Base Manifold Meshes from Skeletons

Michal Piovarči*
Supervised by: Martin Madaras

Faculty of mathematics physics and informatics
Comenius university
Bratislava / Slovakia

Abstract

We propose an algorithm that generates a base manifold mesh from an input skeleton, based on Skeleton to Quad Dominant Mesh (SQM) algorithm which converts skeletons to meshes composed mainly from quadrilaterals. Each node in skeleton has assigned a sphere with a predefined radius. SQM algorithm first creates branch node polyhedrons for each sphere corresponding to a branch node. These polyhedrons are bridged with quadrilaterals in order to create the final base mesh. We have extended the algorithm to support generation of meshes from cyclic skeletons. We have also generalized skeleton nodes to ellipsoids instead of spheres. Finally, we extended the algorithm to generate meshes from linear skeletons without branching and from skeletons which root node is not a branch node. The generated base mesh is tessellated on GPU for better visual results.

Keywords: skeleton, convert, base mesh, manifold

1 Introduction

Skeletal structures are often used in computer graphics to represent basic topology of a model. This representation allows artists to conveniently animate articulated models, by manipulating key points represented as joints in skeletons. Skeletons corresponding to a model, are often provided by an artist, or extracted directly from the model [1]. Since skeletal structures carry an information about the topology of a model, we could apply a reverse process to skeleton extraction and recover the base mesh represented by a skeleton.

Such base meshes, generated directly from skeletal structures, could be used to ease the modelling of base models of articulated characters. An artist would only design the skeleton of the model and the base mesh would be generated automatically. This technique can also be used to procedurally generate articulated models. A base mesh generated from a supplied skeleton can be augmented with procedurally generated displacement maps in order to generate a complex model.

*michal.piovarci@gmail.com

In Section 2, the state of the art methods used in the area are described. In Section 3, the original SQM algorithm and its drawbacks are discussed. In Section 4, our implementation of base mesh generation is described. In Section 5, our proposed solutions to discussed drawbacks of the original SQM algorithm are presented. Finally, in Section 6 the results of our implementation are presented.

2 Related Work

The most notable algorithms generating base meshes from skeletons are B-mesh [4] by Ji et al. and SQM [2] by J. A. Bærentzen et al. The input for both algorithms is a skeleton with a sphere defined for each node of the skeleton which represents the local geometry of desired output base mesh. Both algorithms present a different way how to approach generation of base meshes from the input skeleton.

The former B-Mesh algorithm firstly generates geometry for paths connecting skeletal branch nodes. These paths are then stitched together at each branch node and the resulting mesh is evolved to better approximate the input skeleton. On the other hand SQM algorithm uses a reverse process. First polyhedrons corresponding to each branch node are generated. The generated polyhedrons are joined together via a tube consisting of quadrilaterals. The resulting mesh is subdivided to increase visual quality.

There are more techniques that generate base meshes but are not limited or used on skeletal structures only. In Solidifying wireframes [7] Srinivasan et al. proposed a method similar to B-Mesh. The proposed method firstly generated mesh corresponding to tubular parts of wireframes. These paths are later joined at branch nodes in a similar manner as in B-Mesh. Although the method is more general than B-Mesh it suffers from the same drawbacks mainly the stitching geometry which produces undesired triangular faces. In a more recent paper Leblanc et al. [6] proposed generating base meshes by iteratively combining blocks into cuboid shapes. Since our algorithm should operate on skeletal structures, by limiting the connectivity of blocks to skeletal structures only, we would lose many of the advantages of the original technique. Base mesh could be also recovered from medial axis transform [8]. However, due to the nature of me-

dial axis transform it is not suitable for editing by artists. Taking in account all the previous drawbacks and that "B-Mesh produces three to four times more irregular vertices than SQM" [2], we have decided to base our algorithm on SQM.

3 Original SQM Algorithm

The algorithm consist of four steps and one preprocessing step:

Preprocessing: Skeleton straightening - serves to simplify step number 3 of the algorithm.

Step 1: BNP generation - generation of branch node polyhedrons (BNPs).

Step 2: BNP refinement - subdivisions of BNPs.

Step 3: Creating the tubular structure - bridging of BNPs.

Step 4: Vertex placement - reverting straightened mesh to its original pose.

