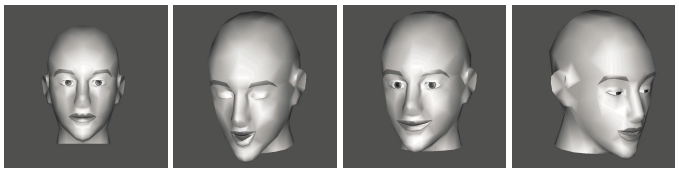
Chapter 7

Applications in animation



Computer animation has evolved to a standard technique in Computer Graphics. In the last decades, a number of different animation techniques have been developed. Starting from the standard frame-by-frame animation introduced by Disney and others, key-frame animation has established itself as the standard technique for describing time-dependent scenes in Computer Animation. Instead of describing every single frame only a sequence of principal frames - so called key-frames - is defined and additional frames are generated by interpolating between two consecutive key- frames using in-betweening. Elaborated techniques have been developed to allow for the automated generation of physically-based behavior, namely kinematics and inverse kinematics, and today animation systems are more and more coupled with simulation engines to facilitate the production of complex and realistic animations. However, key-frames and the concept of describing object states at specific times remain the fundamental philosophy for representing animations and time-dependent aspects in virtual scenes.

Despite its widespread use, the concept of merging object and geometry descriptions has a fundamental problem. While it is easy to specify, design, or output an animation in terms of key-frames, it is difficult to manage or change the time-behavior since all meta-information is lost. In particular, the following problems connected to geometric key-frame animations can be stated:

- Redundancy: A complete object description has to be recorded for each key-frame, even if parts of the object do not change at all. Additionally, repetitive patterns result in repeating the geometry description. Consequently, animation sequences are usually very large and hard to apply in streaming appli-

cations. The compression of such sequences is a problem which is under intense investigation of the Computer Graphics research community today.

- Modification: Exchanging an animated object in a scene while reusing the once specified animation at the same time is an involved task, even though most of the necessary information should be available. In general, after introducing the new model, it has to be animated again by hand. Similarly, it is almost impossible to make use of a once defined animation and to extend or exchange the object's behavior. The aspect of reuse has been addressed for specific object domains. One approach is standardized parameterization of an object and its possible behaviors, thus allowing for the exchange of both, geometry and animation (e.g. Humanoid Animation [Web3D Consortium 1999a] or Facial Animation in MPEG-4 [Ostermann 1998]). Similar but more general is the idea of animation elements [Dörner et al. 1997], general object hierarchies with a defined interface. However, both approaches do not provide a general solution to this problem.

- Level of detail: Another problem of high impact is the reduction of scene complexity in interactive applications. LOD concepts such as progressive meshes [Hoppe 1996] allow static objects to be fitted to the display requirements. Recent techniques try to provide a view-dependent level of detail on static object based on mesh simplification techniques or sometimes by exploiting progressive transmission and decompression schemes. However, if the objects are animated these standard techniques may fail or not be applicable at all. Standard LOD hierarchies could not be applied for animation since this would result in even a much higher redundancy since specific LOD hierarchies would have to be provided for all key-frames. Key-frame interpolation also would become more difficult since an interpolation between objects of different LOD would have to be supported.

  Still, geometric simplification exploits just spatial coherence, while animations exhibit additional temporal coherence. Surprisingly, the application of a LOD concept for animated geometry and animation itself has not been discussed in the literature to the best of our knowledge. For the same reasons small, static geometry features are omitted in standard LOD techniques, small temporal features should be a target of LOD approaches also. Here, an interesting potential for reducing scene complexity is hidden.

Key-frame animations and related concepts inhibit a compound of geometry and animation in their scene description. For all the problems stated above, this composition represents the main obstacle. It makes an easy exchange of either geometry or animation in a scene description impossible and makes the application of LOD concepts and abstraction difficult.

Here, we address this problem and present an alternative representation of animation sequences based on principal animation components, thus decoupling the animation from the underlying geometry [Alexa & Müller 2000]. The idea is find

or use a set of basis shapes and represent each (key) frame as linear combination of the basis objects. In the spirit of this work the basis objects form a space and the animation is a curve through this space.

We present two ways of representing animations in this way. First, we start from a suitable basis of shapes. The basis shapes and their blends are used to author the animation and the animation is then naturally represented in terms of the basis. Second, starting from a key frame animation we try to find a suitable basis consisting of only a few shapes.

7.1 Building animations using base shapes

We demonstrate the idea of building animations from an existing set of base shapes at the example of facial animations [Müller et al. 2000; Alexa et al. 2001].

Models for Facial Animation have been of much interest in Computer Graphics in the last years. One of the first examples was presented by Parke [1979], who used a selected number of key expression poses. Specific expression poses could be generated by interpolating between these key expression poses. Interpolating between these expression poses would then create facial animations. However, this approach has been criticized to allow only a generation of a limited range of facial expressions and to request a significant amount of modeling time to specify the key expression poses [Parke 1982]. The need for more realism in facial animation lead to the development of Facial Animation techniques based on physiological models. Waters [1987] introduced a model for facial animation that incorporates skin and virtual muscles corresponding to the ones in a human face. Activation of these muscles results in different facial expressions. For the control of these muscles FACS, a classification scheme for facial expressions [Ekman & Friesen 1978], was applied. Magnenat-Thalmann et. al. developed a pseudomuscle-based model in which the parameters control abstract muscle actions (AMA, [Magnenat-Thalmann et al. 1988]). While the approach is similar to the one of Water, FACS actions are exchanged here for more complex actions. While the application of muscle-tissue based facial expressions may result in very realistic images, animations are still not easy to control. Moreover, the corresponding models are much to complex and not flexible enough for an application in the context of real-time animation.

Facial animation techniques have been suggested for bandwidth compression in tele-communications for video images of a human speaker. Here the idea is to transfer a geometric representation of the speaker's head and face and their movements rather than a complete image of the speaker every second [Parke 1982]. However, until now prototype systems in this area succeed in making use of the head and shoulder's geometry. The description of speakers in terms of geometric models and facial expressions has gained much interest in the context of MPEG-4 [ISO JTC/WG11 1997]. MPEG-4 allows the definition of data streams with 2d and 3d objects. Moreover, MPEG-4 specifies a set of face animation parameters [Ostermann 1998]. Each one of these parameters corresponds to a particular fa-

cial action deforming a face model in its neutral state. The underlying approach is again based on a physiological model. While the integration of facial animation control techniques in this streaming standard gives a much higher degree of flexibility, solutions for the modeling of animations and specific expressions are not presented.

7.1.1 Representing facial animations

The main use of facial animations in our context is communication. Basically, we are not interested in being able to represent every possible state of a human face. But, we might want to represent facial expressions that are not realistic in order to generate cartoon-like characters. For these two reasons, most of the documented parameter sets used to describe facial expressions are not adequate: First, they usually exhibit a very high degree of granularity and detail, more than might be necessary for the sole purpose of communication. Second, they are derived from the human face and have problems to represent unrealistic facial expressions.

