$ to a square of side size 2δ about $q \in Q$, so that $h(T(p), q) \leq \delta$ using the L_∞ metric. For rigid transformation, this region forms a Minkowski sum of an arc with the same square, since the transformation is not linear in the angle parameter. In this case we approximate the region with the union of convex polytopes. The solution is then approximated in the sense that the distance of the solution from the true optimal solution is bounded by the maximum error of our approximation (see Section 4.3).

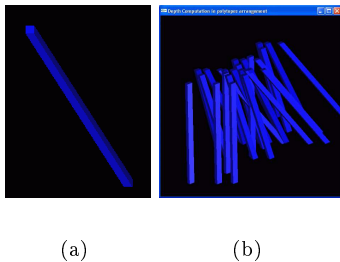


Figure 1: Polytopes in transformation space: (a) polytope created for a single match of p into a 2δ -square about q (b) arrangement of polytopes for all points in P and all points of Q .

For each pair (p, q) , $p \in P$ and $q \in Q$, there is a polytope. We assume that the polytopes that correspond to one point p in P are disjoint in their interior (if this is not the case, we can decompose the union of squares of size 2δ around points in Q to $O(|Q|)$ rectangles disjoint in their interiors as in [4]). We investigate the depth of the polytope arrangement: the depth of a point t in transformation space is the number of polytopes that contain t . A point of maximum depth corresponds to a transformation that brings the maximum number of points in P close to Q [4]. The regions in transformation space that are covered by $|P|$ of these $|P||Q|$ polytopes are thus the locus of transformations that bring all points of P close enough to a point of Q . Since we are looking for matching the largest subset S of P to points of Q , we will search for the maximal depth of that arrangement.

4 Computing the maximum depth

4.1 Using k -levels

In this section we show how we compute the maximum depth in 3D transformation space using the GPU.

Definition 1 Given a set C of hyperplanes, a point p is said to be at k -level, if there are exactly k hyperplanes in C lying strictly below p .

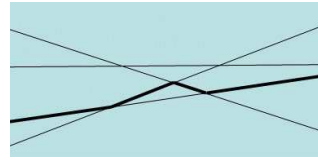


Figure 2: k -levels in arrangement of lines. The bold line is the 1-level.

In an arrangement of convex polytopes, the depth is the number of objects that a ray down to $-\infty$ crosses only once (see Figure 2). The depth can be computed by going through the k -levels, one by one and counting the number of “entering into” and “exiting” polytope events. If we count and remember the maximum, we end up with the point of maximum depth.

4.2 Applying the GPU in the computation of k -levels

We use the GPU to compute a rasterized version of the k -levels for 3D transformation space. (The rasterization induces an error bounded by the pixel size.) Simple rendering using the depth buffer computes the 0-level using the standard z-buffer. The z-buffer performs the test “is $x < a$ ” in its standard process for hidden surface removal[10] while surfaces or planes are being rendered.

Our goal is to “peel” the k -levels one by one when our scene is built from all the polytopes defined above. During this process we count for every pixel all the objects that contain it by counting entry into and exit of polytopes. Using two depth buffers simultaneously plus the previous level, we can perform the test “is $a < x < b$ ” and thus we get the 1-level for all the pixels simultaneously. Similarly we follow to all k -levels.

This process is known as *depth peeling* and uses the shadow map as a secondary depth buffer (see [5] for the detailed algorithm). Throughout the process we count the number of entries and exits (from polytopes) for each pixel, thus getting the maximum depth in the polytope arrangement. By applying fragment program (FPs) [10] we compute the maximum depth at each pixel. The entire process requires L passes over the input scene when L is the number of levels and no CPU involvement is needed during the process.

Once we finish, we have the translation (t_x, t_y) at which the maximum depth is attained but we still have to compute the third parameter of the transformation (scale, or rotation in the rigid case).

For finding the right scale efficiently we have to find the scale coordinate for (t_x, t_y) . We perform another pass similar to the one above but now only on the pixel (t_x, t_y) applying *selection mode* on that pixel. Selection mode is a mode of rendering in which the

depth order of the objects within a given window is saved during the rendering and can be obtained by the application afterwards. The output of the *selection mode* are the ordered faces in the arrangement that meet the ray from (t_x, t_y) to $-\infty$. The scale coordinate of maximum depth in the arrangement is thus found.

