Rig Retargeting for 3D Animation

Martin Poirier* Eric Paquette†

Multimedia Lab, École de technologie supérieure

Abstract

This paper presents a new approach to facilitate reuse and remixing in character animation. It demonstrates a method for automatically adapting existing skeletons to different characters. While the method can be applied to simple skeletons, it also proposes a new approach that is applicable to high quality animation as it is able to deal with complex skeletons that include control bones (those that drive deforming bones). Given a character mesh and a skeleton, the method adapts the skeleton to the character by matching topology graphs between the two. It proposes specific multiresolution and symmetry approaches as well as a simple yet effective shape descriptor. Together, these provide a robust retargeting that can also be tuned between the original skeleton shape and the mesh shape with intuitive weights. Furthermore, the method can be used for partial retargeting to directly attach skeleton parts to specific limbs. Finally, it is efficient as our prototype implementation generally takes less than 30 seconds to adapt a skeleton to a character.

Index Terms: I.3.7 [Computer Graphics]: Animation— [I.3.5]: Computer Graphics—Geometric algorithms, languages, and systems

1 Introduction

One of the most frequently used methods of character animation is the use of a skeleton to drive the deformation of its associated geometry. The transformation of each bone controls how the model is deformed through time and space. Systems like this often define animation sequences as building blocks for later reuse and remixing.

One problem that then arises is how to reuse those action blocks for different characters. Various algorithms were developed to adapt animations of one skeleton to others of different proportions. While this solves the problem of reusing actions, animation studios also try to maintain similar skeletons for their different characters (See Figure 1). This way, animators can work efficiently on several skeletons and reuse special skeleton controls developed on previous projects. Automatically adapting a skeleton to a different character is still very much an ongoing research subject. While certain solutions are able to deal with simple skeletons with few joints, these are usually far from the complex multilayered skeletons used in professional animations. Transfering such skeletons manually between characters is a tedious task. Our discussions with professional artists revealed that transferring complex skeletons such as the ones found in this paper can take many hours. Given a source skeleton and a target mesh, the proposed method retargets complex skeletons in less than a minute, substantially lightening the burden on artists.

*e-mail: martin.poirier.4@ens.etsmtl.ca
†e-mail: eric.paquette@etsmtl.ca

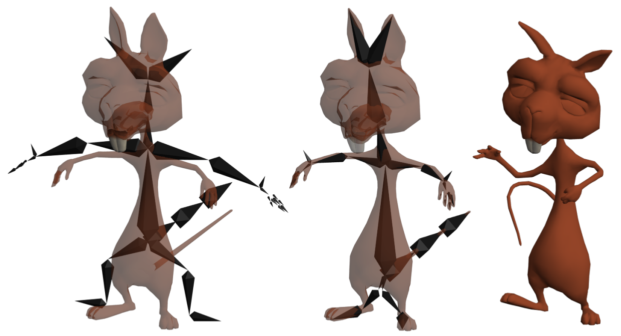

Figure 1: Rinky and a retargeted skeleton with inverse kinematics / forward kinematics switches.

1.1 Contribution

This paper presents a method to use topology graphs to automatically adapt complex deformation skeletons to different input meshes. Complex skeletons are defined here as having full control on all limbs as well as using control bones (bones that directly or indirectly control the movement of other deforming bones). The main contributions of our method are:

1. A multiresolution topology graph filter adapted to retargeting constraints;
2. A precise way to detect subgraph symmetries using a *shape descriptor*;
3. A robust method to match multiresolution subgraphs using the local shape descriptor;
4. A simple weighting function for inner joint placement providing a balanced control between following the original skeleton shape or the input mesh shape;
5. A comprehensive way to deal with control bones.

2 Related Work

2.1 Skeleton Extraction

Extracting a 1D skeleton from static meshes is a well known problem with varied solutions. The following review will only cover representative work for the most common approaches. For a more thorough review of the field, please read the survey by Cornea *et al.* [6].

Skeleton extraction methods can roughly be separated into two main categories based on the processing method they use: a volumetric method based on voxels or a geometric method based on polygonal faces.

2.1.1 Volumetric Methods

The major advantage of volumetric methods is that since the skeleton extraction is based on a volumetric representation of the input object, these methods can handle objects made of multiple parts, with overlaping geometry and disconnected components (like clothes over a body). On the other hand, thin limbs are often problematic because of the size restriction of the voxels, where a size small enough to capture thinner sections of a mesh would result in prohibitive runtimes. Volumetric algorithms are usually limited to closed meshes and are noisy due to the volumetric approximations. Approaches based on a volumetric method often use the medial surface [3, 17] or the discretization of force fields [5].

2.1.2 Geometric Methods

Geometric methods use the object mesh to extract the skeleton and thus will create disconnected graphs for disconnected overlapping geometry. The advantage is that they are typically much faster to compute and the resulting skeleton is smoother than that of volumetric methods. Geometric approaches can deal with thin limbs at no additional runtime cost, but are often peppered with small noisy arcs that do not represent important topology and have to be removed.

A popular approach, strongly rooted in Morse theory, is through the construction of a Reeb graph: a data structure that explicitly represents the topology of a shape. Reeb graphs are strictly 1D structures; it is therefore important to pair them with a 3D space embedding algorithm. This can be done during the graph generation [13] or as a post processing pass [2] using the pairing between the geometry and the graph's arcs or using contour constriction [16], for example. Other methods like fuzzy clustering [11] make a more extensive use of the geometric information to first segment the mesh into common patches and then use these for skeletonization. A more recent approach using a geometric method is to repeat mesh contractions through constrained Laplacian smoothing [1]. Unlike Reeb graphs, this method is noise and pose invariant and does not depend on a global weighting function.

Some approaches typically used with a volumetric method are also applicable to geometric methods. The shape diameter function [15], related to the medial surface, is one of them.

