Integrating Hsplines into Softimage|3D

by

Jean-Luc Duprat

B.Sc., McGill University, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming
to the required standard

The University of British Columbia

July 1997

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Jean-Luc Duprat.

The University of British Columbia
Department of Computer Science
2366 West Mall
Vancouver, BC
V6T 1Z4 Canada

Date: September 5, 1997.

Abstract

Hierarchical splines (hsplines) are a multiresolution representation of spline surfaces with support for local refinement. They have proved to be more flexible for character animation than the surface formulations supported by modern animation systems, especially when driven by a skeleton, a technique known as enveloping.

An extension to the basic hspline formulation allows cylinders to be smoothly connected to non-isoparametric lines of another hspline surface. This allows limbs to be connected to bodies with unprecedented ease for the animator. The region of the attachment is represented with Catmull-Clark surfaces.

This thesis describes how hierarchical splines have been integrated into the Softimage|3D environment, emphasizing both technical and user interface aspects, and including a complete discussion of enveloping. Arbitrary topology surface attachments are described in detail, including current limitations in Softimage that prevent their complete integration. Rendering of hsplines was not implemented in a satisfactory manner due to limitations in the host environment, and these issues will be carefully examined.

Contents

List of Figures

Acknowledgements

I must thank Dave Forsey, my supervisor, whose enthusiasm for animation is highly contagious. His help was invaluable, and often requested at indecent hours during production.

Chantal Guyon who gave me the strength to get through it all must be praised, for her support was not always acknowledged promptly. Los Angeles would have been a nightmare if I had been on my own.

My parents Anne and Pierre, as well as my sisters Séverine and Catherine who are always very supportive and understanding have my eternal gratitude.

I want to thank the members of the computer graphics lab (Imager) at the university of British Columbia, who made my stay stimulating, challenging and rewarding. In particular the following people must be mentioned: Kevin Coughlan, Joel DeYoung, Alain Fournier, Jason Harrison, Paul Lalonde and Dave Martindale.

My officemates Ronald Beirouti, Joel DeYoung and Brian Fuller who didn't mind it when I slept in the office must be thanked for the many entertaining conversations that kept the pressure under control.

My friends Ben and Tristan made sure that I got a healthy amount of laughs on a regular basis.

The DreamWorks production crew in Los Angeles taught me a lot about animation and management, but the most exciting thing was definitely the artwork. I want to thank Ken Harsha, Sylvia Matchett, Tom Sito and Michael Stone for their enduring friendship. Tom Sito must also be thanked for the March of History. I am indebted to the Propellerheads: Rob Letterman, Loren Soman and Andy Waisler for being the first to believe in us. I am grateful to the rest of the programming team: Andy Bruss, Bart Gawboy and Mike Meckler for sharing the excitement and the pain.

The animators who took time to make constructive suggestions helped shape the plugin into what it is today, in particular Anders Beer, Donnachada Daly, Loren Soman and Goesta Struve-Dencher.

I would like to thank Peter Cahoon for taking time out of his busy schedule to be my second reader.

Finally UBC and UCLA must be thanked for indulging my hectic moves, especially Joyce Poon who made it all happen.

Thank you all, for none of this would have been possible without your help, support and friendship.

JEAN-LUC DUPRAT

The University of British Columbia

July 1997

Chapter 1

Introduction

Digital character animation is a very complex undertaking that requires mastery of motion, timing and acting from the animator, as well as expertise with the medium. Animation packages currently available allow users to build complex spline surfaces, however animating them remains a frustrating experience. The medium imposes many constraints, most of which are derived from the mathematical model of spline surfaces.

Surface representation is problematic: the more detail that need to be represented, the larger the number of *degrees of freedom* (DOF) that must be controlled during animation. Another difficulty lies with the *topology* of the characters we are trying to model. Human-like characters cannot be accurately represented with a single spline surface. If they are represented as a collection of separate surfaces, continuity is difficult to maintain during animation. If the surfaces are joined, we get vertices at which more than 4 edges meet. This does not fit the tensor product framework, because the mesh isn't a grid around these points.

The two broad categories of tools currently available to manipulate models

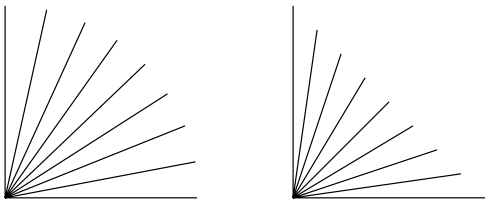
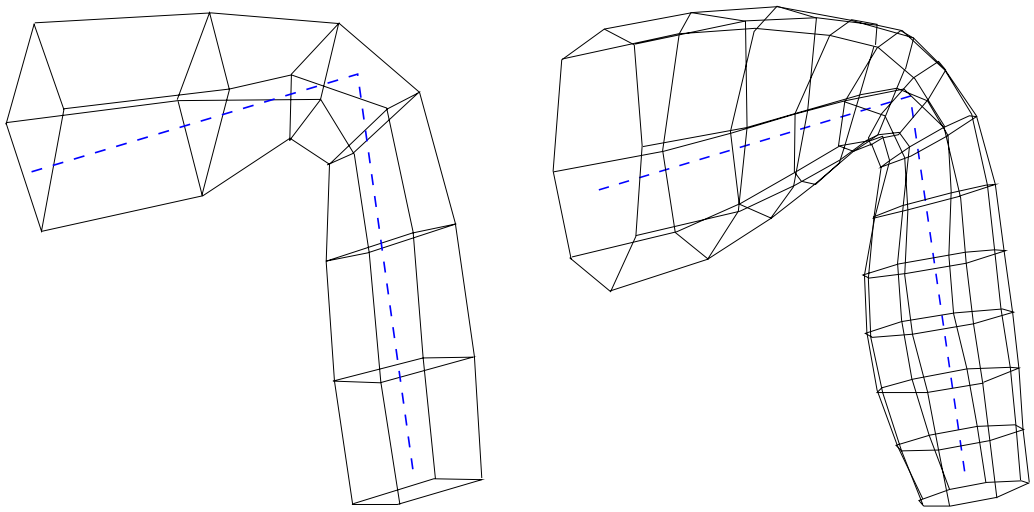


Figure 1.1: Angle vs. endpoint interpolation

are *space deformations* and *enveloping*. Space deformations, exemplified by *lattice deformations* (also known as *free-form deformations* or FFDs), are a local deformation of the space containing the model, which forces its *control vertices* (CVs) to undergo these same deformations. Enveloping assigns control vertices of a surface to the coordinate frame of a controlling *segment* or bone from an underlying skeleton. The segment influences the position of the CVs as it moves in space, thus deforming the surface in a predictable manner. Both of these categories of tools attempt to control most of the DOF in the model.

Although these tools are useful, in practice animators are reduced to directly manipulating CVs in many situations. Lattices are very useful for anthropomorph objects, but less so for human figures since they distort the space in which the geometry is embedded, rather than the geometry itself. This makes them inadequate when complex, subtle, local deformations are required, as in facial animation.

Enveloping is usually implemented as the weighted average of the influence of each one of the individual segment affecting a surface in the neighborhood of a joint. This linearly interpolates the position of each CV between the positions determined by the individual segments, rather than doing an angular interpolation, as illustrated in figure 1.1. Further, even angular interpolation does not prevent the surface from collapsing on the inner side of the joint when it is closed (figure 1.2), not maintaining the volume defined in the region of the articulation. Both of these



(a) Enveloping a Spline surface

(b) Enveloping with a finer parameterization

Figure 1.2: Surface collapse at the joint

problems require animators to spend a lot of time fixing the segment assignments in the regions around articulations.

Hierarchical splines [Forsey 88], or *hsplines* for short, offer multi-level control over surfaces, each level corresponding to a refined area of its parent surface. This allows manipulating and enveloping the surface at multiple levels of detail. Less points need to be explicitly enveloped than with regular spline surfaces, articulations can easily be made to deform convincingly [Forsey 91], and the multi-resolution control over the surface allows one to work with a relatively smaller number of patches. The current implementation of hsplines also supports arbitrary topology attachments, which are invaluable for character animation. They allow cylinders to be attached to non-isoparametric lines of another surface, thereby allowing the modeller to attach limbs to bodies very easily. This is done by extending the basic spline framework with *subdivision surfaces*.

Hsplines have been used in an experimental modeller, but it is not yet a complete animation system, and it lacks many of the surface construction tools familiar to animators. We attempted to integrate hierarchical splines into Softimage|3D, a high-end animation system, to bring their features to a larger audience. The goal was to integrate the new surface primitive in the package in such a way that their use would be non-intrusive to the rest of the system. This work took place in the context of creating a complete pipeline for the use of hierarchical splines in a feature animation production environment. The modifications required affected the whole digital production cycle: from the digitization of the surfaces, to the construction of the hierarchies, to their animation in Softimage|3D and finally to the renderer.

1.1 Overview

It is assumed that the reader is reasonably familiar with the mathematics of splines and with hsplines. This salient points will be discussed in chapter 2 which briefly covers some background material on the mathematics of splines, in particular B-splines and NURBS, the hierarchical spline representation and finally subdivision surfaces.

Chapter 3 describes the implementation within the Softimage/SAAPHIRE framework, and how the different issues were addressed. User interface issues are also examined.

Chapter 4 describes the inherent limitations of the different components used in this implementation. The problems described in that chapter could not satisfactorily be circumvented.

Chapter 5 summarizes the findings and concludes by describing possible improvements to the plugin with current technology, and possible forthcoming technol-

ogy from Softimage.

Finally, appendix A contains a tutorial on using hsplines in Softimage|3D, covering all the tools implemented that are not part of Softimage.

Chapter 2

Background material on Splines

This chapter presents background material on spline curves and surfaces, in particular B-splines and NURBS are examined. It then examines higher-level abstractions based on these concepts: hierarchical splines and subdivision surfaces. The remainder of this thesis will rely heavily on the material presented here.

2.1 B-Splines

A spline curve $\mathbf{Q}(u)$ is defined by a control vector $\mathbf{V}$ and a set of *basis functions* $B_{i,k}(u)$, which are polynomials of order k (degree $k-1$). The curve is then defined as:

$$\mathbf{Q}(u) = \sum_{i=0}^{n} \mathbf{V}_i B_{i,k}(u) \tag{2.1}$$

This is a vector equation, that is, a single u value is used to evaluate the equation for each of the x, y and z component of $\mathbf{Q}$, using the respective components of $\mathbf{V}$.

Similarly, a spline surface $\mathbf{S}(u,v)$ is defined by a control mesh $\mathbf{M}$ and two sets of basis functions $B_{i,k}(u)$ and $B_{j,l}(v)$ of respective order k and l. The spline

surface is then defined as a *tensor product*:

$$\mathbf{S}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{t} \mathbf{M}_{i,j} B_{i,k}(u) B_{j,l}(v) \tag{2.2}$$

In the case of B-splines, the basis functions are recursively defined using the *Cox-deBoor* recurrence relation:

$$B_{i,1}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

$$B_{i,k}(u) = \frac{u - u_i}{u_{i+k-1} - u_i} B_{i,k-1}(u) + \frac{u_{i+k} - u}{u_{i+k} - u_{i+1}} B_{i+1,k-1}(u)$$

The quotient $0/0$, which may arise from the recursion, is defined to be zero. The nondecreasing sequence of real numbers $U = \{u_0, \ldots, u_m\}$ is called the *knot sequence*. The order k, the number of control vertices $n+1$, and the number of knots $m+1$ are related by $m = n + k$.

2.1.1 Properties of B-splines

The following interesting properties are stated without proof. The reader is directed to [Bartels 87] or [Piegl 97] for details.

- **Local support**: $B_{i,k}(u) > 0$ for $u_i < u < u_{i+k}$ and $B_{i,k}(u) = 0$ elsewhere. That is each basis function spans k intervals. To put this differently, the interval $u_i \leq u < u_{i+1}$ is supported by the following k basis functions: $B_{i-k+1,k}, \ldots, B_{i,k}$. This also means that control point $\mathbf{V}_i$ influences the curve $\mathbf{Q}(u)$ only on the interval $u_i \leq u < u_{i+k}$.

- **Convex Hull property**: For $u_{k-1} \leq u < u_{n+1}$, $\sum_{i=0}^{n} B_{i,k}(u) = 1$. The implication is that the curve $\mathbf{Q}(u)$ lies inside the convex hull formed by the $n+1$

control vertices of the control vector $\mathbf{V}$. In surface terms, $\mathbf{S}(u, v)$ lies inside the convex hull formed by the $(n+1) \times (t+1)$ control vertices of the mesh $\mathbf{M}$.

- **Continuity**: Since $\mathbf{Q}(u)$ is a linear combination of the $B_{i,k}$ its continuity is determined by that of the basis functions, which is in turn determined by the knot vector. $\mathbf{Q}(u)$ is at least $k - \mu_i - 1$ continuously differentiable at knot u_i of multiplicity μ_i (that is $u_i = \ldots = u_{i+\mu_i-1}$), and is always $k-1$ continuously differentiable between knots. Thus $\mathbf{Q}(u)$ is anywhere from C^{-1} (discontinuous) to C^{k-2} continuous as determined by U.

- **Knot Insertion and Refinement**: Knot insertion allows inserting a set of new knots U' ($|U'| = m_{u'}$) in the original knot sequence U ($|U| = m_u$), thus changing the vector space basis, without changing the curve geometrically or parametrically. There are two common methods to perform knot insertion: Böhm's algorithm [Böhm 80] and the Oslo algorithm [Bartels 87]. While Böhm's algorithm sequentially inserts single knots values from U' into U, the Oslo algorithm which is described here can insert all of the knots in U' simultaneously. The old basis functions $B_{i,k}$ must be re-expressed as a linear combination of "smaller" basis functions:

$$B_{i,k}(u) = \sum_{r=0}^{n+m_{u'}} \alpha_{i,k}(r) N_{r,k}(u)$$

so that equation 2.1 becomes:

$$\mathbf{Q}(u) = \sum_{i=0}^{n} \mathbf{V}_i B_{i,k}(u) = \sum_{r=0}^{n+m_{u'}} \mathbf{W}_r N_{r,k}(u)$$

where $\mathbf{W}$ is a new control vector related to $\mathbf{V}$ by

$$\mathbf{W}_r = \sum_{i=0}^{n} \alpha_{i,k}(r) \mathbf{V}_i$$

For details on computing the set of functions $\alpha_{i,k}(r)$ the reader is referred to [Bartels 87].

2.1.2 Uniform cubic B-splines

In the particular case of the uniform cubic B-spline, we have $k = l = 4$, and $u_{i+1} = u_i + \Delta u = u_i + 1$. The resulting spline basis functions are:

$$\begin{aligned} B_{0,4}(u) &= \frac{1}{6}\left[-u^3 + 3u^2 - 3u + 1\right] \\ B_{1,4}(u) &= \frac{1}{6}\left[3u^3 - 6u^2 + 4\right] \\ B_{2,4}(u) &= \frac{1}{6}\left[-3u^3 + 3u^2 + 3u + 1\right] \\ B_{3,4}(u) &= \frac{1}{6}u^3 \end{aligned} \tag{2.4}$$

Using these results, equation 2.1 and 2.2 can be written more compactly using matrix notation as:

$$\begin{gathered} \mathbf{Q}_i(u) = \mathbf{u} \cdot \mathbf{B}_u \cdot \bar{\mathbf{V}}_i \qquad \mathbf{S}_{i,j}(u,v) = \mathbf{u} \cdot \mathbf{B}_u \cdot \bar{\mathbf{V}}_{i,j} \cdot \mathbf{B}_u^T \cdot \mathbf{v}^T \\ \mathbf{u} = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} \qquad \mathbf{B}_u = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \end{gathered} \tag{2.5}$$

where $\mathbf{Q}_i$ represents the i^{th} curve segment and $\bar{\mathbf{V}}_i$ is the vector of CVs influencing $\mathbf{Q}_i$, namely $\mathbf{V}_{i-3}$, $\mathbf{V}_{i-2}$, $\mathbf{V}_{i-1}$ and $\mathbf{V}_i$. Similar definitions apply to $\mathbf{S}(u,v)$.