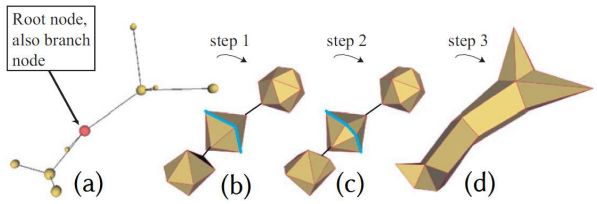


Figure 1: Steps of SQM algorithm. (a) the input skeleton; (b) generated BNPs; (c) refined BNPs; (d) BNPs bridges with quadrilateral tubes; Image from [2].

.

Straightening This is a preprocessing step of the algorithm that simplifies the generation of tubular structures. For each connection node its child is rotated, so that the edge between connection node and its child is parallel with the edge between connection node and its parent. This is useful, because during step 3 the algorithm needs to generate straight tubes only and does not need to take rotation into account.

BNP Generation A Branch Node Polyhedron (BNP) is a polyhedron assigned to a branch node. Vertices of a BNP correspond to a set of points that are generated by intersecting the sphere assigned to a branch node with each edge connected to said branch node. We will call these vertices as intersection vertices. To form a BNP intersection vertices are triangulated. After that each triangle is split into six triangles by inserting one vertex in the middle of each triangle and in the middle of each of the edges of the triangle. These vertices are then projected back onto the sphere associated with a branch node. This projection is needed because if the intersection vertices are coplanar, or nearly coplanar the generated polyhedron would have zero volume, or very small volume, respectively. The result of this step can be seen in Figure 1b.

BNP Refinement During step 3 of the algorithm, the BNPs connected via path are bridged with tubes consisting solely of quadrilaterals. This is done by connecting the one-rings of two corresponding intersection vertices with faces. To ensure that we can use only quadrilaterals the one-rings need to have the same valence. Each BNP is refined so that the valence of two intersection nodes lying on the same path are equal. We take the notion of a Link Intersection Edge (LIE): "An LIE is simply a set of edges in a subdivided BNP which belong to the links of two path vertices", from [2]. During the refinement phase only one representative edge of each LIE is subdivided. Subdivided BNPs are displayed in Figure 1c.

Creating the Tubular Structure After previous step of the algorithm, connected BNPs can be joined by a tube formed by quadrilaterals. The tube is divided into segments. Each of the segments corresponds to a connection node. Vertices corresponding to a certain connection node are projected onto its corresponding sphere. Leaf nodes are terminated with a triangle fan, which central vertex corresponds to the leaf nodes position. The result is illustrated in Figure 1d.

Vertex Placement The base mesh is now finished. All that remains is to reverse the rotations used to straighten the input skeleton. After final vertex placement the resulting mesh is smoothed with three iterations of Laplacian smoothing and attraction scheme.

Discussion Because SQM generates BNPs, it resembles the geometry of the input skeleton even without smoothing or evolution of the mesh. SQM produces small number of triangles because after the joining step triangles remain only in parts of the mesh corresponding to leaf nodes. Limitations of the algorithm are:

1. The root of the input skeleton has to be a branch node, as discussed in McDonells Skeleton Based Interactive 3D Modelling [3].
2. SQM can not generate a base mesh from linear skeletons without branching.
3. SQM supports only a sphere defined for each node of a skeleton to represent the local geometry where a more general input as ellipsoids may be desired.
4. A different termination method for leaf nodes, for example capsule termination, may be desired.
5. SQM can not handle implicitly defined cycles in the input skeleton.

The goal of our adaptation of SQM algorithm will be to improve upon all of the listed drawbacks as well as moving final vertex placement on GPU.

4 Our Base Mesh Implementation

Skeleton Straightening Skeleton straightening is a preprocessing step that simplifies bridging of branch node polyhedrons. Straightened skeleton is a skeleton which nodes in every path between two branch nodes, two leaf nodes, or a branch node and a leaf node are co-linear. In addition we have added an extra condition that angles between branch nodes child nodes should be the same in straightened skeleton as they are in the input skeleton. To achieve the first condition for each connection node, we take the normalized direction of a vector formed by connection nodes parents position and connection nodes position. The direction vector can be seen in Figure 2 as the green arrow. Then we project the child node onto the direction vector. The projected position is the position of the child node in the straightened skeleton. We then calculate rotation between connection nodes child original position and its new position, in respect to the position of connection node. Finally, we rotate all descendants of the connection node. In order to conform to the second condition, at each branch node we do not alter the position of its child nodes.