Our implementation uses Reeb graphs because of their relative simplicity of creation. As a geometric method, their ability to treat thin limbs as well as thick ones, was also considered essential for dealing with full featured skeletons.

2.2 Deformation Skeleton Generation and Retargeting

Generating a deformation skeleton from a mesh or adapting an existing one is also a much studied topic in character animation. While certain methods are based on an implicit definition of the skeleton, like the one used in the game *Spore* [9], this review will be limited to those that are based on generic static mesh input because they are closer in application to the goal of the technique presented here and are the most common in animation. Generating or retargeting a skeleton both require solving a lot of the same problems (such as joint placement), therefore, the techniques used are often similar.

A recent retargeting method [3] is based on simple template skeleton embedding. Skeletons are roughly detailed without smaller limbs like fingers and without any control bones. Embedding in this case is based on packing spheres on the medial surface of the input mesh. Since a volumetric model is used, limbs have a minimum thickness below which they are undetected. On the up side, this technique is well suited for cases where a very simple skeleton is enough such as for children and other non-professional animators. Moreover, it does not exclusively depend on the topology of the mesh, and as such can embed skeletons on models that do not necessarily match in shape.

Some methods use knowledge of anatomy [7] to help in generating usable skeletons. These algorithms are usually limited to the human skeleton, but tend to give better results for their specific case, due to their hyperspecialization.

Other deformation skeleton generation approaches are based on extracting a topology graph from a mesh and creating bones on the resulting arcs. These solutions can also be combined with human anatomical studies [17] or can be made to work with more generic biped and quadruped rules [2]. Graph-based skeleton generation algorithms often have difficulty with the automatic subdivision of arcs (adding inner joints): knees and elbows are typically problematic. While templates or anatomic rules can help with this problem, they do not solve it in a very satisfying manner when limb proportions are out of the ordinary (in a cartoon character, for example).

The algorithm presented in this paper is a graph-based retargeting algorithm. As such, it alleviates the limb subdivision and joint placement problems by using information from the source skeleton.

3 ALGORITHM OVERVIEW

Here is a simplified view of the algorithm, also illustrated in Figure 2.

A. Extract topological graph from mesh
   - Generate topology graph from mesh geometry
   - Filter and build multiresolution structure
   - Detect and tag symmetries

B. Extract graph from skeleton
   - Generate topology graph from skeleton hierarchy
   - Detect and tag symmetries
   - Link control bones to graph

C. Retarget skeleton to mesh
   - Match arcs between the skeleton graph and the multiresolution mesh graph
   - Place inner joints, balancing between fitness to the mesh graph and to the original skeleton
   - Transfer position updates to linked control bones

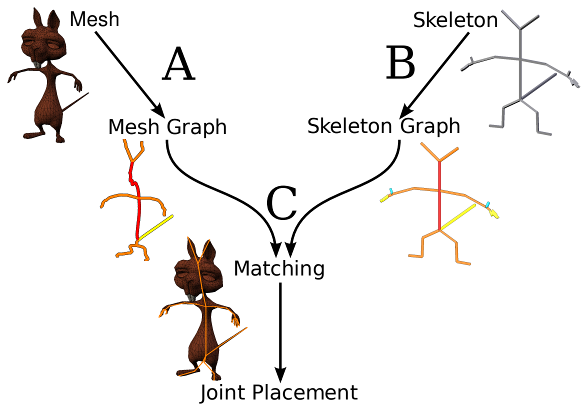

Figure 2: Algorithm overview.

4 SKELETONIZATION

Generating the 1D skeleton graph from the input 1D skeleton is quite simple as the skeleton is already a graph. Nevertheless, both the mesh graph and the skeleton graph will need to share a common starting node that will be used to bootstrap the matching method. This node is identified by the user both on the mesh (selecting a vertex) and on the skeleton (selecting a joint). This node will be referred to as the *head node*. The head node needs to lie on the primary symmetry axis as it will be used at other steps of the method. The head node is typically selected on the head of the character.

Generating the 1D mesh graph from the 3D mesh is much more complex. The goal is to contract the mesh onto itself, replacing 3D limbs (fingers, ears, tail, head, torso, etc.) with curves that fit the geometry. In order to extract this skeleton, the proposed method computes a Reeb graph. For our Reeb graph computation, the user first selects a vertex (the head node) on the mesh. From the head node, distances are computed for every node using a harmonic function based on geodesic distance extrema [2]. The graph generation algorithm is based on Pascucci *et al.* [13] with slight modifications to better deal with lower resolution meshes. The topology graph is embedded in 3D along the contracted mesh. The 3D embedding of a point along the topology graph corresponds to its associated 3D position along the contracted mesh. Figure 2 and Figure 6 display examples of Reeb graphs positioned along their 3D embeddings.

The retargeting algorithms do not depend on properties specific to Reeb graphs; any other topology graph algorithm that correctly preserves all limbs would be equally suitable. As such, this process will not be explained in detail as the indicated literature provides enough information for implementation.

4.1 Filtering

The mesh topology graph typically exhibits too many arcs for our purpose since arcs are created for most topological features of the mesh, big or small. It is thus filtered to remove arcs representing unwanted or unneeded features. Note that filtering is only applied to the mesh graph since retargeting needs all the nodes and arcs from the skeleton graph. As with previous graph filtering methods [2], it is important to remove smaller arcs first in order to keep relevant details. The filtering attribute used is the length of the arcs' path along their 3D embeddings; arcs shorter than a controlled threshold are filtered out. While different attributes could have been used, like the difference in weight between both end nodes, it was experimentally found that using the embedding length of arcs gives a progressive filtering that follows the spatial importance of the mesh more closely, especially with regards to arcs that curve around in 3D space. This last property of filtering by length is important for the next step of the algorithm.