4.3 Computing the depth for rigid transformations

The only difference between rigid transformations and scale + rotation transformations is that the rigid is not a linear transformation. Thus a linear constraint of the square of size 2δ around points of Q in the plane does no longer correspond to a hyperplane in transformation space. The transformations that bring a point $p \in P$ to a point $q \in Q$ form an arc in $3D$ instead of a line as before. For a given error bound, we can approximate this arc by a set of segments.

By evaluating the allowed error we divide our rotation axis into a set of slices such that in any slice the arc is approximated by a line segment. The Minkowski sums of the polygonal lines approximating the arcs with a square of size 2δ yield the polytopes on which we apply the method described in the previous subsection.

4.4 Speeding up by randomization and oriented points

For simplicity we assume $|P| = |Q| = n$. The complexity of the peeling algorithm is $O(L)$ rendering passes over the data which contains n^2 polytopes where L is the number of levels. The number of levels is view dependent but can be $O(n^2)$. Thus we have runtime complexity of $O(n^4)$ (it is typically close to $O(n^3)$ in practical cases since $L = O(n)$ in most cases). A significant speed up can be achieved by applying randomization and oriented points, combined with stencil buffer and occlusion query. (The stencil buffer is used to mask a specific region in the frame buffer to which we want to restrict the rendering. Occlusion query is an Open GL operation [10] that enables the application to count the pixels in the frame buffer which have been updated during a specific rendering process.)

We decompose the problem to a set of smaller problems and use the GPU to quickly reject polytopes that cannot contribute to the optimal solution. Since our polytopes are the Minkowski sum of a $3D$ line (in the scale + translation case) and a planar square of side size 2δ , the polytopes are actually **sticks** and the intersection of a pair of them is expected to be a small region in $3D$ (especially if the angle between the $3D$ lines is large enough). To decompose the problem to smaller problems we use randomization as follows: Given $N = n \cdot n$ polytopes (each $p \in P$ transformed to a square around each point $q \in Q$), we know that

the depth is bounded by n . If we assume that a constant fraction c of the points of P , $c \cdot n$ are matched to points in Q , the probability that the intersection of two random polytopes contains the desired solution is $constant/N$. We sample $O(N \log N)$ pairs of polytopes to get high probability that the intersection of one pair contains the desired solution. We now have $O(N \log N)$ small regions in transformation space. For each region we mask the intersected region on the stencil, rendering all polytopes and using the occlusion query to reject polytopes that do not intersect these regions (their number is expected to be large).

The number of polytopes intersecting the regions is typically $O(n)$ instead of the original size $N = n \cdot n$, thus we need only $O(n)$ passes over $O(n)$ polytopes. In the worst case, we can still get a high number of polytopes if there is a large number of optimal solutions but this is rare in practice.

We can further speed up the process by using oriented points. In many real life applications we get orientation at each point (e.g. from edge detection). We construct only polytopes that correspond to validly matching orientations of points $p \in P$ and $q \in Q$ (when the difference between orientations after transformation is small, up to a threshold). The number of polytopes is thus reduced dramatically in the translation + scale problem. For rigid transformations, the size of the polytopes in transformation space is reduced dramatically.

5 Experiments and results

The tests we present here are all under translation and scale. Rigid transformation is essentially the same and was not implemented. We implement the peeling algorithm with randomization and orientations on points. We used a PC running at 3Ghz with Windows XP and OpenGL. The GPU is the Nvidia GeForce 6600. Below we show a synthetic example where we graph the runtime as a function of pattern points and a real time object recognition example that demonstrates the power of the GPU in real life application.

5.1 Synthetic data

For the synthetic test, we randomly create 1000 oriented points in the range $[-1, 1]$ (the orientations were randomly selected from the range $[0, 2\pi]$) as the set Q . For the set P we randomly select a subset of points of Q and perturb each point with a uniform distribution in a small neighborhood of the selected point. For the square size around each point of Q we picked $\delta = 0.004$. For a pair (p, q) we create a polytope if the difference of orientations of p and q is up to 5 degrees. The raster resolution (pixel size) is 0.002. The scale was bounded to be in the range $[0.5, 2.0]$. We

note that for the GPU implementation the runtime is almost not affected by the size of the search space.

The error on the various axes differs as in translation it is determined by pixel size and in scale axis it depends on the number of bits in each pixel in the depth buffer which is 32 bits in our implementation, and thus pretty precise.

