

Balancing Runtime Space- and Time Complexity in Synthetic Database Driven Hand Posture Reconstruction Systems

H. A. C. de Villiers*, T. R. Niesler* and L. van Zijl†

*Digital Signal Processing Group

Department of Electrical and Electronic Engineering
University of Stellenbosch, Stellenbosch, South Africa
Email: hdevil@sun.ac.za, trn@dsp.sun.ac.za

†Department of Computer Science

University of Stellenbosch, Stellenbosch, South Africa
Email: lynette@cs.sun.ac.za

Abstract—Hand posture reconstruction systems based on large databases of synthetically rendered images of a 3D hand model offer a simple and flexible means of exploiting domain knowledge to provide training data. Such systems may also be applied to other domains in the posture reconstruction field by changing the model under consideration.

Typically, the index structures used to answer similarity queries at runtime explicitly contain the prerendered feature data. However, the combinatorial explosion resulting from the multiple degrees of freedom available to the human hand severely limits the complexity of feature data that may be embedded into the index structure.

The system presented in this paper exploits real-time object-space rendering techniques to rebalance the preprocessing and runtime workloads such that the space complexity of the database relative to the number of degrees of freedom is greatly reduced.

A prototype of the database subsystem is implemented and its properties investigated to obtain insight into its scaling behaviour.

I. INTRODUCTION

Hand posture reconstruction systems must be able to handle the multitude of poses that the human hand is capable of assuming. This is particularly important in applications where high accuracy is desired, such as sign language recognition systems.

The combinatorial explosion in the number of possible poses makes the acquisition of sufficient quantities of real training data challenging. This stands in marked contrast to the amount of domain knowledge humans can provide with respect to the structure and appearance of the hand.

Training data might also be obtained by synthesis and, while a measure of caution is needed in dealing with synthetic data, its use may be justifiable in cases where domain knowledge is extensive. For example, a 3D model of a hand may be created fairly easily.

Systems of the type under consideration in this paper are those which make use of a large database of prerendered feature data, indexed in such a way that a nearest neighbour search can be performed efficiently. Of particular interest are

the systems demonstrated by Athitsos et al. [1]–[6], and Dick, Zieren and Kraiss [7] (detailed in Zieren’s PhD dissertation [8]). The former group focused on unadorned hands, while the latter made use of coloured gloves.

An important common feature of these systems is the presence of both structural (index) information and prerendered feature data within the database. Because the number of feature data entries is the *product* of the number of different settings made available to each individual degree of freedom, the space needed by the feature data grows exponentially in the number of degrees of freedom. Due to this behaviour, a difficult trade-off exists between the complexity of the feature data and the allowable size of the purely structural part of the index. The feature part of a database entry tends to be much larger than the structural portion of that entry, and so becomes a large constant multiplier in calculating the total database size (relative to a purely structural database).

One way of addressing this problem is to select features which can be compactly represented. Zieren et al. followed this route by choosing to approximate the coloured marker regions (the fingers and palm marker) as ellipses. Another way, employed by Athistos and Sclaroff, is to embed the feature data into a low-dimensional space, effectively compressing each feature entry to a manageable size.

In the sections that follow, the authors outline a different approach where synthetic feature data is not stored in the database, but resynthesised as needed at runtime by efficient real-time rendering techniques. The rendering process is primed by a separate database containing preprocessed geometric data obtained from the 3D model. This database, however, grows linearly with respect to the number of degrees of freedom. Because the index structure contains only structural information, there is much greater freedom with regard to the complexity of the features that may be chosen.