Figure 2: Skeleton straightening. Left: input skeleton; Right: straightened skeleton.

Skinning In final vertex placement, we need to revert the rotations applied to the input skeleton during straightening. We have decided that the best solution is to use skinning since it can be implemented on GPU and we wanted to move all post-processing on the GPU. Straightened skeleton represents bind pose for skinning purposes and the input skeleton represents reference pose. Now we can calculate rotations, represented as quaternions, required to transform bind pose to reference pose. Traditionally, this would require to find the rotation between two corresponding nodes in respect to their parent. Rotating all child nodes in bind skeleton using the same rotation and propagate the rotation calculation to child nodes. However, since we know precisely how bind pose was constructed, we can exploit this knowledge and avoid the rotation of child nodes. In fact, we do not even need the bind skeleton itself because the positions can be calculated from reference pose. We want to calculate the rotation that would transform a node from its bind pose to its reference pose. We know that the nodes parent is already in reference pose. We also know that bind pose was constructed in such a way that all connection nodes childes are co-linear and preserve the distances between nodes. That means from nodes parent reference pose we can calculate where would the node be in bind pose, if we would apply on it the same transformation matrices as were applied to its parent node. The distance between parent and child nodes remains constant in both poses. And the direction at which the child node would be in bind pose is the same as the direction from its grandparent node to its parent node. Now we only need to store the rotation between calculated child node position in bind pose and its actual position in reference pose with respect to its parent node. The following formula demonstrates the calculation of a quaternion required to transform one node from straightened bind pose to input reference pose:

$$
\begin{aligned}
node &\leftarrow nodeInReferencePose \\
parent &= node.Parent \\
grandParent &= parent.parent \\
distance &= dist(node, parent) \\
direction &= normalize(parent - grandParent) \\
nodeInBind &= parent + distance * direction \\
u &= normalize(nodeInBind - parent) \\
v &= normalize(node - parent) \\
rotation &= QuaternionBetweenVectors(u, v)
\end{aligned}
$$

BNP Generation We generate BNP as in original SQM algorithm. First we generate intersection vertices. Second we triangulate and subdivide these vertices. The newly inserted vertices now should be projected onto the sphere associated with their corresponding branch node. However, a detailed description of this projection was not given in the original SQM article. We have explored various possible projections. In the end we have decided to use a ray-sphere intersection. The sphere is branch nodes corresponding sphere onto which we want to project new vertices. The origin of the ray is the position of each newly inserted vertex. The direction of the ray is mean normal of the faces that are connected with the vertex. This means that for the vertices in the center of each face the normal of the subdivided face is used. For vertices inserted in the middle of each edge the mean normal of faces corresponding to that edge is used. This method does work if the center of the sphere is not in the generated BNP as well as if the generated BNP is coplanar.

BNP Refinement During BNP refinement we always split only representative edges of each LIE. In order to maintain roughly equal distribution of edges in a LIE we are applying a smoothing scheme after each subdivision. The smoothing is very important, because gener-

ated base mesh quality directly depends on the smoothing scheme. Ideally, the length of each edge in a smoothed LIE would be equal. However since smoothing is applied after every subdivision, the smoothing algorithm should be reasonably fast. We propose three smoothing schemes. These smoothing schemes are illustrated in Figure 3, where the polyhedron from Figure 3a is smoothed with various smoothing schemes.

Averaging smoothing calculates new position for each vertex on a LIE by averaging vertices in its one-ring neighbourhood. We start with the last vertex of a LIE, that is the vertex on the last edge of a LIE and move towards the first vertex. We move each vertex, except the first and the last vertices, to the barycentre of its one-ring neighbourhood and project them back onto the sphere corresponding to BNPs node. The resulting smoothed polyhedron is shown in Figure 3b. This approach is iterative and would need several iteration to achieve global optimum, however we have found that one iteration is enough for our needs.

Quaternion smoothing calculates a quaternion representing the rotation from the first vertex of each LIE to its last vertex. From each quaternion we extract its corresponding axis of rotation and angle of rotation. We smooth only points between the first and the last vertex so the calculated axis of rotation and angle are constant. During each smoothing step we first count the number of vertices in a LIE. Then we divide the angle of rotation by that number and form a new quaternion from already calculated axis of rotation and the newly calculated angle. For each vertex in a LIE between first and last we apply the rotation stored in the quaternion and update its position. This method produces LIEs that lie on small circles of their corresponding sphere. The spacing between vertices is regular and thus its very suitable for our needs. The result of quaternion smoothing is shown in Figure 3c.