When designing conversational user interfaces, the optimal origin for creating facial expressions is a neutral, emotionless representation of a face, to which certain features can be added, such as:

- Movements of the mouth representing phonemes (so called visemes)
- mimic (for example blinking, raising eyebrow, smiling, grumbling)
- overlaid moods (i.e. friendly, sad)

The systems should add these features non-exclusively, i.e. phonemes, mimics and moods can be combined and result in an addition of features. Also, the number of different features should be as small as possible while still satisfying the needs of human communication. Obviously, the quantity of each feature has to be controllable. We will describe the single features we have used for our prototype system in detail.

We will use the quantities of the above mentioned features (visemes, mimic, moods) as the representation for facial expressions. In the following we will explain our idea of connecting this representation to geometries.

Our approach somewhat resembles the idea of Parke [1979]. We use not only one geometry for the neutral state of the face, but an additional geometry for each feature. But, instead of just blending between these geometries as in [Parke 1979] we define a vector space of geometries. This vector space has the neutral face as zero element and the differences of the features' geometries to the neutral face as base vectors. This alleviates one key problem of the blending technique: Features may be added independently to the neutral face. For example, the designer can combine speaking an "A" with a smile, where the blending technique would allow only to trade off between speaking and smiling.

Also, the generation of the geometries can be eased with modern techniques. We can produce several laser scans of a human model and use the morphing techniques composed from the methods discussed in Chapters 2, 3, and 4. to produce a consistent geometry. But even if the set has to be modeled by hand, several advantages warrant the task.

Note that the movements of vertices for each feature are linear, which is obviously not correct for every facial expression. However, in our experiments this simplification has proven to be sufficient. It seems that linear approximations are not too bad, because of the small distances and speed with which vertices move. If necessary one could model movements of vertices in a more elaborated way and use eigenvector analysis to decompose the movement into linear parts.

Distinct facial features

In our system we like to allow a designer to start forming a neutral face and to add several expressions. The main goal of our work is to communicate in an easy readable and easy understandable way. Facial expressions include a huge potential in terms of conciseness and a wide spectrum in terms of communication statements. The given principles of different communication channels such as speech and mimics contribute to an easy understandable communication through redundancy (e.g. synchronous appearance of speech and mimics) and thus prevent misinterpretation. We need a collection of features which allows to conveniently and intuitively design communication by lifelike facial animations. We do not strive for completeness in the sense of being able to represent the whole range of possible facial expressions. But, the variety of facial expressions resulting from the defined features must be diverse enough to be accepted by a human observer.

Obviously, such a set has to be found by analyzing special communication requirements, such as:

- the syntactical, the semantical as well as the pragmatical context of facial expressions (e.g. different meanings and interpretations of facial expressions according to several cultural agreements)
- special communication requests and special facial expression codes of the intended audience (i.e. target group).

After this analysis step, a base of elementary prototypes of facial expressions has to be defined. This base can be refined in further steps by repeated experimentation.

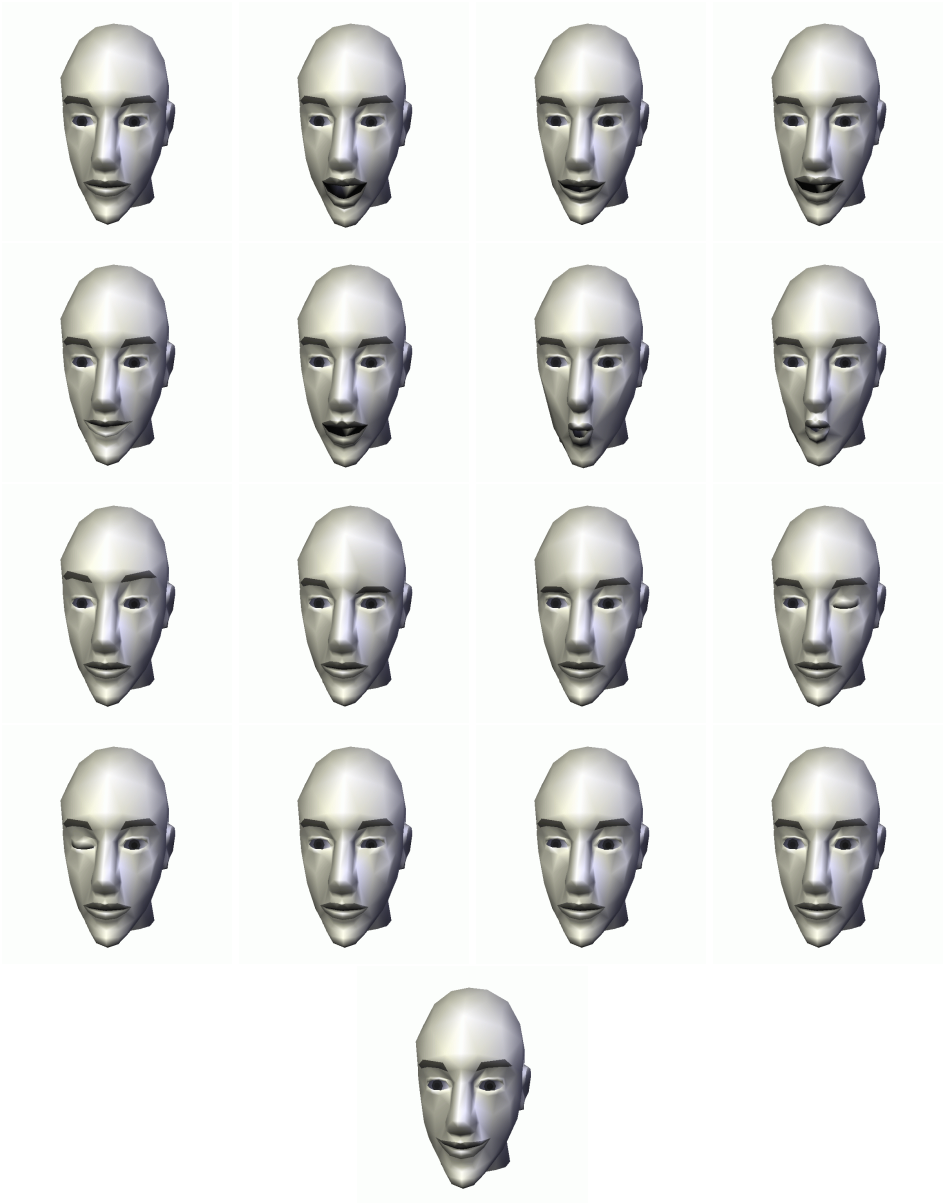


Figure 7.1: Some basic facial expressions

Compact representation of the Animation

A complete representation of an animation consists of the geometry and the change of this geometry over time. In our scheme such a complete representation is given by

- the vertex edge-topology of the geometry
- vertex attributes of the neutral face.
- vertex attributes of the feature facial expressions or the differences to the neutral face
- key-frames given by a feature vector together with a time stamp.