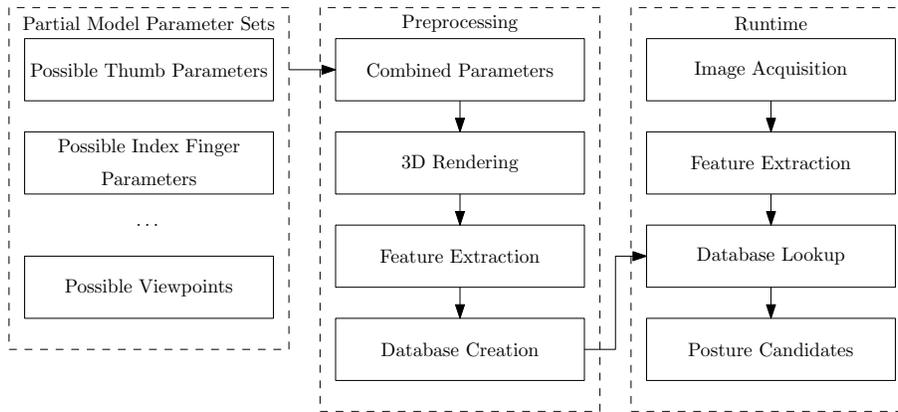


Fig. 1. Conventional System

II. SYSTEM OVERVIEW

A generalised representation of the systems described by Athitsos et al. and Dick, Zieren and Kraiss is shown in Figure 1. Such systems rest on the implicit assumption that the calculation of feature data is “too difficult” to do at runtime, and so is finalised during preprocessing.

This creates a heavy burden in the form of a massive increase in the size of the database. In essence, this represents an extreme design choice in favour of decreased time complexity of runtime database queries at great cost in terms of space complexity (database size). Yet, the index structures employed typically aim to minimise the number of feature comparisons, ideally approaching (in the case of exact query resolution) $\log_2 N$ of the number of database entries (N). This suggests that an increase in computational effort during feature comparison may be justified in order to achieve a significant reduction in the size of preprocessed feature data.

ZeZula [9] provides an excellent introduction to a large class of such index structures (those based around metric spaces), of which the Vantage Point Tree (see Yianilos [10]) is a particularly elegant illustration.

Assuming that the index structure has been created, one can imagine another extreme in which the *entire* rendering operation is performed each time synthetic feature data is required during the runtime query process. While trivial to implement, this solution is unsatisfactory because of its underutilisation of preprocessing time.

The authors propose a compromise between these two extremes, grounded in the observation that each finger of the hand may be rendered separately, and these only affect each other towards the end of the rendering process during hidden surface removal. For each viewpoint, and each allowed individual finger posture, the geometry of the mesh is processed to the point where it must be combined with other component meshes for hidden surface removal to begin. Significant savings in processing time can be made by ensuring that the input to the final rendering stage is in a convenient form.

A block diagram of the resultant system is shown in Figure 2. The core difference between this system and the

conventional system shown in Figure 1 is the splitting of the index structure and the (proto)-feature data. In addition, the runtime system must now also call the rendering subsystem in order to obtain synthetic feature data, which is needed to resolve queries in the index structure.

Both conventional systems discussed previously made use of standard rasterisation techniques during rendering, producing an image at some chosen resolution which must then be processed in a per-pixel fashion towards extracting features. This has the disadvantage of introducing a trade-off between accuracy and speed when choosing the image resolution. However, only the resultant features needed to be stored in these systems, allowing one to discard the rasterised image. In order to stop the process just before hidden surface removal, one would have to store entire synthetic images (along with depth information in the form of a Z-buffer).

Instead, desiring only the shapes of marker regions, we forgo rasterisation, performing all processing steps in object space. This has the advantage of needing only a compact description of the separate model component geometries as input.

The detailed operation of the system will now be discussed, beginning with a description of the workings of the rendering pipeline.

III. RENDERING PIPELINE

For a general overview of object-space hidden surface removal algorithms, the reader is referred to Ghali [11]. In existing literature, algorithms of particular interest are those proposed by Sechrest and Greenberg [12], and Goodrich [13]. The proposed rendering pipeline draws strongly on the ideas presented by these authors. It differs, however, in its emphasis on contour information preprocessing to increase runtime execution speed. It also differs in its removal of non-silhouette vertices of non-planar (though locally planar) surfaces in order to dramatically reduce the space requirements of the preprocessed data.

