Skeleton-Based Modeling of Characters

Héctor D. Rodríguez Figueroa and Etienne Morakotkarn

Abstract

Modeling a character in 3D has a different process from drawing a character in 2D. Whereas in 2D one starts with the structure of the character before working on its form, the process is reversed in 3D. In this paper we present a software for skeleton-based modelling of characters. This aims to be a new way of creating 3D characters by starting with a skeleton and adding meshes on each bone. Through the process of mesh unification, these meshes become one character capable of being posed.

1. Introduction

In general, creation of 3D characters follows this general workflow: first, the mesh is created, which represents the character's details; next, the skeleton of the character is created, which represents the inner details of how the character moves and is structured. However, the creation of 2D characters often follows a different workflow: first, the core details and structure of the character are hammered down in an initial sketch; after that, the fine details of the character are drawn on incrementally.

This project aims to experiment with the creation of 3D characters using a workflow more similar to that used to create 2D characters. In this hypothetical workflow, first the user builds a skeleton containing the core details of how the character moves and is structured. Following that, each bone of the skeleton is assigned an individual mesh part. These mesh parts are unified into one whole mesh that is then able to be deformed according to its skeleton. This process results ideally in a different kind of 3D character from the ones made under a typical workflow.

2. Related Work

The user-centered approach of our project takes inspiration from the approach done by Teddy, described in [4]. Igarashi et al. describe a tool for 3D character creation based off of 2D pen strokes. This shows another way 2D art techniques have been brought into the third dimension.

Much work has been done in the past on the subject of mesh unification. Mesh unification can be discussed as a subset of Constructive Solid Geometry (CSG), a method of describing surfaces as Boolean operations between two other surfaces. In our case, we are looking at unions, also known as an "or" operation. We based our implementation of CSG with triangular meshes on [2], which discusses CSG on triangular meshes as a two-step operation: mesh refinement, and triangle selection. Mesh refinement is a process that modifies mesh geometry so every triangle can be described as strictly "exterior" or "interior." Triangle selection depends on the boolean operation in question, and describes which triangles of the refined mesh to keep. A union operation keeps all exterior triangles and discards all interior triangles. This process is described in Section 3.2.1.

[3] describes a similar process for applying CSG operations on triangular meshes, in particular, it describes mesh refinement in better detail. It is not in the current implementation of

our project, however, its relevance to our project is further described in Section 5.

Previous work has also been done on ways to store skeleton data for efficient use in animation and character creation. Seron et al. propose an actor/skeleton tree hierarchy for use especially in scene graphs. Their data structure boasts real-time performance, and adaptability to direct and inverse kinematics as well as motion capture [5]. Our implementation of character skeletons borrows heavily from their data structure.

3. Implementation

Our implementation is written in the C♯ programming language and renders with OpenGL using the OpenTK binding library. The user interface was created using the Windows Forms library from the .NET Framework.

3.1 Data Structures

3.1.1 Mesh Unification

Several data structures were necessary for the mesh unification process. Firstly, it was necessary to have a data structure for the overall mesh data. Meshes are stored as a set of vertices and a set of indexed triangles. Vertices have two attributes: a 3-dimensional vector for its position, and an integer for its associated bone index. An indexed triangle has 3 integers indicating the index of each of its vertices. This makes it possible to render larger meshes without repeating vertex positions several times in memory.

When performing the mesh refinement step (described in Section 3.2.1), some additional supplemental data structures were necessary. To accelerate the identification of triangle intersections, axis-aligned bounding volumes and octree structures are used. An axis-aligned bounding volume is represented as two points in 3-space, a *starting point* and an *ending point*. Each of the starting point's components is less than or equal to the ending point components. This allows for cheap point-in-volume and ray-intersects-volume determination. A suite of unit tests exists for this structure, however, one case was not accounted for and as a result it is not currently completely functional (specifically, intersecting volumes where neither the starting point nor the ending point are inside any of the two).

Octrees split an axis-aligned bounding volume into eight equal-volume axis-aligned sub-volumes. Leaf nodes contain the indices of the triangles that intersect the leaf's bounding box. These are used to quickly discard any triangle u that definitely does not intersect triangle t. This is because if the bounding volumes of u and t do not intersect, then u and t do not intersect either. The functioning of this was verified with verification of the triangle classification algorithm, with progressively larger and larger meshes.

