The Visual Computer

Automated generation of control skeletons for use in animation

Lawson Wade[1],
Richard E. Parent[2]

[1] ACCAD, The Ohio State University, 1224 Kinnear Road, Columbus, Ohio 43212, USA
E-mail: lwade@cgrg.ohio-state.edu
[2] Computer and Information Science, The Ohio State University, 395 Dreese Lab, 2015 Neil Avenue, Columbus, Ohio 43210, USA
E-mail: parent@cis.ohio-state.edu

Published online: 15 March 2002

This paper describes an algorithm for automatically generating a control skeleton (sometimes called an IK skeleton) for use in animating a polygonal data model. The algorithm consists of several steps, each of which is performed automatically. The basic process involves discretizing the figure, computing its discrete medial surface (DMS), and then using the DMS both to create the skeletal structure and to attach the vertices of the model to that structure. The system can produce a reasonably good control skeleton for any of a variety of figures in as little as 1 or 2 min on a low-end PC.

Key words: Control skeleton – IK skeleton – Skeleton rig – Articulated figure

Correspondence to: L. Wade

1 Introduction

Articulated figures abound in computer graphics. Posing these figures for modeling or animation is accomplished using a control skeleton. The control skeleton consists of a hierarchy of segments and joints along with information for anchoring the surface geometry (or "skin") to that hierarchy so that the geometry can be adjusted when the skeleton is repositioned.

Many modeling and animation packages provide support for users to create a control skeleton for a model. Nevertheless, the creation of the skeleton can be a laborious process requiring several hours of work, and a user typically must possess a fair degree of proficiency with a package to obtain even rudimentary motion via a control skeleton.

In this paper, we present a method for fully automatic generation of a control skeleton, summarizing the relevant work from the first author's dissertation (Wade 2000). After prompting the user for a small number of input parameters, our algorithm converts a polygonal data model into a voxelized representation and approximates the Euclidean distance map (EDM) and the discrete medial surface (DMS)[1] for the voxelized model. Next, a tree-structured voxel path is constructed from the voxels of the DMS, and that tree is then divided into a structure of segments and connecting joints. Finally, the voxel representation and the DMS are used to generate anchors for attaching the vertices of the polygonal data to the segments. Although the method is not without its shortcomings, it is quite fast, and it produces control skeletons of rather good quality.

2 Related work

Perhaps the earliest work on automatic skeleton generation is that of Tsao and Fu (1984). For an object given as a 2D bitmap, a distance map is computed. The DMA of the object is extracted from the map and converted into a graph. Randomly applied node modification routines permit minor bending and repositioning of the graph. Using distance map values at

[1] The medial axis (MA) of a 2D object is defined as the locus of the centers of all disks interior to the object that touch the boundary of the object at two or more points. For a 3D object, spheres are used instead of disks, and the locus yields the medial surface (MS). For discretized figures, the MA/MS is approximated as a subset of the pixels/voxels, and the approximation is referred to as the discrete medial axis/surface (DMA/DMS). The MA/MS is often called the geometric skeleton of an object.

The Visual Computer (2002) 18:97–110
Digital Object Identifier (DOI) 10.1007/s003710100139

each node, an inverse distance transformation (IDT) allows the construction of a new pixelized object, so the graph functions as a control skeleton. A simple 3D example of the procedure is also presented. Because the method operates entirely in the discrete domain and uses a simple IDT, distinct boundary features of the objects (e.g. ridges and valleys) usually disappear in the stochastically generated instances. In our method, the object and the control skeleton exist in the continuous domain, which allows more sophisticated joints, more appropriate segments, and better preservation of boundary features.

A somewhat similar method to Tsao and Fu's is one by Gagvani, Kenchammana-Hosekote, and Silver (Gagvani et al. 1998). It also operates in the discrete domain and has steps to compute a distance map and DMS for a figure. DMS voxels are automatically connected into a graph whose minimum spanning tree is then provided to the user as the control skeleton. An IDT is used to generate posed instances. Gagvani and Silver have implemented the method as a plug-in for Maya^{TM} (Gagvani and Silver 1999). This plug-in can convert the graph into a skeleton in Maya's internal format; however, the resulting skeleton is usually much too complex. Instead, Gagvani and Silver suggest that a user view the DMS while manually constructing a skeleton whose segments run along stretches of DMS voxels. Animation of the Maya skeleton can be exported to drive the animation of the voxelized figure. In contrast, our method is fully automated and generates a control skeleton with a reasonable number of segments and joints in fairly appropriate positions.

Teichmann and Teller (1998) have presented a method for assisting in the generation of control skeletons. Given a closed polyhedral model, their algorithm first computes a Voronoi diagram for a sufficiently dense set of sample points on the surface of the polyhedron. The user then selects Voronoi vertices as endpoints of branches of the control skeleton, and the Voronoi graph is simplified to produce a tree that extends to those vertices. Next, the user selects spanning tree nodes to be joints of the skeleton. Portions of the tree lying between joints and/or endpoints become skeletal segments. A sophisticated network of springs is created to attach polyhedral vertices to the skeleton. Besides using a voxelization approach, our algorithm differs from that of Teichmann and Teller in several ways: it places fewer restrictions on the polygonal data, achieves a higher degree of automation, and is much faster.

In a method proposed by Bloomenthal and Lim (1999), a control skeleton for an object is automatically produced from the MS of an object. The MS is computed using an implicit method and is stored as a polygonal mesh which contains distance-to-surface information. After the skeleton is posed, the MS mesh is updated accordingly; then, an implicitly defined IDT is applied to reconstruct the surface. Apparently a commercial version of their algorithm is planned. A comparison with our algorithm is difficult, because few details are provided by Bloomenthal and Lim with respect to the construction of the control skeleton, the quality of the results, or the execution time. Their method appears to restrict the input object to be a single, closed surface. Also, implicit methods typically require more computation time than non-implicit methods.

Note that many of the steps of the algorithm described in this paper are nearly identical to those in an earlier report on this research (Wade and Parent 2000). The main differences between the two are that the one described here uses the DMS of the voxelized object, that it includes a step to smooth the path tree before creating the skeleton structure, and that it provides the user with slightly more control over the process through the use of a few more input parameters. The implementation here is significantly faster, and the results are slightly better than the results from the previous method.

3 The algorithm

The primary goal of the algorithm is the automatic construction of a control skeleton suitable for animating a given model. In order to have the algorithm perform well in the general case, some assumptions have been made. Chief among these is the belief that the skeleton produced by the algorithm should exhibit a correspondence to the model in both shape and apparent flexibility. To this end, the skeleton produced should be centrally located within the model, and it should have branches that logically correspond to various protrusions. The positions and lengths of its segments should relate to geometric aspects of the surface and volume, and the positions and axes of its joints should seem appropriate both locally and globally. Moreover, the model and skeleton should be attached in a simple but meaningful fashion. Finally, there should be enough of a skeleton to provide some desired level of control, yet there should not

be so much of a skeleton that its manipulation would seem unwieldy.