Representing vertex-edge topology in a compact way is a current research topic. We are aware of schemes that guarantee less than 2 bit per vertex, also enabling a compact way of representing vertex positions. In general, coding the vertex attributes is linear in the number of vertices. For representing the features we can exploit the fact that they differ only a small amount from the neutral face. Thus, all values are small and we can use a small number of quantization levels resulting in compact representations. Anyway, the cost for coding the geometries is independent from the actual animation and might be used for a number of different animations.

The key-frames consist of a time stamp (real valued number) and the representation vector (n real valued numbers). Practice shows that a single precision floating point number (2 byte) is sufficient for the time stamp. For the components of the representation vector not more than 6-8 bits are needed. This sums up to less than 20 byte per key-frame. Even if one likes to specify 25 key-frames per second, the stream is represented with less than 4kBaud, a transfer rate each modem can easily provide. Note that a rate of 25 frames per second eliminates all visual problems originating from interpolation issues.

An important point is that an animation's representation is independent from geometry but based on quantities of facial expressions. Of course, the animation makes sense only with a specified geometry, but, this geometry is exchangeable. In the following two sections we will show how to exploit this idea in two ways. We will show the effects of

- changing the representation of an animation (independent of the geometries used)
- using different geometries for the facial expressions (independent of the actual animation sequence).

7.1.2 Altering and combining animations

In this section we show why the representation of facial expressions in terms of quantities of different facial expressions is useful. For this section it is convenient

to represent the animation as a vector over time rather than defining key frames. Simply speaking, the animation is defined by the quantities of saying-an-a over time, smiling over time, and so on. Formally, an animation sequence is represented by $\mathbf{r}(t) = (r_1(t), \ldots, r_{n-1}(t))$. Note that this is only a conceptual difference and we get back to a key-frame animation by simply discretizing $\mathbf{r}(t)$. In the following, we like to exploit the idea of filtering $\mathbf{r}(t)$ and combining different sequences $\mathbf{r}(t)$ and $\mathbf{s}(t)$.

Filtering animations

Given an animation sequence $\mathbf{r}(t)$. Each of the components describes the influence of one of the facial base expression over time. First, lets look at the effect of operations on only one component at the example of smiling: By adding a constant to the component representing smiling or multiplying it with a factor greater one we make the face 'happier'. Conversely, by subtracting a constant or multiplying with a factor less than one the face appears less 'happy'.

For most geometries we should make sure that each the scalar $r_i(n)$ is in $[0, 1]$. Another way of altering the animation would be limiting the range of values for a component. In this way, one could e.g. set a minimum smile, so that the face is always smiling.

The next level of operations would be to intertwine different components. That is, we could e.g. exchange grumbles by smiles and vice versa. If we only look at linear operations on the components, the idea of changing each component based on the quantity of all components could be modeled by a matrix multiplication.

Even more flexibility is achieved by also involving time. If we let these changes be triggered by some events on the components really interesting effects are possible. For example, we could add a smile following each blinking.

Combining animations

Combining animations is the key to powerful and convenient authoring. We will start to explain the idea by looking at the combination of two animations, $\mathbf{r}(t)$ and $\mathbf{s}(t)$. Basically, we understand combination as a weighted sum of respective components over time: $\lambda_r \mathbf{r}(t) + \lambda_s \mathbf{s}(t)$. As in the previous section, clipping the vector components to $[0, 1]$ might be necessary. Additionally, a phase shift τ might be useful $\lambda_r \mathbf{r}(t) + \lambda_s \mathbf{s}(t + \tau)$, which also introduces the idea of smooth blending from one animation into another (fading one animation out, the other one in).

Some useful example for the case of combining just two animations is this: Typically, a human is blinking from time to time to moisten the eye. One could create an animation, where the face is just blinking from time to time. This could be added to any other animation, thus, eliminating the need to author blinking in every animation.

Of course, the idea of combining animations fosters the paradigm of a component based authoring system. Animators would author only small components and,

once a base of useful components is established, just plug them together as needed. This would be best done with a hierarchical approach. Ideally, a text to speech system would serve for constructing the phoneme/viseme part of the animation.

The idea of animation components is also important for the design of interactive systems. According to user input, the system would pick pre-authored components to react flexibly. Smooth transitions from one animation to another also help to change behavior quickly, but still visually pleasant and realistically.

7.1.3 Streaming and displaying animations

In this section we will show the effects of changing the geometry used to display. Using different geometries serves several needs: First, we can change geometry in order to achieve some changes in visual appearance and, thus, communication behavior of the animation. Second, geometry changes might be necessary to adapt to the actual display capabilities. This will be explained in more detail. Finally, we will share the idea of a multicast scenario, where a number of different and distributed graphical workstation will display an animation.

Geometries and style of animation

Remember that the neutral face as well as each feature is described by a geometry. Obviously, we can change the whole set of geometries, without the need of re-authoring anything. Thus, we can use one animation and play it using a human face, but also e.g. a 'monster' face, dragon/horse/dog/whatever face. Imagine a large library of animations represented in our scheme - a designer, who wants to use his geometry of a face to play-back all these animations. All to do, would be to make the existing face smile, speak an 'A', and so on. But also changing only some of the geometries has interesting effects. One could change the style of some facial expressions, for instance the way a face smiles. This could be used to 'personalize' the facial expressions of a geometry.

Geometries and display capabilities

Of course, not every workstation displaying an animation will be the same. We would consider the geometries presented here neither very small, nor big. Remember that cost for displaying an animation increases linear with the number of vertices of the geometry. Therefore, small geometries will be displayed faster than large ones. On the other hand, large vertex counts might be desirable from a designer's point of view. Thus, we need to trade off between display limitations and designer's demand.

In recent years, simplification of polyhedral meshes has drawn much attention. One can find many schemes in the literature that allow a representation of meshes at different levels of accuracy. If the geometries would be represented at different levels of accuracy (and vertex count, of course) one could pick an appropriate resolution according to display capabilities and current workstation load.

Streaming animations to groups

Because of the very compact size of the animation itself, it can be communicated on demand or in real time from a server to clients. We envision multicast scenarios, where one server streams an animation to several clients. Contrary to most other approaches no quality of service considerations are necessary for the limited and varying bandwidth of communication connections. The amount of bandwidth needed for our representation will be always available, as long as the connection is not broken.

So, we can assume that every animation can be distributed among several clients. Every displaying workstation can adapt to its own limitations and to the user's needs. The former is obvious and was already explained in the previous section. For the latter, we think of needs in terms of usage, experience, and taste. The user might not want a fully detailed geometry, independent of the capabilities of the workstation. Or, younger users might want special characters to speak to them, while - at the same time - experienced users only need a very abstracted face that just conveys the information. In general, the independence of geometry and animation sequence is a key feature in distributed display scenarios.

