Development of a Humanoid Avatar in Java3D

Mihaela D. Petriu
University of Ottawa,
Ottawa, ON, Canada
mpetriu@discover.uottawa.ca

Nicolas D. Georganas
University of Ottawa,
Ottawa, ON, Canada
georganas@discover.uottawa.ca

Thom E. Whalen
Research Centre Canada,
Ottawa, ON, Canada
Thom.Whalen@crc.ca

Abstract

The paper discusses implementation details for the construction and animation of a human avatar taking advantage of the built-in features of Java3D.

1. Introduction

There is an ongoing interest for realistic anthropomorphic models, called *avatars*, for a multitude of virtual reality applications such as: multimedia communications, computer graphics in entertainment, experiments in natural language interactions, interpersonal collaboration, and interfaces for interactive virtual environments, [1] and [2].

Due to its powerful graphics construction tools, VRML proved to be a convenient language for the model development of anthropomorphic avatars in virtual reality environments, [3] and [4]. However, it appears to be less powerful when it comes to avatar animation.

More recently, Java3D, sitting between OpenGL and VRML in the computer graphics capability spectrum, appears to offer more advanced methods for the creation and animation of three-dimensional geometric shapes. It has more built-in graphics manipulation and user interface methods than OpenGL, but less graphics construction methods than VRML. Although, unlike VRML, Java3D can animate the virtual objects it creates.

This paper discusses implementation details for the construction and animation of a human avatar taking advantage of the built-in features of Java3D, [5] and [6].

2. Avatar design

The design of the humanoid avatar in Java3D is based on the *H-Anim* standard [7]. Of particular interest in the *H-Anim* definition are the relative locations of the joints and body segments shown in Figure 1 and the hierarchical organizations of the body parts shown in Figure 2. Some features from the *H-Anim* representation such as the eyebrow and eye features and the clavicle and scapula joints, were deemed unnecessary for the scope of this project and dropped from the Java3D avatar model.

The built-in Java3D *Triangulator* method was used to make most of the avatar's anatomy. It works by taking an array of vertices describing an outer polygonal contour and an optional inner contour and constructing the polygon out of triangles.

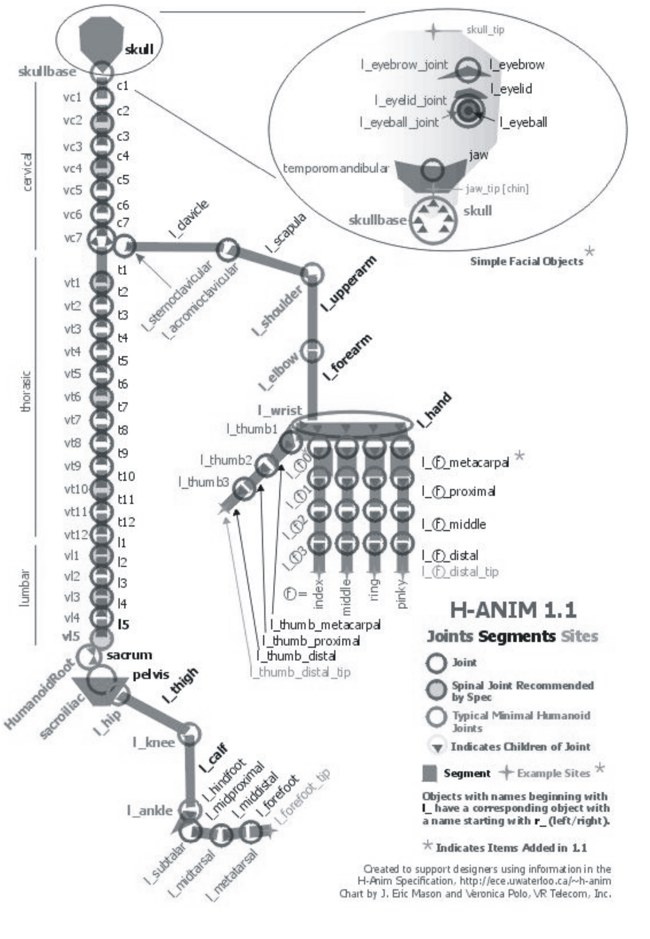

Figure 1. H-anim structure from [7]

The *Triangulator* is powerful, but idiosyncratic in that when given vertices describing a surface curving in three-dimensions, it will draw triangles between vertices that were not meant to be joined simply because they are closer together than the vertices that were meant to be joined, [5] and [6].