Closely associated with the primary goal is the aim of requiring very little user input. Besides the polygonal data, the user can specify seven input parameters, though generally the algorithm performs fairly well using the default values of the parameters. The most influential parameters are the voxel-size parameter, the exposure threshold, and the closeness-of-fit parameter. The voxel-size parameter is simply the desired edge length of a voxel; the exposure threshold helps control the results of the DMS calculation; and the closeness-of-fit parameter helps control the extent of skeletal branches. The various parameters will be discussed in more detail as they arise in the presentation of the algorithm.

The geometric input to the algorithm is currently restricted to sets of polygonal data. The polygons are not required to form a single, closed surface, or really even to be connected at all. What is required is that after voxelization of the polygonal data and classification of each voxel as being either interior or exterior to the object, the interior voxels form a single, connected set. A closed polyhedron works quite well as input, but a figure consisting of overlapping closed polyhedra works equally well. The voxelization and classification process is often rather forgiving of aberrant polygons or of polygonal surfaces that are not closed.

The remainder of this section describes the various steps of the algorithm. Section 3.1 discusses the manner in which the given model is discretized for the computation of the distance map, and Sect. 3.2 tells how the DMS is computed for the discretized model. Section 3.3 describes how the medial surface approximation is used to generate a tree-like structure of voxel paths, which, as detailed in Sect. 3.4, is used to generate the segments and joints of the control skeleton. Also in that section, the method of attaching the original polygonal model to the control skeleton is presented.

3.1 Voxelization and distance map construction

For uniformity, the first step of the algorithm is to transform the model so that its bounding box lies just inside the unit cube. A voxelization of the bounding box is then performed, with the resolution being determined by the voxel-size parameter. After intersecting the polygons with the grid, a filling routine

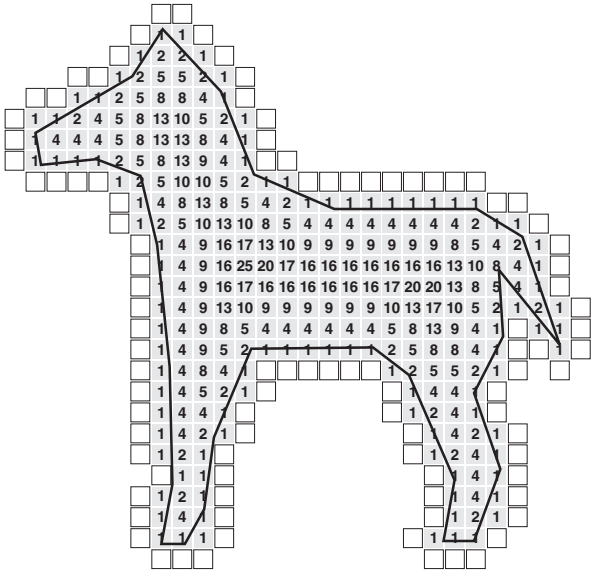


Fig. 1. The 2D Euclidean distance map (EDM) for a discretized, animal-shaped polygon. Each cell's value is the square of the Euclidean distance to the nearest exterior cell

is used to complete the labeling of each voxel as either interior or exterior to the figure. For simplicity, we require that the interior voxels form a single 26-connected group.

The objective is to have a sufficient number of interior voxels. Experiments have shown a voxel size of 0.005, 0.01, or 0.02 units to work fairly well, depending on the manner in which the transformed object fills the unit cube. Anywhere between 20 000 and 200 000 interior voxels is usually sufficient to produce a reasonable control skeleton.

The next step is the generation of an EDM for the interior voxels. Figure 1 shows an example of a 2D EDM; the extension to three dimensions should be clear.[2] Exact, efficient computation of the EDM is not easily implemented, so many people instead use approximations (Danielsson 1980). Nevertheless, linear time algorithms for computing the exact EDM in two dimensions do exist (Breu et al. 1995), and algorithms for linear time computation of 3D maps are being investigated. In our implementation, we use a propagation technique that computes a very close approximation to the EDM and operates in linear time with respect to the number of interior voxels.

[2] The design for Fig. 1 was borrowed from one of the many excellent diagrams appearing in a paper by Ogniewicz and Kübler (1995).

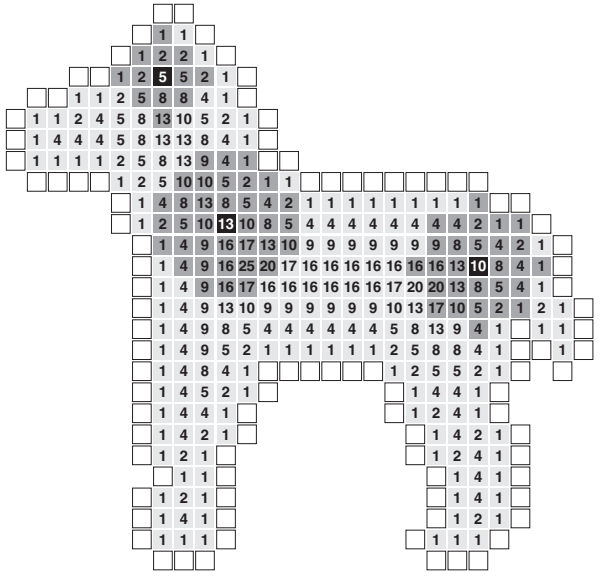


Fig. 2. Examples of the coverage of three disks corresponding to the cells shown in *black*. The radius of each disk is equal to the square root of the respective EDM value. The *dark gray* cells (and the *black* cells) are contained in the disks and are thus considered to be "covered" by the *black* cells

A concept related to the distance map, and one that will come into play during the formation of the path tree, is that of "coverage", an application of the IDT. Each value in the 3D EDM defines a sphere, centered at the corresponding voxel, that just touches the boundary of the object. The radius of the sphere for a voxel, v_i, is $\sqrt{d_i}$, where d_i is the (squared) distance map value for v_i. Any voxel whose center is contained within the sphere is said to be "covered" by v_i. Figure 2 illustrates coverage as it would apply within a 2D EDM.

3.2 Medial surface extraction

After the EDM has been computed, it is fed into a routine for computing the DMS of the object. Unlike the continuous MA/MS, there is no precise mathematical definition for the DMA/DMS, though several desirable properties for the computation have been identified (Ge and Fitzpatrick 1996; Staunton 1996). Methods for the procedure consist of two main approaches: thinning algorithms (Staunton 1996; Lee et al. 1994) and extraction algorithms (Danielsson 1980; Ge and Fitzpatrick 1996). Thinning algorithms work by iteratively removing selected pixels/voxels from a discretized object in an attempt to whittle the object down in topological fashion to a more simple representation. Extraction algorithms involve first computing the EDM for the figure and then constructing the DMA/DMS by attempting to identify the ridges implied by the values within the map.[3] For the 3D case, proper extraction is quite difficult due to the complexities of defining and identifying saddle points along the ridges.