4.2 Multiresolution Filtering

Multiresolution filtering removes the need to tweak a global filtering threshold until the graph shape matches the skeleton. More importantly, it enables the matching algorithm to deal with situations where a single filtering factor would give incorrect results. This case is discussed later in Section 6.1.

The technique used here is different from the multiresolution Reeb graph method used by Hilaga *et al.* [10] and the persistence hierarchy used by Pascucci *et al.* [13] on one major point: Connected limb placement requires that graph nodes present on more than one resolution level stay fixed in 3D space.

This restriction on node placement is achieved with the following prune/merge method, illustrated in Figure 3.

- Internal arcs (a) are merged to their node closest to the *head node*.
- External arcs (b) are pruned entirely.
- Arcs connected by a degree 2 node (c) are merged so that the extent of the graph does not shrink.

Multiresolution filtering is done by copying the high resolution base Reeb graph multiple times. Then, each resolution level is filtered with a progressively increasing threshold on arc length, leaving a coarsely detailed graph at the lowest level while each additional level adds more and more details. Nodes and arcs are linked to their corresponding counterpart at preceding and subsequent resolutions to facilitate navigation between levels.

The filtering threshold varies linearly between zero (no filtering) and a user selected value. A typical value of 10% of the graph was experimentally determined. The implementation uses a fixed number (10) of multiresolution levels which was enough for all of the presented test cases.

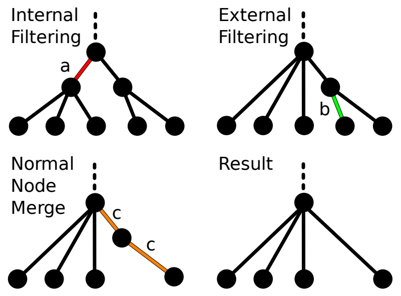

Figure 3: Filtering internal and external arcs of a subtree.

5 Symmetry

Symmetry detection applies to the mesh and skeleton graphs. For the mesh graph, it is computed independantly at each resolution level. The proposed approach is inspired by the simple tree depth and first child degree comparison [2], but is more robust by comparing full subtree shape instead. Like in the depth and child degree method, the user has manually selected a head node lying on the primary symmetry axis. The proposed method is not overly sensitive to the position of this node, but it must lie roughly on the geometrical axis of symmetry if the algorithm is to correctly identify geometrical symmetries. For humanoid models, any node from the top of the head is usually good enough. After selecting the head node, the algorithm works down recursively, grouping subtrees with similar shape into symmetry groups, continuing recursively on each child node. A *symmetry level* is also assigned to each arc corresponding to how far along the graph they are from the primary symmetry axis; 1 being on the primary axis and increasing as it reaches secondary symmetries. This simple method is based on the following assumptions: (a) the graph is actually a tree, and as such all arcs are oriented away from the *head node*, and (b) the subtree shape descriptors of two symmetric subtrees are equivalent. Such restrictions are easy to enforce and are common in the industry.

5.1 Shape Descriptor

To allow for efficient and robust symmetry detection and skeleton matching, a *shape descriptor* is proposed. A subtree shape descriptor is defined here as the number of nodes for each level going down from the root node of a subtree. A single node with two children would have a shape of $\{1, 2\}$ (Figure 4 left). If one child had two of its own while the other had three, the shape would become $\{1, 2, 5\}$ (Figure 4 right). The shape descriptor is defined for nodes, but also for arcs. The shape descriptor of an arc is equal to the descriptor of its first node going towards the leaves of the tree.

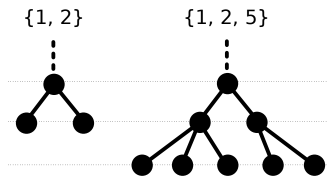

Figure 4: Shape descriptor of a subtree.

While this does not make any distinction as to which nodes the furthest children are connected to, it has proven robust enough for all tested cases. Moreover, using subtree shape for symmetry detection is computationally equivalent to the depth and child degree method [2] while being a better indicator for similarity. Figure 5 illustrates a case where using our shape descriptor correctly identifies symmetries while the depth and child method fails to do so.

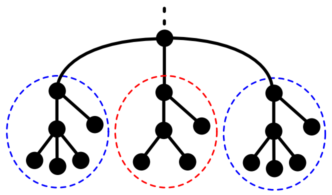

Figure 5: Symmetry detection with subtree shape descriptors. The two subtrees on the left and right have the same shape {1, 2, 3} while the one in the middle is alone with {1, 2, 2}. Using depth and degree, all three subtrees would show depth: 3, degree: 2.

5.2 Grouping

Since the shape descriptor only finds topological symmetries, our implementation further subdivides groups of symmetric arcs using a length variation threshold C. Subgroups are created when the difference between the length of the longest and shortest arcs is greater than C. This enables different arcs with the same subtree shape but different lengths to be correctly checked for symmetry, as shown in Figure 6.

Length grouping can also be supplemented with combinatorial axial and radial symmetry tests. If a group of topologically symmetric subtrees is not geometrically symmetric, the different subgroups are tested to see if there is an axial or radial symmetry between them. This would be useful in cases where, for example, a character's tail is the same length as his legs; only the legs would share a geometric axial symmetry. Nevertheless, length grouping alone was enough to correctly subdivide the groups in all of our tests.

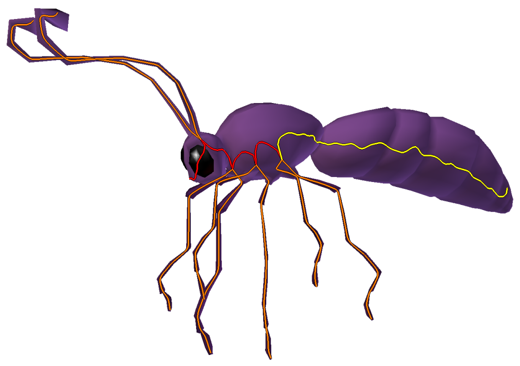

Figure 6: Butterfly (in purple) together with its topology graph. The two posterior legs and tail have the same subtree shape, but the length difference falls outside the threshold. Therefore the two legs are grouped into an axial symmetry while the tail is left alone.