7.2 Decomposing key frame animations

Assume we are given a scene comprising animated shapes described by key frame geometries F_i. All shapes are assumed to have a constant isomorphic topology K. We assume that all base shapes have vectors of same length and that the attributes are arranged identically. The state of an object in a key frame animation can then be calculated by interpolating between two consecutive key frames. Formally, this can be described as

$$A(t) = \sum_i a_i(t) F_i \tag{7.1}$$

where $A(t)$ stands for the object's state at time t and $a_i(t)$ are the weights describing the key frame interpolation, i.e.

$$a_i(t) = \left(0, \ldots, 0, \frac{t_{i+1} - t}{t_{i+i} - t_i}, \frac{t - t_i}{t_{i+i} - t_i}, 0, \ldots, 0\right) \tag{7.2}$$

where t_i is the time stamp of the i-th key frame.

However, if we want to separate geometry from animation an alternative representation would be useful, such as:

$$A(t) = a'_0(t) F'_0 + \sum_i a'_i(t) f'_i \tag{7.3}$$

where F'_0 represents the average geometry of the shape and the sum describes deviations from this representative geometry. This process of extracting a base component could be repeated in order to find the principal deviations from the average

geometry. Generally, it makes sense to sort the geometries based on their geometric importance. We denote this description as

$$A(t) = \sum_i \tilde{a}_i(t) B_i \tag{7.4}$$

where B_0 represents the average static geometry while the remaining factors B_i represent geometric changes with decreasing importance with respect to the reconstruction of the animation. Note that representations (7.1), (7.3), and (7.4) are just basis transformations of each other, i.e. a matrix multiplication transforms one into the other.

However, simple rigid motions of the object may render this approach obsolete because the linear deviations B_i from the base geometry B_0 cannot include e.g. rotations (see also Lengyel [1999]). For this reason we decompose the animation into rigid body motion and an elastic part first: First, all shapes are translated so that their center of mass coincides with the origin. Then, an affine map is computed minimizing the squared distance of corresponding vertices with regard to the first frame. The affine map is restricted to matrix representations with determinants greater zero, since reflections seem not appropriate and the matrix has to be invertible. Results of this approach are depicted in Figure 7.2.

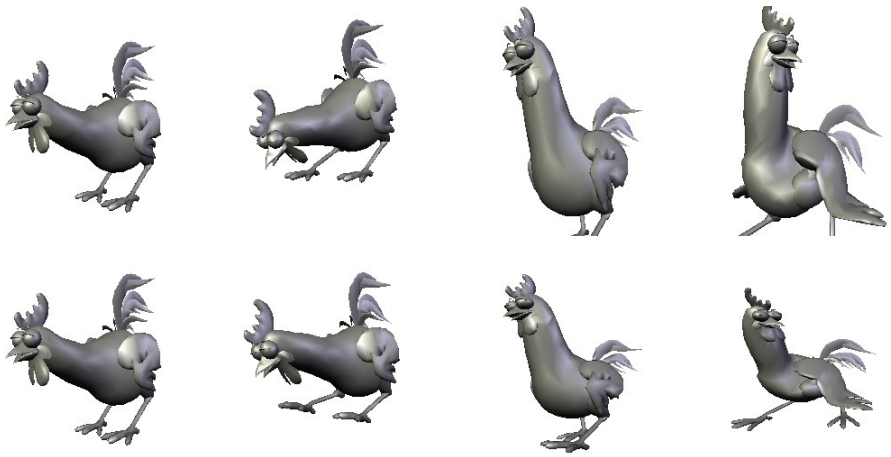


Figure 7.2: The normalization step. The upper row shows frames from a chicken animation translated so that the centers of mass coincide with the origin. The lower row shows transformed shapes where the squared distances of corresponding vertices are minimized.

If we assume our model to be represented in homogenous coordinates, we can write the necessary transformation as a single matrix multiplication. That is, instead of the key frames F_i we use transformed key-frames $T(t_i)F_i$. This has to be taken into account when the animation is reconstructed. Here, we have to use a

slightly different representation:

$$A(t) = T^{-1}(t) \sum_i \tilde{a}_i(t) B_i \tag{7.5}$$

Assumed the B_i behave as described above, we have a representation where animation and geometry are clearly decoupled. The geometry part is described mainly by B_0, while the main animation is described by the T_i and $\tilde{a}_i(t)$. B_1, B_2, B_3, and so on describe all possible deviations of the geometry.

This representation of the animation sequence makes it easy to perform compression and LOD operations. By restricting the representation to the first few components B_i, high compression ratios can be achieved while omitting only unimportant features. Furthermore, metric LOD techniques are directly supported since progressive meshes have to be generated and hold in memory for these few components only.

At the same time, the number of bases used in (7.4) affects the accuracy of the animation. Few B_1 result in a coarse representation of the animation, more B_i in higher accuracy. Thus, this representation is inherently progressive.

Further implications of this representation are shown and discussed in sections 5 and 6. In the upcoming section we explain how to find the above description.

7.2.1 Principal Component Analysis

A process of analyzing the relationship between base vectors of a space is the Principal Component Analysis (PCA). In our scenario, the PCA determines first the average shape that contains the common properties of the shapes in all key-frames. Other components will represent differences to this shape. This is also interesting when it comes to coding the bases of a shapes, as it exploits similarity and turns it into zeros, which are easily compressed using entropy encoding.

There are several ways of finding principal components. In our case we are not only concerned in finding the most important principal components to give a rough approximation of the shapes. In addition, we want to find an alternative basis and cut only a few non-contributing vectors. A way of finding this basis is the singular value decomposition (SVD) [Golub & Van Loan 1989].

Formally, we can write the non-rigid part of the original key-frames in matrix form:

$$F = (T_0 B_0, \ldots, T_{n-1} B_{n-1}) \tag{7.6}$$

Using the SVD we find the following:

$$F = BS\tilde{A} \tag{7.7}$$

The values of the diagonal matrix S are the singular values. The closer singular values are to zero, the closer the original base is to some linear dependencies. The first orthogonal matrix B contains the basis of the space with base vectors corresponding to the singular values, i.e. the rows contain the B_i we are searching for.

The last matrix $\tilde{A}$ contains the new weight vectors $\tilde{a}_i$. The matrix representation and SVD is visualized in Figure 7.3

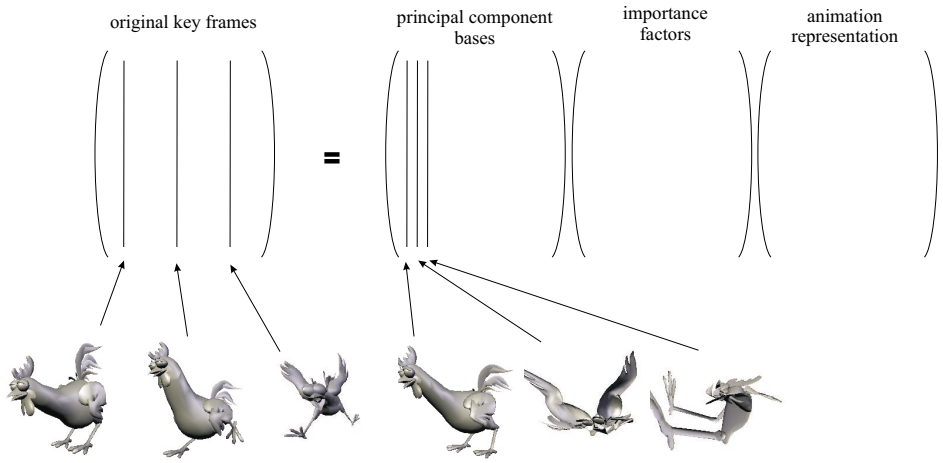


Figure 7.3: The Principal Component Analysis for geometric animations illustrated.