Laplacian smoothing adapts the algorithm described in [1]. The weights used for smoothing are based on the one-ring area of each vertex. We use one iteration of Laplacian smoothing and then project the new vertices back onto their corresponding sphere. The smoothed mesh is shown in Figure 3d. The result is not as good as either Avaraging or Quaternion smoothing.

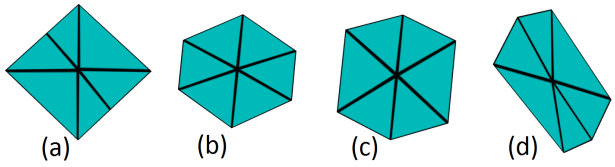


Figure 3: LIE smoothing schemes. (a) original poylehdron; (b) polyhedron after applying averaging smoothing; (c) quaternion smoothing; (d) Laplacian smoothing.

BNP Joining After refinement of BNPs intersection vertices connected via path have the same valence. Now BNPs can be joined by tubes consisting from quadrilaterals only. We loop through each branch node in a depth-first search from skeletons root. We process each BNP in the following manner. We start with the whole BNP Figure 4a. We loop through all intersection vertices corresponding to current BNP. We remove each intersection vertex and its corresponding faces and edges from current BNP. In Figure 4b we can see the removal of third intersection vertex after first and second intersection vertices were joined. After the removal of an intersection vertex we continue joining all nodes on the path that produced the removed intersection vertex. For each node, we generate new vertices and connect them with corresponding vertices from previous node. If the path leads to a branch node we remove the destination branch node corresponding intersection vertex and faces and edges connected to it. The tube generated from connection nodes is then joined with destination intersection vertex former one-ring. This approach is more suitable for our data structure than the approach proposed in SQM. Splitting a quadrilateral face into two faces is equally difficult as creating two new faces. That means the split operation would need more time in our data structure as our approach.

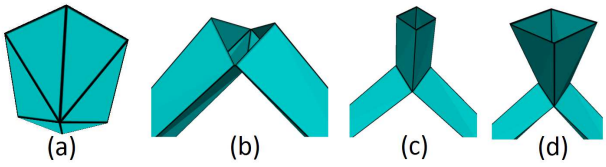


Figure 4: BNP joining process. (a) polyhedron before joining; (b) polyhedron with removed faces corresponding to an intersection vertex; (c) new vertices for connection node before projection; (d) projected vertices of connection node.

Final Vertex Placement We use quaternions calculated during skeleton straightening. For each skeletal node we accumulate the final rotation in a matrix. Matrices are used because they are more suitable for GPU calculations than quaternions. Linear blend skinning, as described in [5], is used to combine skinning matrices corresponding to each vertex on GPU. We apply skinning transformation on GPU in tessellation shaders.

5 Our Base Mesh Improvements

In this Section we propose solutions for several limitations of original SQM algorithm discussed in Section 3. We also describe how we implemented each solution.

1. Root That Is Not a Branch Node If the root of the input skeleton is not a branch node and a branch node is present in the skeleton, we can find it with a depth first search. When we have at least one branch node we can

re-root the tree so that the located branch node would be the root of the tree. This change simplifies the modelling process as user does not need to be aware of the number of neighbours of the root node.

2. Linear Skeletons Linear skeletons, which do not have branch nodes, lack the initial geometry that is generated during BNP generation step. Additional nodes could be inserted into the input skeleton to form at least one branch node, but we have found that it needlessly disturbs the flow of the output mesh. Instead we decided to use a different approach. We introduce an additional input parameter N which specifies how many vertices should be generated, for each node of the linear skeleton. This parameter does not decrease the robustness of our approach, because additional vertices are generated during tessellation and the original number of vertices is negligible.

First step of the algorithm is setting the root to be the head of the input linear skeleton. Next step of the algorithm is straightening of the input linear skeleton. The input skeleton is shown in Figure 5a. Next, N vertices are generated around first connection node, which is a child of the root node. These vertices are distributed regularly around the node by slerping a quaternion, which center of rotation is nodes position, axis of rotation is the direction from connection node to root node and magnitude is $360/N$. Newly generated vertices are then joined with other vertices as in original base mesh algorithm. Leaf nodes form a triangle fan and connection nodes form a tube of quadrilaterals. The joined linear base mesh is shown in Figure 5b. Skinning matrices are used to transform the generated linear skeleton into its input pose Figure 5c.