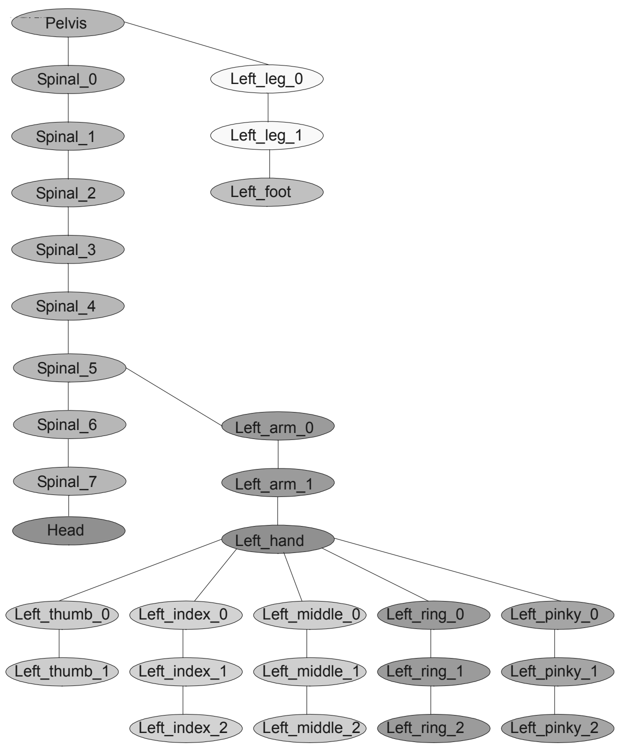

Figure 2. Java3D avatar structure

2.1. The face and hands

The face, Figure 3, and the hands, Figure 4, were designed in greatest detail – every single triangle was planned. Every vertex of one side of the face and one hand, excluding fingers, was determined using Corel Draw 11. Taking advantage of the symmetry of the human body, values of the vertices of the other side of the face and the other hand were defined as the reflections about the y-axis of those original vertices.

Once defined, the vertices, or sometimes whole polygonal contours were passed to the *Triangulator* allowing it to automatically generate the triangles making up the polygon. Other times, groups of three were passed so only one specific triangle was constructed. It must be noted that to make the y-axis reflections of the original side of the face and original hand, the reflection vertices had to be placed in their array in the opposite order of their original counterparts'. This is because Java3D makes a surface visible only if its vertices were defined counter-clockwise in relation to the viewer. It's the test used by Java3D to determine whether a surface is "visible" or not, [6].

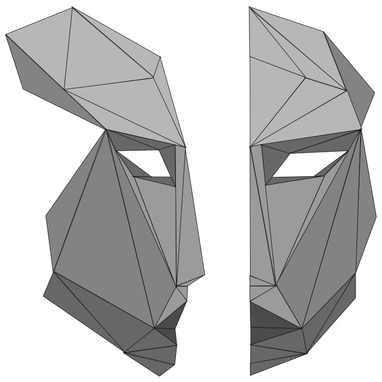

Figure 3. The plan for the face from the profile and front

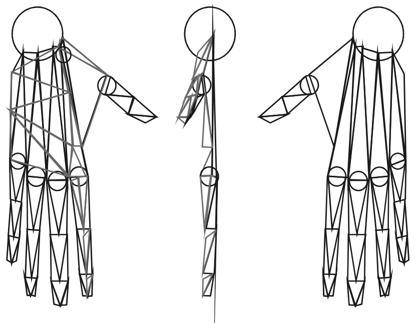

Figure 4. The original plan for the left hand viewed from the front, side and back

2.2. The fingers and arms

The fingers and arms were each constructed not by triangles as originally planned but by attaching the end of a *Cylinder* primitive that would be the segment to a *Sphere* primitive that would be the joint. Using this method, only the radius and the length of the segments had to be determined. However, since *Cylinder* primitives are constructed by Java3D with their centers at origin and their height along the y-axis, they had to be repositioned in each case so that one end was attached to the *Sphere-joint* and that they rested at the proper angle. It is important to note that the rotation must be applied before the translation. This is because rotation is always around origin and moving the object away from the center of rotation will yield unexpected results, [5] and [6].