The local support property can be interpreted as follows for bicubic B-spline patches: each basis function spans four intervals of the spline; that is, each control vertex influences four segment of the curve or surface.

Finally, knot insertion must be looked at, for inserting knots in a uniform knot sequence can affect its shape. In order to preserve uniformity we *refine* the

control mesh, a global operation which turns U into U', a uniform knot sequence where Δu is of the form $1/x$, for x integer.

2.2 NURBS

NURBS are non-uniform rational B-splines, and they differ from the B-spline patches described above in that they are rational functions. They are defined as:

$$\mathbf{Q}(u) = \frac{\sum_{i=0}^{n} w_i \mathbf{V}_i B_{i,k}(u)}{\sum_{i=0}^{n} w_i B_{i,k}(u)} \tag{2.6}$$

The w_i's are called the *weights*, and it is assumed that $w_i > 0$.

There is an elegant geometric interpretation for rational curves, which uses *homogeneous coordinates* to represent geometry, a 4-dimensional coordinate system where coordinates are represented as:

$$P_h = \begin{bmatrix} x & y & z & w \end{bmatrix}$$

and are projected into 3D Cartesian space using the following transformation:

$$P_c = \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix}$$

Technically, Cartesian space is defined as the $w = 1$ hyperplane of 4D homogeneous space. The weight must be different from zero to avoid problems when projecting back to Cartesian space ($w = 0$ represents points at infinity).

Equation 2.1 extended to 4-dimensional homogeneous space reduces to equation 2.6 after the projection into Cartesian space. This makes B-splines a special case of NURBS (the weight associated with each point is set to $w = 1$), and all the results which hold for B-splines also hold for NURBS.

NURBS have many interesting properties, one of the most important arguably being that they can represent conics exactly. The reason they are discussed here is

that most implementations of NURBS, including Softimage's, support trim curves, a technique which allows to remove pieces from NURBS surfaces surrounded by (trim) curves. Most B-spline implementations do not have this facility, however since NURBS can represent B-splines this can usually be done through them.

2.3 Hierarchical Splines

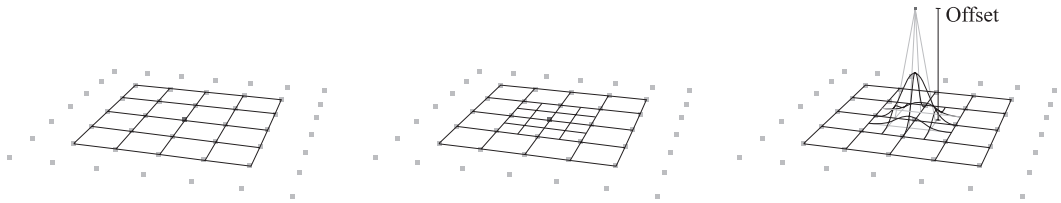
Hierarchical splines are rigorously described in [Forsey 88]; here we will discuss those aspects of hsplines that are relevant to the animator using them. The tutorial in Appendix A illustrates these concepts.

Hierarchical splines are multiresolution representations of surfaces, with spline patches grouped in a hierarchy. The root of the hierarchy, referred to as *level 0*, has the lowest resolution, and its children correspond to refined areas of the parent surface. In particular, the position of the CVs at level $p+1$ are defined by a vectorial *offset* $\mathbf{O}$ relative to the parent level p (figure 2.1c).

$$\mathbf{W}_{i,j} = \mathbf{R}_{i,j} + \mathbf{O}_{i,j} \tag{2.7}$$

where $\mathbf{R}_{i,j}$ is the called the *reference point* on the parent surface. The reference point $\mathbf{R}_{i,j}$ for $\mathbf{W}_{i,j}$ is defined as the point on the surface maximally influenced by $\mathbf{W}_{i,j}$.

There are two major implications that may not be readily apparent from this brief description. First, hierarchical splines allow *local refinement*: unlike regular spline surfaces where refinement is a global operation, hierarchical refinement creates a new patch with a finer parameterization covering the area being refined. Detail has been added to the model only where needed, rather than globally as would be



(a) 7×7 bicubic B-spline patch

(b) parent patch has been locally refined

(c) local refinement has been be edited

Figure 2.1: Local refinement of a bicubic B-spline surface

the case with regular splines. The refined area can be edited independently of the parent surface, subject to the constraint that cracks may not appear between the two surfaces.

To avoid introducing cracks between a patch and its parent, some of its CVs are prevented from moving. The local support property tells us that a single CV influences 4 knot intervals (in the case of cubics), so the surface edge must be 2 knots away from the CV being moved in both parametric directions. In other words, every movable CV must be centered on 7×7 grid of CVs (refer to figure 2.1).

The second implication of the formulation of hsplines is that refined level $p+1$ will follow level p when it is deformed. This comes about because $\mathbf{R}_{i,j}$ is attached to level p, and is updated as the level is deformed. Thus detail modelled on the surface at level $p+1$ will follow the underlying surface as it is deformed. This process is illustrated in the tutorial on editing hspline surfaces, found in section A.3 on page 70.

The scale at which each level influences the surface, which is related to the different parameterization of the levels, is very useful when enveloping. By attaching points at different levels to the two different skeletal segments of a joint, one can avoid having the surface spanning the joint collapse upon itself when the joint is manipulated (this will be discussed in more detail in section 3.5).

Hsplines also make it easy to define secondary actions for animation, since these can be applied to the offsets of the finest level of the surface. For example a simple spring-mass system may be applied to the CVs of the model, for realistic skin dynamics. More details on enveloping hsplines and procedural offset techniques can be found in [Forsey 91].

The hierarchical structure of hsplines greatly reduces redundancy in the data set by providing controls only where needed. A side effect is that it significantly reduces the storage cost of hsplines when compared to an equivalent B-spline surface (see [Wong 95] for details). Most important however is the fact that hsplines can significantly reduce the number of CVs (DOF) that must be manipulated to animate a surface.

2.3.1 Common operations on hsplines

This section describes those operations which were found to be useful when working with hsplines in an experimental modelling system. They arise from the hierarchical nature of the surface representation.

- **Refine**: This operation performs local refinement on the hspline surface, adding detail where required. It compares to global refinement of uniform splines patches.

- **Unrefine**: This operation removes detail from a surface. This usually causes information to be lost, and may be compared to the knot removal techniques described in [Piegl 97].

- **Zero Offsets**: This operation zeroes the offset vector $\mathbf{O}_{i,j}$ at a given vertex, and causes level $p+1$ to lie closer to level p at that point.

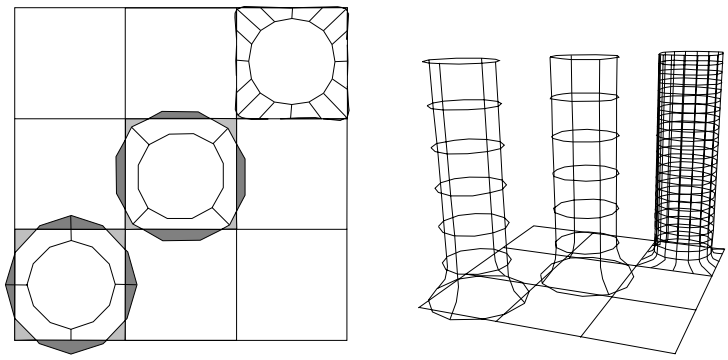
2.4 Arbitrary Topology Attachments

Although it is possible to create a seamless character from a single cylindrical or toroidal surface, this requires severe distortion of the surface, so that is not particularly well suited to manipulate or animate. The parameterization of the surface is likely to be the source of problems, for example when texturing the surface. Limbs and articulations are going to be particularly problematic under such a scheme [Maestri 96].

Animators have devised many approaches to model these regions from separate surfaces, in particular *NURBS trims and blends*. The technique is quite simple: the limb and the body of the character are modelled separately using NURBS. The curve at the extremity of the limb is first projected on the body (and possibly scaled), and then used as a trim curve. Finally a piece of geometry is created, called a *blend surface*, to smoothly join the area of the body around the trim curve to the limb.

Although this technique works well enough for modelling, the animator runs into problems when he tries to move the limb in question. While sophisticated animation packages support animated blends, which generate a new blend surface at each frame for the different positions of the limb, the animator has little or no control on the blending surface. If the surface does not look right it can be modified, however on the next frame the modifications are lost, when the blend surface is regenerated. This usually forces the animator to either tweak the blend surface at every frame—a tedious task indeed—or to simply try to hide the joint from view, which limits the usefulness of the geometry.

The reason attachments are so hard is that the attached limb has cylindrical topology, and we are not trying to attach, in general, to an isoparametric line of the body. Arbitrary curves on a bicubic surface are represented in space by curves of



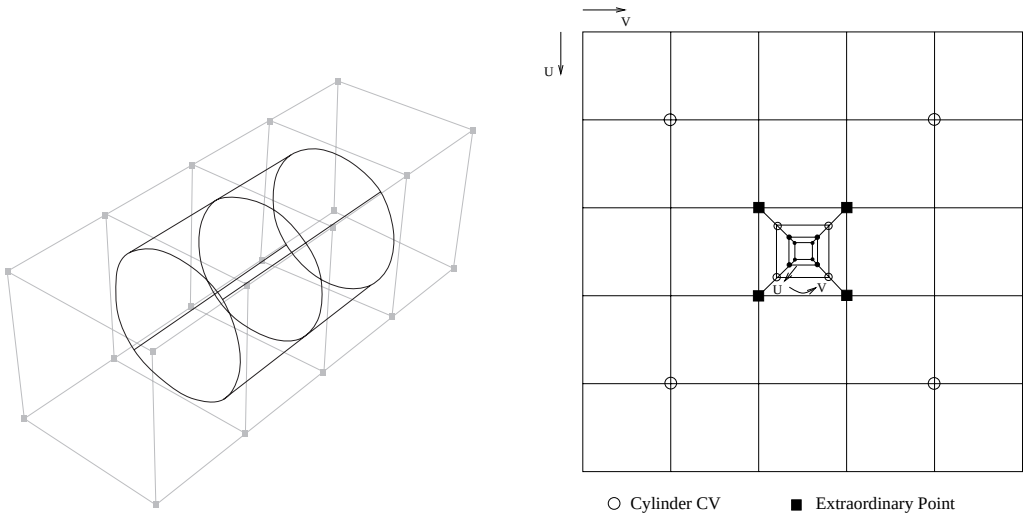
(a) Different attachment schemes

(b) Perspective view

Figure 2.2: Attaching a cylinder to a square patch

degree much higher than 3, and they can't be fit with the edge of a cylinder. Trying to cover the region of attachment with patches will result in gaps and overlaps, despite the parameterization of the attached surface, as shown in figure 2.2. It is impossible to maintain a tensor-product surface across the seam, since some CVs will have more than 4 incoming edges (figure 2.3b).

Hsplines use a different approach, relying on subdivision surfaces, which are described in section 2.4.1. The CVs of the limb and body in the region of the attachment now define the piece of geometry covering the seam. The advantage of this method over animated blends is that the region of the joint can be manipulated directly like other surfaces. Moving the limb affects the surface at the joint rather than just affecting the projected curves that define the blend. An added benefit is that local refinement is possible across the joint, improving control from the animator's perspective.



(a) Bicubic Cylinder and its CVs

(b) Top view of attachment constraints

Figure 2.3: Constraining the cylinder's CVs to the parent surface

2.4.1 Subdivision Surfaces

Catmull-Clark surfaces [Catmull 78] are defined over polygon meshes of arbitrary topology by recursive subdivision of the mesh. The subdivision rules were chosen so that the resulting surface is continuous in tangent and curvature almost everywhere. The process generates tensor product B-spline patches over the mesh where it exhibits a quad structure, except around a finite number of *extraordinary points*, which have $n \neq 4$ incident edges. The surface at the those points is defined as the limit of the subdivision process, where it is continuous in position and in tangents, and it is either discontinuous in curvature (for $n = 3$) or the curvature vanishes at these points (for $n > 4$). [Halstead 93] describes a closed form solution to the position and normal of the surface at the extraordinary point. Further properties of subdivision surfaces are discussed in [Doo 78] and [Ball 88].

The exact subdivision rules are based on an extension B-spline refinement,

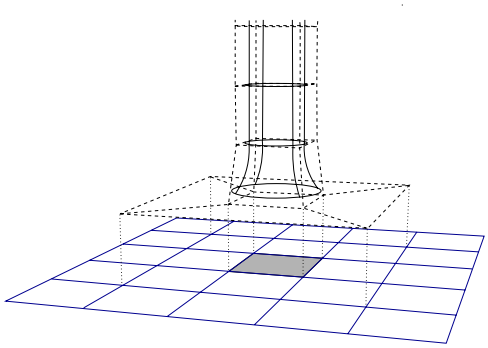


Figure 2.4: Constrained CVs across cylinder and parent patch

and are described in [Catmull 78]. The number of extraordinary points does not increase with successive iterations of the subdivision algorithm, and after the first step all the polygons of the mesh are guaranteed to be quadrilaterals (quads). [Dyn 90] and [Doo 78] explore alternative subdivision schemes with other interesting properties. In this application we will be dealing with $n > 4$ at the extraordinary points, so that curvature is not discontinuous. The subdivision process will be illustrated after some preliminary material is presented.

We will refer to the surface to which the cylinder attaches as the parent surface. The rim of the cylinder is attached to a rectangular area of the parent surface by constraining CVs from the cylinder to CVs of the parent surface. The combined set of CVs is then used to define a subdivision surface that covers the jointed area. There is a restriction on the attachment: only the coarsest level of the cylinder can be attached to the parent surface (finer levels of detail will stride the subdivision surface as expected).

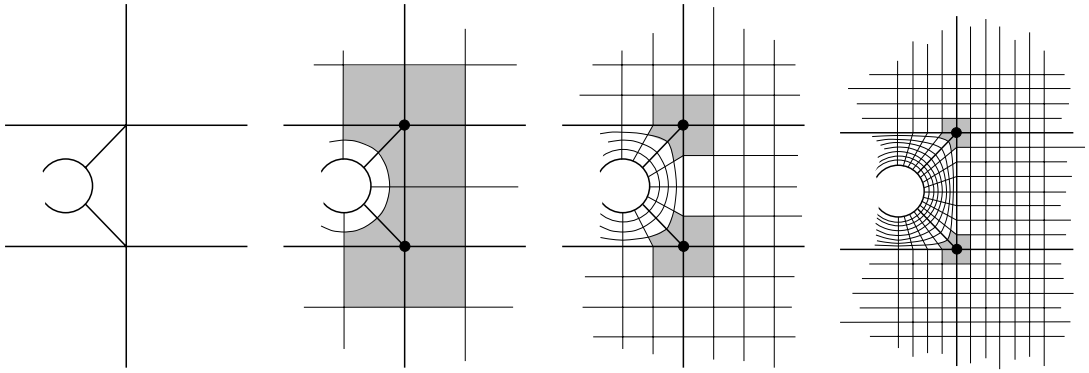
Figure 2.3 shows how the CVs of the cylinder are constrained to those of the parent surface. The only CVs affected are those immediately in the region of the attachment. The next to last row of CVs from the cylinder, which corresponds to the rim of the cylinder in the case of a bicubic patch, is constrained to the CVs

surrounding the rectangular region where the attachment takes place (see figure 2.3 and 2.4), thus creating four extraordinary points (these CVs have now $n = 5$ incident edges). In this case all the polygons are quads even before the subdivision process begins. The control graph of the cylinder and its parent surface must match, but they can be larger than the 1×1 example depicted in figure 2.3b and 2.4.