A severe problem with the SVD is that it is very costly and likely to reach its limits on modern computers when applied to matrices with the size of the vertex count of typical models times the numbers of key frames. A solution is to simplify the base shapes and to not consider every key frame. This is rectified by the spatial and temporal coherence typically exhibited in geometric animations. In particular, it is in many cases sufficient to consider only every second up to every fifth key frame. However, adaptive schemes for the selection of key frames would be desirable.

The representation vectors $\tilde{a}_i$ for key frame which have not been considered in the SVD can be obtained by projecting the key frame into the new basis. Since the basis constructed with the SVD is orthonormal, computing inner products of the key frames and the new base vectors is the desired projection.

7.2.2 Results

In this section we present results of computing a PCA for two animation sequences. We show how the PCA leads to a compressed progressive representation and fosters the exchange of geometry or behavior.

The first example is a part of the Chicken Crossing animation, in particular 400 frames of the chicken's geometry. The sequence is highly non-linear, i.e. it comprises rigid body motion and dynamic soft body changes. The geometry consists of 3030 vertices. Disregarding the topology information, this results in an uncompressed size of 400 frames $\times$ 3030 vertices $\times$ 3 dimensions $\times$ 4byte = 14.544.000

bytes.

To generate the principal component representation we first normalized the frames using the linear least squares fit. The resulting key-frames were composed into a 9090x400 matrix and a SVD was performed. The resulting orthogonal matrix was used to define the new base vectors replacing the key-frames.

Figure 7.4: Principal component analysis applied to the chicken animation. Prior to the SVD base shapes are normalized. The sequences above show frames 0, 80, 160, 240, 320, and 400 using different number of bases in the reconstruction.

We reconstructed the animated sequences using different numbers of base objects. This was done by setting appropriate singular values to zero. The results are shown in Figure 7.4. The animation still looks very reasonable with only 10 base shapes. Even the animation using only 5 bases can be used if viewed from far away. This would correspond to the definition of a complex LOD.

By omitting a number of base objects high compression ratios can be achieved. Table 7.1 shows the compression ratios for the animation. However, this animation is not typical. It comprises a highly deforming object. For typical animation

encoding	size	ratio
original	14,544,000	1
all base shapes	14,548,800	100.01 %
50 base shapes	1,818,600	12.50 %
10 base shapes	364,800	2.51 %
5 base shapes	181,8600	1.25 %
3 base shapes	109,116	0.75 %

Table 7.1: Compression ratios for principal component representation. Note that sizes and ratios include the additional costs for storing the transformation matrices from the normalization step.

sequences even better results can be expected.

In the second example we applied the principal component representation for facial animation. Here we used several facial expressions coded as polyhedral models with isomorphic vertex-edge topology. Key frames are generated by blending several expressions (e.g. speaking an “A” and smiling). In this system, the animation is represented as a vector over time that describes the linear combination of base shapes. Thus, animation representation and geometry are already decoupled in this specific application. In this example, we show how the geometry of the avatar can be exchanged with another geometry making use of the already existing animation descriptions.

We have used a feature based morphing technique to produce a mesh that can represent both, the original avatar and another face. By defining a few vertex-vertex correspondences we make sure that the same vertices represent common features in the source and the target model. This results in the convenient fact that we can combine the new face with expressions of the old one, e.g. we can add the smile defined in terms of the original avatar to the new face. This means, by defining the correspondences between the two neutral faces we get all other expressions of the face automatically. Play-back animations authored for the original avatar can directly be applied to the new face. In addition, we can morph between the two faces while the animation is performed since the avatar and the new face now share the same vertex edge-topology. This is an impressive example of the effectiveness of decoupling animation representation from geometry. Results are depicted in Figure 7.5.

Figure 7.5: Exchanging geometry in existing animations. a) A facial animation defined by a linear combination of base shapes. b) A featured-guided morph between the original avatar mesh and a new mesh. Topological merging is used to produce a mesh which represents both shapes. Feature control assures that the same vertices represent common features (e.g. mouth, eyes, etc.) c) The new mesh can be used with the existing animation with no additional user intervention. d) The morph can even be applied while the animation is performed.

7.3 Implementation in graphical standards

The concept of morphing among several base shapes has proven to be a universal tool in modeling and animation. We envision a construct that is able to perform the necessary operations as a part of modern graphics APIs. Inspired by a scenegraph we call the element *Morph Node* [Alexa et al. 2000].

The basic task a Morph Node should be able to accomplish is to interpolate among the vertex attributes of any number of homeomorphic shapes. Formally, a number of base shapes B_i are defined by different vertex attributes, V_i, where V_i may consist of a coordinate, a normal, color information, a texture coordinate, and possible other information. Given a vector $\mathbf{a}$, we need means to compute a shape $B(a)$, that is,

$$B(\mathbf{a}) = \sum_i a_i B_i = \left\{ \sum_i a_i V_i \right\} \tag{7.8}$$

However, it seems useful to be more flexible as to what is actually interpolated. For instance, the shapes might have color attributes specified, but color is the same for all base shapes. Obviously, it makes no sense to interpolate color. Or the user wants to fix a set of colors for the blended shape, independent of the colors actually set in the base shapes. This seems to be especially true for normals (see the upcoming section on Optimization issues).

Additionally, not all have to be interpolated linearly. A linear interpolation of normals is obviously not the only solution (to say the least). With respect to the general concept of a Morph Node one should be able to exclude specified attributes from the general linear interpolation and interpolate by means of other techniques.

To wrap up the needs for vertex attribute interpolation, we need to be able to

- interpolate linearly between any number of homeomorphic shapes,
- specify which subset of vertex attributes is actually interpolated
- specify fixed values attributes that are not interpolated or
- supply other interpolaters for these attributes.

For playing back animations we need an additional construct to interpolate the given key frame vectors over time. This could be done either linearly or by using splines. However, since the space requirements for a single key frame are so small, we can afford to specify as many key frames as are necessary for linear interpolation (e.g. more than 25 per second).

In the following subsections we will review the current 3D APIs with regard to the above stated requirements as well as propose ex- tensions and changes.