For each segment, a new *TransformGroup* was created for the *Cylinder* and made the child of the *TransformGroup* of the *Sphere*. The *Cylinder's TransformGroup* then had its coordinate system rotated so that its y-axis now stuck out from the *Sphere* at the desired angle. A translation down the y-axis by half its own length was then applied to get the *Cylinder* to attach to the *Sphere* at one end. The next *joint-Sphere's TransformGroup* was made the child of that *Cylinder's TransformGroup* and translated down along the y-axis half the length of the *Cylinder* as shown in Figure 5.

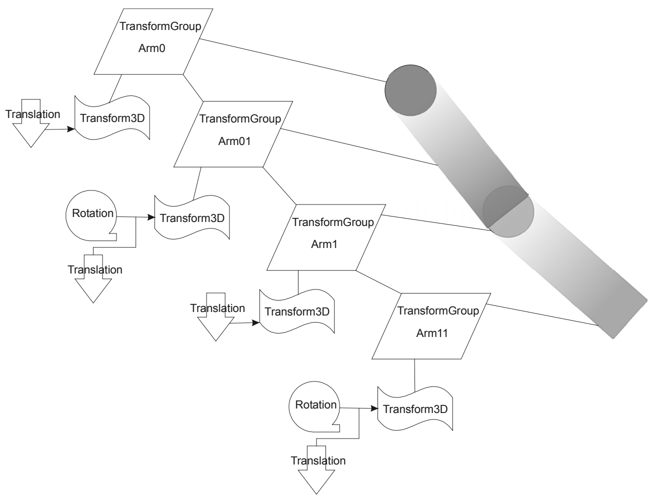

Figure 5. The hierarchy of the geometric, control and managing nodes of the arm

Once more the symmetry of the body was taken advantage of as only the position of one shoulder joint and one set of knuckle joints had to be determined. The position of their counterparts on the other side was simply the reflection along the y-axis of the original position points. Similarly with the angles of the *Cylinders*: the angles of their other side counterparts were simply the negative of the original angles as shown in Figure 6.

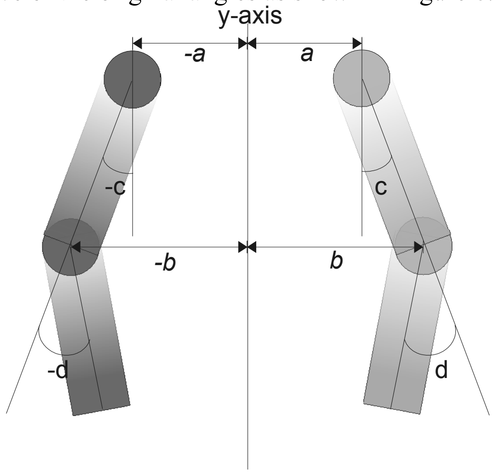

Figure 6. Diagram shows the reflection of the arm about the y-axis and the required negation of joint position (a and b) and segment rotation (c and d)

2.3. The torso and abdomen

The torso had to be able to bend, so it was broken down into eight segments. The segments were designed to overlap slightly so that when they were rotated to produce a torso flex there would be no gaps.

The torso was constructed segment by segment from the waist up to the top of the neck. Each segment was a child of a *TransformGroup* that was itself the child of another *TransformGroup* that was the parent of the segment below. The abdomen was constructed in a similar way, but, as it would not have to bend, all its segments were the children of the same *TransformGroup* and no overlap was required.

In designing the torso and abdomen segments, it became clear that they could be constructed from flattened octagonal tubes. As the cross-sections of these tubes would be symmetric about the z-axis and all centered at their respective origins, their vertices would all have a repeating elements and a repeating pattern. This introduced one level of abstraction: a few variables were used in all the vertices of one segment and in half of the vertices of the next. Thus the code was made more readable and modification of the tubes more efficient as they could be reshaped by only changing one variable instead of a number of vertices, as shown in Figure 7.