From the outset, it seems necessary to process each of the model’s mesh vertices when performing the render operation at runtime. This is, however, not necessarily the case. If only

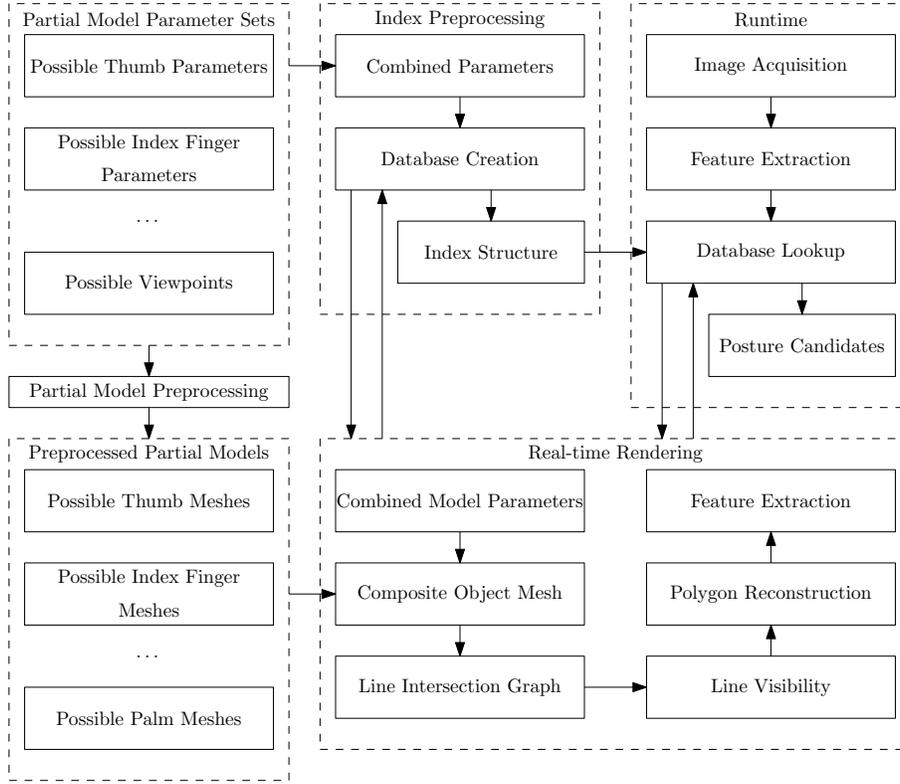


Fig. 2. Overview of the Proposed System

the visible polygons need to be reconstructed (along with some descriptor of which surface is associated with each polygon), a significant reduction in algorithmic complexity can be achieved.

Key to the system presented here is the need to stop just short of hidden surface removal. Usually, a description of the entire surface associated with the 3D model needs to be available. However, if one guarantees that no two surfaces intersect anywhere other than their boundary curves, it can be shown that one no longer needs all the mesh vertices to remove hidden surfaces. Only those vertices need to be stored which, from the viewer's perspective, define the outer edges of the surfaces. This is illustrated in Figure 3, which shows how the depth order of two non-intersecting regions may be determined at the crossing of their boundaries.

This non-intersection criterion is primarily enforced through careful design of the object mesh. However, combinations of joint angles leading to physically impossible intersections of mesh components may be detected during preprocessing, and then excluded from the index structure. Thus, the runtime system never needs to consider this question.

The renderer assumes that the effects of perspective are negligible over the volume of the hand, and performs orthogonal projection. For added simplicity, the image plane is chosen to be the plane where $z = 0$, meaning that projection onto the image plane requires only dropping the z -component of a particular vertex.

Rendering a model in this system consists of a preprocessing

phase and a runtime phase, encompassing several steps each. The output of the various stages is visualised in Figure 4.

A. Preprocessing Phase

- 1) Rotate partial object mesh according to viewpoint.
- 2) Backface culling.
- 3) Remove all internal edges.
- 4) Create reusable boundary representations for fast boundary intersection determination.

B. Runtime Phase

- 1) Combine preprocessed boundary representations.
- 2) Create graph of boundary intersections.
- 3) Determine boundary contour visibility.
- 4) Reconstruct visible polygons.

Each step of the process will now be discussed in greater detail.

A. Preprocessing phase

1) *Partial mesh rotation according to viewpoint:* Changes in viewpoint are simulated by rotating mesh vertices, keeping the viewer located at positive infinity along the z -axis, looking towards the origin.