The mesh refinement step also identifies triangle-triangle intersection lines, which are identified as collections of *segment chains*, which describe an ordered sequence of connected line segments (each represented as two points in 3-space). These were tested with a suite of unit tests.

Finally, the mesh refinement step culminates in the triangulation of polygons, which are represented as a linked list of polygon vertices. Polygon vertices differ from regular vertices in that they have no bone binding and that they are classified as either *reflex* or *non-reflex* as described in [1] and later on in Section 3.2.1.

3.1.2 Skeleton Structure

The data structure containing the skeleton is based heavily off of the work of Seron et al., as mentioned previously. However, it is worth noting that this data structure was somewhat simplified to remove elements we felt were unneeded for this project. The basis of the data structure is a tree with an Actor node at its root. This node represents a character as a whole, including their position in 3D space. It also serves as the parent to the root bone of the skeleton.

Meanwhile, each bone is represented as a Node called a Skeleton Node. Each Skeleton Node has one parent and potentially multiple children Skeleton Nodes. The Skeleton Node mainly keeps track of two matrices. First is the offset matrix, which represents its default position in comparison to its parent. Second is the so-called skeleton matrix, which represents how the skeleton has been additionally deformed from its original position and rotation. The skeleton Node additionally keeps track of which Degrees of Freedom are enabled in order to allow for restraining of certain rotations.

The Actor Node, as the parent to the character tree as a whole, also serves a few other roles not previously mentioned. For one, the Actor Node keeps track of all of the Skeletons under it, as well as their transformation Matrices. Both of these are contained in separate lists which are updated very simply through the use of depth-first traversal. These lists minimize the need for tree traversal which allows for more convenient rendering among other things.

3.2 Algorithms

3.2.1 Mesh Refinement

Before meshes can be united, a *mesh refinement* step described by [2] is undergone. This step works to ensure that, given two meshes A and B, every triangle in A can be described as either "outside of B" or "inside of B," and vice versa. [2] describes a third classification, "on the surface of B," but this classification is, for our intents and purposes, redundant.

Refinement of A to B is performed independently on every triangle t in A. To refine triangle t, we first check if t intersects any triangle u in B. t intersects u if any edge in t intersects u, and vice versa. [2] describes a process for determining whether an edge intersects a triangle. An edge e intersects t if: 1.) both vertices of e are either on opposite sides of the plane determined by t, and 2.) any of the vertices of e is inside the angular zone of the tetrahedron whose base is the triangle formed from any edge of t and a vertex of e and whose apex is the other vertex.

A vertex v is inside the angular zone of the tetrahedron τ if v is on the same side of the four triangles that compose τ. In the case that v is coplanar to any of the triangles composing τ, then v is also considered to be inside the angular zone of τ.

If no triangle is found to intersect t, then t is kept intact. Otherwise, we determine the line of intersection that u makes along t and keep track of it. This line of intersection can be determined in a three-step process. First, we determine which of the edges of u intersect the plane determined from t. Next, we find the the intersection points of the intersecting edges of u with the plane determined from t. Finally, the points are clipped using barycentric coordinates along the ray determined from these two intersection points such that they now lie either within or on the edge of t.

Once the intersecting segments are determined, we join segments which share endpoints into the segment chain structure

described in Section 3.1.1 and create polygons out of these chains and the original triangle geometry. To accomplish this, we walk along the edges of t starting from point A, followed by point B, and then point C, before returning to point A. If we come across an endpoint of any segment chain σ that intersects our current path, then we logically split the execution into two "processes." The parent "process" will advance to the other endpoint of σ but otherwise continue its planned route, while the child "process" will start from the found endpoint of σ, continue the planned route, but stop at the other endpoint of σ. This creates a set of polygons which have the same winding order as the original triangle.

The final step of the triangle refinement process is triangulation. Triangulation is accomplished using the ear-clipping method described in [1]. This process starts by identification of the ear vertices in the polygon. A vertex v is an ear vertex if it is not a reflex vertex (i.e. its angle is less than 180°) and if there are no other vertices within the triangle formed from v and its two adjacent vertices. Then, we can triangulate the polygon by iteratively creating a triangle from every ear vertex. As ear vertices get removed, non-ear vertices may become ear vertices, so this identification step must be performed again. This results in a refined triangle where every sub-triangle is either outside or inside B. A simple example of such a result is shown in Figure 1.