The last row of patches on the cylinder is not rendered, since they are to be replaced by subdivision surfaces as discussed above. Under these circumstances it does not really matter where the last row of CVs from the cylinder goes, because they do not influence the rendered part of the cylinder. In practice however, many modelling packages, including Softimage, do not support subdivision surfaces. The artist is thus forced to render the frame (assuming that the renderer does support them) to look at the joint. While this is unavoidable, we can provide an approximation to the subdivision surface quite easily by constraining the CVs from the last row of the cylinder to the corner diagonally opposite from where the CV on the previous row is constrained (see figure 2.3b), and displaying the last row of patches from the cylinder. The resulting row of spline patches can be seen in figure 2.4.

Figure 2.5 illustrates the first few steps of the subdivision process at the junction of the cylinder and its parent surface. Only two of the four extraordinary points are shown in the figures. The highlighted region in each figure represents the area that cannot be represented by B-spline patches. As shown, the Catmull-Clark surface becomes smaller at each iteration of the subdivision process.

Once the subdivision surface has been created, the user can keep working on the model using the usual editing tools. It is possible to refine the surface across the Catmull-Clark patch since the process used to generate the surface itself relies on subdivisions. This means that the consistency of the interaction model with the



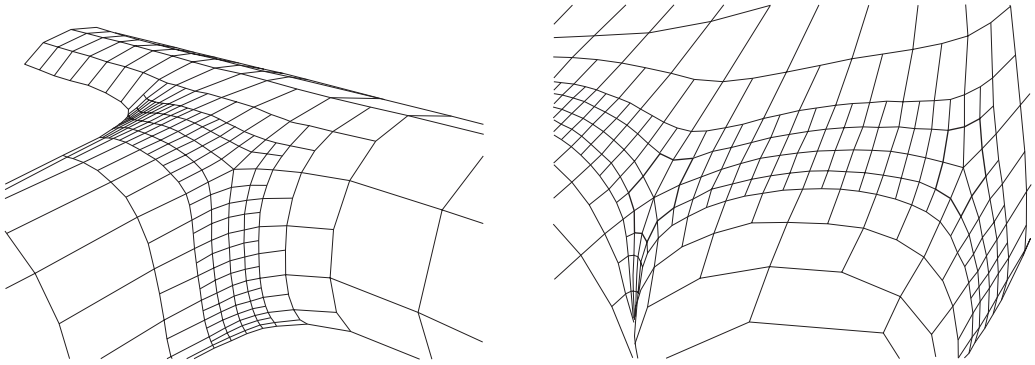
(a) Original (b) After 1 step (c) After 2 steps (d) After 3 steps

Figure 2.5: Illustrating the subdivision process

user is not compromised by the attachment, unlike the blending surface solution to this problem.

Figure 2.6a has two cylinders attached, and one of the extraordinary point is clearly visible, with local refinement. It is also possible to attach cylinders next to each other on the parent surface. This feature is very useful when trying to attach fingers to a hand, for each finger influences the attachment region as it is moved. Figure 2.6b shows what is happening in that case, and there are $n = 6$ edges at the shared extraordinary points.

It should be noted that texture coordinates are defined non-ambiguously across the Catmull-Clark patch, based on both the cylinder and the parent surface texture coordinates, since the new surface is recursively defined in terms of the old ones. However there is a sharp discontinuity in coordinates at the extraordinary point (figure 2.5). The direct consequence is that the two textures associated with this region must blend properly at the extraordinary point if the change in texture is to go unnoticed.



(a) Cylinder attachment with refinement

(b) Fingers attached to the palm of a hand

Figure 2.6: Examples of arbitrary topology attachments

Given the advantages of subdivision surfaces, it is legitimate to ask why they were not used instead of B-spline patches to build hierarchies, as suggested in [Zorin 97]. First off, in this project we wanted to integrate hierarchical modelling into the Softimage|3D framework. Since neither Softimage nor Mental Ray, its renderer, support subdivision surfaces it would have been very hard to use hierarchies based on them. Textures are also more difficult to use on surfaces with complex topologies. Finally, editing tools like trims and fillets cannot work without first converting the region in question to a collection B-spline patches.

The best solution seemed to use the smallest Catmull-Clark surfaces that would cover the arbitrary topology regions, and explicit B-spline patches for the rest of the model. Through subdivision, the Catmull-Clark surfaces can be made arbitrary small. This is useful in getting those regions to render when the renderer has no direct support for them. These options will be further explored in sections 3.4 and 4.3.

Chapter 3

Implementation

Hierarchical splines have been developed for the past ten years within an experimental modeller called Dragon, developed mainly by David Forsey both at the University of Waterloo and the University of British Columbia. This modeller has grown from a simple proof-of-concept experiment to a powerful system with many features that are not found in commercial packages; support for hierarchical splines and arbitrary topology being the most important of them.

Since research was the driving impetus, Dragon lacks many features that animators have come to take for granted. The most obvious features are arguably surface construction tools, *inverse kinematics* (IK), lattice deformations, curve editing features (especially in the animation subsystem) and rendering. Dragon supports bicubic B-spline patches but none of the other spline bases, nor higher order curves; this is somewhat unfortunate since the mathematical formulation of hierarchical splines has none of these restrictions. Finally little attention has been paid to the user interface which has grown idiosyncratic over the years.

This project aims to bring the innovative features of Dragon to Softimage,

to allow animators to use them in a familiar environment. Hsplines have been implemented as a *plugin*, a program that fits in the host package's framework by using a well defined API (Application Programming Interface). The main problem in achieving this goal is that the API provided by Softimage is far from being flexible enough to allow new primitive objects to be added to the system. This forced this implementation to work in terms of the underlying package's spline implementation. As will be discussed in the following paragraphs this approach had many drawbacks both in terms of efficiency and functionality.

3.1 Plugin Architecture

The hspline plugin for Softimage|3D is built around SAAPHIRE (Softimage Advanced API for Relations and Elements). SAAPHIRE's architecture is designed on the same model as an operating system kernel: everything happens through system calls; the core does not interact directly with data structures of the plugin, nor does it provide any memory management facilities.

The advantage of this design is that it is not possible for the user to keep pointers into data structures in the core that could become invalid through some action of the user (for example, keeping a pointer to a list of CVs after the surface to which they belong has been deleted). However this safety comes at a huge price: every system call requires the Softimage kernel to copy data to and from plugin memory. These costs are analyzed in section 3.1.1.

As of September 1996, SAAPHIRE was still an early version of the API intended for Softimage|3D. On the one hand its architecture was inherited from legacy code used in an earlier API, while on the other hand it had limited functionality in many respects. The implementation described here spanned several incarnations of

SAAPHIRE, from version 1.0 to 1.7. As SAAPHIRE matured, we were presented with functionality that became powerful enough to solve some of our problems, and this project provided a lot of feedback to the SAAPHIRE development team.

SAAPHIRE plugins (also known as *Custom Effects* in Softimage parlance) are implemented as DSOs (Dynamic Shared Objects) on IRIX, which can be loaded at run time through services provided by the ELF dynamic linker. Each plugin may define three entry points: an init function, which we will refer to as `init()`—the function name can be chosen by the programmer but must be unique among all plugins—that handles any initialization required by the custom effect; an update function, referred to here as `update()` that applies the custom effect whenever Softimage refreshes the scene; and a cleanup function, referred to as `cleanup()`, that can reclaim storage and do other housekeeping tasks after the custom effect.

There can be many instances of a custom effect so that the same effect can be applied to different models within a scene (multiple hspline objects). Each instance of a custom effect has a private data structure that can be used to keep track of internal state.

The lack of a more flexible callback structure burdens the plugin, since essentially the `update()` function is the only entry point for all operations on hsplines, forcing the plugin to determine the reason for being called. It should be noted that there are no callbacks prior to saving the surface to disk (but `init()` is called immediately after a load), there are no callbacks when the CVs of an object are selected (we would be able to prevent users to select unmovable CVs), etc.

3.1.1 Efficiency of Softimage's plugin model

Graphics data structure tend to be quite large, making the operating system model very expensive. A bicubic B-spline patch needs to have at least 16 CVs (and much more to be useful for modelling purposes). Assuming a `double` for every coordinate of every CV, we have to deal with $16 \times 3 \times \texttt{sizeof(double)} = 384$ bytes for a typical system. The amount information doubles if we require normals at these points and then there are also texture coordinates. A typical model would have something closer to 500 CVs, requiring the copying of over 11Kb from kernel space to plugin memory for the whole control mesh (without normals or texture coordinates).

This copying is required whenever a plugin wants to examine the CVs of a mesh, even if only to find out whether they have moved or not. If the coordinates are to be modified as well, they will also be copied back to kernel memory. This design issue has enormous consequences on the interactive performance of plugins: if a control vertex is interactively manipulated, the plugin which controls the geometry will be called repeatedly. On every call, it needs to retrieve the coordinates of the model in order to find out which CV is being manipulated, and to react appropriately to the edit. There can be 10 to 20 updates per second, as a CV is interactively dragged during a typical modelling operation. This requires between 110 and 220Kb of memory to be copied back and forth between the Softimage kernel and the plugin, even before the computational part of those updates can take place.

The only reason the system is responsive is that the topology of the surfaces involved in general modelling tasks require the user to break surfaces into different patches of different resolution. This allows plugins to query the different patches individually, somewhat reducing the data transfers (assuming that not all patches need to be examined). An alternative would have been for the kernel to send messages

to the plugin with the old and new position of the manipulated CVs, thus avoiding the querying of the kernel. A similar scheme could have been devised to update only some of the CVs that plugins need to modify in the kernel.

This shows that we should try to make as few calls as possible into the Softimage kernel given their individual cost. The hspline plugin implements an extensive caching mechanism to attempt to alleviate this problem. Also, wherever possible the plugin avoids to call on the services of the kernel. Computing tangents and normals is an example of operations performed more efficiently in the plugin than by going through the kernel.

3.2 Flow Control

Softimage plugins come in two flavors: *immediate effects* and *persistent effects*. This distinction is required by Softimage's model of updates in the scene graph. An immediate effect is a plugin that modifies the scene once and for all. For example we could write a 'noise effect' that perturbs the z-coordinate of an object with a noise function. The effect is immediate because after the plugin terminates it does not need to be notified when the user edits the perturbed object. An immediate effect is usually implemented completely in its `init()` function.

On the other hand a persistent effect is added to Softimage's update list and will be called whenever the object under the control of the plugin is updated in any way. It is possible to choose where to insert the plugin in the update list, which allows the plugin to see coordinates in an object's own coordinate system, or after they have been been modified by inverse kinematics for example.

The hspline plugin is a persistent effect, since it needs to enforce constraints on CVs and how they relate to underlying levels (see section 2.3). Others tools

related to hsplines are implemented as immediate effects: 'Zero Offsets' for example modifies the geometry and returns immediately. Whenever the user manipulates CVs from a patch under the control of the hspline plugin, the `update()` function is called. There is no way to to find out directly what triggered the update. A CV could have moved, but it could also be an IK transformation, advancing to the next frame, a deformation applied to the model, or any one of a multitude of operations that affect the model.

As described in section 3.1, it is very costly to determine whether an individual CV has been moved. All the CVs at all the levels of the hierarchy must be retrieved from the Softimage kernel, and compared with the copies kept by Dragon. Incidently this can make it costlier for hsplines than regular spline patches since all levels of the hierarchy must be examined. One solution to this problem would be to use the Softimage scene graph to control explicit updates, by associating a persistent effect with every level in the spline hierarchy. A CV modified at level j would call the appropriate plugin notifying it that the level under its control has been modified. Since a change at level j in the hspline affects the CVs at lower levels $(j+1, \ldots, n)$, we need Softimage to traverse the hierarchy towards the leaves to notify finer levels that the parent surface has been modified. This approach would at least minimize the number of CVs to retrieve from the kernel to the bare minimum.

The problem with this approach is that in Softimage updates propagate up the model tree, towards the root. This renders this approach infeasible, and requires a single instance of the hspline custom effect to manage all the levels of the Dragon hierarchy. This also proves to be a problem with enveloping, as described in section 3.5.

When a CV is modified in Softimage, either directly by the user or indirectly

through a tool, the `update()` function of the instance of the hspline plugin managing the surface will go through the following steps to try to determine the what triggered the update:

- The first step—which really has nothing to do with updates, but is required due to the lack of callbacks—is to check the consistency of the Softimage hierarchy of patches. The user can delete intermediate levels of the hierarchy, or to randomly reparent the patches without the `update()` function being called. This is very frustrating because the plugin must perform expensive checks on the data structures to ensure their consistency. The plugin is able to recover from most of the above user manipulations, except where the user deletes leaves and their direct parents. This is due to SAAPHIRE not being able to confirm whether a given object has been deleted (the object ID could have been reassigned to another object in the meantime). If an object and its associated label are deleted inside the hierarchy (i.e: the object has both a parent and a child) then it is replaced by a Softimage NULL, which makes it easy to detect.

- The second step is to check whether there has been a change in the frame number. This check is cheap and should therefore happen before querying all the CVs of the surface. If the frame number has changed and there is animation data associated with the surface (usually facial animation, see section 3.8), it is updated accordingly.

- Finally the CVs are checked against their last known location. All the patches are looked at, from the root of the hspline tree towards the leaves. When a CV is found to have moved from its earlier position we check whether it is movable

(see section 2.3). If the CV is movable, then the Dragon CV is updated by computing its new offset. If the CV is not movable, its position is modified in Softimage's data structures, according to Dragon's view of where it should be. When a user interactively moves a CV, the `update()` function is called often enough that the hspline is updated smoothly in Softimage's display. By the same token, if the CV is marked as unmovable, it will be updated often enough not to move in Softimage's display.

3.3 Mapping between Softimage and Dragon data structures

While we had a working implementation of hierarchical splines in the form of the Dragon kernel, there was a choice to be made as to whether to reuse the code that already existed or to try to write something designed around Softimage's view of hsplines. Since Softimage's update model does not provide a way to find what triggered the update, it is important for the plugin to keep track of the last known position of every CV, in order to be able to identify changed CVs. Given this need, and the fact that the plugin must also build its own data structures describing the constraints and relations between CVs, it made sense to start from the Dragon kernel which already provided all these features.

Softimage's view of Dragon's data structures is entirely limited to B-spline patches. At first this may seem convenient, but soon one realizes that it is possible to get a situation where a CV in Dragon is present on multiple distinct B-spline patches in Softimage. This is because Dragon deals with a three-dimensional mesh of CVs, whereas Softimage deals with individual surfaces, unaware of their connectivity.