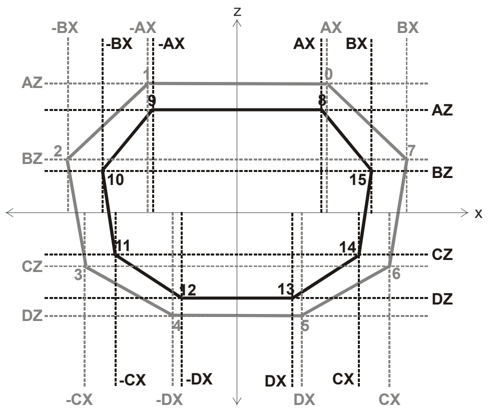

Figure 7. View of a torso segment from top shows the bottom (grey) and top (black) contours and the parameters for the x- and z-vertices' coordinates (the y-values are the same for all the vertices of one contour)

The *createTube()* method which uses the *Triangulator* was developed for tube construction. It takes the top and bottom contours and constructs each rectangular facet individually so as to avoid the *Triangulator's* incorrect vertex joining tendency. The *createTube()* returns an array of facets that make up the tube as shown in Figure 8.

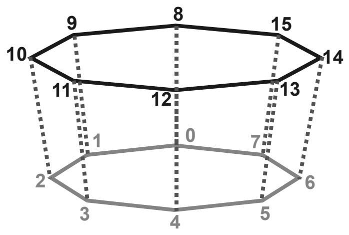

Figure 8. Final product of the *createTube()* method

2.4. The legs and feet

The legs were initially constructed using *Cylinder* primitives like the arms and fingers, but this did not yield the desired tapered look, so the *createTube()* method was used instead. The tubes' ends where designed to fit around the *Sphere-joints* which became successively smaller from hip to knee to ankle. Thus the tampered form of the legs was achieved. The legs were then finished off with feet made from tubes that taper at the ankle and broadened at the soles.

Unlike *Cylinders*, the tubes required the vertices of their upper and lower contours to be determined. However, since these contours were fitted around *Spheres*, they formed a perfect circle. Therefore only one quarter of the vertices had to be determined by measurement, and the rest were then determined through symmetry about the x- and z-axes. The abstraction technique developed in the torso building stage was used once more with even greater efficiency.

For one leg, the measured values needed were the lengths of the upper and lower leg. The feet, however, weren't symmetric along any axis, Figure 9, so the only value that could be parameterized was the height of the feet.

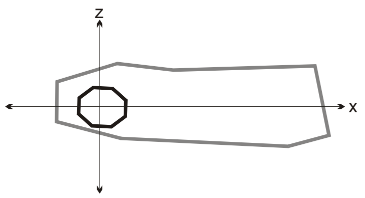

Figure 9. Top view of right foot. The top contour (black) is where the foot attaches to the ankle joint and the bottom (gray) contour is the sole

Once the point defining the position in space of the hip joint was defined, the hip joint position of the other side was determined using the reflection about the y-axis method. Fortunately, unlike with the face and hands, the vertices of the tubes did not have to be put in new arrays in reverse order to be displayed on the other side.

The foot vertices did have to be reflected about the y-axis due to their asymmetric formation, so a new array was required for the foot on the other side. Once vertices have been reflected they have to have an order opposite to that of their original counterparts within their array. Fortunately, because of the way the tube was constructed, this only meant that the top and bottom contours of the tube had to be swapped within their array.

2.5. The back of the head

The back of the head was the last structure to be constructed. The tube edges were arranged in concentric octagonal ellipses and where attached to the sides of the face by a 16-sided tube. The back-most (smallest) 16-edge contour was then converted into an octagonal contour by taking only its even vertices to make an octagonal tube with the second smallest contour, Figure 10. The rest of the head was then made of octagonal tubes with the back-most tube ending with an 8-vertex contour that was simply the same vertex repeated eight times (dot). All the tube segments of the head were made the children of the same *TransformGroup* since there was no need for articulation within the cranium. Once more symmetry was taken advantage of, and abstraction was applied in the form of variables that were used repeatedly in contour definitions.