3.2.2 Mesh Unification

Once the mesh has been refined, mesh unification is a relatively simple step. First, we let $A' = refine(A, B)$ and $B' = refine(B, A)$. Then, for every triangle t in A', if t is outside of B', then we keep t. Otherwise, we discard t. The same process is then performed for every triangle u in B'. To determine whether t is in B', we perform a point-in-solid test per [2]. This test

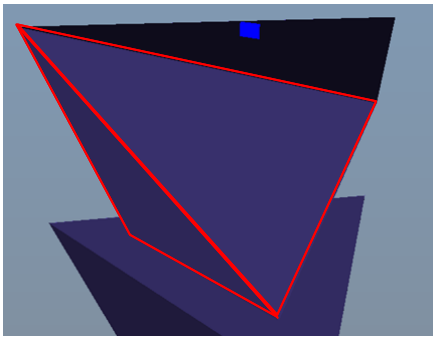

Figure 1 – Result of mesh refinement. The resulting triangles are outlined in red for visibility.

is based on the Jordan Curve Theorem, which states that a closed curve divides its space into two: an interior space and an exterior space. This theorem also applies in 3-space with a closed surface. Thus, we cast a ray from the centroid of t along its normal and count the number of intersections we have with B' along the way. If the number of intersections is odd, then t is inside B'. Otherwise, t is outside of B'. The generated mesh will then be composed of the kept triangles and their vertices.

3.2.3 Vertex Association

In the workflow described in Section 4, meshes are assigned to bones. This indicates which geometry features should be moved along with which bones. When a mesh is associated with a bone, the bone index for each of the vertices in the mesh is set to the index of the bone it is being associated to. When meshes are united, this bone association is preserved.

New vertices may be created during the mesh refinement process. When this happens, several schemes could be used to determine the bone association of the newly-created vertex. The crudely-implemented scheme in the codebase as of this writing inspects the vertices of all triangles that intersect the triangle currently undergoing refinement and selects the most frequent vertex index found. Some other schemes that could have been followed are described in Section 6.

3.2.4 Mesh Deformation

When deforming the mesh in order to reflect the skeleton's current pose, vertices take different model transformations based on their associated bone index. The transformation undergone is as follows. First, we transform the vertex *v* to bone-space. Let O_i and S_i be the offset and skeleton matrices, respectively, for bone *i*. Additionally, let O_{C_i} be the cumulative offset transform matrix for *i*, and let *j* be the parent of *i*. Then, O_{C_i} can be calculated as the multiplication of the bone's offset matrix by its parent's cumulative offset transform matrix:

$$O_{C_i} = O_i \times O_{C_j}$$

The transformation of $\vec{v}$ from model-space to bone-space is the inverse of the cumulative offset transform matrix of its associated bone. Thus, $\vec{v}$ transformed to bone-space $\overrightarrow{v_B}$ is:

$$\overrightarrow{v_B} = {O_{C_i}}^{-1}\,\vec{v}$$

Once *v* is transformed into bone-space, we transform it to its final position $\overrightarrow{v'}$ by transforming it by the cumulative skeleton matrix S_{C_i}:

$$\overrightarrow{v'} = S_{C_i}\,\overrightarrow{v_B}$$

Where the cumulative skeleton matrix S_{C_i} is the multiplication of the bone's parent's cumulative skeleton matrix S_{C_j} by the bone's offset and skeleton matrices:

$$S_{C_i} = S_{C_j} O_i S_i$$

As there are relatively few matrices and these matrices are used several times, a transform matrix T_i for each bone is calculated on the CPU before rendering (as GPUs perform poorly when doing recursive workloads). The bone transform matrix is the total transform matrix that transforms $\vec{v}$ to $\overrightarrow{v'}$:

$$T_i = S_{C_i} {O_{C_i}}^{-1}$$

4. Results and Workflow

Skeleton creation is functional, as is mesh posing. However, mesh unification is currently more broken than not owing to the complexity of the process. Best-case scenario, random triangles in the mesh will be erroneously discarded. Worst-case scenario, the process either hangs indefinitely or crashes. Debugging work in the limited timeframe seems to point at bugs in the octree implementation leading to false positives and negatives. Figure 2 shows a mesh without intersections being posed in the pose editor. The individual body part vertices move as expected with the bones. Figure 3 shows the same skeleton with different body part meshes being posed. As these meshes now