Our implementation follows the extraction model and produces a connected DMS quite suitable for our purposes here. It accepts one input parameter, the exposure threshold, which influences roughly how thick the DMS appears as well as to what degree it extends into each individual surface protrusion of the discretized object. The DMS shown in Fig. 3a is a fairly typical example of the results.

3.3 Path tree generation

After the DMS voxels have been identified, a path tree is generated that effectively simplifies the DMS to a tree structure of 1D pathways (referred to as *chains*). The path tree is developed so as to maintain a tree structure regardless of the genus of the DMS or the object. The formation of the path tree begins by automatically identifying a centrally located voxel referred to as the *heart*. A breadth-first search of the DMS is performed, beginning at the heart in order to identify extreme points in the DMS, which are basically points of local maximum in the search. These extreme points are potential end-effectors of the control skeleton. The length of the path between the heart voxel and the farthest DMS voxel is saved for future use; this length is called the *heart radius*. Figure 3a shows the heart voxel and extreme points for a DMS of a horse.

The process of growing the path tree then begins, and each new branch of the path tree is created to extend to a previously unreached extreme point. When a new branch is added to the path tree, any DMS voxels that lie within any of the corresponding spheres of the new branch are marked as being covered by the path tree. This coverage is used to help weed out insignificant extreme points resulting from spurious extensions of the DMS. When no

[3] The 2D EDM can be interpreted as a height field and viewed as a 3D landscape; the ridges of the landscape represent branches of the DMA. Thus, extracting the DMA (or DMS) from a 2D (or 3D) EDM amounts to finding and following the ridges implied within the map; the main difficulty comes in handling saddle points along the ridges.

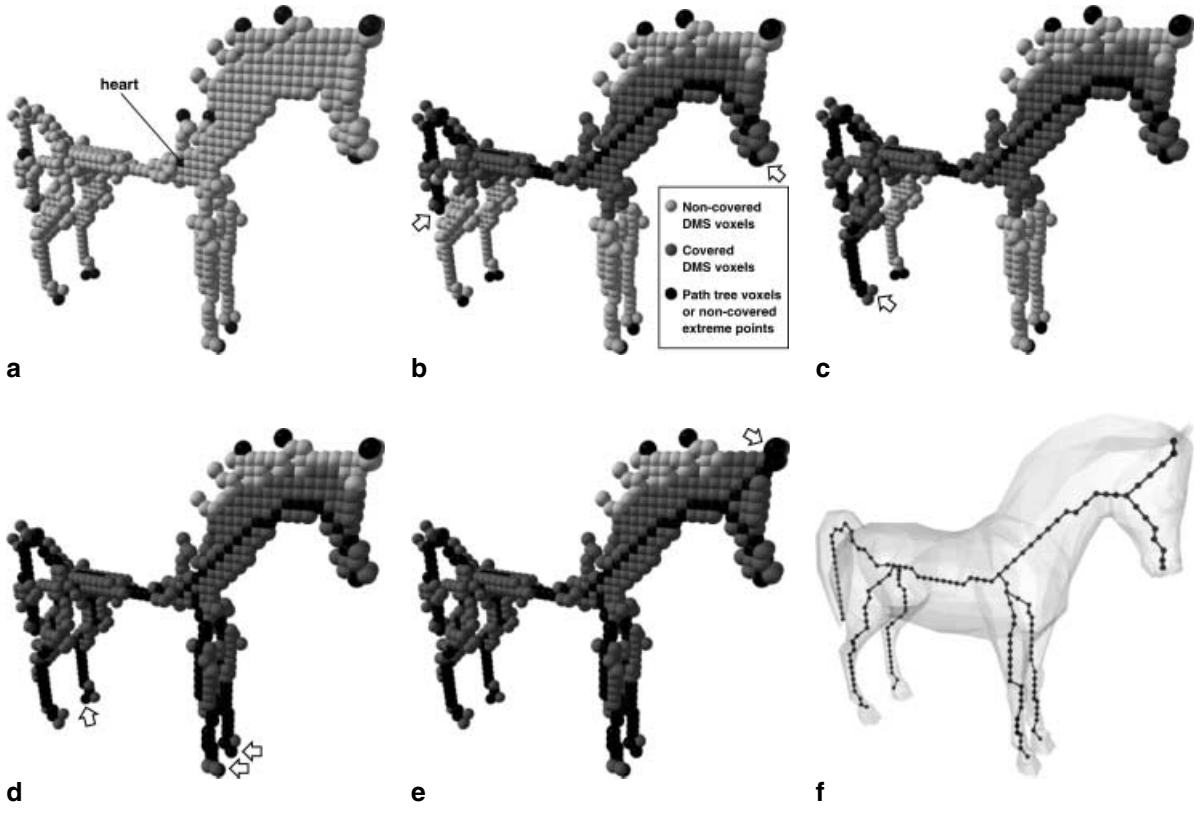


Fig. 3a–f. Forming path-tree extension. **a** DMS of a horse with the heart voxel labeled and the extreme points drawn as *black spheres*. The extreme point at the tip of the tail is the first point of the path tree. **b** The first extension to the path tree reaches from the tail to the nose. **c** The second extension reaches to the right hind hoof. **d** The next three extensions branch out to the other hooves. **e** The final branch extends to the ears. The remaining extreme points are not far enough from the covered region to warrant further extensions of the path tree. **f** The completed path tree without the other DMS voxels; path-tree voxels are drawn as *small spheres* to allow the edges of the path tree to be seen

more path-tree branches can be added that are at least a certain length, path-tree growth stops. Path tree growth is illustrated in Fig. 3. Chains of the path tree are then identified, and the chain vertices are filtered to help smooth the otherwise jagged pathways resulting from stepwise movement between consecutive voxels along the chain. The following subsections describe these processes in more detail.

3.3.1 Forming path-tree extensions

During the formation of the path tree, the algorithm examines connected paths of DMS voxels. Two measures of voxel paths are used during this process: the path length and a special weighted measure. The path length is simply the sum of the distances between the centers of consecutive voxels along the path; the distance between the centers of two adjacent voxels is l, $l\sqrt{2}$, or $l\sqrt{3}$, where l is the edge length of a voxel. The weighted measure, W_P, of a voxel path, P, is based on the EDM:

$$W_P = \sum_{v_i \in P} \frac{1}{d_i^3} ,$$

where d_i is the squared value for voxel v_i as stored in the distance map (the use of 3 as the exponent was arrived at empirically).