5.3 Tagging

For a more stable and consistent retargeting, the proposed approach introduces symmetry tags. Arcs in axial symmetry groups are automatically tagged as *left* and *right* while arcs in radial symmetry groups are ordered by arc length and then numbered. The type of symmetry is also expressed using tags as shown in Figure 6: *primary symmetry* in red, *axial symmetry* in orange and *no symmetry* in yellow. These tags are used during the retargeting process which helps in keeping the correspondence with existing animation blocks. The left and right tags are computed based on the principal axis (X, Y or Z) closest to the vector from one limb to the other. The limb to the positive side of the axis is labeled as left while the limb on the negative side is labeled as right. The legs, arms, ears, etc. of both the mesh and the skeleton will all be consistant but might be inverted as our implementation does not assume the facing direction of the character. If needed, the user could provide this direction to the algorithm in order to determine which direction of the axis (positive or negative), corresponds to the actual left and right of the character.

6 Retargeting

To address the retargeting problem, we propose a multiresolution matching method together with a joint placement method. Compared to previous retargeting methods, the proposed method is more robust. Furthermore, previous approaches were restricted to deformation bones while the proposed method handles deformation as well as control bones. Sections 6.1 to 6.2 describe the robust deformation bones retargeting while Section 6.3 describes the control bones retargeting.

6.1 Multiresolution Matching

The first step of the retargeting process is matching arcs of the skeleton graph to those of the multiresolution mesh graph. The mesh topology graph typically has several arcs that represent details not found in the skeleton we are trying to retarget. As mentionned in Section 4.2, identifying relevant arcs is a difficult task. For example, retargeting the various skeletons of Figure 7 will require the matching method to keep a different subset of the mesh arcs. The

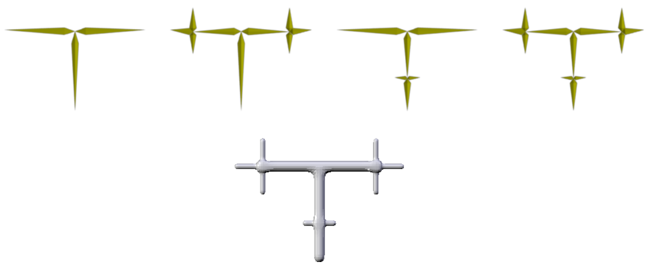

Figure 7: The arc by arc local shape matching is effective and versatile. The various skeleton configurations (top) can all be retargeted to the same mesh (bottom).

third skeleton of Figure 7 is an example where the method keeps the small arcs at the base of the "T" shape while removing the larger arcs at the top of the "T" shape.

Here's a summary of the multiresolution matching method.

- Starting from the head nodes (blue)

1. For each arc from the current node

   1.1 Match the skeleton arc to a mesh arc using symmetry tags and levels.

   1.2 Check the corresponding mesh arcs at increasingly finer resolution levels to find an equal local shape.

   1.3 The next node is the following node along the arc.

   1.4 Check the corresponding mesh nodes at coarser resolution levels to find the coarsest node that has an equal local shape.

   1.5 Recursively work on the next node (step 1).

Algorithm 1 presents further details of the matching method. Figure 8 presents an example of this process and the accompanying video [14] also presents further examples. The matching algorithm compares arcs using symmetry levels and tags and the shape descriptor. While the full shape descriptor was used when detecting symmetry, the matching only uses the first two numbers of the shape descriptor. This is called the *local shape descriptor* (*LShape*

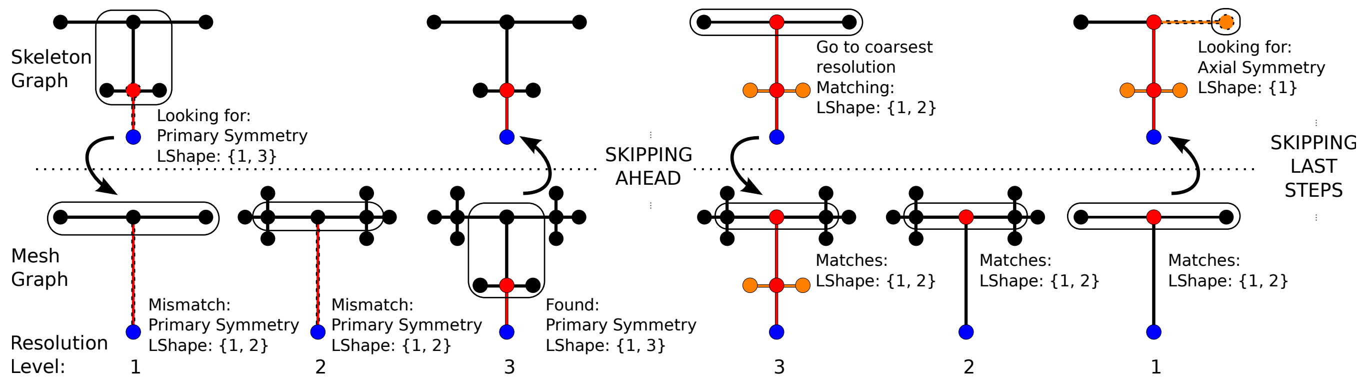
Skeleton Graph
Looking for: Primary Symmetry LShape: {1, 3}
Go to coarsest resolution Matching: LShape: {1, 2}
Looking for: Axial Symmetry LShape: {1}
SKIPPING AHEAD
SKIPPING LAST STEPS
Mesh Graph
Mismatch: Primary Symmetry LShape: {1, 2}
Mismatch: Primary Symmetry LShape: {1, 2}
Found: Primary Symmetry LShape: {1, 3}
Matches: LShape: {1, 2}
Matches: LShape: {1, 2}
Matches: LShape: {1, 2}
Resolution Level:
1
2
3
3
2
1

Figure 8: Multiresolution matching example. Head nodes in blue. Symmetry axis colored when they are matched.