The running times using this data are shown in Figure 3, where we picked P to be of 50, 100, 200 and 400 points. Notice that the graph shows linear dependency of runtime as a function of the number of points in P .

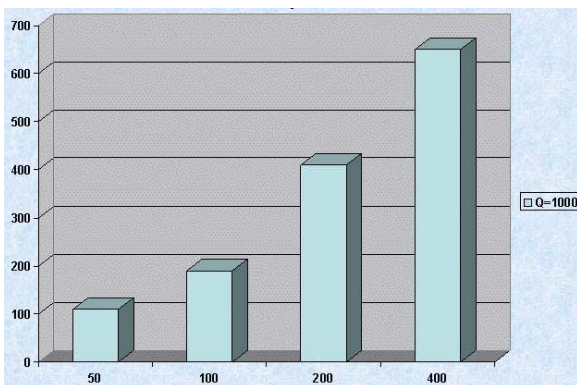


Figure 3: runtime(milliseconds) vs. different number of pattern points, while $Q=1000$. We used all the speed up methods to reduce the computation time

5.2 Application in real time object recognition

We tested our geometric matching method in a real application of model based object recognition. For a 512×512 gray level input image (Figure 4(a)) and a given geometry model (Figure 4(b)), an edge detection was first applied to the input image (Figure 4(c)), and oriented points were extracted from both the image and the model. For the similarly oriented points we constructed the polytope arrangement, applied our method to compute the region of maximum depth in this arrangement and then retrieved the best transformation from this region. The matching was performed allowing translation and scale change. The parameters are the same as in the synthetic case after normalization of the image data to the range $[-1,1]$ as before. The results are shown in Figure 4(d).

6 Summary

Based on our theoretical work [2], on GPU capabilities and on some caveats we present practical real time algorithms and implementations of shape resemblance. The algorithms work for any three parameters transformation (rigid and scale + translation). Our future plans are to investigate the expansion to higher

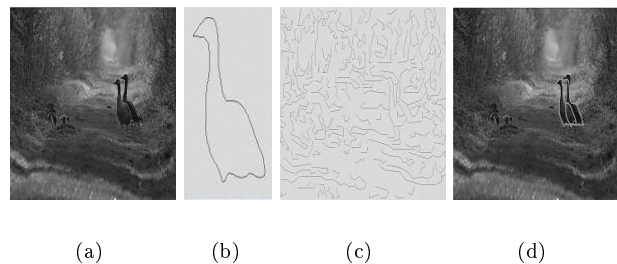


Figure 4: Model based object recognition: (a) input image, (b) model, (c) edge map, (d) detection results. The whole recognition (edge detection + matching) was done in 20 milliseconds while the matching alone took 4 milliseconds

degree of transformation space and to allow the use of line segments instead of points.

References

- [1] P. K. Agarwal, S. Krishnan, N. H. Mustafa, S. Venkatasubramanian, "Streaming Geometric Optimization Using Graphics Hardware". *ESA 2003*: 544-555.
- [2] D. Aiger, K. Kedem, "Exact and Approximate Geometric Pattern Matching for point sets in the plane under similarity transformations", submitted.
- [3] H. S. Baird, "Model Based Image Matching Using Location", *MIT press*, Cambridge, MA, 1985.
- [4] L. P. Chew, K. Kedem, "Getting around a lower bound for the minimum Hausdorff distance", *Comput. Geom.* 10(3): 197-202 (1998).
- [5] C. Everitt, "Interactive order-independent transparency". *Technical report*, NVIDIA Corporation, May 2001.
- [6] K. E. Hoff III, Andrew Zaferakis, Ming C. Lin, Dinesh Manocha, "Fast and simple 2D geometric proximity queries using graphics hardware". *SI3D 2001*: 145-148
- [7] D.P. Huttenlocher, K. Kedem, and M. Sharir, "The upper envelope of Voronoi surfaces and its applications", *Discrete and Computational Geometry*, 9(1993), pp 267-291.
- [8] D.P. Huttenlocher and K. Kedem, "Computing the Hausdorff distance for point sets under translation", *Proceedings of the Sixth ACM Symposium on Computational Geometry*, 1990, pp 340-349.

- [9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krager, A. E. Lefohn, T. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". *Eurographics 2005*, State of the Art Reports, August 2005, pp. 21-51.
- [10] The Industry's Foundation for High Performance Graphics - <http://www.opengl.org>