7.3.1 State-of-the-art

We would like to use web-based 3D application interfaces to play back geometric animations. We propose to use the techniques mentioned in the introduction to

accomplish this task. Thus, to playback an animation a workstation gets

1. the base shapes (several attributes, including position, normal, color, texture coordinate, etc.) and

2. a set of key-frames comprising a time stamp and a weight vector.

We will examine VRML-97, MPEG-4, Java3D, and X3D as to what they offer to play back an animation communicated in this way.

VRML-97

The first version of VRML, presented 1994, was largely a static scene description file format. The lack of animation and interaction was leading to VRML-2.0 which became the ISO standard VRML-97 [ISO JTC124 1997].

Even so, one of the main goals of the original VRML-2.0 proposal "Moving Worlds" was to create a specification which allows animation rich environments, the support for geometric morphing is limited. The designed concentrates on simple linear key-framed animation and supports only the linear interpolation of vertex position values of two shapes. The blending of more than two shapes is not supported at all. Furthermore, the `CoordinateInterpolater` only handles the vertex positions. Other vertex attributes like normal, color and texture coordinate are not handled by the `CoordinateInterpolator`. It is unusual to change the texture coordinate and color on vertex base during a key-frame animation and therefore not directly supported in VRML-97. The normal computation per vertex in contrast is very important but also time consuming and therefore different technics are provided by the specification:

1. The `NormalInterpolator` can be used to compute the normals for faces or vertex. Like the `CoordinateInterpolator`, the `Normalinterpolator` interpolates among a set of multi-valued `Vec3f` values. In addition, all output vectors will have been normalized by the node.

2. If the geometric node does not define any normals (the normal field is NULL), then the system automatically generates normals, using the given `normalPerVertex` and `creaseAngle` settings to determine if and how normals are smoothed across shared vertices.

3. If the base shape includes normal definitions, the fastest and simplest way is to keep the normals unchanged. Only the vertex positions are interpolated, which is not correct, but might be visually acceptable if the position changes are minimal

Beside `CoordinateInterpolator` and `NormalInterpolater` additional Interpolator for rotation and position are defined in the VRML-97 standard. These Interpolators are mainly used for rigid body animation which can be combined (e.g. swimming fish) with the key-frame description to get the desired effect.

There have already been various proposals to extend the animation capabilities of VRML-97. Two proposals lead to officially accepted WEB consortium working groups.

1. H-Anim: To create a standard VRML-97 representation for humanoids [Web3D Consortium 1999a].

2. Natural Language Processing (NLP): Will be used to allow natural language to interact with VRML-97 animation

Although both working groups are involved in animation description, none of these two work items is directly related to the work presented in this paper.

Java3D

Java3D is the API for 3D graphics within Java [Deering & Sowizral 1997]. It follows the scene graph architecture of VRML-97. A Shape in Java3D is represented by the node Shape3D comprising a Geometry and an Appearance node. Geometry is specified as an array of attributes, including coordinate, color (with or without alpha), normal, and texture coordinate. The appearance of a Shape can be controlled in various ways also depending on the geometry. In most application a material description will define the appearance.

Java3D has a node called “Morph”. This node accepts several geometries of exactly the same type, i.e. the same number of vertices with the same combination of attributes specified in the same way (indexed, stripped, etc.). One appearance object can be specified per Morph. The actual geometry can be altered by a call to the setWeights method.

In order to interpolate between key frames, Java3D offers Interpolators that are extended to take the time stamps of key frames into account.

As such, Java3D seems to offers the basic tools for the playback of animations given as base shapes and weight vectors. However, there are limitations in the design which hinder the use of Java3D in this context for now:

- base shapes need to have the same attributes

- and all these attributes are interpolated

- but no other than the predefined vertex attributes can be interpolated

We would like to give the user the flexibility to use one set of colors for all geometries without specifying them for every shape or, even worse, without interpolating them in any case. On the other hand it would be nice to allow the interpolation of materials.

In our example of facial animation the shapes have vertex colors, but these vertex colors do not change for different base shapes. In Java3D these float triples are interpolated whenever setWeights is called. Also, normals are interpolated whether

the user likes it or not (see the section on normal interpolation for other opportunities to set normals). Instead of interpolating only three floats for coordinates, Java3D interpolates nine and additionally normalizes the result of normal interpolation resulting in a more than three times worse performance.

MPEG-4

MPEG-4 represents one of the newest multimedia standards in the MPEG family. MPEG-4 supports the standard video encoding schemes well known from MPEG-2 based on efficient frame en- coding and motion compensation. In addition, MPEG-4 contains language components for 2d and 3d scene elements and scene structures based on VRML-97.

Specific extensions are related to the animation of artificial faces and bodies [Ostermann 1998]. The "facial animation object" in MPEG-4 can be used to render an animated face. The shape, texture, and expressions of the face can be defined by Facial Definition Parameters (FDPs) in BIFs. BIFs are downloadable models to configure a baseline face model or to install a specific face model at the beginning of a session along with the information about how to animate it. Facial Animation Parameters (FAPs) can then be used to animate this model.

Face Animation Table (FAT) within FDPs are used to perform the functional mapping from incoming FAPs to feature control points in the face mesh. Face Interpolation Technique (FIT) in BIFS allow the interpolation between different expressions based on weighted rational polynomial functions which is invoked by conditional evaluation of a Face Interpolation Graph. This functionality can be used for complex cross-coupling of FAPs to link their effects, or to interpolate FAPs missing in the stream using the FAPs that are available in the terminal. Face Animation in MPEG-4 provides for highly efficient coding of animation parameters that can drive an unlimited range of face models. The models themselves are not normative.

In addition, a new standard called "MPEG-4 The Body Animation" is being developed by MPEG in concert with the Humanoid Animation Working Group within the VRML Consortium, with the objective of achieving consistent conventions and control of body models which are being established by H-Anim. This Body Animation, to be standardized in MPEG-4 Version 2, is being designed into the MPEG-4 fabric to work in a thoroughly integrated way with face/head animation. Here, decoding and scene descriptions directly mirror to technology already proven in Face Animation and corresponding Body Definition Parameters (BDPs) and Body Animation Parameters (BAPs) exist.

The facial and body animation elements in VRML-97 represent a very interesting technology in the context of animated User Interfaces and User Interface Agents. The standardization of the control of face animations is one of the major achievements in this context. Technological systems supporting these concepts may, especially when based on FIT, lead to similar problems as stated in the motivation of this paper. The solutions provided in the following sections are therefore

also applicable in this context.

X3D

Another standardization activity in the area of 3d animation systems and exchange formats is X3D [Web3D Consortium 1999b]. The goal of X3D is to represent a next-generation extensible 3d graphics specification which makes use of the VRML-97 structure and expresses it by the means of XML: As such, in the current state of development X3D makes still use of the interpolator functionality known from VRML-97 for morphing.

7.3.2 Proposed extensions and changes

Since neither VRML nor Java3D provide means to implement the concept of a Morph Node as it would be suitable to support anima- tion and elaborate modeling techniques we propose the following changes and extensions to the existing standards.