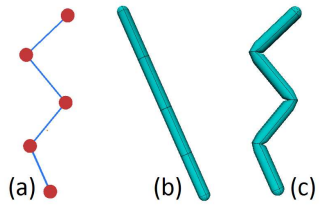


Figure 5: Linear base mesh generation. (a) input linear skeleton; (b) straightened and joined linear skeleton; (c) final linear base mesh.

3. Ellipsoid Nodes An ellipsoid can be defined as a sphere with associated transformation matrix. We take advantage of this representation of ellipsoids. Instead of more complex ray-ellipsoid intersection that would have to be computed at each ellipsoid node, we have decided to represent each ellipsoid node as a sphere and a transformation matrix. First our base mesh algorithm is evaluated as described in Section 3 with spherical nodes. After that we send the transformation matrices corresponding to each ellipsoid node to GPU. The vertices corresponding to each ellipsoid node are transformed directly in vertex shader. Thanks to this, ellipsoid nodes require minimal extra computing resources from CPU. The results can be seen in Figure 6.

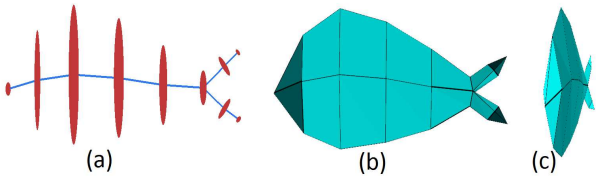


Figure 6: Ellipsoid nodes. (a) skeleton with ellipsoid nodes specified; (b) base mesh generated from skeleton; (c) base mesh from different angle.

4. Capsule Ending A capsule is a hemisphere generated at each leaf node of the input skeleton. Generation of capsules can be approached in two ways. The first is to generate a capsule at each leaf node corresponding to its radius. The second is inserting additional nodes into the input skeleton with decreasing radius that would approximate a capsule. We have implemented the second approach because it fits nicely into our pipeline. Capsules generated this way, can be directly tessellated on the GPU without any additional processing. At each capsule leaf node, we insert additional nodes into the input skeleton, proportional to the radius of the capsule node. The radius of each node is decreased according to the following equation: $newRadius = \sqrt{nodeRadius^2 * (1 - step^2)}$ where $nodeRadius$ is the radius of capsule node and $step$ is a number between $(0 - 1]$, that represents the distance from center of the capsule to its edge. Final tessellated capsule is shown in Figure 7.

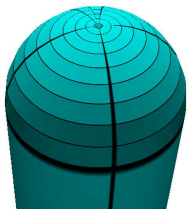
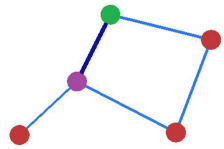


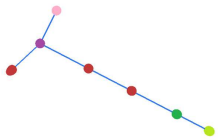
Figure 7: Capsule generated by our algorithm.

5. Cyclic Skeletons Our last improvement is generation of base meshes from cyclic skeletons. The cycle can be located anywhere in the input skeleton. The base algorithm could not be modified to allow generation of cyclic meshes, because during BNP refinement step of the algorithm a cycle could cause an infinite loop. However we can modify the input skeleton in a way that would allow us to generate cyclic skeletons. As the input we have a cyclic skeleton Figure 8a. Cyclic edge is marked with dark blue color and cyclic nodes with dark green (upper node) and dark violet (lower node) colors. First, we split the cycle

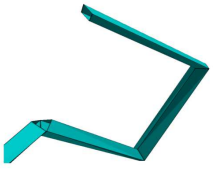
by removing the cyclic edge. To each cyclic node we add an extra child node as shown in Figure 8b. Light green node for dark green cyclic node and light pink node for dark violet cyclic node. These new nodes serve to preserve the skinning matrices that will rotate tubes generated from cyclic nodes to face each other. This can be seen in Figure 8c. Base mesh was generated as described in Section 4 with one exception. We do not generate geometry for light green and light pink nodes. Now the gap between cyclic nodes should be closed. We first project vertices associated to each cyclic node to a plane with origin at $O(0,0,0)$ and normal $n(0,1,0)$. Next, we normalize the vertices so that vertices associated with violet node lie at a circumference with radius 1 and vertices associated with green node lie at circumference with radius 2. The position of projected points is shown in Figure 8e, outer points correspond to dark green node and inner points correspond to dark violet node. Now we execute a Delaunay triangulation on the transformed points. After the triangulation is done, we exclude triangles generated solely between inner or outer vertices. The remaining triangles, Figure 8f represent the faces that should be generated between vertices of cyclic nodes in our generated base mesh in order to close the gap. Final cyclic mesh is shown in Figure 8d.