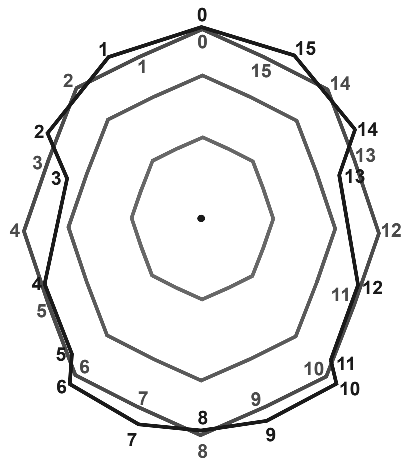

Figure 10. Front view of the contours for the back of the head. The outermost black contour attaches to the sides of the face and forms a tube with the second outermost gray contour

The only problem that arose during this final construction phase was that the face that had been constructed first turned out to be proportionally too large for the rest of the body. This problem was overcome by applying a scaling factor along with a translation to the face's *TransformGroup*.

3. Animation design

Java3D uses a class of objects called *Interpolator* for rotation, translation and/or scaling motions so called because they interpolating linearly from one given point to another. Once defined, the *Interpolator* object is attached as a child-node to the *TransformGroup* it will apply its animation to. The key concept of animation and the reason the avatar was constructed with a definite hierarchical structure is that the animation is applied to not only to its parent *TransformGroup* but also to all of this parent's children and its children's children. For the avatar, only translation and rotation *Interpolations* are of interest, [5].

The *Interpolator* type defines whether it will cause only one kind of motion or a combination. It'll take the end points for each type of motion:

· *rotation end points* – minimum and maximum angles of rotation;

· *translation end points* – minimum and maximum points in space of translation.

The *Interpolator* can be defined as being *increasing* (goes from minimum to maximum), *decreasing* (goes from maximum to minimum) or *both* (goes from minimum to maximum and back to minimum), [5].

Each *Interpolator* also takes an *Alpha object*, which defines the following parameters for its *Interpolator*:

· *trigger time* – the time allowed to elapse from the start of the Java application run to the time the *Interpolator* becomes active

· *delay* – the wait-time between *Interpolator* activation and the actual start of the animation

· *increasing ramp* – the speed at which the *Interpolator* starts and stops its linear interpolation from minimum to maximum

· *increasing* – the rate of the linear interpolation from minimum to maximum

· *at one* – the pause duration at the maximum

· *decreasing ramp* – the speed at which the *Interpolator* starts and ends its linear interpolation from maximum to minimum

· *decreasing* – the rate of the linear interpolation from maximum to minimum

· *at zero* – the pause length at the minimum

· *loop* – the number of times to repeat the motion.

The actual values of the *Alpha object* parameters can be defined by the programmer and by extension the user if the input mechanism is coded within the application, [5].

Ultimately the avatar requires *Interpolators* only for rotation and for translation: *RotationInterpolator* and, respectively, *PositionInterpolator*. These will need to be attached to the *TransformGroups* containing explicit joints such as the shoulders and the *TransformGroups* acting as implicit joints such as those of the torso segments. Only rotations will be required for the parts of the avatar while rotations and translations will be needed to move the whole avatar through the scene, [6].

3.1. Walking Animation

So far only *stationary walking* (i.e. walking on the spot, without forward or backward motion) animation has actually been implemented. In this animation the legs and arms swing at the same rate while the knees and the ankles flex.

A *createRotation()* method was written to achieve the walking motion. The method takes these parameters:

· *parent TransformGroup* of the *TransformGroup* to be animated

· *minimum* angle of rotation (in degrees)

· *maximum* angle of rotation (in degrees)

· *trigger* time

· *duration* of one increase-decrease

· *pause* duration at the end points

This *createRotation()* sets an *Alpha object* with internal default values and the given values. It then passes the *Alpha object* to the *RotationInterpolator* that is also given the minimum and maximum angles. The *RotationInterpolator* is made the child of a *TransformGroup* and that *TransformGroup* is made the child of the given *TransformGroup*.

The values of the *createRotation()* parameters for each part of the right leg is shown in Figure 11. Note that the *trigger*, *duration* and *pause* values are the same for all animation behaviors, and only the *minimum* and *maximum* angles are different. To achieve the reflection of this animation on the left leg, the same behaviors were applied with the *minimum* and *maximum* angles swapped. For the arms' swing, the behavior of the left hip joint was applied to the right shoulder joint and vice versa.

4. Conclusion

The developed Java3D techniques like the reflection about the axes, reversal of vertex arrays and the *Triangulator* method to make tubes could be further refined. The following are the possible future extensions:

· The use of *createTube()* to construct the face, hands, finger and arms. This would be more efficient and make for cleaner code. It would also allow for a more natural looking arms and fingers.