Figure 3.1 depicts the situation clearly. The surface has been refined around three points as labeled, and the resulting mesh at the new level does not have rectan-

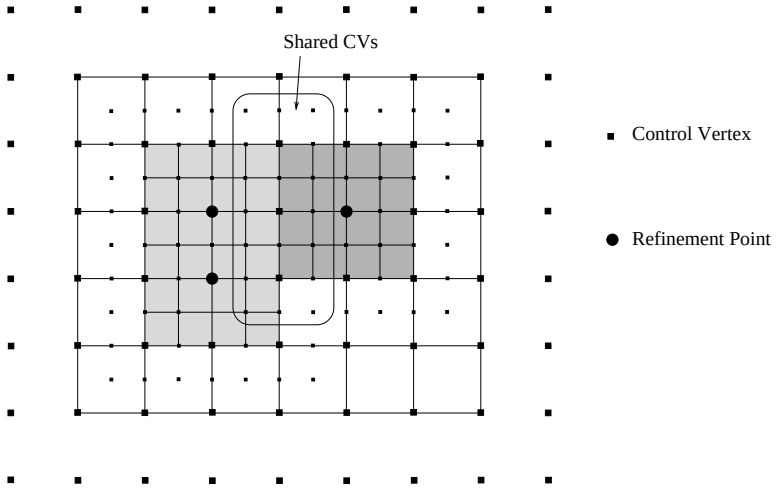


Figure 3.1: Shared CVs among Softimage patches

gular topology. That is, it cannot be represented with a single B-spline patch. One way of breaking the area in two distinct patches is presented in the figure. There are 21 CVs shared between the two patches. While Softimage has no notion of the underlying representation of the surface, the plugin must make sure that when the user explicitly moves a CV on one of the surfaces, the other one gets updated.

The way constraints are enforced among CVs in the Softimage scene is to build a translation table between Softimage's data structures and Dragon's. A control vertex in Dragon can be identified by a quadruplet $(\text{surf}, \text{level}, u, v)$ which describes the surface we are interested in, the required level in the hierarchy and the CV's u and v coordinates. This method is quite general but fairly expensive since it requires the traversal of a fair amount of data structures to locate a particular CV. A CV also has a unique identifier which can be used to locate it more efficiently. While the identifier is often the preferred way to identify a CV it must be noted that the identifier is not preserved across a save and restore of the data structures. In Softimage, individual CVs are identified by pairs (object, i) which represent an index

into an array of CVs, for a given object.

The hspline plugin must be able to translate from a given representation to any of the other ones. This is implemented as a hash table which, given any representations, can retrieve a list containing all the Softimage CVs as well as the Dragon vertex associated with them. What is actually retrieved in the case of Softimage is the pair of identifiers needed to request the CV from the kernel.

As described in section 3.1, the plugin needs to cache as much data as possible during a single update cycle to minimize the number of calls to the Softimage kernel. When updating a surface in Dragon we always recompute the position of affected CVs from the lowest level affected down the hierarchy towards levels with finer details. As the Dragon data structures are traversed in that order, the Softimage patches will be traversed in a totally different order. Since Softimage only allows to retrieve the control mesh for the entire patch, we would need to retrieve the same list of CVs many times over if it were not for caching.

The caching mechanism implemented retrieves the control mesh of a patch and keeps it around, in case it is ever needed again for other CVs of the same mesh. If the mesh is updated, the piece of geometry will be sent back to the Softimage kernel when the cache is cleared. If a given patch has not been modified, its control mesh will be simply discarded. Although this caching technique may sound obvious, it makes all the difference between a plugin that can be used interactively and one that can't.

3.4 Arbitrary Topology

Arbitrary topology is fully supported in the Dragon kernel. Softimage on the other hand currently lacks the major feature required to completely support Dragon's

implementation: neither the modeller itself, nor its renderer (Mental Ray) currently support subdivision surfaces.

Despite these issues, arbitrary topology can be added to the hspline plugin because the region around extraordinary points can be recursively subdivided, with the non B-spline region getting smaller at each iteration (see section 2.4). Thus the user can refine that area until the region not defined by B-splines is smaller than a pixel and can be represented polygonally. These are essentially the steps that the renderer would follow if it supported subdivision surfaces directly.

While this requires effort from the user, once the refinement is done the arbitrary topology attachment will behave as expected. It is possible to implement the refinement procedure in software, which would remove some of the burden from the user; but the downside is that the geometry is affected by the procedure. Whether this will be done or not will depend on the next major release of Softimage and its support for subdivision surfaces.

Arbitrary topology does not present any further technical difficulties than enforcing constraints among different B-spline patches. Dragon enforces these constraints with its own data structures, while the plugin keeps mapping them to Softimage, as with regular hsplines.

3.5 Enveloping

A *flexible envelope* is geometry whose CVs are assigned to the coordinate frame of a segment from an underlying skeleton. Each CV is usually influenced by a single segment, but it is not uncommon, in the area surrounding a joint, to use a linear combination of segments to better control the surface. The envelope can then be driven by the skeleton, either through *forward kinematics*, or inverse kinematics.

Softimage associates two complete sets of coordinates with each CV of each model in the scene. They are labeled *original coordinates* and *deformed coordinates*, and correspond respectively to the world coordinates of the object itself, and to the world coordinates of the object after undergoing deformations. Deformations are a broad class of tools that can manipulate models, to which both enveloping and lattices belong. Both sets of coordinates are accessible from SAAPHIRE.

Until deformations are actually applied to a model, its deformed coordinates are simply ignored. If a plugin requests the deformed coordinates of an undeformed model, the original coordinates will be returned. If a plugin tries to modify the deformed coordinates of an undeformed object, the modification will succeed, but will be ignored by Softimage, since it is not looking at these deformed coordinates.

The existence of two sets of coordinates poses a problem to the hspline plugin. It cannot try to work through the deformed coordinates, because the Softimage kernel ignores the updates made by the plugin to the deformed coordinates of an undeformed model. Similarly, working with original coordinate poses a problem on models that have been deformed since now the plugin and Softimage see the CVs in different locations.

In Softimage, enveloping works at the level of a complete surface, while hierarchical splines require only some of the CVs in a surface to be enveloped. This difference forces the plugin to keep track of whether a CV is enveloped or not, to decide whether to look at its deformed coordinates or its original coordinates. Currently, the only way to do this is for the user to provide this information explicitly by using an auxiliary plugin to mark enveloped CVs. Once this is done, both Dragon and Softimage will agree on the position of CVs in world space.

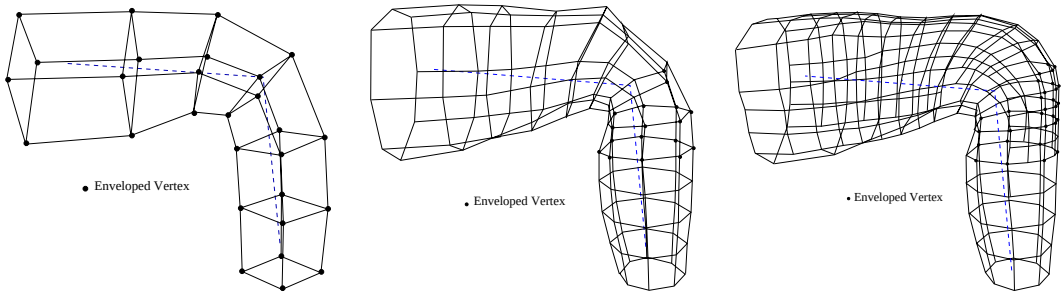
The hierarchical nature of hsplines interferes with Softimage's traversal of

the scene graph. Whenever the whole scene needs to be updated, Softimage sorts the nodes according to internal constraints, and selects parents before children when no other constraints are present. Then, as the nodes of the graph are traversed, all the transformations and effects that a node must undergo are applied, before the following node is updated.

Now lets assume that all the levels of an hspline have been enveloped, as Softimage insists on doing by default. As it traverses its scene graph, updating each node in turn, it will update level 0 of the hspline hierarchy. At that time the hspline plugin is called through its `update()` function, to adjust the lower levels of the hspline to reflect the new situation. As the traversal of the scene graph continues with the next level of the hspline, Softimage overrides the latest changes from the plugin with the envelope data that keeps propagating.

The second function of the enveloping auxiliary plugin is to notify Dragon that since the tagged CVs are enveloped they should not be updated when the underlying level is modified. Their offset is not from the parent surface anymore, but from the frame of reference of the segment which drives them. As far as Dragon is concerned there is no need to know where the frame of reference of the segment is located, since the coordinates from Softimage are in world coordinates, and we might as well assume that it is aligned with the major axes at the origin.

The hierarchical nature of hsplines however determines how the enveloping should be allowed to happen. In general you want to envelope the lowest level of the hierarchy that has the relevant features, and not envelope its children. To ensure that this is the case, when the "Envelope Initial Assignment" dialog box appears, the "Maximum level" box must be checked and set to 1, unselecting in the process the "No maximum" box.



(a) Enveloping at level 0 (b) Enveloping at level 1 (c) Enveloping at level 2

Figure 3.2: Hierarchical enveloping of a joint

Hierarchical spline enveloping allows very realistic joints to be created, by enveloping multiple levels in different ways. The collapse of the surface in the area of the joint may be avoided by refining the area at the joint and assigning CVs from the finer levels to the parent segment of the rotating one. The result is a very realistic joint, on which the skin bends reasonably (figure 3.2c), unlike the default Softimage technique (figure 1.2b) which envelops every vertex.

3.6 Auxiliary Plugins

Immediate effects were implemented to provide additional features to the basic hspline plugin, which only manages updates on the hspline surfaces. These tools are described from the user's point of view in appendix A. Here we will describe some of the implementation issues encountered within the framework provided by SAAPHIRE.

3.6.1 Inter-plugin communication

There are no explicit provisions in Softimage to allow plugins to share data. However all the plugins implemented interact with hsplines and their associated data structures. They need to be able to communicate with the persistent effect and retrieve the Dragon data structures in order to manipulate them.

DSOs share the address space of the process that loaded them. This means that all the plugins are running in the same address space, and all that is needed to retrieve data associated with a particular hspline is to be able to somehow obtain a pointer to it.

Starting with SAAPHIRE version 1.1 it is possible to attach arbitrary data to objects in Softimage. This data is referred to as *user data* and is managed by plugins. SAAPHIRE considers the data as an array of bytes and will make sure that it is never dissociated from the object it was attached to, until either the object is deleted or the plugin detaches the data. In particular, user data is preserved across a save and restore cycle.

A pointer to the Dragon data structures is attached to every surface at the time they are created, either explicitly or through refinement. This pointer can be recovered by all the plugins that request it, thus enabling them to get access to Dragon's data structures. Once the pointer has been recovered, the individual plugins are able to manipulate the data structures directly.

3.6.2 Refining and Unrefining

Refinement affects the geometry of a model in a significant way. New patches are created, corresponding to the new level in the hierarchy. If we were to leave it at that, the organization of the patches that correspond to a given level would depend

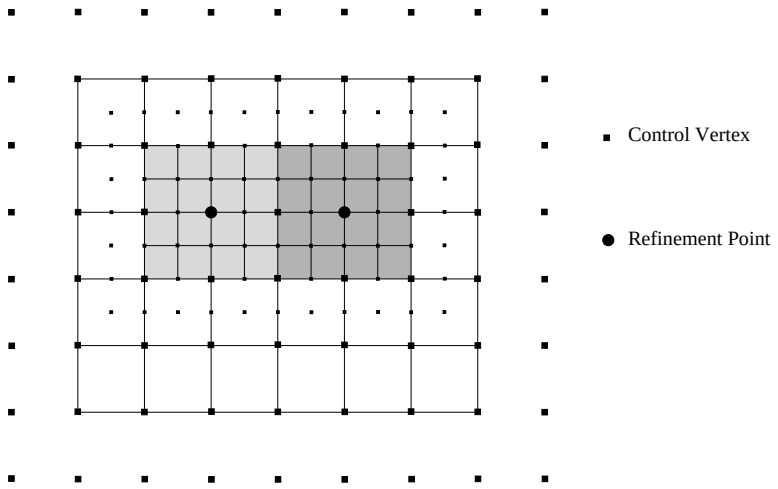


Figure 3.3: Minimizing the number of patches in Softimage

on the order in which the parent surface has been refined—i.e: a single patch at level $n+1$ if all the CVs at level n were refined at the same time, or several patches at level $n+1$ if level n was refined in several steps.

Referring to figure 3.3, it is easy to see that the two patches obtained after refining around two different CVs could very well be represented by a single patch without loosing information. Despite this discrepancy in patches, both configurations define the same surface, and it will behave in the same way when edited (see section 3.3).

The hspline plugin attempts to minimize the number of patches it creates in the Softimage scene. This guarantees that the representation of a given topology is independent of how the user arrived at that result, provides a more consistent view of the geometry to the user and makes the control mesh easier to visualize. It is also more efficient for Softimage to deal with a small collection of large patches, than the opposite.

The set of heuristics that determines the arrangement of patches not only tries to make the patches as large as they possibly can be, but also tries to preserve

as many of the existing patches as possible. The reason for this is that once a model has been created there can potentially be animation data associated with it, some of which is not be accessible to plugins. If we were to simply discard the patches and create new ones all that information would be lost. While it is certainly still possible to loose some information when reorganizing the patches, practice has shown that some of the animation data could be salvaged.

In general, animation systems make no guarantees on the result of modifying models which have already been animated. The hspline plugin makes a stronger commitment: only the level at which patches are generated (and possibly its children) can potentially loose animation. The parent levels will still behave as they did before the refinement. The new patches are also constrained to follow the parent patch during the animation, since they are part of the hspline hierarchy, which minimizes the impact of refinement on the animated sequence.

Unrefining a model is even more drastic than refining it. The topography of the model is simplified, by removing all the CVs whose offset from the parent surface is zero, constrained by the mathematical definition of bicubic B-spline surfaces, as described in section 2.3. Thus all the levels of the surface can possibly be affected, unlike in the refinement case. While this does not affect the surface defined by the hierarchy, it deeply modifies its representation, thus destroying all the animation data associated with the surface. No attempt is currently made to minimize the impact of unrefinement, even though it is possible that the surface representation remains unchanged by the operation.

3.6.3 Load and Save

Users expect to be able to save a scene containing hsplines, and be able to continue working on the scene the next time it is loaded in Softimage. The main problem with that scenario is that Softimage does not provide a way for the plugin to know that the data it manages is being saved to file. The plugin's `init()` function is called once the scene has been restored from file to initialize itself. The patches have already been recreated properly and animation data has been reattached to the models in the scene, but all the internal data structures managed by the plugin have been discarded. It must be in a position to recreate its internal data structures and match them to the patches in the scene from the information restored by Softimage.

Given the cost of manipulating the user data attached to a model, we must again make sure that it is updated as little as possible. This is handled by attaching to the whole hierarchy a description of Dragon's data structures, so that they can be recreated once the scene is restored from file. In essence, we are saving the hspline to a `.hs`[1] file, which is attached to the model. This needs to be updated every time there is a change in topography (refine and unrefine), but not for every update from the system. The assumption is that refinement is an operation that is not performed very often, since it dramatically alters the model.

When the plugin's `init()` function is called, it reads the `.hs` file attached to the hierarchy and recreates its internal data structures. It then needs to be able to map between the Softimage patches and its internal data structures to rebuild the hash table. To that end, the plugin attaches information to every CV in Softimage, at the time patches are created. The information contains the texture coordinates of the CV, u and v, as well as the level to which the CV belongs. If Softimage directly

[1] `.hs` files can be read by the Dragon kernel, shared among all implementations of hsplines.

supported texture coordinates[2] no information would have to be attached to CVs: given the texture coordinates and the depth of the patch in the hierarchy we would be able to retrieve the Softimage vertex matching any Dragon vertex.