The purpose of the weighted measure is to provide a means for favoring centralized paths through the figure that follow along the deepest portions of the DMS. Using a modified version of Dijkstra's short-

est paths algorithm [for the standard version, see Cormen et al. (1990)], the algorithm can find the voxel path through the DMS connecting any given pair of voxels and minimizing the weighted measure of all such connecting paths. Although minimizing the weighted measure does not guarantee that the path will follow along the deepest region of the DMS, experimental results have shown that it appears to do so.

The extreme point farthest from the heart voxel is the starting point for the first branch of the path tree. The path tree is then grown by creating and appending extensions to it until further extensions to the path tree will unnecessarily complicate the structure. Each extension to the path tree is formed by executing the following steps:

1. Mark (or update) the DMS voxels covered by the path tree.
2. Find the extreme point DMS voxel, v_f, farthest from the covered region (note that the group of extreme points containing v_f must not already have had a branch of the path tree extended to one of its members, and also note that any covered extreme points are simply ignored).
3. Find the minimum weight path of DMS voxels connecting v_f to the path tree.
4. Append that minimum weight voxel path to the path tree.

In step 1, note that the coverage of the DMS does not need to be recomputed each time a new branch is added to the path tree; instead, the coverage can simply be updated in the area surrounding the new extension.

In step 2, the algorithm searches for the non-covered v_f that is farthest from the set of covered DMS voxels. If the shortest path length from v_f to a covered DMS voxel is greater than or equal to a certain threshold, then the algorithm proceeds with steps 3 and 4 to extend the path tree to v_f and then repeats the process beginning with step 1. If the shortest path length from v_f is less than the threshold, then steps 3 and 4 are skipped, and no more branches are added to the path tree.

The threshold used in this process is the product of the closeness-of-fit parameter and the heart radius. Observation has shown that a closeness-of-fit value between 0.05 and 0.1 works fairly well for producing a good, simple control skeleton – this means that a new branch will be added if it extends at least $\frac{1}{20}$ to $\frac{1}{10}$ of the length of the heart radius beyond the current coverage of the path tree. Using finer values will usually allow the extension of the path tree into smaller protrusions of the object, such as the fingers of a hand; however, it can also result in the formation of other seemingly spurious branches.

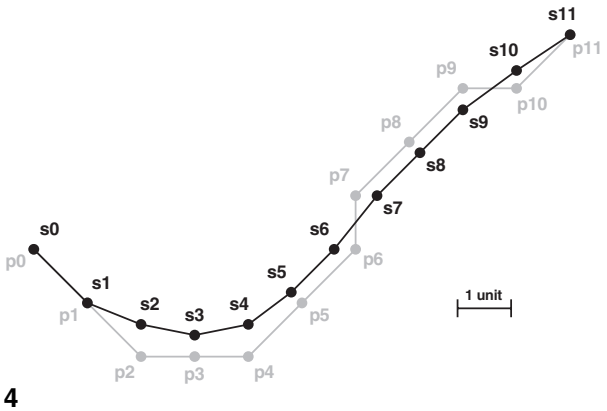
Step 3 makes use of the modified version of Dijkstra's shortest paths algorithm mentioned previously. Each path-tree voxel is assigned a weight of zero and becomes a source point for the shortest paths. The weighted measure is applied as the shortest path search spreads through the DMS. When the search reaches v_f, it is a simple matter to backtrack to find the actual minimum weight path from v_f to the path tree. This minimum weight path is added to the path tree. Its coverage of the DMS voxels is then computed as the process of extending the path tree is repeated from step 1.

3.3.2 Smoothing the path tree

The path tree for the horse has been redrawn as a collection of vertices and edges in Fig. 3f. The vertices can be sorted into three classes. Endpoint vertices have only one adjacent edge – these correspond to the extreme points used. Junction vertices have three or more adjacent edges – this is where the path tree forks. The remaining vertices, termed intermediate vertices, have exactly two adjacent edges. The endpoint and junction vertices split the path tree into a set of connected path segments, or chains.

Due to the regularity of the voxelization, the chains of the path tree can be fairly jagged. The jaggedness may be especially noticeable in parts of the figure where the main direction of a chain section does not align reasonably well with any of the axes of the voxelization. To lessen any peculiar effects the orientation of the voxelization can have on the path tree, and also to diminish the influence of the jaggedness on the later creation of segments and joints, the path tree is subjected to a smoothing operation.

As each chain of the path tree is identified, it is smoothed by applying a filtering process to average positions of consecutive voxels along the chain. A filter radius of three edges usually works well to smooth out any jaggedness of the original chain, although for ease of illustration, a filter radius of two edges is used in Fig. 4. Note that the filter radius is shortened at the ends of the chain. Figure 5 shows the result of smoothing the path tree for the horse.



4

5

Fig. 4. Smoothing of a path-tree chain. To obtain the smoothed chain **s0** . . . **s11**, the chain **p0** . . . **p11** is filtered using a simple box filter at each vertex; for example, s5 = (p3 + p4 + p5 + p6 + p7)/5

Fig. 5. The smoothed path tree for the horse; the result of applying the smoothing operation to the chains of the path tree from Fig. 3f

3.4 Control skeleton construction

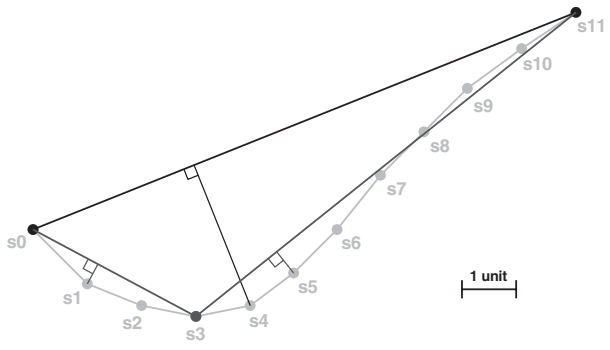
The path tree itself is usually too complicated to use directly as the structure of the control skeleton; instead, an approximation to the path tree is formed in what is called the skeletal graph (which is really a tree). The skeletal graph is the precursor to the final control skeleton structure. It is created so as to approximate the path tree using appropriately sized edges, each of which will become a segment of the control skeleton. The discussion that follows explains how the skeletal graph is constructed, how it is used in to create the segments and joints of the control skeleton, and how vertices of the polygonal model are attached to the skeleton structure.

3.4.1 Creating the skeletal graph

The initialization of the skeletal graph results from a simple conversion of path-tree chains. The endpoint and junction voxels of the path tree are used to create the initial vertices of the skeletal graph. Each chain of the path tree is used to create an initial edge of the skeletal graph.