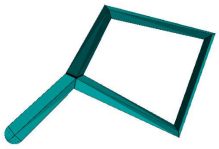
(a) Cyclic skeleton, cyclic edge marked with dark blue color, cyclic nodes with dark green and dark violet



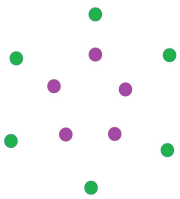
(b) Split cycle with one inserted node for each former cyclic nodes light green for green node and light pink for violet node



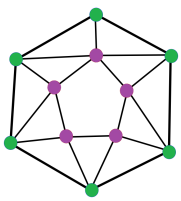
(c) Generated base mesh before the cycle is closed



(d) Generated base mesh after the cycle is closed



(e) Vertices corresponding to green and violet cyclic nodes projected onto the same plane



(f) Faces generated by triangulation that will be used between original unprojected vertices

Figure 8: Cyclic skeleton base mesh generation.

6. Tessellation Tessellation shaders available since OpenGL 4.0 are used to tessellate the generated base mesh. Two connected spherical nodes, a parent and a child, implicitly define a truncated cone between them. The base of the cone has the radius of parent spherical node and the top of the truncated cone has the radius of child spherical node. Each vertex generated during tessellation is projected onto this cone. The projection is done by translating the vertex along its normal until it reaches the surfaces of the cone. A generated base mesh is shown in Figure 9a and after tessellation in Figure 9b. However during this step the generated base mesh gains volume and the newly generated vertices can intersect the tessellated base mesh. This effect can be seen in Figure 9c. To recover from this situation, we detect sharp vertices in the input mesh and apply a radius scaling scheme. Sharp vertices are vertices which faces are forming acute angles. In tessellation shader we have access only to one patch and its vertices. So we compute the sharpness of each vertex by comparing and thresholding the normal of each vertex with the direction of the patch. The smaller the angle between vertex normal and patch direction is, the sharper the vertex is. We apply Bézier curves to modify the radius of the truncated cone. We use Bézier curves that yield values between $[0,1]$. For each tessellated vertex its *distance* from the beginning of the patch is calculated. The distance is equal to tessellation parameter v computed by the GPU. Scaling reduction factor is calculated by sampling a point on Bézier curve at point $t = distance$. The radius of each vertex is then multiplied with calculated factor. The smoothed mesh is shown in Figure 9d. Currently, the scaling bezier curve is constant, but it could be dynamically changed based on the sharpness of the vertices.

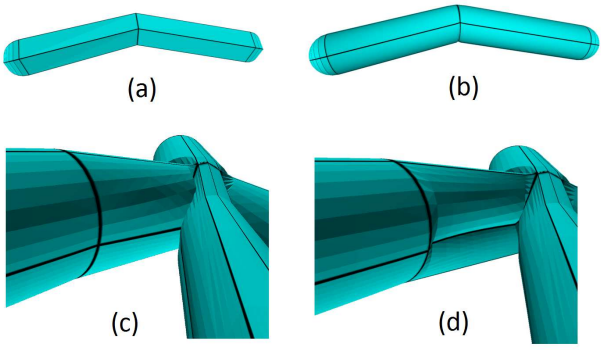


Figure 9: Tessellation. (a) non-tessellated mesh; (b) tessellated mesh with 20 subdivisions; (c) tessellated mesh with self intersection; (d) tessellated mesh with scaling.

Model	Node Distribution				Timing of steps in milliseconds				
	#nodes	#branch	#connection	#leaf	straightening	generation	subdivision	joining	**Total**
worm	23	21	2	0	0	0	0	5	5
dummy	56	2	49	5	0	4	2	15	21
cycle	14	1	12	1	0	4	2	18	24
octopus	131	1	117	13	1	9	5	54	69
dummy 2	140	5	122	13	1	11	6	39	57
goat	150	9	123	18	1	16	10	47	74

Table 1: Table showing statics of base mesh algorithm. From left to right: name of the model, node distribution in skeletal structure, timing of each step of the algorithm measured in milliseconds.