As mentioned in section 3.6.1, there is also a pointer attached to each patch that permits to retrieve the Dragon data structures in memory from the patch. As the patches are read from file, and the plugin is reloaded into memory, these pointers must be updated.

Finally, the position of the CVs must be updated in Dragon, to synchronize it with Softimage's display. Since the file is attached to the hierarchy only when a topological change occurs, all the manipulations of the surface that have occurred after the attachment are unknown to Dragon when it restores its data structures from that file. All that needs to be done is to traverse the restored data structures from the top of the hierarchy down, recomputing the offset of every CV with respect to its parent level.

Besides loading and saving the hspline data with the Softimage scene, it is also possible to export the model to a `.hs` file, which can then be read by other implementation of hierarchical splines. An option is provided to optimize the model for space, which removes all zero offset CVs from the description of the surface, just like the surface obtained through unrefining. Although this is the default in Dragon, it isn't in Softimage as the hspline which gets loaded is different from that which was saved—by different we mean that the hierarchy has been simplified, not the resulting surface. Since Dragon does not support the unrefine operation it made sense to provide it every time the surface was saved to disk, but this is not necessary in Softimage.

[2]This is issue is discussed in section 4.2.3.

3.6.4 Support Tools

While hierarchical splines and arbitrary topology attachments are the two major contributions of this project to Softimage, it should be noted that for the convenience of animators a small number of *auxiliary plugins* were created to assist animators in their use. They will be described briefly here.

- **Save→Export Hspline+**[3] allows to export the hspline surface to a `.hs` file, which can be read by other implementations of hierarchical splines.
- **Edit→Hspline Refine+** refines the selected objects around the tagged vertices.
- **Edit→Hspline Unrefine+** tries to remove redundancy in the model by getting rid of refined areas that are not used. Every CV whose offset is zero (i.e: the CV has not moved away from the underlying level) can potentially be removed. The mathematical formulation of bicubic hsplines however requires a 7×7 mesh of CVs centered on each non-zero offset, which means that in the worst case a single CV is surrounded by 24 CVs that cannot be removed by unrefining. This tool will remove all the redundant CVs from the model, while satisfying the above condition.
- **Edit→Zero Offsets+** will zero the offset of the tagged CVs, so that they lie on the underlying level. If there are no finer levels with non-zero offsets, a subsequent **Edit→Hspline Unrefine+** will remove these levels from the surface, within the constraints stated above.
- **Edit→Set Envelope+** will mark the tagged CVs as enveloped. As described in section 3.5, this is a necessary step, if enveloped hsplines are to behave

[3] This notation means that the 'Export Hspline' command is found under the 'Save' menu. The + symbol is used in Softimage to indicate that the selected item is a plugin.

properly.

- **Tags→Tag Movable+** will tag all the CVs on the selected surface that can be moved by the user.

- **Tags→Untag Fixed+** will untag all the CVs that cannot be moved due to mathematical constraints from the set of tagged CVs on selected items.

3.7 User Interface

The hspline plugin attempts to integrate itself in Softimage at the lowest possible level, in order to work with all the tools already present in the modeller. At the same time it attempts to provide a straightforward interaction model to the user. The only way to reconcile these two goals would be to have a very flexible environment in which plugins are first-class citizens, permitted to set their own interaction model with users.

Unfortunately SAAPHIRE was not designed with that kind of flexibility in mind, and the hspline plugin must act through objects already present in Softimage to define its own interaction model. This issue has been the source of much experimentation ...

Before the advent of SAAPHIRE version 1.1 and the appearance of user data that could be attached to CVs of a model we were restricted to using a polygon mesh to control the splines, as this was the only way to enforce the constraints on the control mesh. We eventually settled on the collection of B-spline patches described in section 3.3, but this is not completely satisfying since it is not possible to interact with the finest representation of the surface without having the internal levels clutter the work space. Appendix A illustrates the current interface, while

section 5.1.1 describes another approach that may be implemented in the future, but even that approach will not be as satisfying as the one adopted in Dragon.

3.8 Integration into the production environment

The hspline plugin for Softimage is the central piece of software written to implement the hspline production pipeline mentioned earlier. A complete set of tools was produced, and they need to be able to communicate with each other to exchange data. The surface hierarchies are described in `.hs` files. This provides a mechanism to exchange data among the different tools, and allows them to remain independent. All the steps preceding animation can therefore take place outside the Softimage environment and we shall not be concerned with them here.

The main exception is the production's proprietary facial animation system, which interacts directly with the hspline plugin. This tool is external to Softimage, and allows the animator to directly animate facial expressions and lip sync. The sequences thus created must then be integrated into the scene managed by Softimage. A library has been written which permits the hspline plugin to update character heads in the Softimage scene using the tracks created by the facial animation system. The advantage of this arrangement is that animators are able to use a well adapted tool to animate faces, and use Softimage to animate the characters themselves.

The production required most of the character motion data to be gathered using motion capture. The motion capture data can be imported in Softimage and used to drive a skeleton. The skeleton is then enveloped with hspline surfaces and drives the animation of the surface, as described in section 3.5.

Chapter 4

Inherent Limitations

Trying to get two pieces of software which understand different concepts to work together is definitively a devious task. This section attempts to analyze the problems and limitations found in the components used, independently of the actual effort to get them to work together (this was discussed in chapter 3).

Dragon and Softimage|3D both have faults of their own, but there was a significant advantage in favor of Dragon: its source code was available. This allowed most of the problems found with Dragon to be fixed promptly, while we were forced to rely on the Softimage development team to fix problems in their modeller. Needless to say, most of the fixes requested never made it in the product during the lifetime of this project. This partly explains why most of the limitations currently lie with Softimage rather than Dragon.

4.1 Dragon

Dragon's code lacks modularity with respect to the spline basis used. As described in [Forsey 88], there is nothing in the formulation of hsplines that limits them to

the uniform B-spline basis. The current version of the Dragon kernel does not make it easy to provide different bases, however it can be argued that uniform bicubic B-splines, when complemented with arbitrary topology attachments, are flexible enough for character animation purposes.

With our implementation committed to uniform B-splines, we had to provide a unique user interface to hsplines inside Softimage. Had the flexibility been present in Dragon, hsplines could have been presented much like spline patches; the dialog boxes related to hsplines being similar to those used for regular spline surfaces that are familiar to the user.

4.2 Softimage|3D

Softimage currently has many limitations, most of which can be traced to the lack of maturity of some of its components. Both SAAPHIRE—the plugin API—and Mental Ray—the renderer—are recent additions to the Softimage|3D package. SAAPHIRE currently offers access to a subset of the functionality present in Softimage, and it might gain in flexibility in future releases.

4.2.1 Saaphire architecture

SAAPHIRE is impeded by the cost of system call to the Softimage kernel, as described in section 3.1. This is the result of architectural decisions made during the design of the API, and isn't expected to change in future releases. This architecture offered some fairly obvious advantages to the designers of the API: plugins can never have pointers inside Softimage data structures, so their contents can only be changed through the published API, with the guarantee that the modeller can check the sanity of the values written. This also ensures that the maintainers of the modeller

will be free to modify these internal data structures as needed in future releases of the software.

These design decision make for an API that spends much of its time copying data to and from plugin memory. Although the impact can be somewhat minimized by implementing extensive caching mechanisms, intended to minimize the number of system calls into the kernel, it definitely burdens the plugin programmer for the convenience of the Softimage developers.

SAAPHIRE does not allow plugins to animate parameters under their control. *Motion graphs* (called *fcurves* in Softimage) cannot be attached to an arbitrary animated parameter. The problem is that hsplines are really controlled by their offsets, and there is currently no way to keyframe them. If fcurves could be associated with offsets the hspline surfaces would behave much better in Softimage. Currently one must make sure that all the levels are accounted for when keyframing the surface, which puts the burden on the user.

4.2.2 Plugin entry points

The aspect of SAAPHIRE which proved most inappropriate for our purposes was the lack of entry points to the plugin. With only three functions available (`init()`, `update()` and `cleanup()`), two of which used for bookkeeping purposes, we are left with a single function that needs to do a lot of extra work every time it is called to make sure that the state is consistent. This further adds to the delays caused by the cost of system calls. The following additional entry points have been suggested to the SAAPHIRE development team:

- `save()` would be called before the geometry managed by the plugin is saved to file, with the rest of the scene. This would have allowed the plugin to

be conceptually much simpler than it is now. The workaround described in section 3.6.3 could have been avoided, and the refinement operation would have been significantly faster.

- `select()` would be called whenever the user tags CVs associated with geometry managed by the plugin. This would have allowed the plugin to prevent the user from tagging unmovable CVs making for a more consistent user interface. It would also have made both of the following auxiliary plugins unnecessary: **Tags→Tag Movable+** and **Tags→Untag Fixed+**.

- `delete()` would be called whenever geometry managed by the plugin is deleted by the user. The plugin should have the option of refusing to let the user delete the geometry, or decide to terminate (in which Softimage would automatically ensure that the `cleanup()` function is properly called).

- `render()` would be called whenever Softimage is ready to dump geometry managed by the plugin to the renderer. The plugin should have a way to write out specific commands directly to the renderer, and possibly prevent Softimage from dumping the geometry (in case the plugin does this itself, or the geometry is not intended to render).

- `deformed()` would be called after Softimage envelopes geometry managed by the plugin. This would allow the plugin to retrieve the assigned weights, and possibly modify them. The hspline plugin would then be able to work without support from the **Edit→Set Envelope+** auxiliary plugin.

Since the `update()` function is the only one called during normal operations of the plugin, it serves many different purposes. A mechanism through which it could

find what triggered the update event would help in keeping conceptually different parts of the code separate.

In particular, it is currently very costly to determine which CVs of the hspline are being interactively manipulated by the user. A mechanism which could identify those CVs readily would certainly be helpful. If the mechanism was able to provide both the old position of the control vertex and the new one, the hspline plugin could be redesigned to be much smaller, discarding most of the Dragon kernel. Movable CVs would be updated to their new position, while unmovable CVs would be updated to their old position. A side benefit would be that plugins are now able to find out how fast a CV is being moved. This would be useful, for example, in implementing automatic generation of secondary action through dynamics.

4.2.3 Texture coordinates

One of the most surprising aspects of Softimage, is that it does not support explicit textures coordinates associated with B-spline patches. Implicitly, all patches in the scene are parameterized from zero to one along the main local axes of the patch. Theoretically this should not be a problem since we can provide a texture for each patch, obtaining the same result than with explicit texture coordinates for each patch.

In practice however, an object composed of multiple patches cannot be associated with a single texture. Instead, the texture must be cropped to match the area covered by each patch, and each of these must be associated with the appropriate spline patch. This is much less convenient than having a single texture and associating texture coordinates to the different patches in such a way that they each map to a different area of the image.

In the case of hsplines, this alternative approach would have two significant advantages. First, as described in section 3.6.2, whenever refinement occurs existing patches are rearranged to minimize the number of new patches required to represent accurately the hspline in the Softimage scene. All the patches that are modified in this fashion need to be associated with a new custom cropped texture. The second significant advantage of using a single texture is that when an hspline model is viewed at a lower resolution than the maximum level of detail in the model, the areas that would be hidden by finer detail and are now exposed can be mapped to the same texture, giving the user a good idea of the surface being deformed.

4.2.4 Scene Graph

The Softimage scene graph is unfortunately very poorly documented in the current SAAPHIRE litterature, and some aspects of its behavior are surprising. Updates in the graph in particular do not behave as one would expect, since changes to a node do not propagate towards the children of the node. This is surprising since the state of the children is partly dependent on the parent (graphics context, cumulative transformation matrix, etc).

As described in section 3.2, updates have been found to propagate towards the root of the hierarchy, as far as the plugins are concerned. This forces a single plugin to handle the whole hierarchy of B-spline patches. If instead updates propagated down the scene graph, we could associate one instance of the plugin with each level of the hierarchy. This would make sure that only levels which have been modified and their children need to be updated, limiting in this manner the number of calls to the Softimage kernel required to retrieve the geometry.

Another issue was raised when the behavior of enveloping was examined in

section 3.5, for after the skeleton controlling the skin is updated, the whole hierarchy of patches making up the envelope is traversed top-down. This required the intervention of an auxiliary plugin to notify Dragon that some of the CVs are enveloped, and that they are not controlled by the parent level anymore.

4.3 Mental Ray

Mental Ray is a mature product in itself, and it has just recently been added to the Softimage environment. Most of the limitations experienced arised from its partial integration in the environment, with one exception. Mental Ray does not currently support subdivision surfaces. This can be worked around, since Catmull-Clark surfaces can be recursively subdivided into B-spline patches, however this functionality really belongs in the renderer. Section 3.4 discusses a possible workaround that may be implemented if the next version of Mental Ray offers no native support for subdivision surfaces.

SAAPHIRE currently offers no help in interfacing to the renderer. This is a problem since you do not want to render all the defined B-spline patches that make up a hierarchical spline. Areas that have been refined should not render, since there is a finer representation for them. It is wrong to assume that the finer surface will hide the lower level, for it could very well intersect the underlying level and be hidden behind it. This also means that it is currently difficult for the user to visualize her work at the finest level of detail, by rendering the frame without hiding the underlying levels.

Section 5.1.1 offers a possible solution to this dilemma using NURBS and trim curves, but what is really needed is for the plugin to be able to supply the geometry to be rendered to Mental Ray, independently of the data in the scene. Currently,

this can only be done by forcing the plugin to dump the collection of patches that represent the hspline to a file, then having the user hide the geometry in the scene, so that Softimage won't try to add it to the scene description itself. This requires an additional auxiliary plugin that must be applied to each hspline model in the scene before starting the renderer, an inelegant way to solve the problem.

Chapter 5

Conclusions and Future Work

This thesis has described how hierarchical B-splines were integrated into the Softimage|3D environment, using the SAAPHIRE API. Three major features were implemented: basic support for hsplines, arbitrary topology attachments and skeleton enveloping. Those features were used daily in a feature animation production environment for several months and have triggered a lot of positive responses from the animators working on the project.

There was one major issue left to be resolved at the close of this project: how to render the hsplines. This is conceptually simple to accomplish, but we were prevented from doing so by the poor integration of Mental Ray in the Softimage environment. Since Mental Ray is the latest addition to the package, this issue is expected to be resolved shortly after the next release of Softimage. A temporary solution using NURBS will be proposed in section 5.1.1, which also has the potential of significantly improving the user interface to hsplines, and this possibility should be explored.

Besides rendering there are several improvements that could be made both

to the tools already present and the user interface. More feedback would be required from animators in order to assess which directions to take next.

5.1 Possible Improvements and Future Extensions

This section describes possible extensions to the hspline plugin, which would improve the user interface to hierarchical splines. The first section suggests a major architectural change in the plugin, while the following section suggests auxiliary plugins that could be implemented. It is expected that as animators work with hsplines in Softimage more changes and improvements will be suggested.

5.1.1 NURBS patches

The Softimage implementation of NURBS supports trim curves both in the shaded viewports and the renderer. As described in section 2.2, NURBS are a superset of the B-splines used to represent hsplines in Softimage, they could indeed be used instead of B-splines. The hspline plugin would have to enforce the following constraints to make sure that they are strictly equivalent to B-splines:

1. The knots must be uniformly spaced, and the user should not be allowed to change the parameterization of the surface.

2. The weight associated with each knot is 1.0, and may not be modified by the user.

Within these restrictions, the surface is identical to that represented by the collection of B-spine patches, with one major distinction: trim curves may now be used wherever the surface has been refined. It is now practical to view the surface

represented by the hspline in the viewports, without seeing the underlying levels, since they have been trimmed off.