After the initial edges and vertices of the skeletal graph are formed, tests are performed to determine which edges should be split. Splitting of a skeletal edge is accomplished by inserting an intermediate vertex into the skeletal graph (at the location of a specially selected chain vertex from the corresponding path-tree chain) and replacing the edge with two new edges. Each new edge then corresponds to a subsection of the original chain. The edges for any particular chain may be split repeatedly in order to form closer approximations to the chain or in order to have more appropriate lengths.

Two input parameters are used to specify a range of desired edge lengths (the specified range is actually applied not to the skeletal edges but to their corresponding section of a path-tree chain). The parameters, called min-fraction and max-fraction, are entered as values between zero and one (default values are 0.1 and 0.3, respectively). The lower limit of the range is the product of min-fraction and the heart radius; the upper limit of the range is the product of max-fraction and the heart radius. Any skeletal edges whose chains are already shorter that the lower limit



6

7

Fig. 6. Error and splitting of a skeletal graph edge. The error for the skeletal edge **s0–s11** is the length of the perpendicular segment to **s4**, 2.79 units. The best split for the edge comes from inserting a skeletal vertex at **s3**, as this results in the smallest maximum error (0.50 units) for the two replacement edges (the error for **s0–s3** is 0.41 units, and for **s3–s11** it is 0.50 units)

Fig. 7. The skeletal graph for the horse. Each edge of the graph is used to create a segment for the control skeleton. Joints are created at interior vertices of the graph, but more than one joint may be created at a vertex depending on how many edges are incident to that vertex

will not be split. Any skeletal edges whose chains are longer than the upper limit will definitely be split. Edges whose chain lengths are within the range may be split based upon how closely they approximate the corresponding chain section.

Skeletal edges are assigned error values according to how closely they approximate the corresponding chain section of the smoothed path tree. This error is simply the maximum distance between the skeletal edge and one of the vertices of its related chain section. Figure 6 provides an illustration of the error computation and the splitting of a skeletal edge.

The splitting of skeletal edges is performed incrementally; at each step, the entire skeletal graph is compared to the entire path tree to determine which edge should next be considered for possible splitting. In this way, the skeletal graph gradually becomes more complex while providing an acceptable approximation to the path tree at any stage of the splitting process. The reason for this global approach is to provide the best approximation given the constraint imposed by the number-of-segments parameter (each skeletal edge corresponds to one segment of the control skeleton). Another input parameter controls the error tolerance allowed for the approximation.

The processing is accomplished in an efficient manner through the dynamic use of a heap whose node weights are the error values of the skeletal edges – the node at the root represents the next edge to be considered for possible splitting. If the number-of-segments parameter has been set, then the splitting process is repeated until the skeletal graph contains an equivalent number of edges (or until the heap is empty and there are no more edges to be split). If the number-of-segments parameter is left unspecified by the user, then splitting stops when the heap is empty. Figure 7 shows the skeletal graph computed by allowing the heap to empty.

3.4.2 Creating segments and joints

Creating the segments of the control skeleton is simple. Each edge of the skeletal graph essentially becomes a segment of the control skeleton. A deep seg-

ment is then selected to host the root joint for the control skeleton, or rather to be the only segment connected to the root joint. The root joint itself is positioned at the midpoint of the skeletal edge for that segment (note that it does not divide the segment).
The location of the root joint imposes proximity relationships on the skeletal graph edges and thus on the control segments. For each proximal-distal pair of adjacent segments, a joint is created at the shared joint-voxel (note that this results in coincident joints at the branching points of the tree structure). Each joint other than the root joint thus has one proximal and one distal segment; the root joint has only a distal segment. Vertices of the skeletal graph that are not used for joint creation become end-effector points of the control skeleton.
Each joint has three rotational degrees of freedom defined about an orthogonal set of vectors: the z-axis points outward along the distal segment, the x-axis is formed to be perpendicular to the plane defined by the proximal and distal segments, and the y-axis is then formed to complete a right-handed coordinate system. Degeneracies in this method (e.g. parallel vectors) are handled by searching proximally or distally to find other suitable base vectors.

3.4.3 Anchoring skin vertices

Once the segments and joints have been assembled, each vertex of the polygonal data is anchored to one or more segments. Each control segment corresponds to a section of a chain of the path tree, and each voxel along that section of the path tree has a sphere which covers voxels of the figure (see Sect. 3.1 for an explanation of coverage). The collective coverage for a particular section of the path tree essentially defines a volume within which the corresponding control segment exerts influence. In the actual implementation, each voxel gathers and maintains a set of pointers to those control segments that cover (or influence) it.
Not all voxels interior to the figure are necessarily covered by spheres of path-tree voxels. A voxel that is not covered by some path-tree voxel will not at first have an influencing set of control segments; instead, such a set must be created. The sets for such voxels are constructed by propagating sets from covered voxels into non-covered regions of the voxelization. This propagation is performed in a breadth-first manner moving away from the covered region.
For each vertex, an anchoring equation is created. The voxel that contains a specific vertex provides the list of control segments that influence that vertex. The coordinates of that vertex can then be expressed within the local coordinate frame for each influencing segment. When the control skeleton is moved, the (fixed) local definitions of that vertex are converted into global positions and used in a weighted sum formula that computes a new global position for the vertex:

$$p_v = \sum_{s_i \in S} w_i \times (\text{origin}_i + p_i \times \text{RtoWorld}_i) \; .$$

Here, S is the set of control segments that influence a specific vertex v, p_v is the global position of v, and p_i is the local position of v in the frame of segment s_i. The origin of the local frame of s_i is given by origin_i, and RtoWorld_i is the 3×3 rotation matrix used to help transform a point from the local frame of s_i to the global frame of the figure. The amount of influence s_i exerts on v is represented in the weight w_i. If only one segment influences v, w_i is set to one; otherwise, each weight is computed so that closer segments will have larger weights and so that the weights will sum to one:

$$w_i = \frac{\text{totaldist} - \text{dist}_i}{(n-1) \times \text{totaldist}} \; ,$$

where n is the number of segments in S, dist_i is the shortest distance between v and s_i, and $\text{totaldist} = \sum_{s_i \in S} \text{dist}_i$. Note that w_i, p_i, dist_i, and totaldist are constants, computed only once during the set-up. Values that are updated each time the control skeleton is repositioned include origin_i, RtoWorld_i, and, of course, p_v.