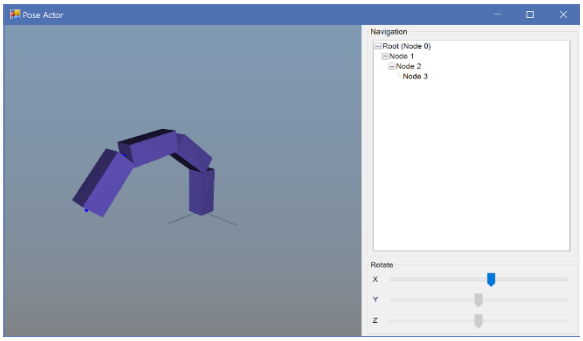

Figure 2 – Posing a snake-like skeleton with non-intersecting meshes.

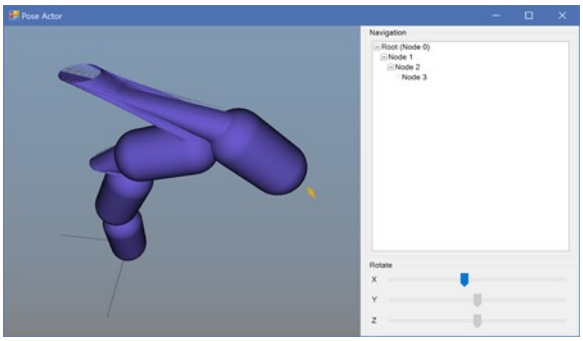

Figure 3 – Posing a snake-like skeleton with intersecting meshes

intersect with each other, the broken state of the mesh refiner is now visible.

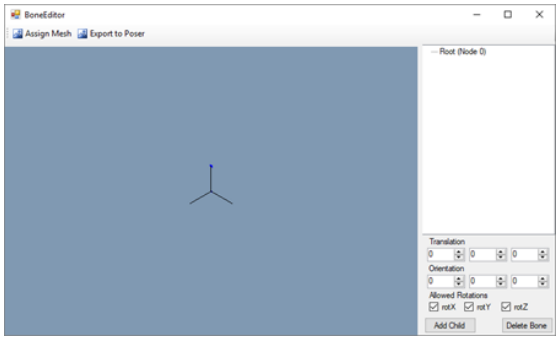

Figure 4 – Bone editor window.

Although the mesh unification stage is still incomplete, the end-user workflow of the tool is in a functional state. The program starts in the Bone Editor window, shown in Figure 4. On the left side of this window is a 3D view with a set of axes representing the X,Y, and Z axes, as well as a single pre-placed bone (represented in the unselected state as a line with blue squares on each end). In this 3D view, as with other windows in this software, the user can rotate the camera by clicking and dragging the mouse. Additionally, the user can zoom the camera in and out by holding Shift while dragging the mouse, and can translate the camera by holding Ctrl.

On the right side of the Bone Editor window is a panel with the hierarchy of the Skeleton Nodes, under which are controls used for placing new bones. The hierarchy can be explored using the mouse; selecting a bone will cause it to be highlighted in orange. Selecting a bone in the hierarchy, and then clicking the Add Bone button in the bottom right, will create a new bone according to the control values, and add it as a child of the currently selected bone. The user can also press the Delete Bone button to delete any bone other than the root bone, which sets all of its children to be children of the deleted bone's parent, while keeping the children in the same global position.

On the top left of this window is a button labeled "Assign Mesh". Clicking this window while a bone is selected will bring up the Mesh Editor window, shown in Figure 5, where the user can position a mesh relative to the bone. Closing this form will assign the mesh to the bone in the Bone Editor as well. Various controls are possible within this form; selecting the folder will allow the user to select an OBJ model from the computer to assign to the bone. On the other hand, selecting the cube icon in the controls toolbar allows the user to move the mesh in relation to the bone (by holding down Shift), and rotate the mesh around the bone (by holding down Alt).