2) *Backface culling:* All triangles which have their back-faces turned towards the viewer are removed.

3) *Remove all internal edges:* Assuming the object is represented by surface meshes built from triangles, each edge is associated with at most two triangles. Internal edges are those edges which have two forward facing triangles associated

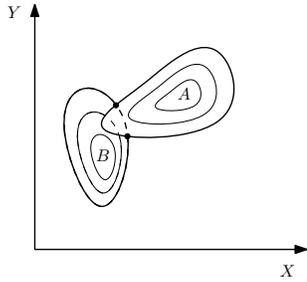


Fig. 3. An example showing two surfaces A and B , with A being closest to the viewer. Note the intersecting projection boundary points which determine the visibility of the overlapping surface regions.

with them. Since backface culling has been performed, all triangles are turned towards the viewer, and this step consists of removing all edges which have two triangles associated with them. This leaves only the boundary contours.

4) *Create reusable boundary representations*: The line intersection algorithm makes use of a priority queue to order the contours in a convenient way. The sort operation needed during queue construction can be performed on the partial mesh boundaries, easing the subsequent runtime merging operations.

First, boundary contours are decomposed into curves with vertices arranged in lexicographic order. For vertices v_1 and v_2 , v_1 is lexicographically before v_2 if and only if either $x(v_1) < x(v_2)$, or $x(v_1) = x(v_2)$ and $y(v_1) < y(v_2)$. The curves are annotated to indicate which side the associated surface (if any) is attached to, as well as the descriptor of that surface.

These curves are inserted into a priority queue, sorted by the first vertex in each curve. This priority queue serves as input to the runtime rendering system. Note that, at this point, the space complexity of the data need only be $O(k)$, where k is the number of boundary points. Essentially, this means that a representation has been found consisting of *only* the projected “shape” of the object. The space complexity, therefore, scales roughly with contour length and complexity, rather than surface area and complexity. This similarly reduces the subsequent time complexity of runtime rendering operations.

B. Runtime operation

1) Combine preprocessed boundary representations:

The rendering operation commences with merging (non-destructively) the priority queues containing the boundary information from each mesh component. The details of this operation depend on the representation of priority queues. For complex models, a Fibonacci heap [14] delivers constant time merge operations (balancing the work of merging structures across later dequeuing operations, which each have amortised $O(\log(m))$ complexity). However, in the case of simpler models, a simple sorted list representation may be preferable to avoid unnecessary overhead.

2) *Create graph of boundary intersections*: The boundary contour intersections are now determined using a version of the Bentley-Ottmann algorithm [15]. The authors’ implementation

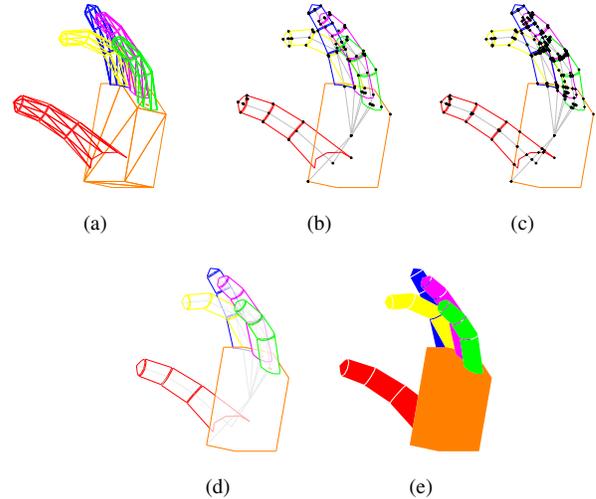


Fig. 4. The stages of the rendering pipeline, illustrated using a complete hand mesh — Preprocessing phase: (a) Wireframe hand model; (b) Mesh component boundaries decomposed into lexicographically sorted curves — Runtime phase: (c) Line intersection graph construction; (d) Determination of line intersection; and, (e) Visible polygon reconstruction.

closely follows that of [16], but takes additional advantage of the continuity of curves.