Extensions to VRML-97

We propose extend VRML-97 by three nodes in order to accomplish the ideas of the Morph Node.

- A `VectorInterpolator` node, which handles interpolation of general float-type vectors.
- A `CoordinateMorpher`, which interpolates linearly among coordinate sets.
- A `NormalMorpher`, which interpolates linearly and normalizes normal sets.

In the following we give specification and details about these nodes.

```
VectorInterpolator {
        eventIn             SFFloat      set_fraction
        exposedField        MFFloat      key []
        exposedField        MFFloat      keyValue []
        eventOut            MFFloat      value_changed
}
```

The node is designed as an additional VRML-97 interpolator in the scene that defines a piecewise-linear function, $f(t)$, on the interval $[infty, \infty]$. The general aim of the node is the ability to interpolate between a set of vectors of any size. The node sends multiple-value results like the `CoordinateInterpolator` and `NormalInterpolator`. Therefore, the `keyValue` field is an $n \times m$ matrix of values, where n is the number of values in the key field and m the size of

the vector. The number of scalar values in the `keyValue` field shall be an integer multiple of the number of key-frames in the `key` field. That integer multiple defines only implicit the size of the vector and therefor the number of elements which will be contained in the `value_changed` events. The `value_changed` output is used in our proposal as input for the `set_weights` eventIn slot in the following `CoordinateMorpher` and `NormalMorpher` Node.

```
CoordinateMorpher {
        eventIn             MFFloat     set_weights
        exposedField        MFVec3f     keyValue []
        eventOut            MFVec3f     value_changed

}
```

The CoordinateMorpher node interpolates linearly among a set of `MFVec3f` values. Unlike the `CoordinateInterpolator` it does not interpolate two key frames but is able to blend any number of shapes. The number of coordinates in the `keyValue` shall be an integer multiple of the number of key-frames in the `key` field. That integer multiple defines how many coordinates will be contained in the `value_changed` eventout slot.

```
NormalMorpher {
        eventIn             MFFloat     set_weights
        exposedField        MFVec3f     keyValue []
        eventOut            MFVec3f     value_changed
}
```

The NormalMorpher node blends among a set of normal vector sets specified by the keyValue field. The output vector, `value_changed`, shall be a set of normalized vectors.

Changes in Java3D

The changes necessary to fit the existing Java3D MorphNode into our concept follow directly from the state-of-the-art section. In addition to what the Java3D API defines up to now we need ways to

1. specify the attributes which are actually interpolated,
2. supply sets of fixed attributes, and
3. supply other interpolators for specified attributes.

Furthermore, Java in general needs more accurate timing mechanisms to support synchronized playback of geometric animations.

Proposal for X3D

One of the main goals of the X3D proposal is to build the new technologies based on a small, lightweight core. Since the VRML-97 specification is considered to be large and complex to implement the base set will not include all elements of the predecessor. We propose to include the `CoordinateMorpher` instead of the `CoordinateInterpolator` into the core X3D specification. The `CoordinateMorpher` is a more general approach to shape animation and can substitute the Interpolators without any lose in functionality or performance.

7.3.3 Optimization issues

The general concept and implementation of a Morph Node is rather simple. However, a naive approach results in a complexity of O(nma), where n is the number of base shapes, m is the number vertices per base shape, and a this the number of attributes to be interpolated. That is, deciding which attributes really need to be interpolated is quite crucial. Especially normals are difficult to treat, since a simple linear interpolation does not yield correct results. We discuss ways to handle normals in the upcoming subsection.

Generally, base shapes might not differ in all the attributes for all vertices, and the system should exploit this fact as much as possible.

How to interpolate normals?

As said before, linear interpolation of normals might not be correct. First note that normals are not necessarily normalized after linear interpolation, which could cause shading to fail. Thus, we need to normalize the vectors after interpolation. But still, the result of linear interpolation seems to be not what one would expect. Instead we should use SLERP/quaternion interpolation. Unfortunately, even a quaternion based, correct direction interpolation of normals does not lead to the normals a linearly interpolated geometry actually has. For these diverse reasons we see the following for different ways to treat normals:

1. use only one set of normals (e.g. the normals of B0, or a set of mean normals among $B_0, \ldots, B_{n-1}$)
2. interpolate normals linearly and normalize
3. interpolate in quaternion space
4. compute new normals after geometry interpolation

While option one seems unacceptable at first sight it produces surprisingly good results. See also Figure 7.6, which compares an animation of swimming dolphin with correct normals (recalculate in each frame) with no normal interpolation (the corresponding movies can be downloaded from http://www.igd.fhg.de/ alexa/morphnode/).

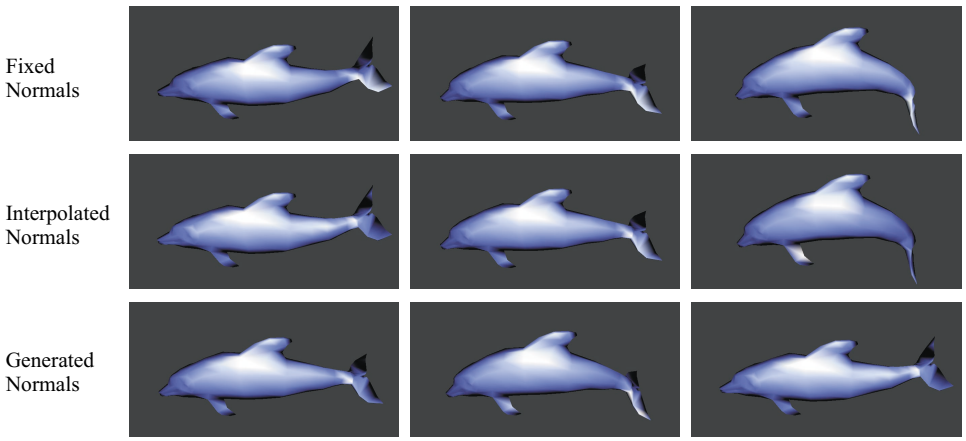


Figure 7.6: Comparison of normal strategies: The upper row shows different states of a dolphin animation with a fixed set of normals (i.e. the normals are the same in all three images). The middle row shows the same states with linearly interpolated and normalized normals. The normals in the lower row generated form the geometry for each frame.

	fixed	interpolated	generated
dolphin 1000 vertices	5.6 sec	6.8 sec	9.8 sec
pig-horse 14269 vertices	192.8 sec	245.7 sec	407.3 sec

Table 7.2: Times needed to render 1000 frames of an animation with different normal calculation strategies.

Of course, recalculation of normals might be necessary in a number of cases. We mainly considered linear interpolation and recalculation. Quaternion interpolation does not produce the correct normals of the interpolated shape yet is still more expensive than linear interpolation. A comparison of the remaining three approaches is given in table 1 below. Note that timings are heavily influenced by several other tasks the animation engine has to accomplish.