6 Results

The algorithm was implemented in C++ in Visual Studio 2012. Mesh is stored in open source half-edge data structure OpenMesh 2.2. OpenGL 4.3 is used to visualize the algorithm and tessellate the generated base mesh. We have also developed an interactive system, where the user can create, save and load skeletons. Nodes can be edited directly with mouse input, or using node property inspector. New nodes can be inserted into the skeleton with mouse clicks.

Performance of the algorithm is shown in Table 1. The table shows from left to right: the name of measured model, distribution of branch, connection and leaf nodes in the model and time required for each step of the algorithm measured in milliseconds. Time was measured on Intel® Core™ i7-3615QM a four core processor with each core clocked at 2.3 GHz. From the table, we can see that the joining step, during which the tubular structure is generated, took the most time. Therefore, it is a candidate for optimization since other steps of the algorithm took nearly no time to execute. However, even at current speed we can generate base meshes at interactive frame rate.

Our algorithm is also capable of generating base meshes from skeletons on which SQM would fail. For example, in Figure 10a we can see a fish produced in SQM without ellipsoid nodes. The generated base mesh resembles an eel. Our algorithm with ellipsoid nodes, Figure 10b, produces a base mesh that corresponds to a fish. In Figure 10d we can see a cyclic mesh generated by our algorithm. We tried to generate similar mesh in SQM from the same skeleton, Figure 10c. Producing a similar mesh in SQM is not possible as the cycles do not lie in symmetrical region of the input skeleton and SQM would not close them.

7 Conclusion

We have managed to improve all the drawbacks discussed in Section 2. Our algorithm is capable of generating base meshes from linear skeletons, explicitly defined cyclic skeletons, as well as from skeletons with root that is not a branch node. We have moved Final Vertex Placement step of the algorithm on GPU. Lastly, we can set arbitrary ellipsoids at each skeletal node and improve visual quality of generated base mesh in tessellation shaders.

References

[1] Oscar Kin-Chung Au, Chiew-Lan Tai, Hung-Kuo Chu, Daniel Cohen-Or, and Tong-Yee Lee. Skeleton extraction by mesh contraction. *ACM SIGGRAPH 2008 papers*, pages 1–10, 2008.

[2] J. A. Bærentzen, M. K. Misztal, and K. Welnicka. Converting skeletal structures to quad dominant meshes. *Computers & Graphics*, 36(5):555–561, 2012.

[3] Michael Mc Donnell. Skeleton-based and interactive 3d modeling. Master's thesis, Technical University of Denmark, 2012.

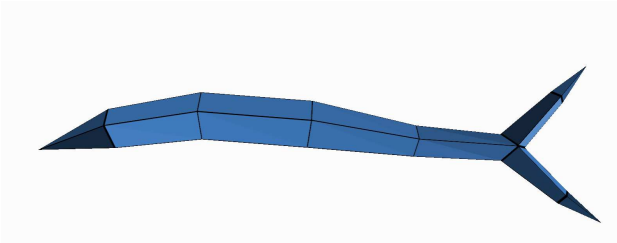
[4] Zhongping Ji, Ligang Liu, and Yigang Wang. B-mesh: A fast modeling system for base meshes of 3d articulated shapes. *ComputGraphForum*, 29(7):2169–77, 2010.

[5] Ladislav Kavan, Steven Collins, Jiri Zara, and Carol O'Sullivan. Skinning with dual quaternions. In *2007 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 39–46. ACM Press, April/May 2007.

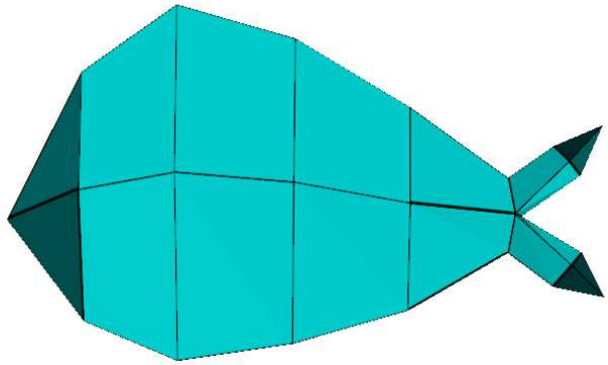
[6] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Modeling with blocks. *Vis Comput*, pages 555–563, 2011.