Essentially, the Bentley-Ottmann algorithm operates by considering events of interest in lexicographic order. Unprocessed events are stored in a priority queue called the X -structure. Initially, this structure contains only the start events of each curve, provided by the previous step in the rendering process.

A list of currently active curves is maintained in what is called the Y -structure. When a curve is first encountered, an entry for it is created in the Y -structure in a way which maintains the current y -ordering of the active curves. In this way, only the neighbours of a curve within the Y -structure need to be considered to find the next possible intersection. A further reduction in curve comparisons may be achieved by ensuring that intersections are removed from each mesh component’s preprocessed boundary information, eliminating the need for comparing curves from the same mesh component.

Intersection events, when detected, are inserted into the X -structure to be handled when intervening events have finished processing. An intersection essentially changes the order of elements in the Y -structure.

Changes of curve direction are also scheduled as events, and these trigger checks for future intersections along the new curve segment. The termination of a curve leads to its removal from the Y -structure.

When a group of events have the same image plane projection, representing some combination of the preceding cases, somewhat more complex behaviour results. Ultimately though, the proper order of the active curves within the Y -structure must be maintained after all events have been processed. This may be done by performing a small sort operation on the relevant subsequence of the Y -structure.

The output of the algorithm is a graph. At each point in the

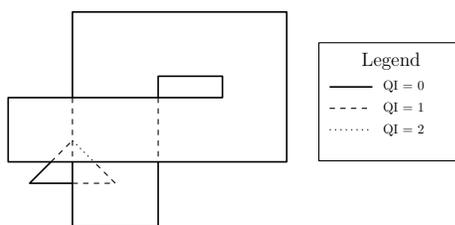


Fig. 5. Illustration of Quantitative Invisibility

image plane where an event is processed, a node is created which connects the curves passing through it. The curves are noted as edges between nodes in the output graph.

The algorithmic complexity of this entire rendering step depends on the time complexity of the X -structure merging, insertion and dequeuing operations, and the average number of active contours within the Y -structure during operation. Specifically, encountering a new curve triggers a search for the proper insertion point in the Y -structure. However, careful use of backpointers to Y -structure elements in events inserted into the X -structure make the search operation constant time for the other operations. The number of events processed equals the sum of curve beginnings, changes in curve direction, curve endings, as well as the number of intersections between curves.

3) *Determine boundary contour visibility*: Appel [17] introduced the concept of Quantitative Invisibility (QI), the number of surfaces between the viewer and the contour. By considering all boundary intersections, QI may be “propagated” through the resulting graph to determine contour visibility.

Figure 5 illustrates line visibility in terms of QI. Note that changes in QI occur only at boundary crossings.

There are two caveats to take into consideration. Firstly, only the depth *order* is determined by boundary crossings. Thus, only the relative order of the two surfaces within the global list of surfaces present at the boundary intersection is known, and not their absolute positions within the list of surfaces. Secondly, it may occur that there are no boundary intersections between two surfaces (for example, where a surface is either entirely in front of or behind another surface).

It is clear that the *absolute* QI needs to propagate along continuous contours, changing at intersections. In order to provide a graph which is fully connected in 3D, one may provide auxiliary contours which are unassociated with a surface, but pass through regions which are known to be free of surfaces, meaning QI may be safely propagated along them. Connecting isolated boundary contours with such auxiliary contours allows absolute QI to propagate throughout the intersection graph. Note their presence (running through the finger cores and the palm) in Figures 4b through 4d.

QI propagation takes the form of a depth-first traversal of the boundary intersection graph, only following continuous connections. The traversal must start at a point of known QI. We select the first point considered by the Bentley-Ottmann algorithm. At least one curve must start at this point, and the top-most one should then have a QI of zero.

Intersections with boundary contours have the effect of either increasing or decreasing the current QI by one, or by leaving it unchanged, depending on whether the intersecting boundary curve passes in front of or behind the current curve. If the intersecting curve passes in front of the current curve, its associated surface and the relative directions of the two curves within the image plane determine whether the change is an increase or a decrease, representing respectively the occlusion or the uncovering of the current curve.

4) *Reconstruct visible polygons*: Visible polygon reconstruction is performed in two steps. During the first step, the depth order of surfaces present in the arc between neighbouring edges around a node is determined.