4 Results

In general, the algorithm is quite effective in producing a useful control skeleton in a short period of time. The quality of individual results is highly dependent on the object and the input parameters. In some cases, the algorithm performs extremely well, but in some other cases, the algorithm does only a mediocre job. For most objects that an animator might wish to animate by using a control skeleton, the skeleton produced by the algorithm is at least a reasonable start worthwhile for finer hand-editing.
Table 1 shows the results of several executions of the algorithm on various polygonal models. The models themselves are shown in both the default stance (as

Table 1. Statistics from executions of the algorithm on various models. Each row represents a single execution in which the voxel-size parameter was specified as in the first column, the number of control segments was determined automatically, and all other input parameters used default values. Rows in **bold** correspond to the set-ups used for Figs. 7, 10, and 11. The rightmost column shows the execution time required on a Silicon Graphics™ O2™ (R5000 Processor), though tests on a PC with a 133 MHz Intel™ Pentium™ processor running under Linux™ had similar timings (only about 5% longer)

Voxel size	Number of segments	Grid dimensions	Total voxels	Interior voxels	Time (min:s)
		Horse (681 vertices, 1354 polygons)			
0.02	**32**	**51 × 42 × 15**	**32 130**	**7 324**	**0:02**
0.01	36	101 × 84 × 29	246 036	48 626	0:11
0.005	31	201 × 168 × 58	1 958 544	352 971	1:29
		Human (349 vertices, 694 polygons)			
0.01	29	37 × 101 × 17	63 529	14 044	0:04
0.005	**27**	**73 × 201 × 33**	**484 209**	**92 934**	**0:24**
0.0025	31	145 × 401 × 65	3 779 425	675 120	2:58
		Octopus (2347 vertices, 4690 polygons)			
0.01	52	101 × 59 × 78	464 802	42 772	0:11
0.008	**57**	**126 × 74 × 97**	**904 428**	**79 926**	**0:21**
0.00675	51	149 × 88 × 115	1 507 880	129 474	0:34
		Jellyfish (2526 vertices, 5048 polygons)			
0.008	**102**	**119 × 126 × 110**	**1 649 340**	**157 150**	**1:04**
0.007	103	136 × 143 × 125	2 431 000	227 784	1:36

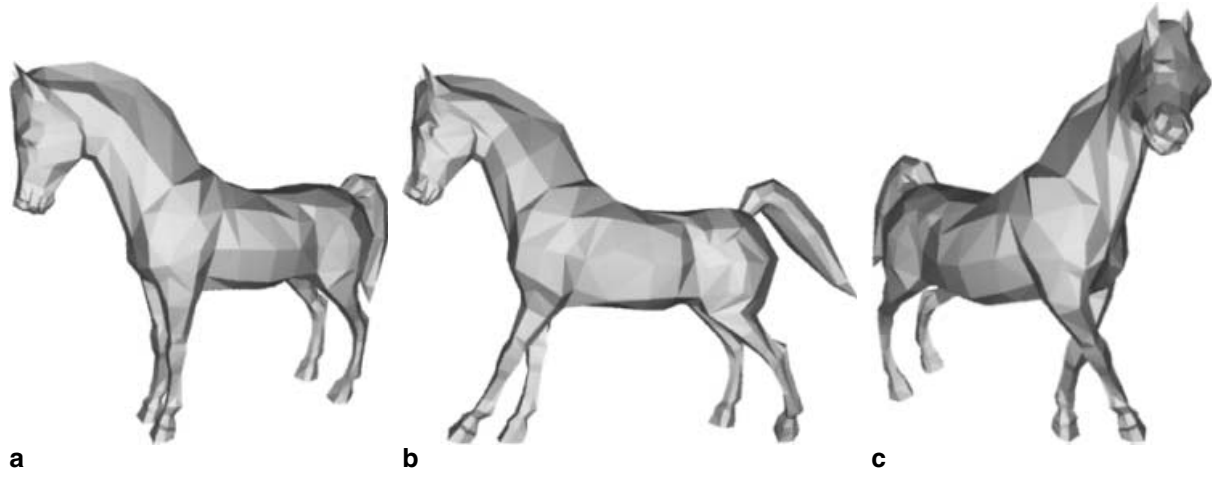


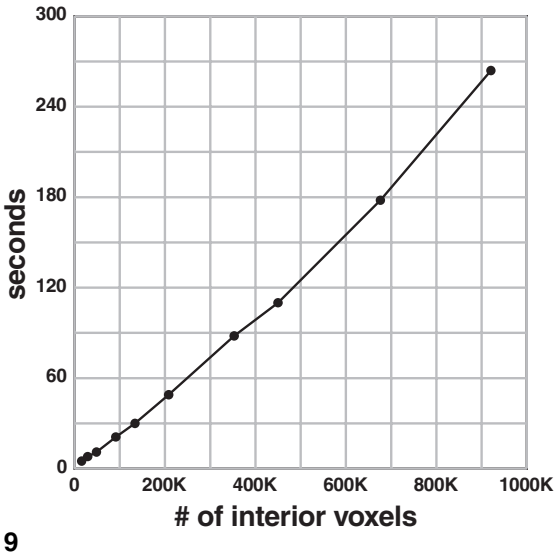
Fig. 8. The horse and two random poses. In **a**, the horse is shown in its default pose (as input for the algorithm). The other images are selected random poses of the horse generated using the skeleton resulting from Fig. 7

input to the algorithm) and in selected random poses using the generated control skeletons (see Figs. 8, 10, and 11). Since each model has been scaled to fit inside the unit cube, a grid size of 0.01 will allow approximately 100 voxels along the edge of the unit cube.

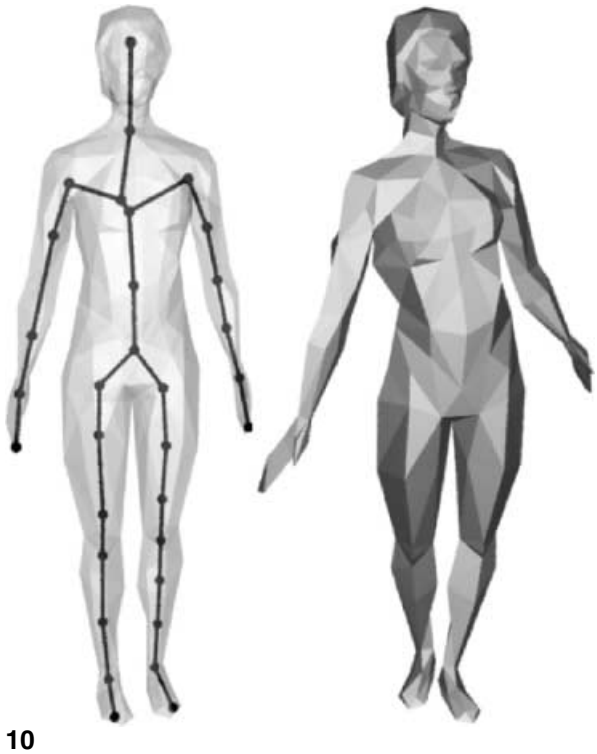
The graph in Fig. 9 illustrates the unproven but apparently superlinear time complexity of the algorithm (superlinear with respect to the number of interior voxels). For the various executions, all input parameters used default values except for voxel-size. The data in the figure can be approximated fairly well by the function

$$f(x) = \frac{x^{1.2}}{54\,000},$$

where x is the number of interior voxels and $f(x)$ is the number of seconds required to create the control skeleton. Although the analysis is derived using the horse, graphs created for other objects were very similar.