[7] Vinod Srinivasan, Esan Mandal, and Ergun Akleman. Solidifying wireframes. *Proceedings of the 2004 bridges conference on mathematical connections in art, music, and science*, 2005.

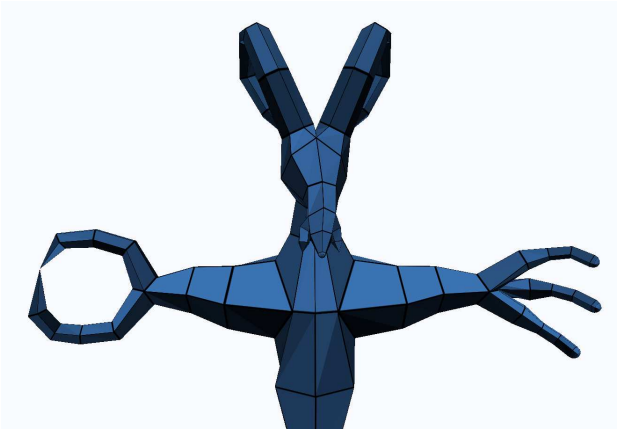
[8] Roger Tam and Wolfgang Heidrich. Shape simplification based on the medial axis transform. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 63–71, Washington, DC, USA, 2003. IEEE Computer Society.



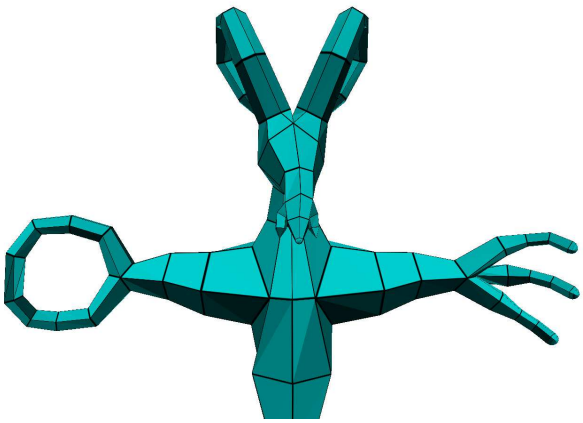
(a) Fish without ellipsoid nodes as would be generated by SQM



(b) Fish with ellipsoid nodes generated by our algorithm



(c) Mesh from cyclic skeleton as would be generated by SQM



(d) Mesh from the same skeleton generated by our algorithm

Figure 10: Comparision of SQM to our implementation.

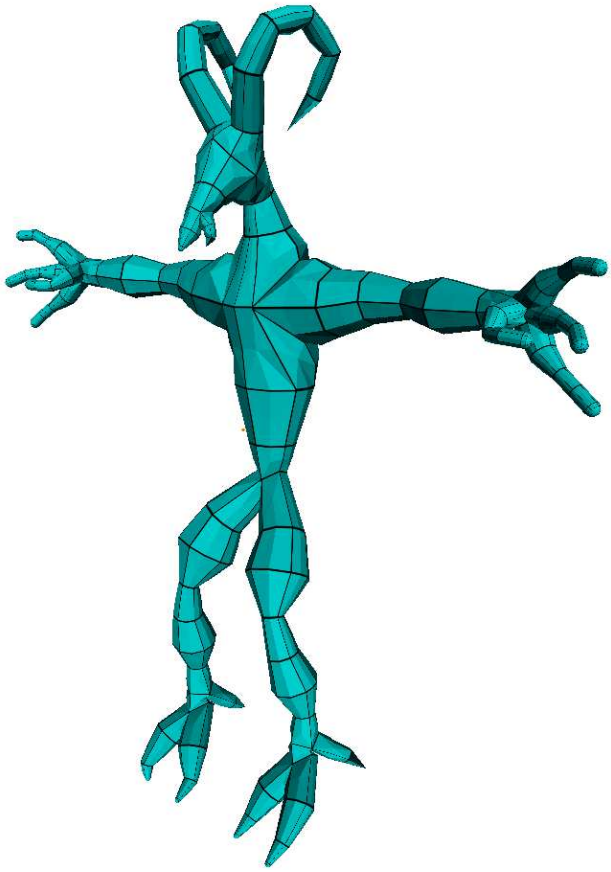


Figure 11: Goat creature generated with our base mesh algorithm.