Algorithm 1: Match(node$_M$, node$_S$, arc$_S$)

```
/* Given a skeleton arc (arc$_S$) connected to a skeleton node
(node$_S$), find a matching mesh arc (arc$_M$) connected to the
mesh node (node$_M$). */
foreach unused arc$_M$ do
    if SymLevelTags(arc$_M$) = SymLevelTags(arc$_S$) then
        /* symmetry level & tags match */
        while LShape(arc$_S$) $\neq$ LShape(arc$_M$) do
            arc$_M$ $\leftarrow$ arc$_M$ at finer resolution
        if LShape(arc$_S$) = LShape(arc$_M$) then
            /* symmetry level & tags + local shape match */
            node$_M$ $\leftarrow$ new node reached through arc$_M$
            node$_S$ $\leftarrow$ new node reached through arc$_S$
            mark arc$_M$ (at all resolutions) and arc$_S$ as used
            node$_{CM}$ $\leftarrow$ node$_M$ at coarser resolution
            while LShape(node$_S$) = LShape(node$_{CM}$) do
                node$_M$ $\leftarrow$ node$_{CM}$
                node$_{CM}$ $\leftarrow$ node$_M$ at coarser resolution
            foreach unused arc$_S$ connected to node$_S$ do
                Match(node$_M$, node$_S$, arc$_S$)
```

in Algorithm 1) and it provides the information needed to match the current skeleton arc to the relevant mesh arc. Using the local shape enables the method to match to different resolution levels on an arc by arc basis. As previously mentionned, in the third example in Figure 7 the two smaller lower arcs need to be kept while the longer ones at the top need to be removed. This is a case where a unique filtering threshold would not yield the correct result, while the multiresolution matching does. The Match function is first called using the skeleton and mesh *head nodes* previously indicated by the user, processing all its connected skeleton arcs.

This matching method handles cases where the mesh graph or the skeleton graph have more arcs than the other. The multiresolution matching takes care of cases where the mesh graph has more arcs than needed, as was shown in Figure 7. Also, the matching algorithm can easily detect superfluous skeleton arcs and remove them automatically, like the antennas in Figure 9 or the fifth finger in the humanoid of Figure 10. Nevertheless, this subgraph matching method limits the retargeting to situations when the topology of the mesh makes sense with respect to that of the skeleton. While this is commonly the case, it is less generic than the method used by Baran and Popović [3] which can retarget a simple humanoid skeleton to a donut for example.

6.2 Joint Placement

The matching method determines the correspondance between skeleton arcs and mesh arcs. While the arcs now correspond, the position of the skeleton joints still have to be set. This positioning is trivial for skeleton arcs made of a single bone: the extremities of the bone are positioned at the first and last 3D embedding positions of the mesh arc. The positioning is much more complex for skeleton arcs that contain two bones or more. Such arcs correspond to the legs of the insects in Figure 9 where four joints have to be positioned along each leg of the insect. Considering such a skeleton arc matched to a mesh arc, the corresponding skeleton bones will be positioned in 3D along the mesh arc 3D embeddings. Consider a skeleton arc with t bones and $t+1$ joints and consider mesh arc embeddings composed of an ordered list of points $V=[v_1,v_2,\ldots,v_m]$. Each of the $t+1$ joints will be assigned an index $k_i \in \{1,\ldots,m\}$ corresponding to an embedding position v_{k_i} with respect to the mesh. The first joint is positioned at $v_{k_1}=v_1$. Each of the $t-1$ *inner* joints between the t bones are assigned to a specific $v_{k_i} \in \{v_2,\ldots,v_{m-1}\}$. Finally, the last joint is assigned to $v_{k_{t+1}}=v_m$. The inner joint positions $v_{k_2},v_{k_3},\ldots,v_{k_t}$ are selected by minimizing the following weight cost:

$$weight = \sum_{i=1}^{t-1} \underbrace{\gamma_\theta|\theta_i|}_{\text{angle}} + \sum_{i=1}^{t} \Big[\underbrace{\gamma_l|l_i|^2}_{\text{length}} + \underbrace{\gamma_x x_i}_{\text{distance}}\Big] \tag{1}$$

$$\theta_i \rightarrow \text{angle joint } i - \text{initial angle joint } i$$

$$l_i \rightarrow \frac{\text{length bone } i - \text{initial length bone } i}{\text{initial length bone } i}$$

$$x_i \rightarrow \begin{array}{l}\text{maximum distance from bone } i \text{ to the mesh arc 3D} \\ \text{embeddings from } V \text{ at indexes between } k_i \text{ and } k_{i+1}\end{array}$$

The γ user defined parameters control whether the result should conform more to the original shape of the skeleton (parameters γ_θ and γ_l) or to the shape of the mesh (parameter γ_x). The difference in joint angle term $\sum_{i=1}^{t-1}\gamma_\theta|\theta_i|$ will penalize changes from the skeleton's joint angles. The bone length difference (length bone i - initial length bone i) is not very meaningful since the same difference is significant for a small bone while it is negligible for a large bone. Instead, we use a relative measure: the length difference as a ratio of the original length $\left(\frac{\text{length bone } i - \text{initial length bone } i}{\text{initial length bone } i}\right)$. When using the ratio directly, it is very difficult to properly adjust the γ_l parameter for some meshes, such as the mesh of Figure 9(c). The squared ratio $\left(|l_i|^2\right)$ provides an amplification which is useful in situations where the bone length of the original skeleton and the retargeted skeleton differ in a significant manner. The distance term

$\sum_{i=1}^{t} \gamma_x x_i$ will penalize bone lengths and skeleton poses that do not fit the mesh. This is computed as the maximal distance from a bone to the corresponding mesh arc 3D embeddings.