A more elaborated approach could compute geometry and normals asynchronously, i.e. updating normals only every f-th update of geometry. This would generate only slightly incorrect normals (which are acceptable, as was demonstrated before) and yet provide a high frame rate with respect to geometry.

Don’t touch equal attributes

In many animations only parts of the attribute change. If we take a look at the facial animation example again we easily see that

- color - even though it is set - does not change at all for different base shapes

- vertex coordinates for several regions do not change or do not change much
- thus also the normals in theses regions do not change (much)

We have illustrated this for the facial animation example in Figure 7.7 (see also the corresponding animation at http://www.igd.fhg.de/ alexa/morphnode/). Here, we use the `ColorManipulator` node of the Avalon system [Behr 1999] to color code vertices depending on their velocity. Standard gray vertices mean fixed vertices, red vertices are moving. Note that only small number of vertices is turning red during the animation. Generally speaking, a subset of vertex attributes might

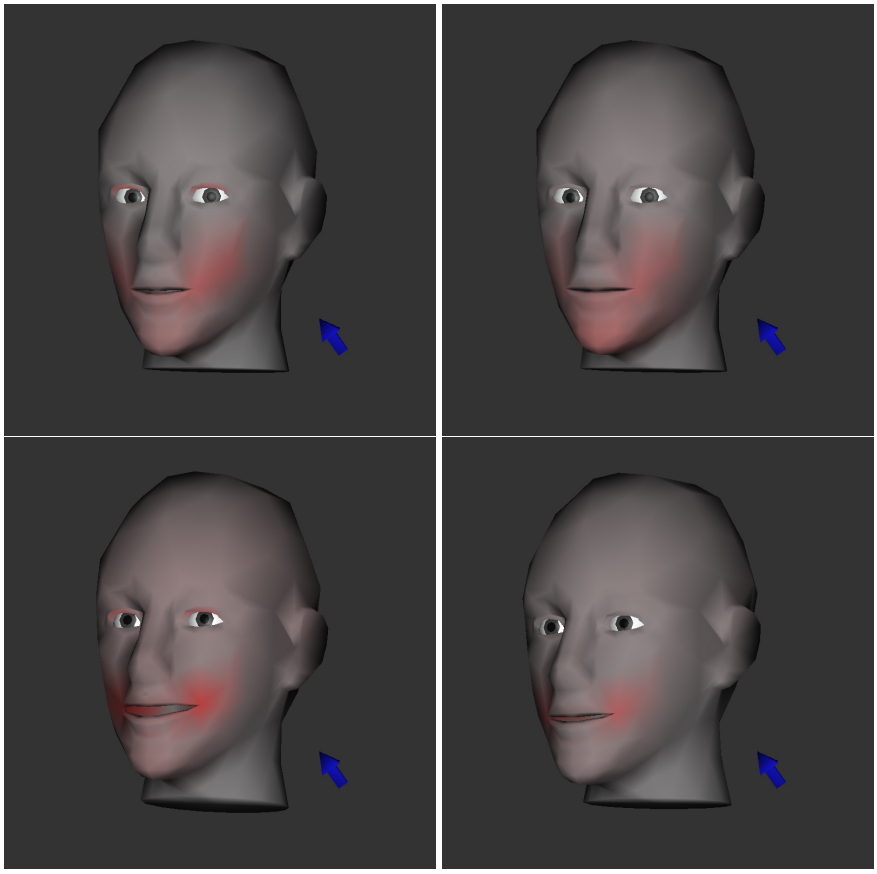


Figure 7.7: Color coding of moving vertices. Grey depicts static vertices, red color reflects high velocity.

be exactly or almost the same for all base shapes. The system should first iterate over all attributes for base shapes and identify similar attributes. This would result in information regarding which vertices and which attributes actually need to be interpolated.

A simple trick to exploit this information is to re-sort the vertices in each shape in a such a way that differing vertices appear first in the list/array. The loop nec-

essary for interpolation would only iterate over the first vertices not touching the fixed vertices.

Know what weights do to vertices

We can extend the concept of the last subsection to individual base shapes: some vertices might not change for a subset of base shapes. If the weights for the base shapes of this subset are all zero we do not need to recalculate these vertices. The basic problem is how to exploit the fact that some base shapes only need to interpolate a very small subset of vertices.

We propose the following solution. We reorder the vertices so that the following conditions describe the sequence of vertices:

1. None of the follwing cases.
2. $B_1 \neq B_0 = B_2 = B_3 = \ldots$
3. $B_2 \neq B_0 = B_1 = B_3 = \ldots$
4. and so on.

When looping over vertices we can include or exclude sets of vertices depending on the weight value of the respective base shape. For instance can we dismiss operations in block 2 if the weight corresponding to B_1 is set to zero.

7.4 Conclusions

We present an approach for representing animation sequences based on the principal components of key frame geometries. The principal component representation allows for an easy and adaptive lossy compression of animation sequences with factors up to 1% accepting loss in animation accuracy. It would be interesting to quantify that loss relative to the compression ratio. However, measures of geometric deformation seem to be topic of current research and not yet applicable. Moreover, standard compression techniques were not exploited at all in this calculation. Additional compression can be achieved by compressing the base shape matrices. Again, the principal component analysis reorganizes the base shapes so that bases with higher index will contain more zeros. This could be exploited with an additional entropy encoding.

The order of base objects naturally supports progressive transmission of the animation base shapes. At the same time, the inherent hierarchy could be used for LOD techniques. This could go hand in hand with a progressive mesh. The importance ordering of base shapes additionally eases mesh simplification since only the geometric features of a few meshes have to be taken into account for simplification. In first experiments we have found that standard simplification techniques can be extended to handle more than one mesh.

The animation itself is represented by small vectors if the number of necessary base shapes is small. Note that the number of necessary base shapes is bounded by the number of key frames. While the number of base shapes in our examples is much less than the number of key frames, it might be useful to break very long animation sequences into pieces. It would be interesting to exchange only parts of a base while streaming an animation.

Having small vectors representing the animation allows for streaming over virtually every network in real time. The decoupling of animation and geometry enables managing and changing animations, e.g. exchanging the animated object according to the client's display capabilities. Mapping existing animations to a new object offers new ways of authoring.

We also analyzed how to incorporate the necessary operations to play back animations defined as linear combinations of a small set of base shapes in state-of-the-art 3d graphics systems such as Java3D and VRML-97. Current shape blending methods based on interpolation nodes always perform an interpolation between the complete geometry of two base shapes and recalculate all object attributes such as normals and color anew. As a consequence, these systems lack performance when morphing is applied in real-time applications.

We propose a new morph node with enhanced flexibility and performance, supporting morphing between several objects and giving enhanced control over the morphing process. Most important, morphing calculations between attributes leading to no additional visual effect can be avoided in this way. For a number of test objects time we could prove a performance enhancement of upto 100%. Therefore, we recommend the consideration of our morph node in the ongoing X3D and

MPEG4 specification processes.