Although this sounds like an attractive alternative, the potential issue of the renderer must be studied before this can be attempted. The renderer must make sure that the boundary of the surface at the trim curve will be tessellated in the exact same way as the NURBS patch that fills the hole, despite their different parameterizations. If the tessellation is not consistent, some pixel along the boundary of the trim region will end up not belonging to either surfaces, due to numerical imprecisions, and will show up black in the rendered frame. Although Mental Ray is apparently able to handle things properly when told to, plugins are currently not able to communicate directly with the renderer. As a last resort, it is possible to run Mental Ray in a mode where it tests curves and surfaces against each other, to make sure that boundaries are always tessellated consistently, but this adds a terrible computational burden to the renderer, which will in general not be tolerated.

5.1.2 Auxiliary plugins

The following auxiliary plugins could possibly be implemented, on top of those described in section 3.6.4:

- **Tags→Tag along U+** and **Tags→Tag along V+** would select all the CVs on the same parametric line as the CVs in the current selection set. This functionality has proved useful in Dragon in a slightly different form.

- **Tags→Tag Neighbours+** would tag up to 8 CVs surrounding every CV currently tagged. This allows to quickly tag large areas of a model, by repeated use of the plugin.

This is different from using the rectangle selection tool to tag vertices, since it tags all the CVs falling in the rectangular area, regardless of the portion of the surface they belong to. When trying to tag CVs on the surface of a cylindrical object for example, one usually ends up selecting CVs from both the front and the back of the cylinder, half of which must then be unselected.

- **Tags→Tag Non-Zero Offsets+** would tag all the CVs of the selected patches whose offsets from the parent level is non-zero. This may be followed, if needed, by **Edit→Zero Offsets+** and **Edit→Hspline Unrefine+**.

- **Tags→Tag Refined CVs+** would tag all the CVs of the selected patches which have been refined. This is a useful visualization tool when trying to understand the structure of an hspline surface before animating it.

- **Edit→Toggle Hspline Symmetry+** would allow all changes made to one side of a symmetric surface to be reflected on the other side, when symmetry is turned on. This is another feature that has proved to be quite useful in Dragon, but which is currently absent from the Softimage implementation.

- **Edit→Hspline Unrefine+** should be improved to minimize the impact it has on the hspline representation in Softimage. As described in section 3.6.2, the current implementation will loose all the associated animation data, even if the surface is not modified in the process. It should also be possible to set the level at which to start the unrefinement procedure.

- **Edit→Remove Transforms+** would make sure that all the levels of the hierarchy render in the same reference frame as level 0.

  It is currently possible to translate and rotate individual levels, to allow users

to edit surfaces in a less cluttered area of the scene, if required. Although this may be convenient for modelling and animating, it must be avoided at rendering time, or the rendered surface will not look as expected. By applying this effect to the hspline before rendering, all the levels will render in the same reference frame (as long as the transformations on the individual levels have not been keyframed).

Bibliography

[Ball 88] A. A. Ball, D. J. T. Storry, "*Conditions for Tangent Plane Continuity over Recursively Generated B-Spline Surfaces*", ACM Transactions on Graphics, **7**(2), pp. 83–102, 1988.

[Barghiel 95] C. Barghiel, R. H. Bartels, D. R. Forsey, "*Pasting spline surfaces*", In M. Dæhlen, T. Lyche and L. L. Schumaker, editors, "*Mathematical Methods in computer aided geometric design III*", Academic Press, 1995.

[Barry 88] P. J. Barry, R. N. Goldman, "*A recursive evaluation algorithm for a class of Catmull-Rom splines*", Computer Graphics (ACM SIGGRAPH '88 proceedings), pp. 199–204, Atlanta, Georgia, 1988.

[Bartels 87] R. H. Bartels, J. C. Beatty, B. A. Barsky, "*An introduction to splines for use in computer graphics and geometric modeling*", Morgan Kauffmann, 1987.

[Bartels 89] R. H. Bartels, J. C. Beatty, "*A technique for the direct manipulation of spline curves*", Graphics Interface '89 proceedings, pp. 33-39, London, Ontario, 1989.

[Blans 95] C. Blans, C. Schlick, "*X-splines: A spline model designed for the end-user*", Computer Graphics (ACM SIGGRAPH '95 proceedings), pp. 377–386, Los Angeles, California, 1995.

[Böhm 80] W. Böhm, "*Inserting new knots into B-spline curves*", Computer Aided Design, **12**(4), pp. 199–201, 1980.

[Catmull 74] E. E. Catmull, R. J. Rom, "*A class of local interpolating splines*", In R. R. Barnhill and R. F. Riesenfeld, editors, *Computer Aided Geometric Design*, pp. 317–326, Academic Press, 1974.

[Catmull 78] E. Catmull, J. Clark, "*Recursively generated B-spline surfaces on arbitrary topological meshs*", Computer Aided Design, **10**(6), pp. 350–355, 1978.

[Culhane 88] S. Culhane, "*Animation from script to screen*", St. Martin's Press, 1988.

[Doo 78] D. Doo, M. Sabin, "*Behaviour of recursive division surfaces near extraordinary points*", Computer Aided Design, **10**(6), pp. 356–360, 1978.

[Dyn 90] N. Dyn, D. Levin, C. A. Micchelli, "*Using parameters to increase smoothness of curves and surfaces generated by subdivision*", Computer Aided Geometric Design, **7**(1), pp. 129–140, 1990.

[Eck 95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, W. Stuetzle, "*Multiresolution Analysis of Arbitrary Meshes*", Computer Graphics (ACM SIGGRAPH '95 proceedings), pp. 173–182, Los Angeles, California, 1995.

[Eck 96] M. Eck, H. Hoppe, "*Automatic reconstruction of B-spline surfaces of arbitrary topological type*", Computer Graphics (ACM SIGGRAPH '96 proceedings), pp. 325–334, New Orleans, Louisianna, 1996.

[Farin 93] G. Farin, "*Curves and surfaces for computer aided geometric design: A practical guide*", Third Edition, Academic Press, 1993.

[Foley 92] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, "*Computer Graphics: Principles and practice*", Second Edition, Addison-Wesley, 1992.

[Forsey 88] D. R. Forsey, R. H. Bartels, "*Hierarchical B-spline Refinement*", Computer Graphics (ACM SIGGRAPH '88 proceedings), pp. 205–212, Atlanta, Georgia, 1988.

[Forsey 91] D. R. Forsey, "*A surface model for skeleton-based character animation*", Eurographics Workshop on Animation and Simulation (proceedings), pp. 55–73, Vienna, Austria, 1991.

[Forsey 95] D. R. Forsey, R. H. Bartels, "*Surface Fitting with Hierarchical Splines*", ACM Transactions on Graphics **14**(2), pp. 134–161, 1995.

[Fowler 93] B. M. Fowler, R. H. Bartels, "*Constraint based curve manipulation*", IEEE Computer Graphics and Applications, **13**(5), pp. 43–49, 1993.

[Golub 89] G. H. Golub, C. F. Van Loan, "*Matrix computations*", Second Edition, John Hopkins University Press, Baltimore, 1989.

[Grimm 95] C. M. Grimm, J. F. Hughes, "*Modeling surfaces of arbitrary topology using manifolds*", Computer Graphics (ACM SIGGRAPH '95 proceedings), pp. 359–368, Los Angeles, California, 1995.

[Halstead 93] M. Halstead, M. Kaas, T. DeRose, "*Efficient, fair interpolation using Catmull-Clark surfaces*", Computer Graphics (ACM SIGGRAPH '93 proceedings), pp. 35–43, Anaheim, California, 1993.

[Hoppe 92] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, "*Surface reconstruction from unorganized points*", Computer Graphics (ACM SIGGRAPH '92 proceedings), pp. 71–78, Chicago, Illinois, 1992.

[Hoppe 93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, "*Mesh optimization*", Computer Graphics (ACM SIGGRAPH '93 proceedings), pp. 19–26, Anaheim, California, 1993.

[Hoppe 94] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, W. Stuetzle, "*Piecewise smooth surface reconstruction*", Computer Graphics (ACM SIGGRAPH '94 proceedings), pp. 295-302, Orlando, Florida, 1994.

[Kochanek 84] D. H. U. Kochanek, R. H. Bartels, "*Interpolating splines with local tension, continuity and bias control*", Computer Graphics (ACM SIGGRAPH '84 proceedings), pp. 33–42, Minneapolis, Minnesota, 1984.

[Krishnamurthy 96] V. Krishnamurthy, M. Levoy, "*Fitting smooth surfaces to dense polygon meshes*", Computer Graphics (ACM SIGGRAPH '96 proceedings), New Orleans, Louisianna, 1996.

[Kurihara 93] T. Kurihara, "*Interactive surface design using recursive subdivision*", In N. M. Thalmann and D. Thalmann, editors, "*Communicating with virtual worlds*", Springer-Verlag, 1993.

[Lasseter 87] J. Lasseter, "*Principles of traditional animation applied to 3D computer animation*", Computer Graphics (ACM SIGGRAPH '87 proceedings), pp. 35-44, Anaheim, California, 1987.

[Loop 87] C. Loop, "*Smooth subdivision surfaces based on triangles*", Master's thesis, Department of Mathematics, University of Utah, Salt Lake City, Utah, 1987.

[Loop 90] C. Loop, T. DeRose, "*Generalized B-spline surfaces of arbitrary topology*", Computer Graphics (ACM SIGGRAPH '90 proceedings), pp. 347-356, Dallas, Texas, 1990.

[Loop 94] C. Loop, "*Smooth spline surfaces over irregular meshes*", Computer Graphics (ACM SIGGRAPH '94 proceedings), pp. 303-310, Orlando, Florida, 1994.

[Lounsbery 92] M. Lounsbery, S. Mann, T. DeRose, "*Parametric surface interpolation*", IEEE Computer Graphics and Applications, **12**(5), pp. 45–52, 1992.

[Maestri 96] G. Maestri, "*Digital character animation*", New Riders, 1996.

[Nasri 91] A. H. Nasri, "*Surface interpolation on irregular networks with normal conditions*", Computer Aided Geometric Design, **8**(1), pp. 89–96, 1991.

[Piegl 97] L. Piegl, W. Tiller, "*The NURBS Book*", Second Edition, Springer, 1997.

[Rogers 89] D. F. Rogers, J. A. Adams, "*Mathematical elements for computer graphics*", Second Edition, McGraw-Hill, 1989.

[Softimage 96] Softimage, "*The Softimage|3D Reference Guide*", Version 3.5, Microsoft Corp., 1996.

[Softimage 96b] Softimage, "*Working with Softimage|3D*", Version 3.5, Microsoft Corp., 1996.

[SAAPHIRE 96] Softimage, "*The* SAAPHIRE *Reference Guide*", Version 1.1, Microsoft Corp., 1996.

[Sheikh 97] H. S. Sheikh, R. H. Bartels, "*Towards a generic editor for subdivision surfaces*", Shape Modelling and Applications '97 proceedings, pp. 000–000, Aizu-Wakamatsu, Japan, 1997.

[Thomas 81] F. Thomas, O. Johnston, "*The illusion of life: Disney animation*", Hyperion, 1981.

[Wong 95] D. Wong, D. R. Forsey, "*Multiresolution surface Reconstruction for hierarchical B-splines*", Technical Report TR-95-31, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, 1995.

[Zorin 96] D. Zorin, P. Schröder, W. Sweldens, "*Interpolating Subdivision for Meshes with Arbitrary Topology*", Computer Graphics (ACM SIGGRAPH '96 proceedings), pp. 189-192, New Orleans, Louisianna, 1996.

[Zorin 97] D. Zorin, P. Schröder, W. Sweldens, "*Interactive Multiresolution Mesh Editing*", Computer Graphics (ACM SIGGRAPH '97 proceedings), pp. 259-268, Los Angeles, California, 1997.

Appendix A

Tutorial: using Hierarchical B-Splines in Softimage|3D

This tutorial on using hsplines in Softimage|3D assumes that the reader is familiar with the Softimage environment. All the functions used in this tutorial that are not directly part of Softimage|3D will be described in detail, while you may refer to [Softimage 96] for further information on Softimage functions.

Figure A.1 shows the Softimage display as it typically appears, with its toolbars and viewports. The hierarchical spline functions were all added to the **Model** module, and are not accessible from the other Softimage modules.

Hierarchical splines are limited to the B-spline basis in the current implementation, and all hspline objects are in fact hierarchies (in the Softimage sense) of B-spline patches. The level with the least amount of detail, referred to as *level 0*, is at the top of the hierarchy. Regions with refined detail will be added to the hierarchy as children of the patch they refine.

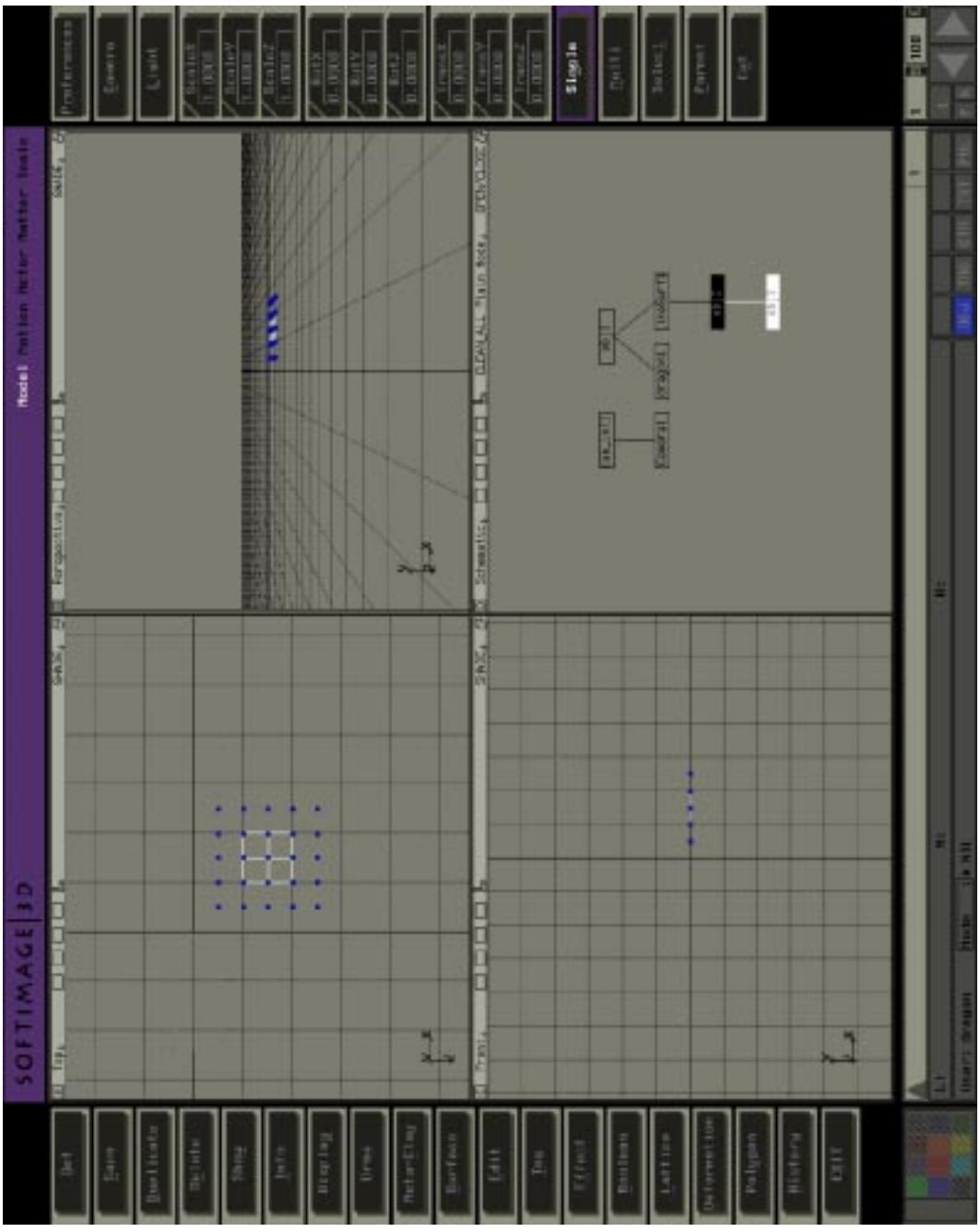


Figure A.1: The Softimage|3D Display

A.1 Creating an Hspline in Softimage

To add a hierarchical spline in the current scene, from the **Model** module select **Get→Primitive→Hspline+**. The + symbol indicates that the selected item is a plugin. The dialog box of figure A.2 will appear.