Hard restrictions are also enforced in the solver algorithm to prevent backtracking on the arc and zero-length bones ($i < j \rightarrow k_i < k_j$). The problem is translated into the solveInnerJoints function (Algorithm 2) that finds the best inner joint positions given the position of the previous two joints and the remaining number of inner joints to position. Solving the problem is done by calling solveInnerJoints$(1, 1, t-1)$.

Algorithm 2: solveInnerJoints(k_p, k_c, j)

```
$k_p, k_c \in \{1, \ldots, m-1\}$: previous, current joint position indexes
$j \in \{0, \ldots, t-1\}$: number of inner joints left
if $j > 0$ then
    result.weight $\leftarrow \infty$
    foreach $k_n \leftarrow$ all position indexes for next joint do
        weight $\leftarrow$ AngleCost ($k_p$, $k_c$, $k_n$)        // $\gamma_\theta |\theta_i|$
            + LengthCost ($k_c$, $k_n$)                  // $\gamma_l |l_i|^2$
            + DistanceCost ($k_c$, $k_n$)                // $\gamma_x x_i$
        if weight $\geq$ result.weight then
            /* skip recursion: weight too high */
        else
            child $\leftarrow$ solveInnerJoints ($k_c$, $k_n$, $j$ - 1)
            weight $\leftarrow$ weight + child.weight
            if weight $<$ result.weight then
                result.weight $\leftarrow$ weight
                result.positions $\leftarrow$ [$k_n$ : child.positions]
else /* $j = 0$ */
    $k_n \leftarrow m$ // last joint on arc
    result.weight $\leftarrow$ AngleCost ($k_p$, $k_c$, $k_n$)        // $\gamma_\theta |\theta_i|$
        + LengthCost ($k_c$, $k_n$)                      // $\gamma_l |l_i|^2$
        + DistanceCost ($k_c$, $k_n$)                    // $\gamma_x x_i$
return result
```

Comparatively, the method used by Baran and Popović [3] also uses bone length and distance penalities, but relies on the orientation of bones instead of the angle of each joint, which renders it sensitive to the orientation of limbs in the target mesh such as the arms from a vertical orientation to a horizontal orientation in Figure 10.

6.2.1 Optimizing the Placement Problem

The complex combinatorial problem of joint placement is reduced to a simpler problem using the dynamic programming technique called memoization. Since solveInnerJoints is a recursive function and most of the recursive calls will be done several times, the memoization approach caches the intermediate results, transforming the exponential runtime of the brute force algorithm into a quadratic one. While this approach is still exhaustive, tests done using heuristic methods (gradient descent and simulated annealing) showed a noticeable loss in quality without a useful decrease in runtime.

6.3 Control Bones

Control bones are defined as non-deforming bones that are used to drive deforming bones, to add a common parent to disconnected bone chains or other such purposes. These kinds of bone are very frequent in animation-quality skeletons because they enable users to create more flexible skeleton hierarchies, more easily controllable actions (finger flex, foot roll, foot skating prevention, etc.) and are generally considered essential for professional animation. The accompanying video [14] shows the effect of various types of control bones. Most known approaches to the retargeting problem do not deal with control bones.

These bones also need to be repositioned but since their position is rarely dictated directly by the shape of the character (as is the case with its deforming skeleton), it leaves us with a different problem. The approach used is to link each control bone to a single deforming bone before retargeting and to propagate the transformation from deforming bone to the control bones. When considering a control bone, the method first tries to find a deformation bone which is constrained by the control bone. If there is no such bone, the method checks if the control bone has a parent. Again, if there is no such bone, the method tries to find a child bone. Deforming bones are thus selected using the following priorities:

1. Constraint target bone: Bones used as inverse kinematics and Pole targets, action controls and the like. Such a control bone is linked to the constraint owner.
2. Parent bone: Bones with a parent, like those in chains of control bones. Such a control bone is linked to its parent. If the parent bone is also a control bone, the method follows the links to the last control bones and then to the deformation bone linked to this last control bone. The current control bone is linked to this deformation bone.
3. Child bone: Bones with children, like root bones. Since such a control bone can be parent of many child bones, the method needs to select the relevant one. The bone is thus linked to its child on the first symmetry level. This links a spinal chord root to the spine instead of the legs.

At linking time, the offset between the control bone and deforming bone is calculated. Control bones are then repositioned by deriving a rotation and scaling factor from the transformation of their deforming bone. The offset and length of control bones are scaled and the bones themselves are then rotated with respect to the tip of their deforming bone.

This technique is generic and could be paired with other methods of transforming skeletons: mixed with other retargeting techniques or even as a way of automatically adjusting control bones when an artist modifies deforming bones and joints.

7 Results

7.1 Implementation

Implementation has been done using the Open Source software Blender [4] for mesh and skeleton processing. Blender provides an automatic skinning function to attach a skeleton to a mesh using bone heat equilibrium [3]. This skinning approach was used for all of our tests. Solving the harmonic equation for Reeb graph node weights is done by a sparse matrix direct solver using SuperLU [8] through the OpenNL library [12]. The algorithm currently used for Reeb graph generation can result in embedding problems such as arcs that sometimes puncture through the surface of the mesh. Nevertheless, methods to solve these problems have been devised and it would be strictly a matter of applying one of the known solutions [1, 13] to fix them.

User input is limited to simple parameters (joint placement weights, Section 6.2, and length variation threshold) as well as indicating one matching *head node* on the mesh and the skeleton.

7.2 Applications

The following images show our retargeting results:

- A biped rig with inverse kinematics (IK) / forward kinematics (FK) switches retargeted to a squirrel (Figure 1).
- A generic insect rig retargeted to a butterfly mesh and a fly mesh (Figure 9) to showcase the joint placement algorithm in multi-jointed limbs. In these cases, the penalty function was given more weight to the shape of the target mesh than the original skeleton.