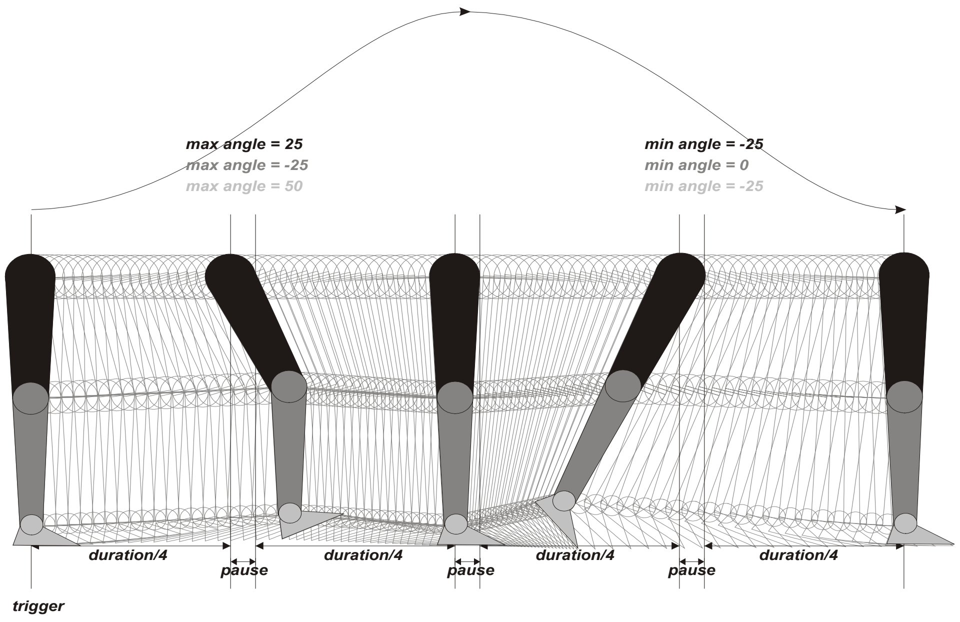

Figure 11. Animation for the upper leg, lower leg and foot

· The development of more *Interpolator*-based animation methods to achieve a broader range of motion. This would require applying an Interpolator to every joint and the definition of more animation variables to increase abstraction.

· The development of a user-interface to allow for user-control of avatar posing and animation. The ultimate goal is to have an interface that would actually allow the user to click and drag a piece of the avatar to pose it instead of just inputting numerical values into a field box.

5. Acknowledgements

This work was funded in part by the Natural Sciences and Engineering Research Council of Canada, the National Capital Institute of Telecommunications, and the Communications Research Centre Canada.

The first author thanks Dr. Xiao Li Yang for her encouragement and valuable discussions during the implementation of this project.

6. References

[1] N.M. Thalmann and D. Thalmann, *Synthetic Actors in Computer Generated 3D Films*, Springer-Verlag, Berlin, Germany, 1990.

[2] H.J.W. Spoelder, E.M. Petriu, T. Whalen, D.C. Petriu, M. Cordea, “Knowledge-Based Animation of Articulated Anthropomorphic Models for Virtual Reality Applications,” *Proc. IMTC/99, IEEE Instrum. Meas. Technol. Conf.*, pp. 690-695, Venice, Italy, May 1999.

[3] D.C. Petriu, X.L. Yang, and T.E. Whalen, "Behavior-Based Script Language for Anthropomorphic Avatar Animation in Virtual Environments," *VIMS'02, IEEE Intl. Symposium on Virtual and Intelligent Measurement Systems*, Anchorage, AK, USA, May 2002.

[4] X.L. Yang, D.C. Petriu, T.E. Whalen, and E.M. Petriu, “Script Language for Avatar Animation in 3D Virtual Environments,” *Proc. VECIMS'03, IEEE Intl. Symposium on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pp. 101-106, Lugano, Switzerland, July 2003.

[5] S. Daniel, *Java 3D Programming*, Manning Publications Co., 2002.

[6] Sun Microsystems, *Java 3D API Documentation*, http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_3_API/j3dapi/index.html (last visited April 2003).

[7] WEB3D Consortium, *Humanoid Animation Working Group of the WEB3D Consortium*, http://www.h-anim.org/ (last visited April 2003).