A depth-first traversal is performed, selecting a known visible edge by, again, considering the first node outputted by the Bentley-Ottmann algorithm. There are no surfaces at a point just anticlockwise from the closest edge clockwise from the vertical. The initial (empty) surface list is located here.

From this edge, going clockwise, the algorithm enumerates the surface lists between each pair of edges. At each edge, the current surface list is recorded as the surface list anticlockwise of the edge. If the edge has an associated surface, the edge’s QI is used as an index into the current surface list for insertion or removal according to the direction in which the surface extends. The resulting surface list is recorded as the surface list clockwise of the current edge.

If an edge connects to an unvisited node, the edge is traversed (after all edges at the current node have been considered) and the process repeated. A surface list bordering the new node is obtained by observing that a surface list clockwise of an edge at the first end-node is the surface list anticlockwise of the edge at the other end-node.

The final step in polygon reconstruction considers each node in lexicographic order. At every visible edge (edges with $QI = 0$) which has a surface associated with it, the surface list clockwise of it is taken as a starting point. The surface list is marked as “visited”, but if it has already been marked previously, the current edge is ignored instead. Assuming the list was unmarked, the next visible edge is found and its **anticlockwise** surface list marked as visited. A recursive traversal is performed, following the boundary in an anticlockwise fashion, marking the appropriate clockwise and anticlockwise surface lists as needed. When a surface list is discovered that has already been visited, it must be the first edge where the traversal began. The edges visited are now concatenated into a list of polygon vertices which, together with the surface descriptor, form one element in the output list of polygons.

IV. EXPERIMENTAL METHODOLOGY

A prototype system was implemented in order to test the scaling behaviour of the preprocessed partial feature database.

A hand model was created based on a set of component meshes representing the fingers and palm. Each finger is controlled separately by selecting from predefined sets of joint parameters. The sets of joint parameters were purposefully

chosen to be the same as those presented by Zieren [8], in order to use that system as a baseline. Accordingly, the thumb, index finger, middle finger, ring finger and pinky have 7, 9, 8, 9 and 9 possible settings respectively.

The set of possible viewpoints was, again, chosen to match Zieren [8] as closely as possible, in which the database contained 105 viewpoints. For the prototype, the viewpoints were derived from a spherical covering of 130 points, made available in tabular form by Sloane et al. [18].

All the necessary preprocessing for runtime operation was performed. In the next section, key characteristics of the resulting partial feature database are examined and contrasted with the baseline system.

V. RESULTS AND DISCUSSION

The number of component mesh entries in the prototype database equals $130 \cdot (7 + 9 + 8 + 9 + 9) = 5460$. This contrasts sharply with the baseline database which, without elimination of impossible combinations, contains $105 \cdot 7 \cdot 9 \cdot 8 \cdot 9 \cdot 9 = 4286520$ feature data entries (after elimination, this was reduced to 2451960). At runtime, the baseline system is reported to have used approximately 477 Mb for storing feature data, each entry using 204 bytes. While retrieval of feature data is trivial in this case, it should be clear that even at relatively rough quantisation of finger postures, the combinatorial growth of the database places severe constraints on the complexity of feature data. In order to reduce feature complexity, fingers were approximated as ellipse-shaped markers.

In the prototype database, each component mesh entry consists of a list of curves, each curve containing a list of vertices and up to two associated surface descriptors. It was found that the total number of vertices is 542937 (of which 305271 are unique). The number of pregenerated curves equalled 197948, implying an average of approximately 2.74 vertices per curve entry.

Assume each vertex is composed of a triplet of double-precision floating point numbers (64 bits each). The total space requirement to store the 305271 unique vertices is then approximately 7 Mb. If a curve is composed of an array of vertex references (32-bit pointers), an array length (32-bit integer) and a pair of 32-bit integer surface descriptor identifiers, the 197948 curve objects require only $(542937 \cdot 4) + (3 \cdot 4 \cdot 197948) \approx 4.33$ Mb. Similarly, defining a component mesh entry as an array of curve references along with an array length, the component mesh entries require $5460 \cdot 4 + 197948 \cdot 4 \approx 0.77$ Mb of storage. In total, the prototype database can be stored in under 20 Mb.