This dialog box is divided in two halves; the top half allows one to create a simple surface of specified topology: planar, cylindrical or toroidal. The user can specify the number of knots in both u and v. The cylinder is wrapped in v—at the current time it is not possible to request a cylinder wrapped in u.

Figure A.2: Create Hspline dialog box

The bottom half of the dialog box allows one to import a hierarchical spline surface from a `.hs` file. It behaves in a similar way to the Softimage file dialog boxes, and allows you to easily navigate directories through quick access entries: HOME, ROOT, ORIGINAL and PARENT, with their usual meanings. A `.hs` file is a file stored in a format that permits to transfer a collection of hspline surfaces across a wide range of systems.

Files in the current directory are listed in file box, and the current directory may be changed either by navigating the file box or by typing a new path in the **Directory** box and using the **Scan** button to update the list of files. Once a model is selected, it will be imported in the current scene.

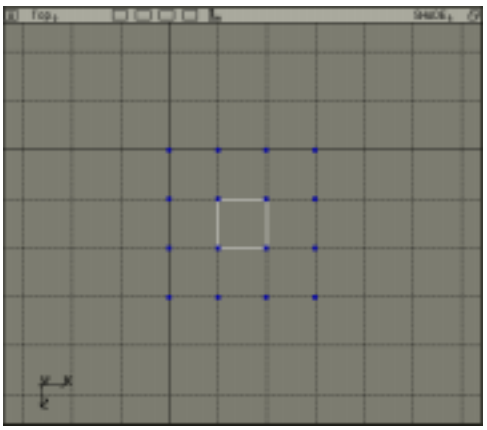


Figure A.3: Square hspline patch

For this tutorial, we will use the default surface: a square B-spline patch with 4 knots in both the u and v directions. The patch appears in the scene, along the XZ-plane. Note that the control vertices are represented by black boxes, which, when selected for manipulation (tagged) become slightly larger. The patch at this point is simply a B-spline patch, and it will respond as expected to the Softimage|3D tools.

Figure A.4: The hspline hierarchy

As can be seen in figure A.4, the hierarchical view of the scene has the model parented to a NULL. Note that the patch is selected in the Softimage hierarchy, and its ancestors and sibling are hidden. Plugins that implement persistent effects need to be represented by an icon, with associated geometry in the scene. This feature is not useful for the hspline plugin, and the icon for the plugin is simply a NULL. This NULL is parented to another NULL which serves as the root of the hierarchy. The main reason for things to be parented in this way is historical, and this might be changed in the future.

A.2 Refining Hspline Surfaces

We are going to refine the patch we just created, but first we need to tag the CVs around which the refinement is to happen. Refer to figure A.5 where the 16 control vertices of the level 0 patch have been tagged. The patch is refined by selecting **Edit→Hspline Refine+**. Figure A.6 shows the new level that was created (called level 1), along with its CVs. Note that level 0 has been unselected (tags have been removed on its CVs) and level 1 is now the selected level in the hierarchy.

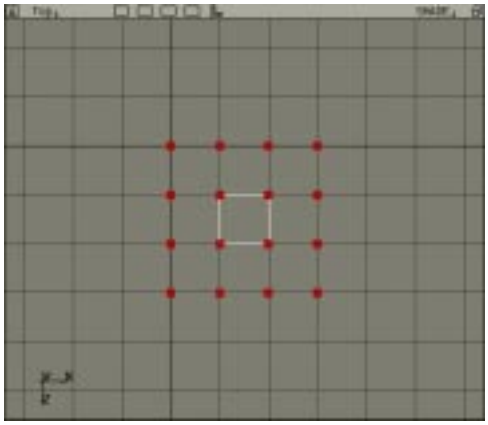


Figure A.5: Tagged CVs for refinement

There are now 25 CVs influencing the surface, however each CV now influences a region only half the size of that influenced by CVs on the parent level, in accordance with the mathematical description of B-spline refinement. This allows for finer control of the shape of the surface, and is the primary reason why you would want to refine a surface. We will discuss how to edit the surface at different levels of the hierarchy in the next section.

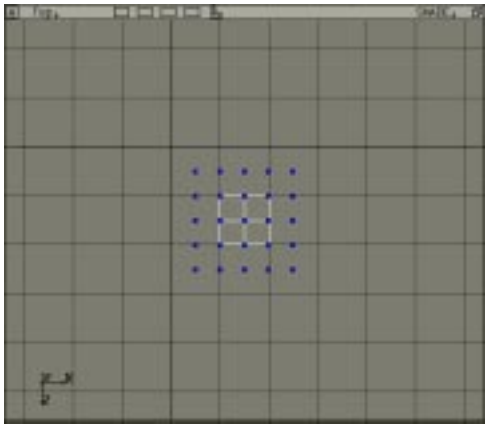


Figure A.6: Level 1 of the refined square patch

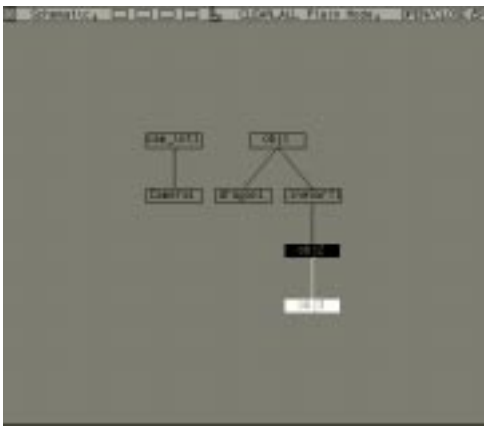


Figure A.7: A 2 level hierarchy

Figure A.7 shows the new level parented to level 0. Note that there can be more than one child to a level, when the refined area cannot be represented with a single B-spline patch. This is common in advanced modelling situations, and is discussed at length in sections 3.3 and 3.6.2. Although in these cases the spline hierarchy is a complex tree, it easier to think in terms of levels.

A.3 Editing Hspline Surfaces

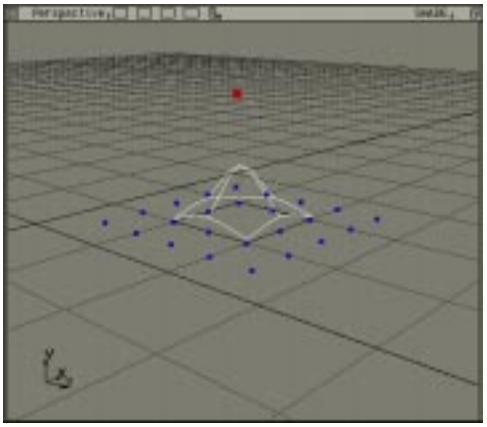


Figure A.8: Editing the patch at level 1

As can be seen in figure A.8, all that is required to edit the hspline surface is to tag the appropriate CV and move it. This is identical to what would be done to edit CVs from a regular B-spline patch. We will see in section A.4 that there are constraints on certain CVs which may prevent them from moving. If you are using the surface in this example, however, all CVs are movable at this point. Note how the surface is affected by the tagged CV.

We are going to refine the surface twice. Tag all the CVs at level 1 and refine the surface. If you look closely at figure A.9 you will notice that with the surface deformed at level 1, the new refined level follows the parent level exactly. The surface is identical to that before the refinement, with more control points. Make sure you understand what happened in the hierarchical view window before continuing with this tutorial. Refine the surface once again, to get a four level hierarchy (levels 0 through 3).

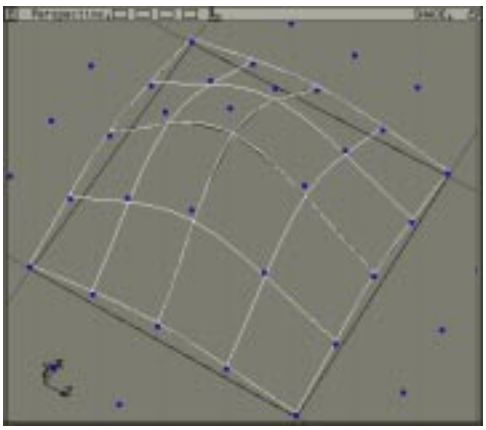


Figure A.9: Refining the patch to level 2

We are now going to edit the surface at level 3. Figure A.10 shows an example of moving two such CVs. Note the area affected by the CVs and compare it to what we were moving at level 1 (figure A.8). You should still be able to see the surface at level 1 in the viewport, but it should be unselected. This makes it useful as a reference since you can easily see what the local deformations are. The scene may become cluttered quickly and we will see how to deal with this at the end of this section.

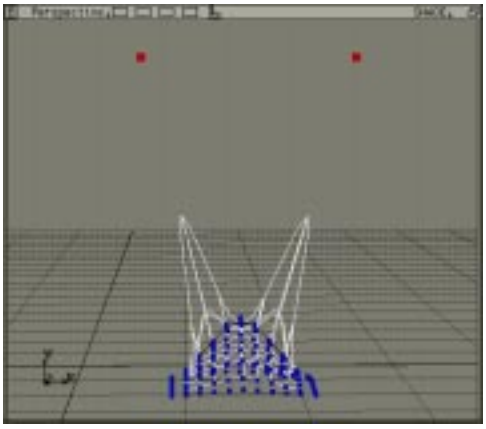


Figure A.10: Editing the patch at level 3

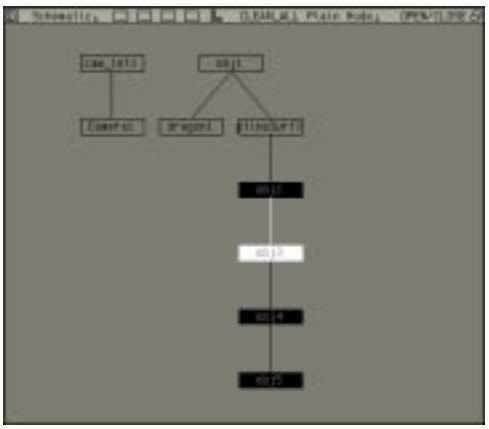


Figure A.11: Selecting CVs at level 1 of a 4 level hierarchy

We can still edit the surface at level 1. To do so, make sure that you are in single selection mode and select level 1 in the hierarchical view of your scene. Level 1 is the third node from the end of the chain—two levels above the finest level of detail (refer to figure A.11). Notice that level 3 is now unselected in the different viewports, and its parent is highlighted. You should still be able to distinguish the contribution of each level to the complete surface.

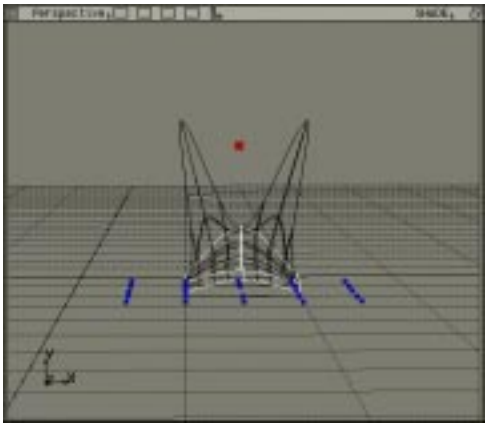


Figure A.12: Selected CVs at level 1

The CV you manipulated earlier at level 1 should still be tagged. Notice how the whole surface is affected when it is moved around. This is the only CV at level 1 which allows us to easily observe the changes to the surface, since the other ones are off towards the edges of the patch. You will notice that the deformations at level 3 "follow" the changes. This is the feature of hierarchical splines that you will find most useful: surfaces may be edited at different levels of resolution. After refining the surface it is still possible to do broad modifications while manipulating very few CVs.

Since the changes at level 1 do not destroy the work done at level 3 you can always go back and work on the surface at that level later if you are not satisfied with it. It is not difficult to understand what is happening intuitively: if you refer to figure A.13 and A.14, you will notice that the changes at level 3 happen relative to the surface at level 2, wherever it may be. Level 2 (which we have not modified) in turn follows the changes in level 1. In practical terms this means that you never have to worry about not being able to do broad level manipulations of the surface once detail has been added to it, since the hierarchy is still there.

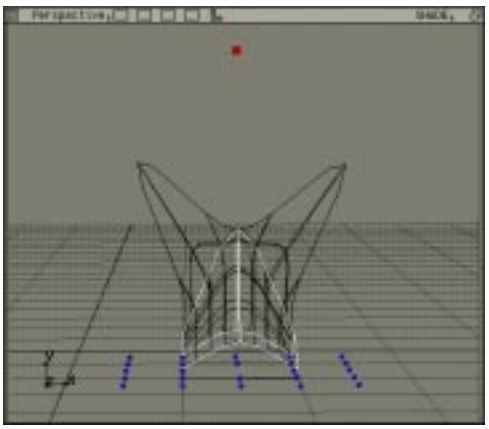


Figure A.13: CV at level 1, moving up

This is the most powerful feature of hierarchical splines and you should make sure that you understand what the consequences are on your modelling habits. We will look at a real life example in the section A.4. Even though the different levels all appear on top of one another, there is nothing in the software to enforce that constraint. It may be useful to translate levels so that they do not overlap for complex models. This will reduce the clutter of having everything superimposed on the screen, and might be of great help to the mod-

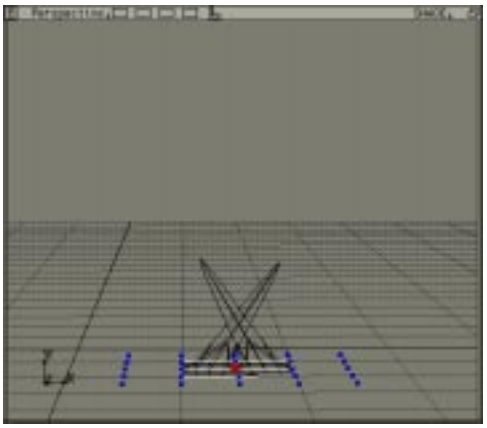


Figure A.14: CV at level 1, moving down

eller. Note that even when the levels are separated in space, they are still connected and manipulations at one level affect lower levels in the hierarchy. You can also hide some levels from view, by picking the level in question and selecting **Display→Hide→Toggle & Deselect Hidden**. Even if a level is hidden it is still be updated internally.