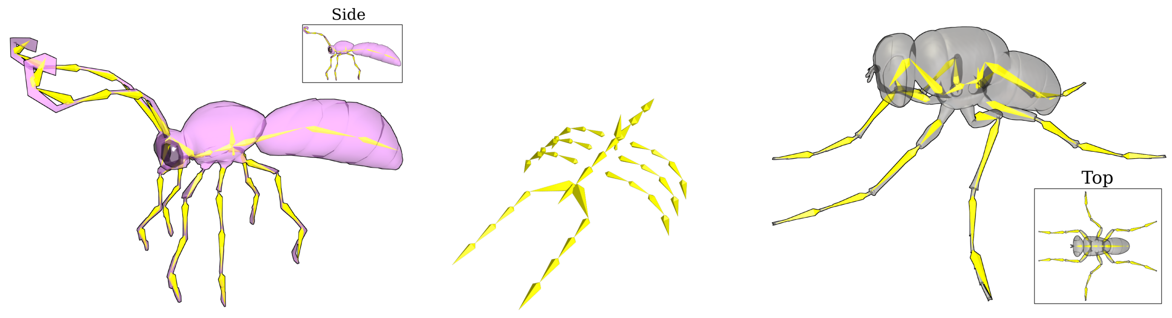

Figure 9: (a) Butterfly mesh and retargeted skeleton (b) The generic insect rig used (antennas were removed when retargeting to the fly) (c) Fly mesh and retargeted skeleton.

- A full featured human skeleton with IK and pole constraints, finger flexors and foot roll controls retargeted to a cartoon character, Figure 10(a), as well as partial retargeting for the arm, Figure 10(b), and the leg, Figure 10(c). In these cases, joint placement is better when the limbs are half bent since the weight function forces the joint to position itself where the limb is bent. We consider this skeleton to be complex because of its many limbs and control bones. Furthermore, this skeleton was provided by a professional animation studio and has been used in animated shorts and commercials.

The accompanying video [14] also features other retargeting examples and animation sequences. As can be seen in these examples and in the video, the proposed multiresolution filtering, symmetry grouping, symmetry tagging, and shape descriptor provide a robust retargeting method. Also note that the method can retarget skeletons that present a large geometric difference from the meshes (see Figure 10).

7.3 Performance

7.3.1 Graph from Mesh

Weight calculation on the mesh is done using the well known Dijkstra's *SPF* algorithm. The complexity of this operation is $O(e \log v)$ where v is the number of vertices and e the number of edges.

The theoretical lower bound for Reeb graph creation is $O(v \log v)$. Post processing filtering is at worst $O(n^2)$ where n is the number of nodes in the graph which, barring degenerate cases, makes it faster than the former.

7.3.2 Graph from Skeleton

Skeleton graph creation is $O(j)$ where j is the number of joints while its post processing (including control bones linking) is $O(j^2)$. Because of the smaller number of joints compared to the size of the input mesh, this step is always dwarfed by others: the slowest test case is the full human skeleton, taking only 2 milliseconds.

7.3.3 Retargeting Skeleton to Mesh

Matching has an upper bound of $O(ak\Delta_M)$ where a is the number of arcs in the skeleton graph, k the number of levels and Δ_M the maximum node degree of the multiresolution mesh graph.

Inner joint placement is a combinatorial problem of potential complexity $O(m^t)$ where t is the number of inner joints to position and m the number of embedding points to consider. Memoization (a dynamic programming principle) reduces the problem to filling in a cache table of $O(tm^2)$ complexity in time and space, trading off higher memory consumption for a faster runtime. Furthermore, inner joint placement for different arcs is inherently easy to parallelize and has been implemented to take full advantage of today's multi-core processors.

7.3.4 Global Runtime

The element that has the largest impact on runtime is the input itself. For skeletons with a lot of inner joints, runtime is dominated by the retargeting joint placement while large meshes will have a longer mesh graph creation (See Table 1).

Models and skeletons used are production grade, having been used in professional animation work (short movies, advertisements, etc.). Runtime performance on high poly characters are comparable to other solutions [3] while being much faster in other cases and offering unprecedented results in skeleton complexity and control bone repositioning.

Table 1: Performance (Core^{TM}2 Duo 2.4GHz / 4 GB RAM).

Model	Faces	Joints	Graph	Retarget	Total
Butterfly	2,428	51	0.3s	1.5s	1.8s
Fly	8,834	41	1.2s	4.5s	6.0s
Rinky	8,562	42	1.9s	0.4s	2.7s
Toon	27,232	136	7.5s	23.8s	32.6s

7.4 Comparison

Table 2 shows a point by point comparison with other methods for reusing skeletons between characters. This shows that the presented method, while not topologically independant like Baran and Popović [3], is better suited to handle professional level animation characters especially with its use of multiresolution matching and its handling of control bones.

8 Conclusion and Future Work

This paper presented a method to retarget skeletons to meshes. The method proposes a simple and effective shape descriptor to compare graph subtrees. It introduces an algorithm that matches the skeleton to the mesh using the shape descriptor and a robust symmetry tagging and grouping. Skeleton joints are then positioned using the developed penalty function with control between mesh and skeleton shapes. A linking system is also proposed to reposition control bones. The method is intuitive with few simple parameters: head node, length variation threshold and weight balancing parameters. It can handle complex skeletons with control bones and was applied to data from real production work. While professional artists need hours to retarget skeletons such as the humanoid presented in this paper, the proposed method requires less than one minute of computation. Finally, compared to previous work, the method provides better joint placement and can handle skeletons of greater complexity.