It is immediately apparent that the space requirements are far lower in the prototype database than the baseline system. If one further notes that the prototype database also allows a far greater amount of detail to be represented (fully detailed silhouettes, as opposed to approximating ellipses), the advantage of this approach becomes clear.

VI. CONCLUSION

It has been shown that a significant reduction in the runtime space complexity of a database driven posture reconstruction

system can be achieved by foregoing hidden surface removal during database construction. In support of this, an efficient, object space, procedure for hidden surface removal and visible polygon reconstruction has been presented that limits the necessary increase in time complexity. The rendering pipeline was implemented and used to generate a prototype feature database. The resulting feature database compares favourably in terms of compactness and the amount of detail available during query processing at runtime.

ACKNOWLEDGEMENTS

The authors would like to thank the Wilhelm Frank trust for financial support during the course of this work.

Parts of this work were performed using the High Performance Computing facility at the University of Stellenbosch.

REFERENCES

- [1] V. Athitsos and S. Sclaroff, "Estimating 3D hand pose from a cluttered image," in *Proc. IEEE Comput. Soc. Conf. Computer Vision and Pattern Recognition*, 2003, pp. II:432–439.
- [2] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios, "Boostmap: A method for efficient approximate similarity rankings," in *Proc. IEEE Comput. Soc. Conf. Computer Vision and Pattern Recognition*. IEEE Computer Society, 2004, pp. II:268–275.
- [3] —, "Learning Euclidean embeddings for indexing and classification," Tech. Rep. 2004-014, Boston University, Apr. 2004.
- [4] V. Athitsos and S. Sclaroff, "Database indexing methods for 3D hand pose estimation," in *Gesture-Based Communication in Human-Computer Interaction*. Springer, 2004, pp. 288–299.
- [5] —, "An appearance-based framework for 3D hand shape classification and camera viewpoint estimation," in *Proc. 5th IEEE Int. Conf. Automatic Face and Gesture Recognition*, 2002, pp. 45–50.
- [6] V. Athitsos, A. Stefan, Q. Yuan, and S. Sclaroff, "ClassMap: Efficient multiclass recognition via embeddings," in *Proc. 11th IEEE Int. Conf. Computer Vision*, 2007, pp. 1–8.
- [7] T. Dick, J. Zieren, and K. Kraiss, "Visual hand posture recognition in monocular image sequences," in *Proc. of the 28th DAGM Symp.* Springer, 2006, pp. 566–575.
- [8] J. Zieren, "Visuelle Erkennung von Handposituren für einen interaktiven Gebärdensprach Tutor," Ph.D. dissertation, RWTH Aachen, 2007.
- [9] P. Zezula, *Similarity search: The metric space approach*. New York: Springer, 2006.
- [10] P. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proc. 4th ACM-SIAM Symp. Discrete algorithms*, 1993, pp. 311–321.
- [11] S. Ghali, "A survey of practical object space visibility algorithms," *SIGGRAPH 2001 Course 6 Tutorial Notes*, 2001.
- [12] S. Sechrest and D. P. Greenberg, "A visible polygon reconstruction algorithm," *ACM Trans. Graphics*, vol. 1, no. 1, pp. 25–42, 1982.
- [13] M. Goodrich, "A polygonal approach to hidden-line and hidden-surface elimination," *CVGIP: Graphical Models and Image Processing*, vol. 54, no. 1, pp. 1–12, Jan. 1992.
- [14] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987.
- [15] J. L. Bentley and T. A. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 643–647, 1979.
- [16] U. Bartuschka, K. Mehlhorn, and S. Näher, "A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem," in *Proc. Workshop on Algorithm Engineering*, 1997, pp. 124–135.
- [17] A. Appel, "The notion of quantitative invisibility and the machine rendering of solids," in *Proc. 22nd ACM Nat. Conf.*, 1967, pp. 387–393.
- [18] N. J. A. Sloane, R. H. Hardin, and W. D. Smith, "Tables of spherical codes." [Online]. Available: <http://www.research.att.com/~njas/packings/>