Going back to the Bone Editor window, the last button on the toolbar opens the Poser. The poser is similar to the Bone Editor, but instead of allowing the user to add or delete bones, it allows the user to rotate the bones according to the constraints the user defined in the Bone Editor by using the sliders in the bottom right. It is also at this point that the program attempts the mesh unification process, as described in the previous section.

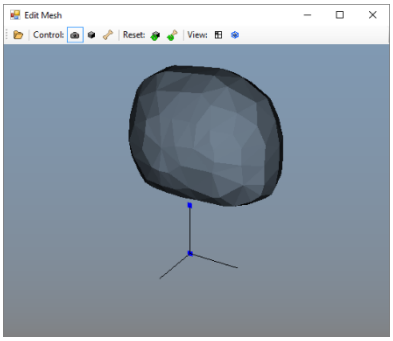

Figure 5 – Editing a bone's associated mesh.

5. Future Work

While the core functionality of this project has been, for the most part, finished, there are quite a few various areas to improve upon. Most notably is the area of mesh unification, which is still rather buggy and prone to mishaps. Late in the project's implementation period, an additional source on CSG application on triangulated meshes was found which has a more thorough explanation on the subdivision of intersecting triangles, specifically, a constrained Delaunay triangulation can be performed on the triangle and its intersecting edges [3]. This has the advantage of handling closed intra-triangle intersection lines, which do not intersect with any of the triangle's edges, and, as a result, are not handled by our triangle refinement method. Thus, rewriting the mesh refinement process to use this alternative mesh refinement method can be included as work that could be done.

Additionally, as can probably be clearly seen in the images, the user interface could definitely use some work; ideally the need to type in values could be done away with in favor of manipulating the bones directly through use of mouse or other input devices. The windows could also probably be combined in some way to make them more convenient, and it could be made easier to modify skeleton offsets after they are placed. This is especially important for the root bone, which currently cannot be changed from its default placement.

As for future work that we considered outside the scope of this project, there are various things that could be done to expand on this project. One idea for future work would be to have the mesh parts be automatically weight-painted when they are added to a skeleton, making for smoother deformations.

Additionally, this software is currently unable to save its models for later access in the program. One area for future work on this program would be to allow it to save in either a unique file type, or more preferably, one of the more widely used file types for 3D characters such as OBJ.

6. Distribution of Work

Héctor worked on the mesh refinement process, which turned out to be much more involved than we all expected. He also worked on mesh association and setting up the project.

Etienne worked on implementing the skeleton data structure and operations related to it.

Between the two of us the project took about 72 man-hours, of which a majority was spent trying to figure out mesh refinement.

7. References

[1] D. Eberly, "Triangulation by Ear Clipping," Geometric Tools, Redmond, WA, Accessed on: April 22, 2019. [Online] Available: https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf

[2] F. R. Feito, C. J. Ogayar, R. J. Segura, M. L. Rivero, "Fast and accurate evaluation of regularized Boolean expressions on triangulated solids," *Computer Aided Design*, vol. 45, no. 3, pp. 705-716. March 2013. Accessed on: April 2, 2019. [Online]. Available https://linkinghub.elsevier.com/retrieve/pii/S0010448512002746.

[3] S. Landier, "Boolean Operations on Arbitrary Polyhedral Meshes," *Procedia Engineering*, vol. 124, pp. 200-212. 2015. Accessed on: April 22, 2019. [Online].

Available https://linkinghub.elsevier.com/retrieve/pii/S1877705815032348.

[4] T. Igarashi, S. Matsuoka, H. Tanaka, "Teddy: A Sketching Interface for 3D Freeform Design," in *SIGGRAPH '99, Proceedings of the 26th annual conference on Computer Graphics and interactive techniques*, Los Angeles, CA. 2002, pp. 409-416. Accessed on: January 17, 2019. [Online]. Available: doi: 10.1145/311535.311602

[5] F. J. Seron, R. Rodriguez, E. Cerezo, A. Pina, "Adding support for high-level skeletal animation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 4, October 2002. Accessed on: April 1, 2019. [Online]. Available http://ieeexplore.ieee.org/document/1044521/