You should never manipulate the hierarchy of surfaces directly. Trying to delete a level by playing with the hierarchy will not work, and the level will be recreated as soon as the plugin notices that it is missing. Section A.6 will examine some of the issues involved in simplifying an hspline and removing levels. If you want to delete the hspline from the scene, you should delete the icon for the hspline plugin, and all the patches will be properly removed from the scene.

A.4 Local Refinement

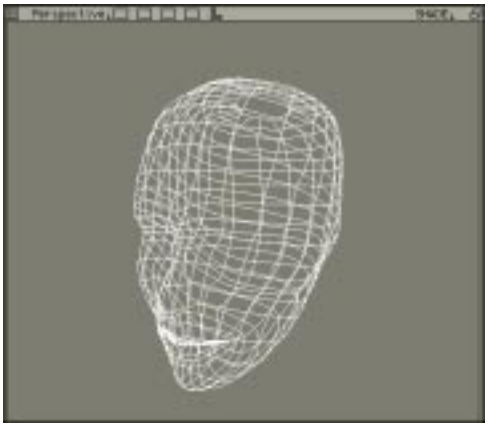


Figure A.15: Head before local refinement

This tutorial is going to tackle a more realistic example. The challenge is to model eyes on a human head. The head has been modelled from a cylinder, with one end located inside the mouth and the other at the top of skull. So far, the cylinder has been uniformly refined, so that every part of the head has the same amount of detail (see figure A.15). There aren't enough controls in the region where the eye is to appear, so we will refine the surface locally in that region.

The goal is to avoid inserting rows of CVs at the back of the head, as would be required with regular spline surfaces using knot insertion techniques. The region of the eye is tagged, and we want to make sure that we grab enough CVs to cover the region of interest (figure A.16). The area is refined as usual, but since we only selected a subset of the parent surface to refine, the refined region is local to the eye. Figure A.17 shows that there are more controls around the eye.

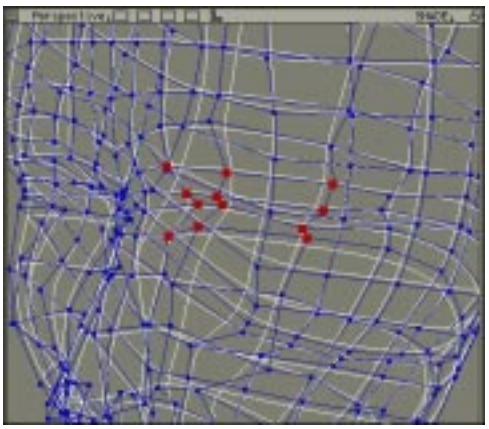


Figure A.16: Selected CVs for local refinement

Since the new patch is a local refinement of the parent surface, there are going to be constraints on its CVs. The hspline plugin always ensures that no cracks can appear between the two levels of the hierarchy. When manipulating bicubic B-splines, this requirement can be understood in the following simple manner: the only CVs that can be manipulated are those that lie at least 2 knots away from the patch edge in every direction. This is a direct consequence of the mathematical formulation of B-splines, and will be strictly enforced by the plugin.

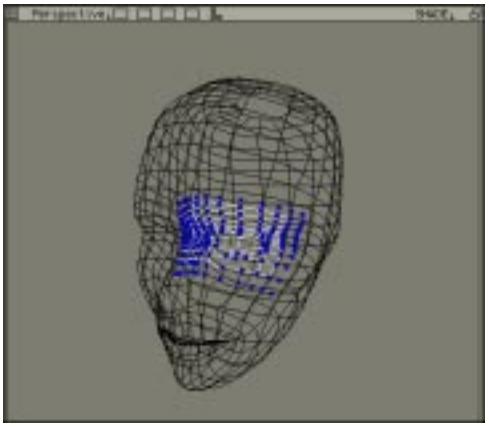


Figure A.17: Head with locally refined patch

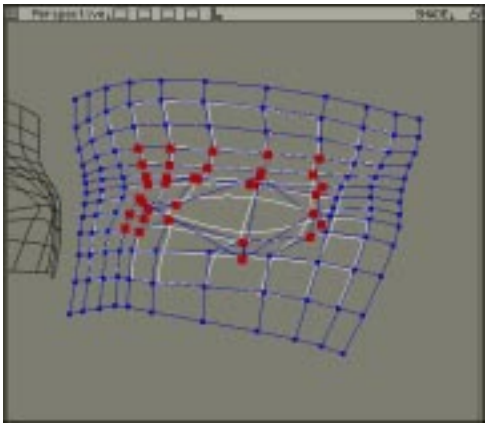


Figure A.18: Editing the new patch

There are a few cases where this may look not to be true, and they should be described for completeness. The original spline surface in figure A.3 did not have this restriction (otherwise none of its CVs would have been movable), since the edge of the surface is part of its description. Once the surface has been refined (figure A.6), the new level still influences the edges of the original patch, and thus all its CVs are also allowed to move. In most cases however, the locally refined patch will not be adjacent to the edge of the level 0 patch, and the 2 knot rule will apply.

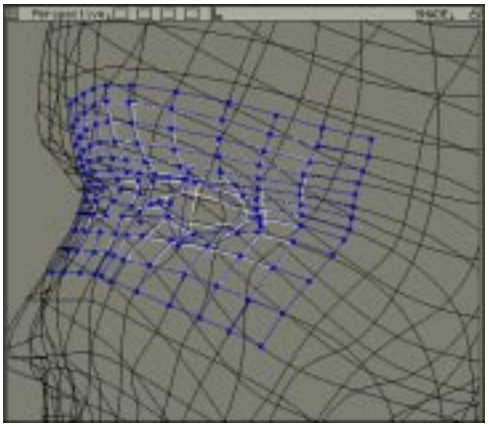


Figure A.19: New head, with locally refined areas

With these restrictions in mind, the new patch can modelled into an eye socket (figures A.18 and A.19). The eyeball can be added later and should be ignored for the rest of this tutorial. The back of the eye socket does not need to realistically surround the eyeball since it will be invisible once rendered. Note how the parent surface was hidden in figure A.18, to avoid cluttering the window. The isolated patch is much easier to edit than with the rest of head present, and we are guaranteed that the surface boundaries will perfectly match.

The plugin does allow the user to move the child patch away from the parent by modifying the transformation matrix associated with it, but you must remember to move it back to the parent surface before rendering, or the output will not meet your expectations. Figure A.20 has the other eye socket in place and is ready for more work. Eyebrows should probably be refined from our latest two patches.

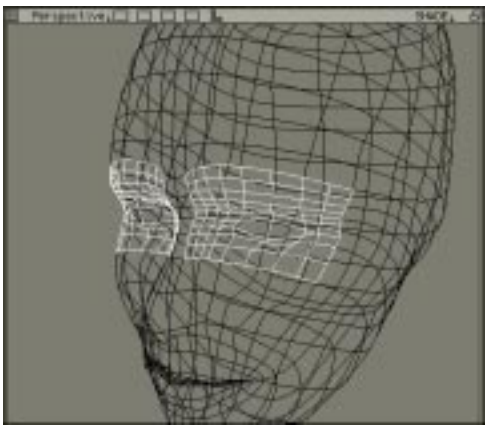


Figure A.20: Final head

If the shape of the head was to be modified at level 0, to model a different character, all the work we have done on the eyes would still be there, although we'd probably want to change the shape of the eye sockets slightly to avoid having many characters with the same features.

A.5 Managing Tagged Points

The rules that govern which CVs are movable and which aren't will become a second nature in almost no time, but there are cases where the surface has been heavily modified and the connectivity of the control mesh is hard to understand, as in figure A.22. Another possible source of confusion has to do with the configuration of patches. The movability of a CV is determined by the hspline representation and not the Softimage patch hierarchy. A CV may be on the edge of a patch and still be movable, if there is a patch with the same parameterization next to the CV. This was illustrated in figure 3.1.

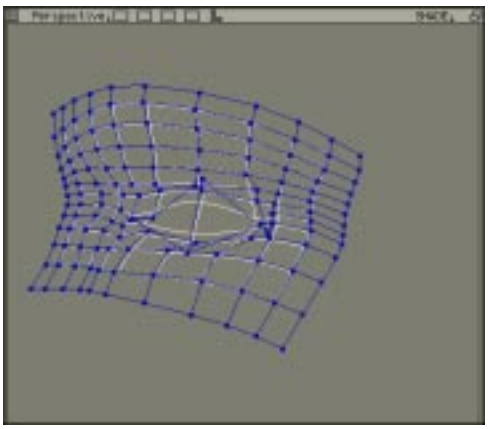


Figure A.21: Untagged B-spline surface

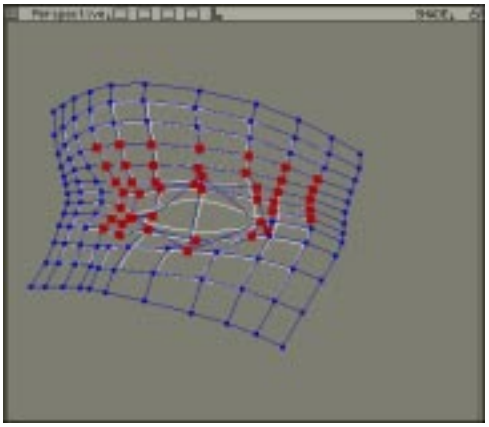


Figure A.22: Movable CVs tagged

An auxiliary plugin has been provided which will tag all the movable CVs on the currently selected surface. It is invoked through **Tags→Tag Movable+**, and is often used to help visualize how a surface has been built. In a similar vein, the **Tags→Untag Fixed+** plugin untags CVs from the currently selected set, so that it contains only movable CVs. This operation is often used during modelling, since it allows the user to make sure that the selected CVs will move.

Figure A.22 shows the result of using the **Tags→Tag Movable+** plugin on the patch of figure A.21. Finally it is possible to request that a level be moved back to where the parent surface lies, by selecting **Edit→Zero Offsets+**. This is used when trying to simplify a surface, or when trying to undo some edits even after the package has forgotten the edit history. This option should be used with caution because subsequent unrefines may remove those CVs, as described in the next section.

A.6 Unrefining

In order to discuss the steps involved in unrefining a surface, we should go back to a simple example, so that the effects of unrefinement may appear clearly. We are going to use the model from the first part of this tutorial. Figure A.23 is identical to figure A.12, and figure A.24 is a top view of the same patches. We can see from these figures that all the levels of the hierarchy are fully refined. One CV at level 1 and two CVs at level 3 have non-zero offsets, while the other ones remain unchanged. The assumption is that the model can be simplified through unrefining, getting rid of unnecessary CVs inside the hierarchy. This need arises when surfaces are refined more than they need to be in the construction process, a fairly common mishap.

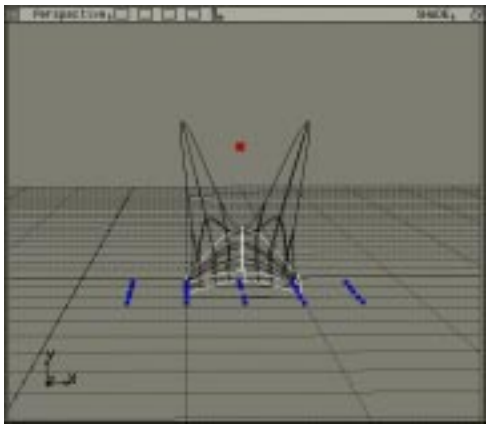


Figure A.23: Before unrefinement

The mathematical restrictions described in section A.4 force CVs on local patches to be at the center of a 4×4 grid. The CVs at level 3 refine CVs at level 2, which must provide support for them, and are thus submitted to those same restrictions. Let us now unrefine the surface, by selecting the **Edit→Hspline Unrefine+** auxiliary plugin. The result of this operation is shown in figure A.25, and we can see that

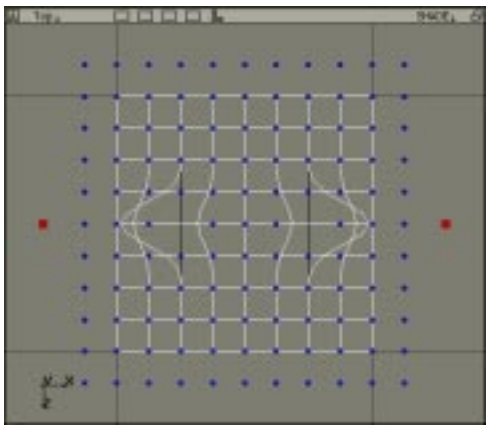


Figure A.24: Top view, before unrefinement

level 3 is now smaller than it was.

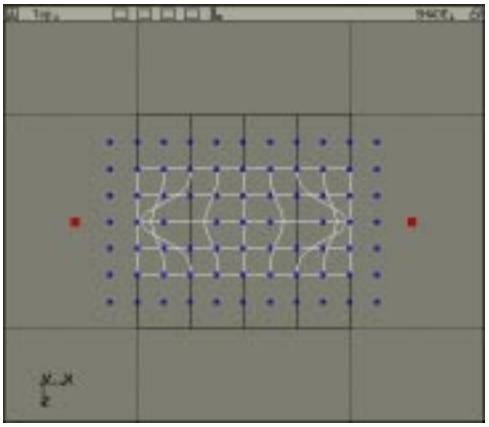


Figure A.25: Top view, after unrefinement

Each non-zero offset CV is now at the center of a 4×4 grid, and because these touch they are part of the same patch at level 3. The CVs that were not in use at level 3 have been removed by the unrefine process, thus trimming the set of movable CVs at that level. It is to be noted that the current implementation of the unrefine plugin is not very user friendly, and all the animation data associated with the model before the operation is lost in the process.

A.7 Adding key bindings in Softimage

It is often practical to define keyboard accelerators (Swift Keys) for some of the functions described above. Refining the tagged area by selecting **Edit→Hspline Refine+** is the prime example of a function that you may want to be able to invoke through a keyboard accelerator.[1]

Follow the steps below to create a keyboard accelerator for surface refinement:

- Select **Preferences→Keyboard Setup→Learn**
- Softimage is now waiting for you to choose a menu cell command that you want associated with a keyboard accelerator. Select **Edit→Hspline Refine+**.

[1] More information on the topic can be found in [Softimage 96b].

- The Key Sequence Setup dialog box is now displayed. It represents your keyboard, and you can choose the accelerator by clicking on the appropriate keys in the dialog box. Both 'r' and 'Shift+r' are already bound to the **Refresh Display** command, so you may want to use 'Ctrl+r' to avoid conflicts. It is important not to rebind Softimage key bindings as people not used to your setup and working at your station may hit 'r' or 'Shift+r' expecting to refresh the display, but causing them to alter your model. Softimage allows you to select either control key to be associated with your keyboard accelerator. You should pick the one that you are most comfortable with, and stick with it.

- There may be other keyboard accelerators that you want to set up. They can be created by repeating the above steps until you are satisfied.

- Now select **Preferences→Keyboard Setup→Save** to save your new keyboard setup to file. If you want to have the accelerator present every time you run Softimage, it is advised to save the file in your home directory and give it the name '`.softimage-keys.sks`'.

In general, you need to select **Preferences→Keyboard Setup→Load** in order to load a new keyboard definition into the current Softimage session. It is possible however to have Softimage|3D load your custom keyboard setup each time it is run by modifying your `.softimage` file. Ask your system administrator to change the `soft` alias in that file to look like this:

```
alias soft $SI_LOCATION/3D/bin/soft $SI_LOCATION/3D/rsrc \
                    -k $HOME/.softimage-keys.sks
```