9



10

Fig. 9. An analysis of execution times for the algorithm. Each *dot* represents one execution of the algorithm on the horse model. The time complexity appears to be superlinear with respect to the number of interior voxels (see Sect. 4 for an approximating function)

Fig. 10. a,b Control skeleton for a human figure and one pose. **a** Skeleton generated using a voxel-size parameter of 0.005. **b** Figure from selected random pose of the skeleton

With respect to the goals described at the beginning of Sect. 3, the algorithm performs reasonably well. As can be seen in the figures, the control skeletons are centralized, and they have branches that reach to the ends of the major protrusions of the objects. The segments and joints relate fairly well to the surface features, though there is still room for improvement. As for the idea of having "just enough but not too much" of a control skeleton, results are rather highly dependent on the objects given as input. For the most part, skeletons produced by specifying the voxel size and using default values for the other input parameters are reasonably succinct. With a closeness-of-fit value below 0.1, the algorithm can extend the skeleton into shorter surface protrusions such as the fingers of a hand; when doing so, however, it also usually produces at least a few spurious branches in other parts of the figure.

One area where the algorithm can have noticeable difficulty is with multi-junction points, such as where two "arm" sections of the path tree might joint a "spine" section of the path tree (often the arm sections join the spine section at different points). Increasing the exposure threshold so that the DMS is rather lean often collapses the area involved in the multi-junction point and sometimes results in better joining of path-tree extensions in the area of the junction.

As can be expected, the quality of the skeleton is dependent on the quality of the voxelization of the object. The use of finer grids allows better approximations of the surface details of the object, but not without a cost – simply halving the voxel-size pa-

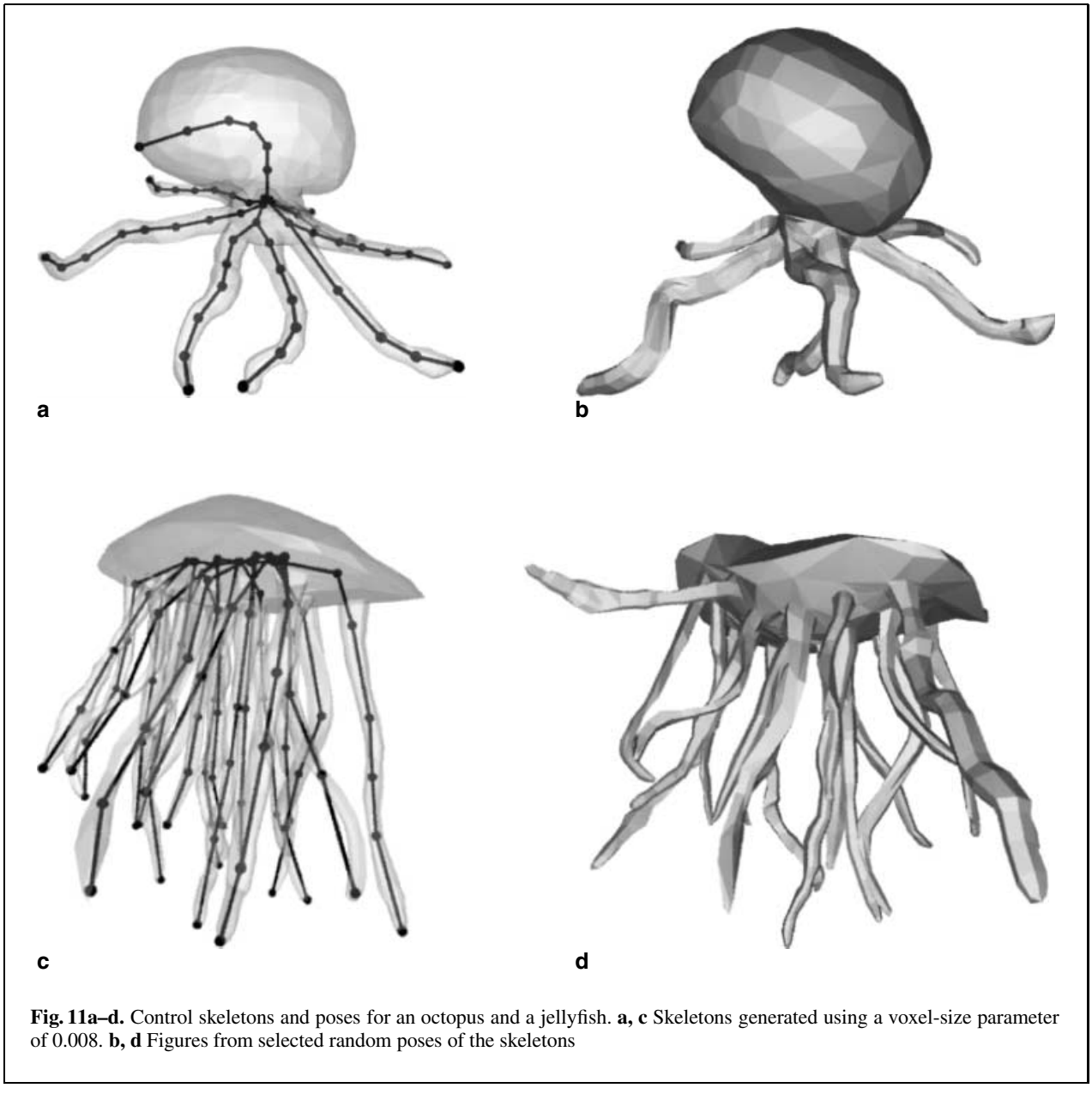


Fig. 11a–d. Control skeletons and poses for an octopus and a jellyfish. **a, c** Skeletons generated using a voxel-size parameter of 0.008. **b, d** Figures from selected random poses of the skeletons

rameter will produce roughly eight times the number of interior voxels and result in a related increase in the running time. As long as the topology of the grid is not compromised, a coarse grid can still produce reasonable results; the main benefits of a finer grid are better centralization of the control skeleton and better determination of surface protrusions (the latter allows better application of the closeness-of-fit parameter).

Several shortcomings of the algorithm have been identified:

- Because the algorithm produces a tree-structured control skeleton, it does not work very well for objects with holes; for example, when given an object such as a doughnut, it will create a C-shaped skeleton. With some additional programming, the algorithm might be extended to

produce a kinematic constraint that effectively closes the "C" during animation.