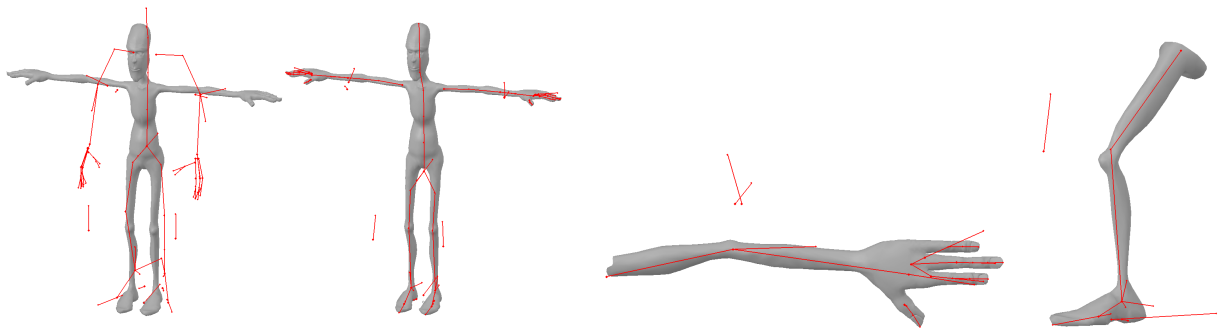
Figure 10: (a) Cartoon character with retargeted skeleton. (b) Cartoon arm with retargeted skeleton: IK and pole constraints as well as finger flex controls. (c) Cartoon leg with retargeted skeleton: reverse foot rig and roll controls.

Table 2: Comparison with previous work.

	Baran and Popović [3]	Aujay *et al.* [2]	Proposed Method
Skeletonization	volumetric	geometric	geometric
Skeleton Complexity	simple skeletons	biped / quadruped templates	full-featured professional skeletons
Control Bones	no	no	yes
Joint Placement	fixed weights	fixed template	controlable and balanced weights
Filtering	n/a	single threshold	multi-resolution
Topology Independant	fully	no	through multi-resolution
Orientation Independant	no	yes	yes

A change to the joint placement approach that might yield benefits would be to remove the assumption that start and end joints on limbs must correspond to those of the underlying mesh arc. This could give better results for forking limbs, like hands, which tend to be positioned a bit too far up. However, such a change would remove the independent status of each retargeted arc, most likely complicating the placement problem by a non-negligible factor. Mesh segmentation algorithms could also be used as guides for the inner joint placement problem. On the other hand, such algorithms are typically much slower than the current runtime.

A completely different issue is how to deal with deforming bones not based on topology like those used for muscle bulges and facial deformations. Those might be dealt with by mixing the control bone linking mechanism and a procedure to recognize the underlying attached mesh shapes. This would however introduce a dependency on the mesh originally attached to the skeleton.

9 Acknowledgements

The authors would like to thank the Blender Foundation, especially the Big Buck Bunny open movie project for all available data; ProMotion studio for humanoid skeletons and constructive discussions; Daniel Salazar and Bassam Kurdali for their 3D models.

References

[1] O. K.-C. Au, C.-L. Tai, H.-K. Chu, D. Cohen-Or, and T.-Y. Lee. Skeleton extraction by mesh contraction. *ACM Transactions on Graphics*, 27(3), 2008.

[2] G. Aujay, F. Hétroy, F. Lazarus, and C. Depraz. Harmonic skeleton for realistic character animation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 151–160, 2007.

[3] I. Baran and J. Popović. Automatic rigging and animation of 3D characters. *ACM Transactions on Graphics*, 26(3):72, 2007.

[4] Blender-Foundation. Blender 3D, 2008. www.blender.org.

[5] D. Brunner and G. Brunnett. Fast force field approximation and its application to skeletonization of discrete 3D objects. *Computer Graphics Forum, Vol. 27, No. 2 (2008), Eurographics 2008*, pages 261–270, 2008.

[6] N. D. Cornea, D. Silver, and P. Min. Curve-skeleton properties, applications, and algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):530–548, 2007. Member-Deborah Silver.

[7] F. Dellas, L. Moccozet, N. Magnenat-Thalmann, M. Mortara, G. Patanë, M. Spagnuolo, and B. Falcidieno. Knowledge-based extraction of control skeletons for animation. In *SMI '07: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007*, pages 51–60, 2007.

[8] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[9] C. Hecker, B. Raabe, R. W. Enslow, J. DeWeese, J. Maynard, and K. van Prooijen. Real-time motion retargeting to highly varied user-created morphologies. In *SIGGRAPH '08: ACM SIGGRAPH 2008 Papers*, pages 1–11, 2008.

[10] M. Hilaga, Y. Shinagawa, T. Kohmura, and T. Kunii. Topology matching for fully automatic similarity estimation of 3D shapes. In *SIGGRAPH01*, pages 203–212, 2001.

[11] S. Katz and A. Tal. Hierarchical mesh decomposition using fuzzy clustering and cuts. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 954–961, 2003.

[12] B. Levy. OpenNL: Open numerical library. alice.loria.fr/index.php/software/4-library/23-opennl.html.

[13] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust on-line computation of reeb graphs: simplicity and speed. *ACM Transactions on Graphics*, 26(3):58, 2007.

[14] M. Poirier and E. Paquette. Rig retargeting for 3d animation - video, 2009. http://profs.logti.etsmtl.ca/epaquette/Research/Papers/Poirier.2009/.

[15] L. Shapira, A. Shamir, and D. Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *The Visual Computer*, 24(4):249–259, 2008.

[16] J. Tierny, J.-P. Vandeborre, and M. Daoudi. Enhancing 3D mesh topological skeletons with discrete contour constrictions. *The Visual Computer*, 24(3):155–172, 2008.

[17] L. Wade and R. E. Parent. Fast, fully-automated generation of control skeletons for use in animation. In *CA '00: Proceedings of the Computer Animation*, page 164. IEEE Computer Society, 2000.