- The step-wise greedy approach to splitting the control segments in order to produce a desired number of them is probably not the optimal method, especially since it only considers bifurcations of the segments.
- Long, straight sections of the path tree are sometimes segmented in a seemingly arbitrary fashion. This can be the case when a figure's arm is posed without a bend at the elbow – sometimes no elbow joint is generated, at other times, numerous joints are generated along the straightaway (observe the arms and legs of the human in Fig. 10, for instance). In contrast, when the input figure has bent limbs, the algorithm does very well at producing joints at the expected locations.
- For many objects, a tree being one example, it is probably not desirable to have the root joint centrally located with respect to the articulation points of the control skeleton. An input flag could be provided to request that the root joint be placed at the lowest end-voxel of the path tree, or better yet, a user could simply select the root once the control segments have been generated.
- The surface attachment scheme is rather simplistic, which is an advantage in terms of easy understanding and implementation, and it is quite effective under moderate repositioning of the control skeleton. However, the repositioned surface can sometimes suffer from interpenetration problems, especially when joints are bent beyond about 20 or 30°. In the vicinity of the joints, sufficient numbers of vertices are necessary to minimize the penetrations, and the algorithm could be extended to produce extra vertices near the joints; regardless, a better and probably more complicated attachment scheme is necessary to avoid the penetration problem.
- Often the algorithm produces a control skeleton that overall is quite good but that could use some tweaking. Since the focus of this research has been the automation of the control skeleton construction, there is currently no interface for tweaking it; nevertheless, such an interface would definitely be useful and indeed would be required for widespread use of the algorithm. A better idea would be to convert the implementation into a plug-in for a software package designed for modeling and animation and to allow tweaking of an automatically generated skeleton via the skeleton-control interface of that package.

5 Conclusion

In this paper we have described an approach to automating the process of generating control skeletons. The method presented achieves a higher degree of automation than previous approaches; furthermore, the algorithm is very fast, quite general, and fairly robust. With very little user input, the algorithm produces control skeletons of relatively good quality, sometimes good enough for immediate use in animation. At the very least, the algorithm is generally useful for providing an initial skeleton that an animator could hand-tune. It is especially useful for producing skeletons for more complex objects like trees or jellyfish, where creating a skeleton by hand would be a tedious and time-consuming process.

The algorithm is intended as a general solution to the problem of automatic generation of control skeletons. It must be emphasized that the algorithm constructs a control skeleton based solely upon a geometric analysis of the object. The algorithm has no knowledge of what kind of an object it is dealing with, nor of any semantic relationships between the parts of an object. Of course, this does not prevent a user from having definite ideas about what kind of skeleton should be produced based on what type of object was provided as input. Nonetheless, even in the face of possibly unrealistic assumptions on the part of the user, the algorithm can often produce acceptable results.

Other research by the author has investigated the automated generation of anatomically appropriate control skeletons for human-like and animal-like models (Wade 2000). Drawing on basic anatomical knowledge and patterns, a simple semantic analysis of the model's shape is conducted, and various assumptions and heuristics are employed with the goal of generating a control skeleton that mimics the expected anatomical flexibility of the model. The system is also capable of producing individual bone models tied to the control skeleton, providing a possible foundation for further anatomically based modeling of the figure.

Future research into control skeleton generation might involve adaptive voxelization of the figure. Various regions of the figure could be partitioned in automated fashion using differently sized voxels, thus providing an appropriate number of voxels in each particular region so that the corresponding skeleton structure in that region could be generated at an appropriate level of detail. It may also be possible to generate a sort of hierarchical control skeleton offering varying levels of articulation (LOA) for a figure. Perhaps the levels of articulation of the skeleton structure could even be mapped to different level of detail (LOD) representations of the model, allowing efficient switching of both LOA and LOD for an articulated figure during animation.

Acknowledgements. We would like to thank Meg Geroch, Matt Lewis, and Pete Carswell for lengthy discussions about the research. Matt also provided several humanoid data models for our use. We extend thanks to Peter Gerstmann for creating several other models for us. Additionally, we wish to thank Steve May, Rephael Wenger, and Wayne Carlson for their comments and suggestions. Finally, a special thanks goes to John Warren for the spirited discussion that inspired this research.

References

1. Bloomenthal J, Lim C (1999) Skeletal methods of shape manipulation.
   Available at http://www.unchainedgeometry.com
2. Breu H, Gil J, Kirkpatrick D, Werman M (1995) Linear time Euclidean distance algorithms. IEEE Trans Pattern Anal Mach Intell 17:529–533
3. Cormen TH, Leiserson CE, Rivest RL (1990) Introduction to algorithms. MIT Press, Cambridge, Mass.
4. Danielsson P-E (1980) Euclidean distance mapping. Comput Graph Image Process 14(3):227–248
5. Gagvani N, Silver D (1999) Realistic volume animation with alias. In: Volume Graphics, chap 15. Springer, Berlin Heidelberg
6. Gagvani N, Kenchammana-Hosekote D, Silver D (1998) Volume animation using the skeleton tree. In: IEEE Symposium on Volume Visualization, pp 47–54; ISBN 0-8186-9180-8
7. Ge Y, Fitzpatrick JM (1996) On the generation of skeletons from discrete Euclidean distance maps. IEEE Trans Pattern Anal Mach Intell 18(11):1055–1066
8. Lee T-C, Kashyap RL, Chu C-N (1994) Building skeleton models via 3-D medial surface/axis thinning algorithms. CVGIP: Graph Models Image Process 56(6):462–478
9. Ogniewicz RL, Kübler O (1995) Hierarchic voronoi skeletons. Pattern Recognition 28(3):343–359
10. Staunton RC (1996) An analysis of hexagonal thinning algorithms and skeletal shape representation. Pattern Recognition 29(7):1131–1146
11. Teichmann M, Teller S (1998) Assisted articulation of closed polygonal models.
    Available at http://graphics.lcs.mit.edu/~marekt
12. Tsao Y-F, Fu K-S (1984) Stochastic skeleton modeling of objects. Comput Vis Graph Image Process 25:348–370
13. Wade L (2000) Automated generation of control skeletons for use in animation. PhD thesis, Ohio State University
14. Wade L, Parent RE (2000) Fast, fully-automated generation of control skeletons for use in animation. In: Proceedings of Computer Animation 2000, pp 189–194; A lengthier version is available as Technical Report OSU-ACCAD-9/99-TR3 (1999) Ohio State University, Advanced Computing Center for the Arts and Design

LAWSON WADE received his B.A. in Mathematics and Computer Science from Capital University in 1991 before continuing on in Computer and Information Science at The Ohio State University, where he received his M.S. in 1993 and his Ph.D. in 2000. His research interests include computer graphics and animation, computational geometry, and algorithms.

RICK PARENT is an Associate Professor in the Department of Computer and Information Science at Ohio State University. He received his Ph.D. in Computer Science from Ohio State University in 1977 and joined the faculty there in 1985. His research focuses on motion control algorithms for computer animation, and he is particularly interested in the modeling and animation of the human form.